Student Workbook for *Murach's Mainframe COBOL*

murachbooks@murach.com • www.murach.com

How to use this workbook

This workbook is designed as a companion to our book, *Murach's Mainframe COBOL*. If you are using this workbook as part of a college course or corporate training program, your instructor may give you specific information about how to use the material it contains. In general, though, you can use this material to determine if you understand and can apply the information presented in each chapter of the book. The topics that follow present some additional information about how to use each of the components of this workbook.

Objectives

To help you prepare for taking the tests and doing the projects your instructor assigns, this workbook includes objectives for each chapter. These objectives describe what you should be able to do after reading the chapter. If you want to, you can review the objectives for a chapter before you start reading it so you know what material to focus on. Keep in mind, however, that you may not fully understand the objectives at that point. In any case, you'll definitely want to review the objectives once you've finished the chapter.

If you study the objectives, you can see that the first objectives for most chapters are what we refer to as *applied objectives*. These ask you to apply what you've learned as you develop COBOL programs. If you can develop the applications described in the projects at the end of this workbook, you have met these objectives.

After the applied objectives for each chapter, you'll find what we refer to as *knowledge objectives* (for some chapters, which focus on introductory or conceptual material, all of the objectives are knowledge objectives). These objectives define skills like identifying, describing, and explaining the required concepts, terms, and procedures. In general, you should be able to do the knowledge objectives, even if you have trouble with the applied objectives.

Summaries

This workbook also includes a summary of the information presented in each chapter. After you read a chapter, you can use this summary to review the main concepts and programming techniques presented in it. Then, if you aren't clear about any of these summary points, you can review the topics that present them. That will help you to prepare for the chapter test and the projects your instructor may assign.

Terms

In addition to the summary, a list of terms is presented for each chapter. After reading a chapter, you should scan the list to check that you understand each term. If you don't, you'll want to look back in the chapter to find the terms that are confusing to you so you can see how they're used. As you do that, keep in mind that you're not expected to write definitions of the terms. You just need to understand how they're used in the context of COBOL programming.

Projects

At the end of this workbook, you'll find projects that you should be able to complete after reading each chapter. These projects are intended to test your ability to develop COBOL programs from scratch. Although the program specifications for each project are detailed enough for you to complete the project, you'll need to determine minor details on your own.

Most likely, your instructor will assign one or more of these projects to be completed after reading a particular chapter or section of the book. Your instructor may also give you additional details about how each project should be implemented. If your instructor doesn't assign these projects, you may want to develop one or more of them on your own to be sure that you've mastered the programming techniques that were presented in that chapter or section.

Chapter 1
Introduction to COBOL programming

Objectives

Knowledge

- Given a COBOL listing for a simple interactive program like the ones in this chapter, explain what each statement in the program does.
- Distinguish between a set of COBOL standards and a COBOL compiler.
- Identify the location and use of these portions of a line of COBOL code: indicator column, A margin, B margin.
- List three rules for forming any name in a standard COBOL program.
- Describe the purpose of a Picture clause and a Value clause.
- Distinguish between alphanumeric, numeric, and numeric edited items.
- Distinguish between a group item and an elementary item.
- Describe the operation of any of the statements presented in this chapter:

Accept	Display	Perform
Add	If	Perform Until
Compute	Move	Stop Run

Summary

- COBOL is a standard programming language that can be run on any computer that has a *COBOL compiler* to create the machine language that will run on the computer. The current compilers for IBM mainframes are based on the 1985 standards.
- An *interactive program* gets its input data from the keyboard of a mainframe terminal or PC, and it displays its output data on the screen of a terminal or PC monitor. In contrast, a *batch program* processes the data in one or more disk files without interacting with the user.
- A COBOL program consists of four divisions: Identification, Environment, Data, and Procedure.
- When you code a program, some lines must start in the *A margin* and some in the *B margin*. If a line has an asterisk in column 7, it is treated as a *comment*, which means that it is ignored by the compiler. To make a program easier to read, you can use *blank comments* or blank lines for vertical spacing and extra spaces for horizontal spacing.
- The Working-Storage Section in the Data Division defines the *data items* that are used by the program. For each item, you code a Picture clause to define the format. You can also code a Value clause to give the item a starting value. A data item can also be referred to as a *field* or *variable.*

- The Procedure Division consists of *procedures*, or *paragraphs*. When a program is run, the computer executes the statements in this Division in sequence starting with the first one in the first procedure, unless a Perform or Perform Until statement changes this sequence by performing one of the other procedures.
- The Accept and Display statements are input and output statements. The Move statement moves data from one field in storage to another, sometimes with editing. The Compute and Add statements are arithmetic statements. The If statement is a logical statement. And the Stop Run statement ends the execution of a COBOL program.

Terms

IBM mainframe
structured programming
structured COBOL
object-oriented COBOL
COBOL standards
COBOL-85
COBOL-2002
compile
COBOL compiler
extension
interactive program
batch program
comment
indicator column
blank comment
A margin
B margin
syntax
data name
data item
reserved word
variable
variable name
field
alphanumeric item
numeric item
numeric edited item
byte
literal
alphanumeric literal
numeric literal
figurative constant
group item
elementary item
procedure name
paragraph name
procedure
sending field
receiving field
editing
arithmetic expression
arithmetic operator
order of precedence
delimiter
relational operator
nested If statements
processing loop
loop

Chapter 2
How to compile, test, and debug a COBOL program

Objectives

Knowledge

- Describe the process of compiling, linking-editing, and executing a program on a mainframe using the proper terms for the input and output files in each step of the procedure.
- Describe the basic procedure for developing COBOL programs in terms of what the programmer has to do.
- Describe the difference between testing and debugging.
- Describe the difference between compile-time and run-time errors.
- Explain why you should start your new programs from old programs.
- Explain how you can use COBOL statements to get debugging information.

Summary

- When you test a COBOL program, the compiler first compiles the *source program* into an *object module*. Next, the *linkage editor* links the object module with any other object modules that are required and prepares a *load module*. Then, the computer executes the load module.
- A *compile-time error* occurs when the compiler can't compile a statement because its syntax is incorrect. A *run-time error* occurs when the computer can't execute a statement in the load module.
- *Testing* refers to the process of finding errors. *Debugging* refers to the process of finding the causes of the errors and fixing them.
- When a program finishes with a Stop Run statement, it is called a *normal termination*. When a program finishes with a run-time error, it is called an *abnormal termination*, or *abend*.
- *Diagnostic messages* are part of the compiler output. These messages identify the compile-time errors.
- The most common cause of a run-time error, or *exception*, is a *data exception*. Other common causes are *decimal-overflow exceptions* and *decimal-divide exceptions*.
- You can use the *debugging tools* that your compiler offers to speed up the process of debugging. Another alternative is to add COBOL statements to your programs that will provide the debugging information that you need.

Terms

source program
object module
compiler listing
compile-time error
linkage editor
link editor
load module
run-time error
source code
clean compile
normal termination
abnormal termination
abend
testing
debugging
diagnostic message
exception
data exception
decimal-overflow exception
decimal-divide exception
debugging tools

Chapter 3
How to write a program that prepares a report

Objectives

Applied

- Given the specifications for a report-preparation program like the ones in this chapter, develop a program that prepares the report.
- Given the specifications for a report-preparation program, prepare a test plan for it.

Knowledge

- Describe the characteristics of a sequential file on disk.
- Describe the use of a print chart.
- Describe the use of a switch.
- List two ways that you can get the current date and time into a program.
- Describe each of these types of errors: operation exception, data exception, decimal-overflow exception, and decimal-divide exception.

Summary

- A *sequential file* on disk consists of *records* that are read in sequence. Each record consists of *fields*.
- To plan a report, you create a *print chart* that shows the content and format of each line to be printed.
- A *switch* is a field that can have a value of either yes or no (or on or off). All business programs require one or more switches.
- An *input file* is a file that is read by a program, and an *output file* is a file that is written to. A *print file* is an output file that contains a report that can be printed.
- For each file in a program, there must be a Select statement in the Environment Division and an FD statement and record description in the Data Division.
- Before you can read from or write to a file, you must open it. Before a program ends, you must close all open files.
- A Read statement for a sequential file reads the next record in the file, starting with the first record. A Write statement for a print file writes one record for each line of the report.
- If you're using a compiler that supports the *intrinsic functions*, you can use the Current-Date function to get the current date and time from the system.

- A *test plan* lists the test phases and describes the test data to be used in each phase.
- Four common types of run-time errors on a mainframe are *operation exceptions*, *data exceptions*, *decimal-overflow exceptions*, and *decimal-divide exceptions*.

Terms

sequential file
record
field
print chart
print file
switch
file name
system name
ddname
VSAM (Virtual Storage Access Method)
spooling
print area
input file
output file
intrinsic function
test plan
operation exception
data exception
decimal-overflow exception
decimal-divide exception

Chapter 4
How to design, code, and test a structured program

Objectives

Applied

- Given input file specifications and report specifications like the ones in this chapter, create a structure chart or structure listing for the program.
- Given program specifications and a structure chart or a structure listing for a program, write pseudocode for the critical modules.
- Given program specifications and a structure chart or structure listing for a program, create a test plan for testing all the modules. This plan should include a description of the test data to be used for each phase.
- Given program specifications, a structure chart or structure listing, and a top-down test plan, code and test all the modules of the program.

Knowledge

- Explain how control fields and control breaks are used in programs that prepare summary reports.
- Given a structure chart for a program, identify the calling modules, called modules, common modules, and work modules in each chart leg.
- Distinguish between a structure chart and a structure listing.
- List three guidelines for coding program modules so they're independent from one another.
- Describe the process of top-down coding and testing.
- Explain what is meant by "commenting" a line of code or "stubbing off" a module.
- (If you have experience with an object-oriented language) List three ways that COBOL programs differ from object-oriented programs.

Summary

- *Structured programming* refers to the development techniques that include *structured design*, *structured coding*, and *top-down coding and testing*.
- A report that includes totals for groups of records is called a *summary report*. The summary lines in a report like this are printed whenever the value in a *control field* changes, which is referred to as a *control break*.
- With *structured design*, you design a program from the top down. To do that, we recommend you create a structure chart.

