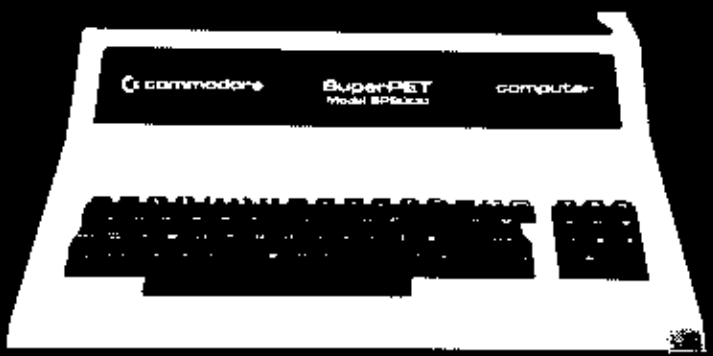
commodore **SuperPET** computer

Waterloo microCOBOL©



Waterloo microCOBOL

Tutorial

and

Reference Manual

P.H.Dirksen

J.W.Welch

Copyright 1982, by the authors.

All rights reserved. No part of this publication may be reproduced or used in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping or information storage and retrieval systems - without written permission of the authors.

Disclaimer

Waterloo Computing Systems Limited makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Waterloo Computing Systems Limited, its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claim for lost profits, fees or expenses of any nature or kind.

Preface

Waterloo microCOBOL is intended to be a substantial implementation of the standard COBOL language. The language supported is suitable for both teaching purposes and for programming many business problems.

It is intended to make available a number of different microCOBOL processors. At the time of writing, interpreters are available for the Commodore SuperPET and for the IBM VM/CMS operating system. Interpreters are being tested for the IBM Personal Computer and for DEC VAX VMS systems. As well, compilers are being developed for the systems mentioned.

This manual is presented in two parts. The first part is a collection of annotated examples intended to introduce the reader to many of the features of microCOBOL. In this way, a novice is presented with a staged introduction to the language. An experienced programmer could use the examples to compare microCOBOL to other COBOL implementations or to other languages.

The second part is a comprehensive language reference manual for Waterloo microCOBOL. Essentially, the language supported includes level one of the **NUCLEUS, SEQUENTIAL I-O, RELATIVE I-O** and **TABLE HANDLING** modules described in COBOL Standards (ANSI X3.23-1974 or ISO 1989-1978). Parts of level two in these modules have also been implemented, including full support for the **PERFORM, STRING** and **UNSTRING** verbs. A few items have been omitted from level one:

(1) The **I-O-CONTROL** paragraph in the **ENVIRONMENT DIVISION** is not supported.

(2) The **DELETE** statement in **RELATIVE I-O** is not supported.

(3) Paragraph and section names must contain at least one alphabetic character.

(4) Continuation of a line is not supported. Syntactic units, such as data names or