- A *structure chart* is a design document that shows all the *modules* of a program and their hierarchy. A *calling module* connects to the modules below it, and a *called module* is subordinate to the module above it. Each module should represent a single function.
- *Pseudocode* is a programmer's shorthand you can use to plan the logic for the critical modules of your program or to plan the coding of any module.
- Each module in a program should be as independent as possible. To accomplish this, a module should contain *all* of the code for the function represented by the module and only the code for that function.
- With *top-down coding and testing*, you begin by coding and testing the top module of the program along with a few subordinates. When these work correctly, you code and test a few more modules at a time, until all modules are coded and tested.
- When a module calls a subordinate module that isn't required for a testing phase, you can code a *program stub* in place of the subordinate. The stub can contain just enough code to complete the testing phase.

Terms

structured programming
structured program
structured coding
summary report
detail report
control field
control break
structured design
structure chart
hierarchy chart
calling module
called module
control module
subordinate module
work module
common module
chart leg
leg of a chart
structure listing
pseudocode
top-down coding and testing
top-down testing
commenting out a line
uncommenting a line
program stub
dummy module
global scope
local scope
method

Chapter 5
How to use the COBOL features for structured coding

Objectives

Applied

- Given the specifications for a two-level summary report like the one in this chapter, develop a program that prepares the report.
- Use any of the features presented in this chapter as you develop programs of your own.

Knowledge

- Describe the three valid structures of structured programming.
- Explain how the principles of substitution and combination lead to a proper program.
- Explain what is meant by "shop standards." Then, list three typical Data Division standards and three typical Procedure Division standards.
- Describe the characteristics of relation, sign, and class tests.
- Explain how the use of condition names can improve coding.
- Describe the proper use of an inline Perform statement.
- Describe the characteristics of a case structure.
- Explain how the use of Set to True statements can improve coding.
- Explain what structured delimiters are and when you might want to use them.

Summary

- In *structured programming* terms, a *proper program* is made up of *sequence*, *selection*, and *iteration structures*.
- In COBOL, statements that don't include conditions implement the sequence structure. The If statement implements the selection structure. The Perform Until statement implements the iteration structure.
- You should use your *shop standards* for structured programming if your COBOL shop has established them. Otherwise, you should follow the recommendations in this book.
- For the condition that's used in an If, Perform Until, or Evaluate statement, you can code a *relation*, *sign*, or *class test*. You also can code *compound conditions*.
- *Condition names* describe the conditions that are related to switches and flags. A *switch* is a one-character field that can have either a Yes or a No value. A *flag* is a one-character field that can have other values.

- In general, *nested If statements* become difficult to read when they reach the third or fourth level. However, *linear nests* are easy to read no matter how many levels deep they go.
- *Inline Perform statements* are useful when they let you code a single function in a single paragraph. They shouldn't be used to combine two or more functions in a single paragraph.
- You can use the Evaluate statement to improve the clarity of nested If statements that are three or more levels deep. The Evaluate statement also can be used to implement a *case structure*.
- You should use Set to True statements to turn on all conditions that are defined as condition names. And you should use *structured delimiters* to end statements that are coded within other statements.
- You can use Not clauses whenever they make sense, which is normally in input/output statements.
- A *two-level summary report* shows group totals at two different levels.

Terms

structured programming
sequence structure
Do structure
selection structure
If-Then-Else structure
iteration structure
Do While structure
Do Until structure
Perform Until structure
entry point
exit point
proper program
structured coding
shop standards
condition
relation test
sign test
class test
compound condition
logical operator
implied subject
implied operator
condition name
switch
flag
nested If statements
linear nest
inline Perform statement
case structure
structured delimiter
two-level summary report

Chapter 6
Other ways to define, move, and initialize fields

Objectives

Applied

- Apply the coding features and recommendations presented in this chapter to the programs that you develop.
- Given EBCDIC hex codes for a zoned-decimal or packed-decimal field, tell what value is stored in the field.

Knowledge

- Explain how Usage clauses improve the efficiency of a program.
- List three ways you can initialize fields in working storage, and name the preferred way.
- Explain why you should code an S in the Picture clause for a numeric field and why you should code an odd number of digits in the Picture clause for a packed-decimal field.

Summary

- The Picture clause lets you define five different data types. Of these, you'll use *alphanumeric*, *numeric*, and *numeric edited* fields most of the time.
- The Usage clause lets you specify the way that you want numeric fields to be stored. Because this improves the efficiency of a program, you should code either Packed-Decimal or Binary usage for all numeric fields that are going to be used in computations or numeric comparisons.
- The Redefines clause lets you define a field in two or more ways.
- When you use the Move statement to move data to a numeric edited field, the data is *edited*. When you move data from a numeric edited field, the data can be *de-edited*.
- The Initialize statement can be used to initialize fields in working storage. Most of the time, though, it's better to use Value clauses to initialize fields.
- On an IBM mainframe, EBCDIC is used to represent the characters in the *character set*. This code also determines the *collating sequence* of the computer.
- Two *hexadecimal* digits can be used to represent the data in one byte of storage. Decoding hex data is often useful when you're debugging.
- Numeric data can be stored in five different formats on an IBM mainframe. The three that you normally use with COBOL programs are *zoned-decimal*, *packed-decimal*, and *binary*.

Terms

alphabetic item
alphanumeric item
alphanumeric edited item
numeric item
numeric edited item
sending field
receiving field
editing
de-editing
zero suppression
fixed dollar sign
floating dollar sign
check protection
floating plus sign
floating minus sign
byte
binary digit
bit
zone bits
digit bits
hexadecimal code
hex code
character set
EBCDIC
ASCII
collating sequence
zoned-decimal format
packed-decimal format
binary format

Chapter 7
How to use arithmetic statements and intrinsic functions

Objectives

Applied

- Apply any of the functions or arithmetic statements presented in this chapter to the requirements of the programs you develop.

Knowledge

- Explain how the intermediate result fields in Compute statements can lead to computational errors and what you can do to fix them.

Summary

- When you code a Compute statement that requires two or more operations, *intermediate result fields* are produced. These fields can cause computational errors if they don't provide for enough decimal positions.
- Besides the Compute statement, COBOL provides Add, Subtract, Multiply, and Divide statements. You should use the Compute statement for most arithmetic operations, though, because it's easier to use and understand.
- The 1989 addendum to the 1985 COBOL standards defined 42 *intrinsic functions* that were made standard in COBOL-2002.
- Most functions require one or more *arguments*. The functions themselves are normally coded within Move or Compute statements.

Terms

intermediate result field
intrinsic function
function
argument
mean
median

Chapter 8
How to work with dates

Objectives

Applied

- Apply any of the features presented in this chapter to the requirements of the programs you develop.

Knowledge

- Describe the Y2K problem.
- Explain how date expansion and century windowing were used to solve the Y2K problem.
- Explain how IBM's Millennium Language Extensions were used to solve the Y2K problem.

Summary

- The 1985 COBOL standards provide for four variations of the Accept statement that get the standard date, time, Julian date, and day of the week. Both dates, however, are formatted with two-digit years.
- The 1989 extensions to the COBOL standards provide for six date functions that can simplify date-handling routines. These functions work with dates that have four-digit years.
- The *Y2K problem* was caused by the use of dates that contained two-digit years. Many of these routines would no longer work when the current year became 2000.
- *Date expansion* and *century windowing* are the two basic solutions to Y2K problems. Expansion of all dates is the long-term solution, while windowing is a short-term solution.
- To help solve the Y2K problem, IBM provided *Millennium Language Extensions*. After you use the Date Format clause to tell the compiler where the years are located in date fields, the statements in the Procedure Division automatically expand two-digit years to four digits.

Terms

Julian date
function
argument
Gregorian date
Y2K problem
date expansion
century windowing
century window
Millennium Language Extensions
date pattern
compiler option
fixed century window
sliding century window

Chapter 9
How to work with characters

Objectives

Applied

- Apply any of the features, functions, or statements presented in this chapter to the requirements of the programs you develop.

Knowledge

- Explain how reference modification can be used in conjunction with the String, Unstring, or Inspect statement.

Summary

- *Reference modification* lets you refer to characters within a field by providing *offset* and *length* values.
- Eight functions for working with characters became available with the 1989 addendum to the 1985 standards. Of these, the Numval, Numval-C, Upper-Case, and Lower-Case functions are the most useful.
- The String statement lets you combine two or more fields into a single field, while the Unstring statement lets you extract two or more fields from a single field.
- The Inspect statement can be used to count or replace characters within a field.

Terms

reference modification
offset
length
intrinsic function
function
argument
string
concatenate
delimiter
unstring
character string

Chapter 10
How to work with tables

Objectives

Applied

- Given specifications for a program that uses a one-level or a multi-level table, develop the program using either subscripts or indexes. This may require loading a table, searching a table, or processing the entries in a table.

Knowledge

- Explain why using indexes is more efficient than using subscripts.
- List three ways that you can refer to a table entry using subscripts and three ways that you can refer to a table entry using indexes.
- Describe the difference between a sequential and a binary search.
- In general terms, describe the difference between working with a fixed-length table and a variable-length table.
- Explain how you can use the entries in a table with intrinsic functions.

Summary

- When you work with a table, you can use either *indexes* or *subscripts*. An index represents a *displacement value*, and a subscript represents an *occurrence number* that must be converted to a displacement value. That's why using indexes is more efficient than using subscripts.
- A *one-level table* contains data that depends on a single variable factor. A *multi-level table* contains data that depends on two or more variable factors. A table can be defined with constant values or it can be loaded from a file at the beginning of any program that uses it.
- If you use subscripts to work with a table, you can refer to a table entry using the subscript name, an *occurrence number*, or a *relative subscript*. If you use indexes, you can refer to a table entry using the index name, an occurrence number, or a *relative index*.
- To load a table, you use a Perform Varying statement that varies the value of the subscript or index for the table. To search a table, you can use a Perform Varying statement for each level of the table.
- When you define a table with indexes, you can use the Search statement to perform a *sequential search* on the table. If the entries are in sequence by a *key field*, you can also use the Search All statement to perform a *binary search*.
- You use the Set statement to work with indexes or *index data items*. You can use this statement to assign an index value or to increase or decrease an index value.

- If you define a table as variable-length, the actual number of entries in the table must be stored in a working-storage field that's identified in the table definition. When you search a variable-length table using the Search or Search All statement, only the occurrences that contain entries are searched.
- You can use one Perform Varying statement to vary the indexes or subscripts for all the levels of a multi-level table.
- You can use the values in a table as the arguments for an intrinsic function that requires a series of values.

Terms

one-level table
table entry
subscript
occurrence number
relative subscript
multi-level table
index
displacement value
relative index
index data item
sequential search
serial search
binary search
key field

Chapter 11
How to use copy members and subprograms

Objectives

Applied

- Given the documentation for available copy members, use them in your programs.
- Given the documentation for available subprograms, use them in your programs.
- Given the specifications that you need for creating a copy member on your system, create a new copy member.
- Given the specifications for a subprogram, write it in COBOL.

Knowledge

- Explain why it's usually better to implement a processing routine as a subprogram instead of as a copy member.

Summary

- *Copy members* contain source statements that can be copied into COBOL programs by using the Copy statement. On most platforms, each copy member is stored in a *copy library*.
- *Subprograms* are compiled programs that do commonly-used routines. To *call* a subprogram from a *calling program*, you use the Call statement.
- On a mainframe, subprograms are compiled into *object modules* that are stored in an *object library*. Then, the object modules for the calling program and the subprograms are *link edited* into a *load module* that is ready for execution.

Terms

partitioned data set
copy library
source statement library
copy member
copy book
subprogram
calling a subprogram
calling program
passing fields to a subprogram
parameter
argument
object module
link editing
load module
subprogram library
object library
nested subprograms

Chapter 12
Concepts and terms for working with disk files

Objectives

Knowledge

- Describe the way that data is stored on a disk drive.
- List the four steps that are taken when disk data is read or written.
- Explain how blocking improves the efficiency of I/O operations.
- Describe sequential file organization.
- Describe indexed file organization for a file that has only a primary index.
- Describe the use of alternate indexes with indexed files.
- Describe relative file organization.
- Describe the operation of an interactive update program.
- Describe the operation of a random update program in a batch system.
- Describe the operation of a sequential update program in a batch system.

Summary

- A *disk drive* consists of *disks* that have recording surfaces that consist of *tracks*. These tracks are divided into *sectors* that have unique *disk addresses* so they can be read and written on a random basis.
- To access data on a disk drive, the *actuator* moves the *read/write heads* to the *cylinder* that contains the requested sector. Then, one read/write head is turned on, the sector rotates to the head, and the sector is read or written. In this process, the *actuator movement* and *rotational delay* take the most time.
- *Blocked records* make better use of disk storage and reduce rotational delay, so they are commonly used with all types of disk files. This includes files that have *fixed-length records* and files that have *variable-length records*.
- The records in a *sequential file* are read and written in sequence by a *key field*.
- The records in an *indexed file* are accessed through a *primary index* or an *alternate index*. Although the keys in the primary index have to be *unique*, *duplicate keys* are allowed in alternate indexes. Since the keys in an index are kept in key sequence, the records in an indexed file can be read with *sequential* or *random access*.
- The records in a *relative file* can be accessed randomly by *relative record number*.

- *VSAM* is the set of access methods on an IBM mainframe that provide for the use of sequential, indexed, and relative files. Today, VSAM files are used for indexed files, but non-VSAM sequential files are still in common use.
- Typical *interactive programs* include *inquiry*, *update*, and *maintenance programs*. Since these programs get data from a user and perform the requested file processing, random access of the master records is required.
- Typical *batch programs* include *random update*, *random maintenance*, *sequential update*, *sequential maintenance*, and *report preparation programs*. These programs are often run in off hours after the transactions have been collected during the day.
- Before sequential update or maintenance programs can be run, the transactions need to be sorted into the key sequence of the master file. Then, the update or maintenance program reads the *old master file* and writes a *new master file* that has been updated by the transactions.
- Two benefits of sequential updating and maintenance are (1) the automatic creation of a backup file and (2) improved efficiency due to reduced actuator movement and rotational delay.

Terms

disk drive
disk
spindle
track
sector
gigabyte
disk address
DASD (direct access storage device)
actuator
read/write head
cylinder
actuator movement
rotational delay
millisecond
microsecond
nanosecond
blocked records
unblocked records
blocking factor
fixed-length records
variable-length records
sequential file organization
sequential file
sequential access
key field
key
indexed file organization
indexed file
data component
index component
index
primary key
primary index
unique key
duplicate keys
direct access
random access
alternate index
alternate key
non-unique key
relative file organization
relative file
relative record number
record area
randomizing routine
VSAM (Virtual Storage Access Method)
data set
ESDS (entry-sequenced data set)
KSDS (key-sequenced data set)
RRDS (relative record data set)
interactive program
system flowchart
inquiry program
maintenance program
transaction
editing a transaction
update program
batch program
edit program
random update program
random maintenance program
sort program
sequential update program
old master file
new master file
backup file
sequential maintenance program

Chapter 13
How to work with sequential files

Objectives

Applied

- Given the specifications for a sequential update or maintenance program, develop the program.
- Given the specifications for a program that processes a file of variable-length records, develop the program. The variable-length records can contain two or more record types or a varying number of segments.

Knowledge

- Explain how you use the File Status clause of the Select statement in a program.
- List the four Open modes you can use with a sequential file along with the I/O statements that you can use with each mode.
- In general terms, explain how matching record logic works in a sequential update or maintenance program.
- In general terms, describe the error processing that's done in the COBOL programs that are developed for IBM mainframes.
- Explain how you determine whether a test run for a sequential update or maintenance program has worked correctly.
- Describe two techniques that you can use to work with files that contain variable-length records.

Summary

- You can open a sequential file in *input mode*, *output mode*, *extend mode*, or *I-O mode*. The mode you use determines what I-O operations you can perform on the file.
- You can use the File Status clause in the Select statement for a sequential file to name a field that will receive a *file status code* after each I/O operation for the file. That value will indicate what the result of the operation was.
- A *sequential update program* creates a new master file from an old master file and a file of transactions that update fields in the old records. A *sequential maintenance program* creates a new master file from an old master file and a file of transactions that can add, delete, and change records in the master file.
- Sequential update and maintenance programs use *matching record logic* that's based on the comparison of *control fields* in the master and transaction records. This logic determines when records are read from the transaction and old master files and when records are written to the new master file.

- Error processing is a critical part of the coding in update and maintenance programs on IBM mainframes. You often need to *abend* a program when a serious error occurs.
- A file of *variable-length records* can have two or more record types with different lengths or a segment that occurs a varying number of times. You can process a file of variable-length records using either the Record Is Varying In Size or the Record Contains clause of the FD statement.

Terms

fixed-length records
ddname
input mode
output mode
extend mode
I-O mode
file status code
sequential update program
matching record logic
control field
matched transaction
unmatched transaction
apply a transaction
sequential maintenance program
abend
storage dump
variable-length records
segment
root segment
repeating segment

Chapter 14
How to work with indexed files

Objectives

Applied

- Given complete program specifications, develop a program that processes an indexed file sequentially, randomly, or dynamically. The program may use the primary key or an alternate key to access the records in the file.

Knowledge

- Name the three access modes you can use for an indexed file.
- Describe two ways that a program can detect I/O errors for indexed files.
- Describe the operation of the Read, Write, Rewrite, and Delete statements when used with random access of indexed files.
- Describe the operation of the Start statement and name the two types of access that it can be used with.
- Explain how you can use file status codes to determine when all of the records for a alternate key value have been processed.
- In general terms, describe dynamic processing.
- In general terms, describe skip-sequential processing.

Summary

- You can process an indexed file using *sequential access*, *random access*, or *dynamic access* that's based on the file's *primary index* or an *alternate index*.
- You can use file status codes to determine whether an I/O operation on an indexed file was successful. You can also use the Invalid Key clause of the Read, Write, Rewrite, Start, and Delete statements to catch the four file status codes that indicate an invalid key condition.
- You can use the Start statement to start sequential processing of an indexed file at a specific location.
- To process an indexed file by alternate keys, you code the Alternate Record Key clause on the Select statement for the file. If *duplicate keys* are allowed, you must also code the With Duplicates phrase.
- To read a record randomly using an alternate key, you must include the Key clause in the Read statement. To read records sequentially based on an alternate key, you can use the Start statement to position the file and then Read statements to read the records sequentially.

- You can use the File Status field to determine when all of the records with an alternate key value have been processed. A file status code of 02 indicates that the file contains at least one more record with the same key. A file status code of 00 indicates that there are no more records with the same key.
- You can use *dynamic processing* when you want to read the first record in a group randomly and then read the other records in the group sequentially. You can also use *skip-sequential processing* to process the records in a group.

Terms

primary index
sequential access
random access
dynamic access
alternate index
duplicate keys
unique keys
dynamic processing
concatenated key
skip-sequential processing

Chapter 15
How to work with relative files

Objectives

Applied

- Given complete program specifications, develop a program that processes a relative file.

Knowledge

- In general terms, describe how you process a relative file sequentially.
- In general terms, describe how you process a relative file randomly.

Summary

- You can access a *relative file* sequentially, randomly, or dynamically. If you access the file randomly or if you use a Start statement to position the file, the Select statement for the file must identify the field that will contain the *relative record number* of the record to be accessed.
- The Relative Key field that's named in the Select statement for a relative file must be defined in working storage.
- You code the I/O statements for a relative file the same way you do for an indexed file. The difference is that the records are retrieved by their relative record numbers rather than by their key values.
- When you read a relative file sequentially, the program automatically skips over any record areas that don't contain data.
- When you process a relative file randomly, you can use the Invalid Key clause of the Read, Write, Rewrite, Delete, and Start statements to trap invalid key conditions just like you do for indexed files.

Terms

relative file
relative record number
randomizing routine

Chapter 16
How to use the sort/merge feature

Objectives

Applied

- Given complete program specifications, develop a COBOL program that includes an internal sort or merge.

Knowledge

- In general terms, explain how you perform an internal sort or merge from a COBOL program.
- Explain why an internal sort or merge is typically more efficient than a standalone sort or merge.

Summary

- When you *sort* a file, the records in the file are sequenced based on one or more *key fields*.
- When you *merge* two or more files, the records in the files are combined into a single file and are sequenced based on one or more key fields.
- Most systems provide a *sort/merge program* that you can use to sort and merge the records in one or more files. When you execute this program directly, it's called a *standalone sort* or *standalone merge*. When you execute it from a COBOL program, it's called an *internal sort* or an *internal merge*.
- To perform an internal sort from a COBOL program, you use the Sort statement. To perform an internal merge, you use the Merge statement. In either case, you must define a work file that will provide a work space for the sort/merge program.
- For a sort operation, you can use an *input procedure* to select the records to be sorted from the input file. This procedure must include a Release statement to pass the selected records to the sort/merge program. If you don't use an input procedure, all of the records in the input file are sorted.
- After the records are sorted, you can use an *output procedure* to process the sorted records. This procedure must include a Return statement to retrieve the records from the sorted file. If you don't use an output procedure, the sorted records are written to an output file.
- You can't use an input procedure for a merge operation, which means that all of the records in the input files are merged. However, you can use an output procedure. If you don't, all of the merged records are written to an output file.

Terms

sort
merge
sort/merge program
sort program
key field
sort key
collating sequence
internal sort
standalone sort
internal merge
standalone merge
sort work file
input procedure
output procedure
merge work file
concatenate
sort register

Chapter 17
Introduction to IBM mainframes

Objectives

Knowledge

- Identify the three basic components of an IBM mainframe processor.
- Identify the main difference between the z/Architecture found on IBM's zSeries processors and earlier system architectures.
- Name and briefly describe four types of I/O devices that are commonly found on an IBM mainframe.
- Describe these five features of mainframe computer operating systems: virtual storage, multiprogramming, spooling, batch processing, and time sharing.
- Describe how MVS, OS/390, and z/OS are related to one another.
- Name three subsystems or facilities that are commonly found on an OS/390 or z/OS system.

Summary

- An *IBM mainframe* consists of a *central processing unit* (*CPU*), *main storage*, *channels*, and *I/O devices*. Today, mainframes are also called *servers* to highlight their adaptability to e-commerce applications.
- *Terminals* are the workstations that are used on mainframes, although PCs running *emulation software* are commonly used as terminals today.
- *Direct access storage devices*, or *DASDs*, are the disk devices that are used on a mainframe.
- *Printers*, *magnetic tape* devices, and *optical disk* devices are other types of I/O devices used on a mainframe.
- *Virtual storage* is a software feature that lets a large amount of main storage be simulated by a processor that has a smaller amount of *real storage*. Then, *multiprogramming* is used to run more than one program at a time in the virtual storage.
- *Spooling* manages printer output by intercepting the output and storing it on disk so it can be printed later on. Spooling runs as one of the programs that's being multiprogrammed.
- *Job control language*, or *JCL*, describes the *jobs* that are to be run by the system. The system's *job scheduler* controls the execution of the jobs that are submitted for *batch processing*.
- With *time sharing*, many users can do *interactive processing* at the same time. The time sharing program runs as a batch job in the multiprogramming environment.

- The *MVS* (*Multiple Virtual Storage*) operating system makes up the core of the *OS/390* and *z/OS* operating systems, which are the primary operating systems used on mainframes today.
- Besides the operating system, the IBM mainframe environment consists of many facilities and subsystems that the COBOL programmer needs to use.

Terms

The terms that follow are those that are most critical to your understanding of IBM mainframe systems as you develop COBOL programs. A number of other terms were presented in this chapter, however, and you're likely to hear them used in your conversations with other application programmers, systems programmers, and operators.

processor
CPU (central processing unit)
main storage
cache
channel
multiprocessor system
channel
input/output (I/O) device
z/Architecture
mainframe server
3270 display station
terminal
3270 emulation software
DASD (direct access storage device)
line (impact) printer
character (dot-matrix) printer
page (laser) printer
magnetic tape
optical disk
operating system
z/OS
OS/390
systems programmer
virtual storage
real storage
multiprogramming
spooling
batch processing
job
job control language (JCL)
job scheduler
time sharing
interactive processing
online processing
foreground processing
background processing
MVS (Multiple Virtual Storage)
subsystem
JES/JES2/JES3
TSO
ISPF
PDF
CICS
DB2
SQL
utility program

Chapter 18
How to compile and test programs on an IBM mainframe

Objectives

Applied

- Use ISPF to create partitioned data sets for storing your COBOL source programs and the jobs for compiling, link-editing, and executing them.
- Use the ISPF editor to create a COBOL source program. Then, create and submit a job that uses a cataloged procedure to compile and link-edit the program.
- Use ISPF to create a sequential data set that will contain the data used as input to your program.
- Use SDSF to display the output from a compile-and-link job. If the job output contains compile-time errors, correct the errors and compile and link-edit the program again.
- Create and submit a job to execute your COBOL program, or use TSO commands to execute it. If a run-time error occurs, correct the problem and compile, link-edit, and execute the program again.

Knowledge

- Describe the three steps involved in compiling and testing a program. Include a description of the input and output to each step.

Summary

- *ISPF* provides a menu-driven, full-screen interface to most of the features of TSO. You can use ISPF to develop your COBOL programs and the jobs that compile, link-edit, and execute them.
- The files on an IBM mainframe are called *data sets*. The two types of data sets you'll use most often are *partitioned data sets* and *sequential data sets*.
- A partitioned data set consists of a *directory* and one or more *members*. You use partitioned data sets to store your source programs, object modules, load modules, and the JCL to compile, link-edit, and execute them.
- Before you can use a data set, you must create, or *allocate*, it. You can do that using ISPF's Data Set utility.
- You can use the ISPF editor to create new members and edit existing members. You can also use the editor to enter data into a sequential data set.

- When you use Job Control Language (JCL), a *job* is identified by a JOB statement, and each job consists of one or more *job steps* that are identified by EXEC statements that execute programs or procedures. To allocate the data sets required by a program or procedure, you code DD statements.
- IBM provides *cataloged procedures* that include most of the JCL you need to compile, link-edit, and execute your COBOL programs. To use a cataloged procedure, you code a job that executes it and supplies the required input and output data sets.
- To execute an existing program, you can use JCL or TSO commands. TSO commands are especially useful for running additional test runs for the same load module without recompiling and linking.
- You can use *SDSF* to work with jobs and their output. To do that, you use SDSF options to display the jobs in the *input queue* and the job output in the *output queue* and the *held output queue*.

Terms

compile-link-and-go procedure
source program
compile
object module
link
linkage editor
load module
command area
panel
program function (PF) key
data set
partitioned data set
PDS
library
sequential data set
directory
member
allocating a data set
qualifier
high-level qualifier
edit profile
edit data display
heading area
primary command
line command area
line command
screen window
identifier field
name field
operation field
parameters field
job step
job name
step name
ddname
cataloged procedure
loader
compiler option
concatenating libraries
SDSF (System Display and Search Facility)
input queue
output queue
held output queue
job-id
job log
return code
JCL listing
message log

Chapter 19
How to use Access Method Services to work with VSAM files

Objectives

Applied

- Given the specifications for an AMS job that uses any of the commands covered in this chapter, code the job.

Knowledge

- Explain the function of the Integrated Catalog Facility (ICF) for VSAM files.
- In general terms, explain how VSAM files are stored.
- In general terms, explain how alternate indexing works and how it affects the efficiency of VSAM file access.
- List three AMS commands that you're likely to use and describe what they do.

Summary

- The *Integrated Catalog Facility* (*ICF*) for OS/390 and z/OS provides for one *master catalog* and an unlimited number of *user catalogs* that identify user data sets. By convention, the *high-level qualifier* in a data set name is the name of the user catalog in which it is entered.
- A *key-sequenced data set* (*KSDS*) in VSAM is a *cluster* that consists of an *index component* that contains the primary keys for the records and a *data component* that contains the data. To facilitate the addition of records, *free space* can be assigned to the data component.
- An alternate index is actually a KSDS with the alternate keys in the index component and the primary keys in the data component. This leads to significant processing overhead.
- *Access Method Services* (*AMS*) is a utility that provides commands for working with both VSAM and non-VSAM files.
- You use the DEFINE CLUSTER command to define a VSAM file, and you use the LISTCAT command to list information for the entries in a catalog.
- You use the PRINT command to print a data set, the REPRO command to copy a data set, and the DELETE command to delete a data set.
- You use the DEFINE ALTERNATEINDEX command to define an alternate index, you use the DEFINE PATH command to define the path that lets you access the base cluster via the alternate index, and you use the BLDINDEX command to build an alternate index.

Terms

KSDS (key-sequenced data set)
ESDS (entry-sequenced data set)
RRDS (relative record data set)
record management
volume serial number
ICF (Integrated Catalog Facility)
master catalog
user catalog
high-level qualifier
alias
cluster
control interval
control area
free space
control interval split
control area split
index set
sequence set
free pointer
buffer
base cluster
AMS (Access Method Services)
IDCAMS
functional command
primary space allocation
secondary allocation
retention period
expiration date
reusable data set
cross-region share option
cross-system share option
generic entry name
upgrade set
path

Chapter 20
How to use CICS to develop interactive programs

Objectives

Knowledge

- Distinguish between a transaction and a task.
- Name two CICS modules that support application programming.
- Explain how pseudo-conversational programming works and why it's recommended for CICS programs.
- Describe two functions that the CICS translator performs.
- Describe the use of the Execute Interface Block and its EIBCALEN and EIBAID fields in a COBOL program.
- Describe the use of the DFHAID copy member in a COBOL program.
- Describe the purpose of a BMS mapset, and explain how you code one.
- In general terms, describe the coding of the first procedure in a CICS program.
- Describe how I/O operations are handled in a CICS program and how that differs from a standard COBOL program.
- Explain the benefits of web-enabling CICS applications instead of writing new web applications.
- List two facilities that IBM provides for web-enabling CICS applications.

Summary

- Every CICS program is associated with a *transaction* that can be invoked by entering a *transaction-id*. When you invoke a transaction, CICS starts a *task* that runs within CICS and uses CICS storage.
- CICS provides functional modules that support CICS programs. The terminal control and *BMS* (*Basic Mapping Support*) modules control interactions with the display station, and the file control modules control interactions with the access methods like VSAM.
- To use BMS, you code an *assembler language* program called a *mapset* that contains the definitions of one or more *maps*. Then, you assemble the mapset to create a *physical map* that's used to determine the appearance of the data displayed on the screen and a *symbolic map* that's used by the COBOL program.

- CICS programs are typically implemented using *pseudo-conversational programming* to minimize the system resources they use. With this type of programming, a program ends after it sends data to a terminal. Then, CICS restarts the program when the user completes an entry.
- To use CICS services from a COBOL program, you code CICS commands. Then, before you compile the program, the *CICS translator* converts the commands to COBOL code that can be compiled by the COBOL compiler.
- The CICS translator also inserts the *Execute Interface Block* (*EIB*) into the Linkage Section of a COBOL program. The EIB includes an EIBAID field that contains the one-byte value that represents which key the user pressed and an EIBCALEN field that provides the length of the *communication area* that's passed to the program.
- Since the files that are used by a CICS program are defined in a CICS table called the *File Control Table*, you don't code Select or FD statements in the program. You also use CICS commands for all I/O operations.
- Because pseudo-conversational programming is so similar to the way web applications work, it's relatively easy to web-enable COBOL/CICS programs. Two facilities that help you web-enable COBOL/CICS programs are (1) the *CICS Transaction Gateway* and (2) *CICS Web Support*.

Terms

CICS (Customer Information Control System)
transaction
transaction identifier
trans-id
task
BMS (Basic Mapping Support)
mapset
map
pseudo-conversational programming
attention identifier (AID) key
conversational program
CICS translator
Execute Interface Block (EIB)
communication area
program function (PF) key
program attention (PA) key
assembler language
assembler
physical mapset
symbolic mapset
macro
attribute byte
CUA (Common User Access)
screen painter
symbolic mapset
symbolic map
FCT (File Control Table)
web browser
HTTP (Hypertext Transfer Protocol)
HTML (Hypertext Markup Language)
CICS Transaction Gateway
CICS Web Support (CWS)
External Presentation Interface (EPI)
stateless environment
presentation logic
business logic
External Call Interface (ECI)
external CICS interface (EXCI)
Java class
class library
Java gateway application

Chapter 21
How to use DB2 to develop database programs

Objectives

Knowledge

- Describe how a table in a relational database is organized, and explain what a primary key and an index are.
- Explain how two tables in a relational database are related.
- List the three types of relationships that can exist between two tables in a relational database.
- Name the SQL statement that you use to retrieve data from a relational database, and describe the results of that statement.
- Name the three SQL statements that you can use to modify the data in a relational database.
- Explain what a join is, and describe the most common type of join.
- In general terms, describe the syntax that you use to include an SQL statement in a COBOL program.
- Explain what a cursor is and when you need to use one in a COBOL program.
- Name the four SQL statements you use to process a cursor-controlled result table within a COBOL program, and explain the purpose of each statement.
- Explain what a host structure is and how you use it in a COBOL program.
- Describe the purpose of the DB2 precompiler and list its two types of output.
- In general terms, explain what binding a program means.

Summary

- A *relational database* consists of one or more *tables*. Each table contains one or more *rows*, or records, and each row contains one or more *columns*, or fields.
- Most tables contain a *primary key* that uniquely identifies each row in the table and one or more *indexes* that improve access to the rows in the table.
- Three types of relationships can exist between tables: a *one-to-one relationship*, a *one-to-many relationship*, and a *many-to-many relationship*. Tables are typically related by a *foreign key* in one table that refers to a primary key in another table.
- To work with the data in a database, you use *SQL* (*Structured Query Language*). To retrieve data, you use the SQL SELECT statement to create a *result table* that can consist of one or more rows. To modify data, you use the SQL INSERT, UPDATE, and DELETE statements.

- A SELECT statement can *join* data from two or more tables. The most common type of join is an *inner join* or *equi-join*. With this type of join, rows from the two tables are included in the result table only if their related columns match.
- To access DB2 data from a COBOL program, you use *embedded SQL*. Embedded SQL statements can refer to fields that are defined in the COBOL program, called *host variables*.
- If a result table will contain two or more rows, you have to use a *cursor* to process one row at a time. DB2 provides special SQL statements for defining and processing a *cursor-controlled result table* within a COBOL program.
- You can use a DB2 utility called *DCLGEN* (*Declarations Generator*) to generate a description of a DB2 table, called a *host structure*, from the table declaration for the table. You use a host structure within a COBOL program just as you use a record description for a sequential or indexed file.
- Before compiling a program that accesses DB2 data, you have to run the *DB2 precompiler* to translate the SQL statements into COBOL statements that invoke DB2 functions. This produces a modified source program and a *database request module* (*DBRM*) that contains information about how your program will use DB2.
- After you compile and link edit a program to create a load module, DB2 must *bind* the program to check all the DB2 functions it uses. You can bind a program directly into an *application plan*, or into a *package* and then into a plan.

Terms

relational database
DB2 (Database 2)
relational database management system (RDBMS)
table
row
column
value
string value
null value
primary key
index
unique index
one-to-many relationship
foreign key
one-to-one relationship
many-to-many relationship
SQL (Structured Query Language)
result table
query
action query
calculated value
join
inner join
equi-join
outer join
left outer join
right outer join
full outer join
embedded SQL statements
embedded SQL
host variable
cursor
cursor-controlled result table
host structure
DCLGEN (Declarations Generator)
table declaration
SQL communication area
DB2 precompiler
modified source program
DBRM (database request module)
binding a program
application plan
package
DB2 catalog
DB2 directory
DB2I (DB2 Interactive)

Chapter 22
How to become an effective maintenance programmer

Objectives

Applied

- Given the specifications for a program change, the source code for the program, and any other available documentation, make the required change.

Knowledge

- Describe the operation of any of the COBOL statements or features presented in this chapter.
- Explain how you should document a program within its source code.

Summary

- Perform Thru statements let you execute more than one procedure in a single Perform statement. Within the performed procedures, Go To statements are commonly used to pass control to the exit procedure.
- Prior to COBOL-85, Go To Depending statements were often used to implement the case structure.
- A program can be divided into *sections* within the Procedure Division. Then, you can use Perform statements to perform sections.
- You use *qualification* to identify non-unique data names. You can also use the Corresponding phrase in Move, Add, and Subtract statements to move, add, or subtract the non-unique data items in a group.
- A Use statement within a *declarative section* is executed when it is triggered by an I/O error produced by the specified file.
- To help you deal with the poorly-written or unstructured code that you're likely to encounter when you maintain a program, you should use a structured maintenance procedure.
- The primary documentation for any program is its source code. In the Identification Division of this code, you should include comments that provide a program description and a *modification log*.

Terms

legacy program
maintenance programmer
branch to a paragraph
fall through to the next paragraph
section
section name
qualification
declarative section
declaratives
modification log

Student projects

Guidelines that apply to all of the projects

Install the project files on your C drive

- **If you downloaded this Workbook from our web site,** the download includes all of the files that you need to do the projects, and they've been installed on your system in folders that start with C:\Murach\Mainframe COBOL\Projects.

  If you got this Workbook as part of a class, your instructor may provide the project files you need. Or, you can download them from our web site: Go to www.murach.com, go to the page for *Murach's Mainframe COBOL*, choose the Download link for the book, and download the Student Materials file (it's an executable zip file named mcb2_student_materials.exe). Double-click on the file in the Windows Explorer or My Computer display to unzip all of the files so they're stored in a folder structure that starts with C:\Murach\Mainframe COBOL\Projects.

- Within the Projects folder, you'll find a subfolder named Data that contains the data files that you need for the projects. The extension used for these files is dat.
- You'll also find a subfolder named Copy that contains the copy member files that you can use with the projects. The extension for these files is cpy.
- If you're going to develop the projects on your PC, you should copy the data and copy member files to appropriate folders before you use them. For example, you may want to copy them to C:\MCOBOL\Data and C:\MCOBOL\Copy. That will make them easier to work with. Later, if you want to restore the original state of a file, you can do that by copying it from the original folder.
- If you're going to develop the projects on some platform other than a PC, you'll need to store the data and copy member files in a way that's appropriate for that platform. You may also need to do some other types of file conversion.

Don't enter the record descriptions into your programs

- On the next two pages, you'll see two record descriptions that are used by the projects. These record descriptions are stored in copy member files in the C:\Murach\Mainframe COBOL\Projects\Copy folder.
- Although you can type the record descriptions into your programs, that's both time consuming and error-prone. Instead, use one of these alternatives: (1) **If you haven't read chapter 11** in the text yet, use the techniques that are appropriate on your system to copy a record description from its copy member file into your program; (2) **If you have read chapter 11,** use the Copy statement instead.

Always start a new program from an old program

- As the COBOL book emphasizes, you should always start a new program from an old one. At first, you can start your new programs from the program examples in the book (you can download these from our web site). Or, if your instructor has assigned any of the exercises for the book, you can use those as starting points. Eventually, of course, you'll have a set of your own programs to use as prototypes.

Use consistent names for your programs

- To keep the names for your programs simple and consistent, we recommend that you name each program SP (student project) followed by the project number. For instance, you can use SP02-3 as the program name for project 2-3 and SP10-1 as the name for project 10-1.

The record description for the student master file

```
01  STUDENT-MASTER-RECORD.
    05  SM-STUDENT-ID                  PIC 9(09).
    05  SM-STUDENT-STATUS              PIC X(01).
        88  ENROLLED                   VALUE "E".
        88  INACTIVE                   VALUE "I".
    05  SM-STUDENT-NAME-AND-ADDRESS.
        10  SM-STUDENT-NAME            PIC X(25).
        10  SM-DATE-OF-BIRTH.
            15  SM-DOB-YEAR            PIC 9(04).
            15  SM-DOB-MONTH           PIC 9(02).
            15  SM-DOB-DAY             PIC 9(02).
        10  SM-STUDENT-ADDRESS         PIC X(25).
        10  SM-STUDENT-CITY            PIC X(11).
        10  SM-STUDENT-STATE           PIC X(02).
        10  SM-STUDENT-ZIP-CODE        PIC 9(05).
        10  SM-STUDENT-ZIP-CODE-EXT    PIC 9(04).
    05  SM-STUDENT-PROGRESS-SUMMARY.
        10  SM-CLASS-STANDING          PIC 9(01).
            88  FRESHMAN               VALUE 1.
            88  SOPHOMORE              VALUE 2.
            88  JUNIOR                 VALUE 3.
            88  SENIOR                 VALUE 4.
        10  SM-MAJOR                   PIC X(04).
        10  SM-UNITS-COMPLETED         PIC 9(03).
        10  SM-TOTAL-GRADE-POINTS      PIC 9(03).
        10  SM-UNITS-IN-PROGRESS       PIC 9(03).
```

88 levels

- Several of the definitions in the record description are at the 88 level, which isn't presented until chapter 5. Until you read that chapter, then, you should know that these levels indicate what the values in the fields above can be; they don't require any additional bytes of storage.
- The 88 levels for sm-class-standing, for example, just indicate that this one-digit field can store values of 1, 2, 3, and 4, which stand for freshman, sophomore, junior, or senior class standing.

File sequence

- The primary sequence of the records in this file is by sm-class-standing. This means that all the freshmen are first in the file, followed by sophomores, juniors, and seniors.
- The records are also in sequence in two other ways. First, the records are in sequence by sm-student-id within sm-class-standing. Second, the records are in sequence by sm-student-id within sm-major within sm-class-standing. In the real world, the same file of records couldn't be in sequence in both of these ways, but we set up the data this way so you could use one file for preparing two different types of summary reports.

The record description for the course registration file

```
01  COURSE-REGISTRATION-RECORD.
    05  CR-COURSE-KEY.
        10  CR-DEPARTMENT-CODE          PIC X(04).
        10  CR-COURSE-NUMBER            PIC 9(03).
        10  CR-SECTION-NUMBER           PIC 9(02).
    05  CR-COURSE-INFORMATION.
        10  CR-COURSE-TITLE             PIC X(20).
        10  CR-COURSE-START-DATE.
            15  CR-COURSE-START-YEAR    PIC 9(04).
            15  CR-COURSE-START-MONTH   PIC 9(02).
        10  CR-COURSE-UNITS             PIC 9(01).
        10  CR-COURSE-DAYS              PIC 9(01).
    05  CR-REGISTRATION-INFORMATION.
        10  CR-TEACHER-NUMBER           PIC 9(03).
        10  CR-STUDENT-INFORMATION.
            15  CR-STUDENT-ID           PIC 9(09).
            15  CR-STUDENT-NAME         PIC X(25).
            15  CR-CLASS-STANDING       PIC 9(01).
                88  FRESHMAN            VALUE 1.
                88  SOPHOMORE           VALUE 2.
                88  JUNIOR              VALUE 3.
                88  SENIOR              VALUE 4.
            15  CR-MAJOR                PIC X(04).
    05  CR-GRADING-INFORMATION.
        10  CR-6-WEEKS-GRADE            PIC X(01).
        10  CR-12-WEEKS-GRADE           PIC X(01).
        10  CR-FINAL-EXAM-GRADE         PIC X(01).
        10  CR-SEMESTER-GRADE           PIC X(01).
```

88 levels

- In chapter 5, you'll learn what the 88 levels mean. For now, you should know that the 88 levels for cr-class-standing just indicate that this one-digit field can store values of 1, 2, 3, and 4, which stand for freshman, sophomore, junior, or senior class standing.

File sequence

- The primary sequence of the records in this file is by cr-class-standing. This means that all the freshmen are first in the file, followed by sophomores, juniors, and seniors.
- The records are also in sequence in two other ways. First, the records are in sequence by cr-student-id within cr-class-standing. Second, the records are in sequence by cr-student-id within cr-major within cr-class-standing. In the real world, the same file of records couldn't be in sequence in both of these ways, but we set up the data this way so you could use one file for preparing two different types of summary reports.

About the projects for chapter 2

These projects require the COBOL skills that you learned in chapter 1 and the program development skills that you learned in chapter 2. These projects are simply designed to get you going and to give you some ideas for creating projects of your own.

Project 2-1 Calculate grade point average

This interactive program accepts user entries for grade points and credits. For each set of entries, the program should calculate the grade point average (GPA) by dividing grade points by credits and rounding to two decimal places. The interactive session should be something like this:

```
------------------------------------------------
Calculate another GPA (Y/N)?
Y
------------------------------------------------
Enter the number of grade points for the semester.
41
Enter the number of credits taken.
15
The grade point average is 2.73.
------------------------------------------------
Calculate another GPA (Y/N)?
N
End of session.
------------------------------------------------
```

Project 2-2 Calculate totals and averages

This program accepts a series of numeric entries. For each series, the program should calculate the total and average of the numbers. The interactive session should be something like this:

```
------------------------------------------------
End program (Y/N)?
N
------------------------------------------------
Enter a series of numbers that range from 1-999.
Press the Enter key after each entry.
To end the series, enter 0.
------------------------------------------------
13
871
411
12
0
------------------------------------------------
The total of the numbers is    1,307
The average of the numbers is    326.75
------------------------------------------------
End program (Y/N)?
Y
End of session.
```

About the projects for chapters 3, 4, and 5

The goal of chapters 3, 4, and 5 is to teach you how to prepare report-preparation programs the way the best professionals prepare them. As a result, the projects that follow ask you to prepare a variety of reports that require a variety of design structures and logical approaches.

Project 3-1 Prepare a student listing

This program reads the student master file and prepares a listing of students.

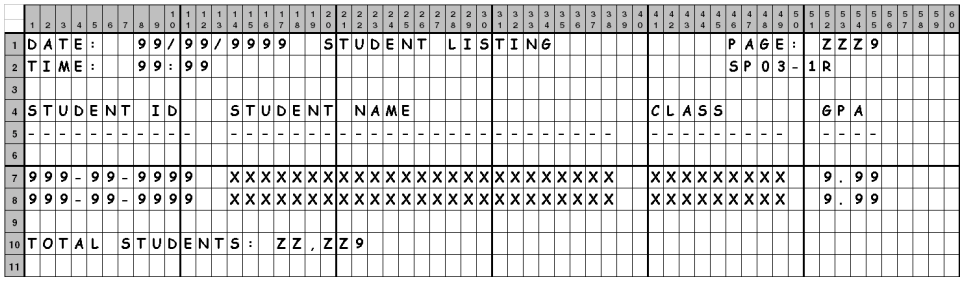
Files used by the program

Filename	Description	Mode
STUMAST	Student master file	Input
STULIST	Student listing	Printer output

Specifications

- Print one student line on the report for each record in the student master file.
- The student ID and student name should be taken from the master record. The GPA should be calculated by dividing the total grade points in the record by the units completed.
- The data in the class column should be derived from the number that's stored in the class standing field in the student master record where 1 = FRESHMAN, 2 = SOPHOMORE, 3 = JUNIOR, and 4 = SENIOR.
- The total number of students in the listing should be printed at the end of the report.

Print chart



Project 3-2 Prepare an honor student report

This program reads the student master file and creates a report that lists honor students.

Files used by the program

Filename	Description	Mode
STUMAST	Student master file	Input
HONORRPT	Honor student report	Printer output

Specifications

- The report should include only those students with a GPA of 3.5 or higher.
- The student ID and student name should be taken from the master record. The GPA should be calculated by dividing the total grade points in the record by the units completed.
- The data in the class column should be derived from the number that's stored in the class standing field in the student master record where 1 = FRESHMAN, 2 = SOPHOMORE, 3 = JUNIOR, and 4 = SENIOR.
- If a student has a GPA of 3.8 or higher, the report should print RANKING SCHOLAR to the right of the student's GPA.
- The total number of students listed and the number of ranking scholars should be printed at the end of the report.

Print chart

```
DATE:  99/99/9999          HONOR STUDENT REPORT                PAGE: ZZZ9
TIME:  99:99                                                   SP03-2R

STUDENT ID    STUDENT NAME               CLASS     GPA
-----------   -------------------------- --------- ----

999-99-9999   XXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXX 9.99   XXXXXXXXXXXXXXX
999-99-9999   XXXXXXXXXXXXXXXXXXXXXXXXXX XXXXXXXXX 9.99   XXXXXXXXXXXXXXX

TOTAL STUDENTS:   ZZ,ZZ9
RANKING SCHOLARS: ZZ,ZZ9
```

Project 4-1 Prepare a student roster

This program reads the student master file and creates a student roster. This report includes summary lines for each class (freshman, sophomore, junior, and senior).

Files used by the program

Filename	Description	Mode
STUMAST	Student master file	Input
SLSTROST	Single line student roster	Printer output

Specifications

- Print one student line for each record in the student master file. These records are in sequence by class standing.
- When the class changes, print the total number of students in the class and the grade point average for the class in two lines as shown in the print chart. For readability, skip a line before and after these summary lines.
- The data in the class column should be derived from the number that's stored in the class standing field in the student master record where 1 = FRESHMAN, 2 = SOPHOMORE, 3 = JUNIOR, and 4 = SENIOR.
- These printed fields should be taken from the master record: student ID, student name, units completed, and units in progress. The GPA should be calculated by dividing the total grade points by the units completed.
- At the end of the report, print the total number of students and the grade point average for all of the students as shown in the print chart. Your program should skip two lines before printing them.

Print chart

```
DATE:     99/99/9999                    STUDENT ROSTER                                  PAGE:  ZZZ9
TIME:     99:99                                                                         SP04-1R

                                                                    UNITS          UNITS IN
CLASS      STUDENT ID     STUDENT NAME                          COMPLETED       PROGRESS      GPA
----------  -----------  ----------------------------------  -----------  -----------  ----

XXXXXXXXX   999-99-9999  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX        ZZ9          ZZ9        9.99
            999-99-9999  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX        ZZ9          ZZ9        9.99
            999-99-9999  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX        ZZ9          ZZ9        9.99

TOTAL STUDENTS IN CLASS:  ZZ,ZZ9
CLASS GPA:                9.99

STUDENT TOTAL:  ZZ,ZZ9
STUDENT GPA:    9.99
```

Project 4-2 Prepare a multi-line student roster

This program reads the student master file and prepares a student roster that includes the complete address for each student. This report also includes summary lines for each class (freshman, sophomore, junior, and senior).

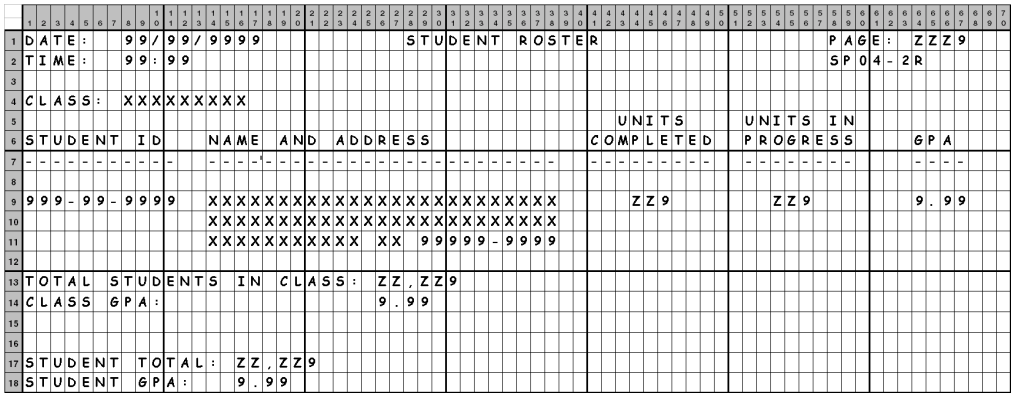
Files used by the program

Filename	Description	Mode
STUMAST	Student master file	Input
MLSTROST	Multi-line student roster	Printer output

Specifications

- Print three student lines for each record in the student master file, and skip one line after the lines for each student.
- The student records are in sequence by class standing. When the class changes, print the total number of students in the class and the grade point average for the class in two lines as shown in the print chart.
- The students for each class should start on a new page. The fourth heading line on this page should include the class name: FRESHMAN, SOPHOMORE, JUNIOR, or SENIOR.
- These printed fields should be taken from the master record: student ID, student name, complete student address, units completed, and units in progress. The GPA should be calculated by dividing the total grade points by the units completed.
- At the end of the report, print the total number of students and the grade point average for all of the students as shown in the print chart. This should be on the same page as the totals for the senior class with two lines skipped before the total lines.

Print chart



Project 4-3 Prepare a student roster summary

This program reads the student master file and prepares a student roster summary with one summary line for each class.

Files used by the program

File	Description	Mode
STUMAST	Student master file	Input
SRSRPT	Student roster summary	Printer output

Specifications

- The student records are in sequence by class standing.
- For each class, this program should print one line that summarizes the number of students, units completed, units in progress, and class GPA.
- At the end of the report, print grand totals for number of students, units completed, and units in progress. Also print the GPA for all the students.

Print chart

```
DATE:     99/99/9999    STUDENT ROSTER SUMMARY          PAGE:  ZZZ9
TIME:     99:99                                         SP04-3R

                NUMBER OF        UNITS       UNITS IN
CLASS           STUDENTS         COMPLETED   PROGRESS    GPA
----------      ---------        ---------   ---------   ----

FRESHMAN          ZZ,ZZ9          ZZZ,ZZ9     ZZZ,ZZ9     9.99
SOPHOMORE         ZZ,ZZ9          ZZZ,ZZ9     ZZZ,ZZ9     9.99
JUNIOR            ZZ,ZZ9          ZZZ,ZZ9     ZZZ,ZZ9     9.99
SENIOR            ZZ,ZZ9          ZZZ,ZZ9     ZZZ,ZZ9     9.99
                  =======         =======     =======     ====

TOTALS:           ZZZ,ZZ9         ZZZ,ZZ9     ZZZ,ZZ9     9.99
```

Project 5-1 Prepare a two-level student registration report

This program reads the course registration file and prepares a student registration report. This two-level report includes unit totals for each student and for each class.

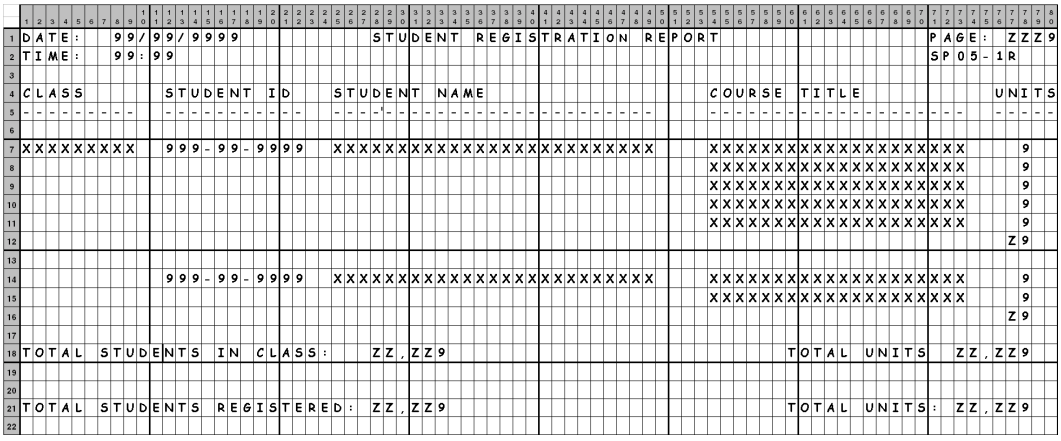
Files used by the program

File	Description	Mode
CRSEREG	Course registration file	Input
REG1RPT	Student registration report	Printer output

Specifications

- The course registration file is in sequence by student ID within class standing.
- When the student ID changes, this program should print the total units for the student with one line skipped after this total.
- When the class standing number changes, the program should print the total number of students in the class and the total units for all these students with one line skipped before this total and two lines after.
- The data in the class column should be derived from the number that's stored in the class standing field in the course registration record where 1 = FRESHMAN, 2 = SOPHOMORE, 3 = JUNIOR, and 4 = SENIOR.
- At the end of the report, print the total number of students in the report and the total number of units, with two lines skipped before the totals.

Print chart



Project 5-2 Prepare a three-level student registration summary

This program reads the course registration file and prepares a student registration summary. This three-level report includes summary lines that show the class counts and unit totals for each student, for each major, and for each class.

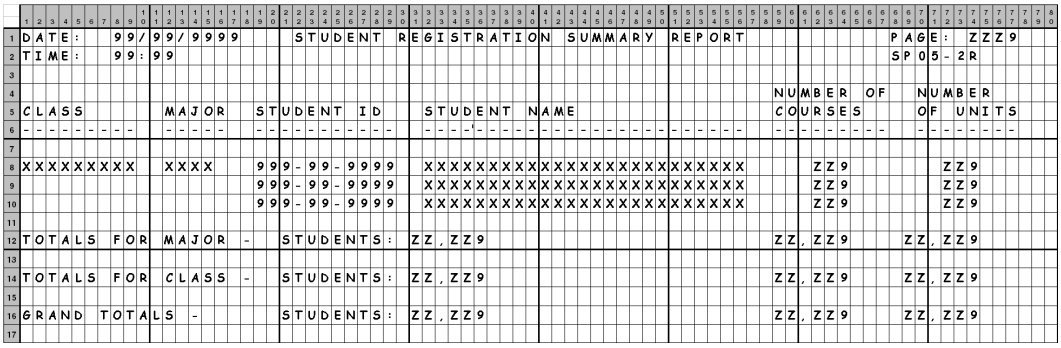
Files used by the program

File	Description	Mode
CRSEREG	Course registration file	Input
REG2RPT	Student registration summary	Printer output

Specifications

- The course registration file is in sequence by student ID within major within class standing.
- When the student ID changes, this program should print one summary line for the student that includes the total units for the student.
- When the major changes, the program should print a summary line for that major.
- When the class standing number changes, the program should print a summary line for that class.
- At the end of the report, print the total number of students in the report and the total number of units.

Print chart



Project 6-1 Apply packed-decimal or binary usage to an old program

Modify any one of the programs you've already developed so the proper usage is applied to all fields in working storage that are used in arithmetic operations or numeric comparisons. If needed, adjust the pictures, too, so they're appropriate for the new usage. From this point on, you should use the proper usages in all of your new programs.

About the projects for section 2

The chapters in section 2 present skills that can be applied to all types of programs, not specific types of programs. As a result, it's hard to design realistic projects for each of the chapters in this section. Instead, the projects for these chapters just force you to use the skills presented in the chapters…sometimes with a spirit of fun.

Project 7-1 Develop a number guessing game

This is a simple interactive program that uses the Random function to generate a number from 1 through 100. The user is then given seven tries to guess what that number is. The interactive session should be something like the one that follows, but you can enhance it to make it more fun.

The interactive session

```
Guess what number I'm thinking of between 1 and 100.
10
Too low!
50
Too high!
25
Too high!
17
Ding! Ding! Ding! You guessed it in 4 tries!
Want to play again? Yes or No?
Yes
```

Specifications

- To generate a random number between 1 and 100, multiply the Random function by 100 because this function returns a decimal fraction between 0 and 1. To make this number different every time the user plays the game, you should code the function with one argument. That argument should be the hours and minutes that you get when you use the Current-Date function.
- Allow only seven tries. If the user doesn't guess the number by then, display the number along with an appropriate message.

Project 8-1 Prepare a student age listing

This program reads the student master file and prepares a listing of the senior students that includes their current ages as well as their projected ages at graduation. This forces you to work with dates. If you prefer to do this on an interactive basis instead of preparing another report, you can do project 8-2 instead.

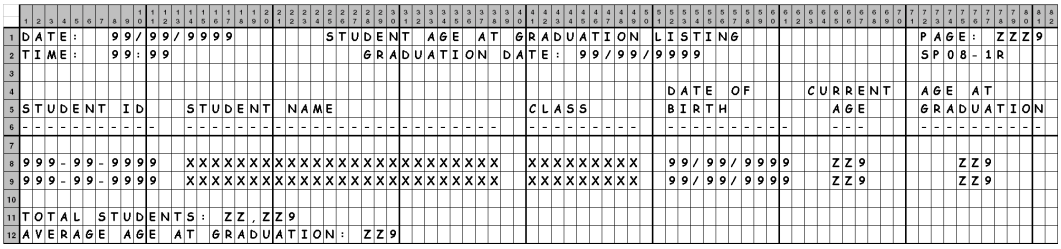
Files used by the program

File	Description	Mode
STUMAST	Student master file	Input
AGELIST	Student age listing	Printer output

Specifications

- The second heading line of the report should include a graduation date like 06/21/2005. Then, the report should list the projected age of each senior at this graduation date.
- At the end of the report, the report should include the number of graduating seniors as well as the average graduation age.

Print chart



Project 8-2 Project graduation age and days until graduation

This interactive program should accept two entries from the user: (1) date of birth and (2) projected graduation date. The program should then display (1) the projected age of the user upon graduation and (2) the number of days until graduation. You design the interactive session.

Project 9-1 Translate English to Pig Latin

This interactive program translates any word the user enters into Pig Latin.

The interactive session

```
-------------------------------------------
Enter any word to see what it looks like in
Pig Latin.
To quit, type Uitqay.
-------------------------------------------
string
The Pig Latin equivalent is: Ingstray
-------------------------------------------
unstring
The Pig Latin equivalent is: Unstringlay
-------------------------------------------
Uitqay
Oodbyegay!
```

Specifications

- If the word starts with a consonant, move the consonants before the first vowel to the end of the word and add *ay*. If the word starts with a vowel, just add *lay* to the end of the word.
- The word entered can be up to 15 characters long, and the Pig Latin equivalent can be up to 18 characters long. The Pig Latin equivalent should start with an uppercase letter and be followed by all lowercase letters. If a user enters more than one word, translate just the first word.
- To end the program, the user must type the word Uitqay, which is Pig Latin for Quit.

Project 9-2 Unstring name, number, and password

This interactive program should accept one line of user entries that contains last name, account number, and password in this format:

```
Menendez,123-45,jtb
```

The program should then unstring the entries, store them properly in their own fields, and display them. To make this user friendly, the program should unstring the three items whether they're separated by commas or by commas and spaces. You design the interactive session.

- The last name must be one or two words followed by a comma. The program should convert this name to all capital letters and store it in an alphanumeric field.
- The account number must be five digits with one hyphen somewhere between the first and last digit. The program should extract the hyphen and store the five digits in a numeric field.
- The password can be up to eight characters. The program should convert any letters in this field to capitals.

Project 10-1 Use a state table to validate states and zip codes

This interactive program accepts a two-character state code and a five-digit zip code from the user. It then checks the validity of these fields and displays a message that says whether or not they're valid. This type of editing routine should be used in all programs that validate input data.

Specifications

- A copy member for a state table is available in a file named stattbl.cpy. If you haven't read chapter 11 yet, you can use the techniques that are appropriate on your system to copy the code for this table into your program. Otherwise, you can use the Copy statement to copy it.
- If you study the code for this table, you'll see that it consists of 50 occurrences of state code, the lowest valid zip code for that state, and the highest valid zip code for that state. As a result, you can use it to validate the state and zip code entries.
- If a state code is in the table, it's valid. Then, if the zip code is in the range of the lowest and highest zip code for that state, it is valid.
- Develop a simple interactive session that tests your validation routine.

Project 10-2 Prepare GPA range summary

This program prepares a summary of the number of students in each GPA range. To prepare this summary, you need to read all of the records in the student master file and use that data to create a table in working storage. After all the records have been read, the program prints the data in the table as indicated by the print chart.

Print chart

```
1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678
DATE:      99/99/9999    GPA RANGE SUMMARY    SP10-2R
TIME:      99:99
                                 NUMBER OF
                GPA RANGES       STUDENTS
                ----------       ---------
                3.5 - 4.0        ZZ9
                3.0 - 3.4        ZZ9
                2.5 - 2.9        ZZ9
                2.0 - 2.4        ZZ9
                1.5 - 1.9        ZZ9
                1.0 - 1.4        ZZ9
                BELOW 1.0        ZZ9

                TOTAL STUDENTS:  Z,ZZ9
```

Project 11-1 Use copy members

Go back to any one of the programs you developed that uses the student master file. Then, delete the code for its record description, and replace that code with a Copy statement that copies the description into your program. From this point on, you should use Copy statements in any program for which there are copy members.

Project 11-2 Create a subprogram that converts numbers to words

This subprogram should receive a four-digit number and return a word description of the value. If, for example, the subprogram receives the number 1423, it should return these words:

```
One-thousand four-hundred twenty-three
```

This type of subprogram is sometimes used in check printing programs. To test this subprogram, you need to write an interactive program like the one that follows.

```
-----------------------------------------------------
Enter a 1 to 4-digit number.
1423
The number is: One-thousand four-hundred twenty-three
-----------------------------------------------------
Convert another number?
N
End of session.
```

Project 11-3 Create a subprogram that unstrings name, number, and password

If you've read chapter 9, you're ready to create a subprogram that unstrings three fields that are separated by commas. This subprogram should receive one alphanumeric field that contains the unstrung fields and return the three fields. The details for this routine are given in project 9-2. If you've done that project, you can convert its code to the subprogram, and modify the interactive session so it tests the subprogram. Otherwise, you need to create your own test program.

Project 11-4 Create a subprogram for state and zip code validation

If you've read chapter 10, you're ready to create a subprogram that receives a state code and a zip code and returns the values in two switches. The first switch indicates whether the state code was valid. The second switch indicates whether the zip code was valid.

To do this validation, you can use the copy member for the state table that's described in project 10-1. If you've done that project, you can convert its code to the subprogram, and you can use the interactive session to test your subprogram. Otherwise, you need to create your own test program.

About the projects for section 3

The goal of this section is to teach you how to develop programs that update and maintain master files as well as prepare reports that require input data from two or more files. The projects that follow give you a chance to test those skills.

Some of the projects for this section use an indexed version of the student master file. The key for this file is the student ID. Project 14-1 asks you to write a program that creates this file. If you're not assigned to do that project, though, your instructor will provide you with an indexed file named stumasti.

Project 13-1 Prepare a student registration report

This program reads the course registration file and the student master file to create a student registration report.

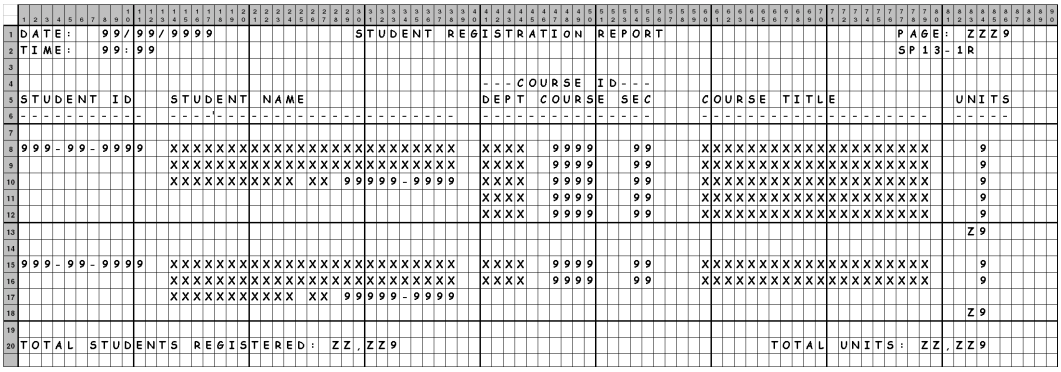
Files used by the program

File	Description	Mode
CRSEREG	Course registration file	Input
STUMAST	Student master file	Input
REG3RPT	Student registration report	Printer output

Specifications

- The student address, city, state, and zip code come from the student master file. The other fields come from the course registration file. Both files are in sequence by student ID, and you need to use matching record logic to prepare this report.
- If you want to make this program more difficult, you can develop a two-level report with summary totals by class standing as well as by student. For each class, you can skip to the top of the next page and print the class name in the heading of the report as shown in the print chart for project 4-2.

Print chart



Project 13-2 Update the student master file

This program updates the student master file with grade point information that's supplied by the course registration file. The result is a new student master file that contains the updated information.

Files used by the program

File	Description	Mode
CRSEREG	Course registration file	Input
STUMAST	Old student master file	Input
NEWMAST	New student master file	Output
ERRRPT	Missing students report	Printer output

Specifications

- The course registration file can have more than one record for each student in the master file.
- For each student record that's matched by a course registration record, add the number of course units to the units completed field, and add the calculated course grade points to the total grade points field.
- To calculate the course grade points, multiply the numeric grade by the course units. To get the numeric grade, convert the letter to a number using a four-point system: A=4, B=3, C=2, D=1 and F=0.
- If a course registration record does not have a matching student ID in the student master file, print the error transaction as a line in the ERRRPT report. You decide the print layout.

Project 13-3 Maintain the student master file

Write a program that maintains just the name and address fields in the student master file. For this program, you design the layout of the transaction file, and you create the test data. Note, however, that your program doesn't have to provide for deleting or adding records to the file.

Project 14-1 Create an indexed student master file

Write a program that converts the sequential student master file to an indexed file named stumasti. The key for the file is the student ID.

Files used by the program

File	Description	Mode
STUMAST	Sequential student master file	Input
STUMASTI	Indexed student master file	Output

Project 14-2 Prepare student registration report

Prepare the report that's shown for project 13-1. This time, though, use the indexed student master file instead of the sequential student master file. Then, read the course registration file sequentially as you prepare the report, but read the student master file randomly whenever you need to get the address data for a student.

Files used by the program

File	Description	Mode
CRSEREG	Course registration file	Input
STUMASTI	Indexed student master file	Input
REG4RPT	Student registration report	Printer output

Project 14-3 Update the student master file

This program updates the indexed student master file with grade information from the sequential course registration file.

Files used by the program

File	Description	Mode
CRSEREG	Course registration file	Input
STUMASTI	Indexed student master file	Input-Output
ERRRPT	Missing students report	Printer output

Specifications

- The course registration file will be read sequentially and the indexed student master file will be read and updated randomly.
- The other specifications for this program are the same as those for project 13-2.
- When you want to return the student master file to its original data, you can rerun the program that you developed for project 14-1.

Project 16-1 Prepare a student listing in descending GPA sequence

This program prepares a student listing after the student master file has been sorted into descending sequence by GPA. As a result, the student listing is in that sequence.

Print chart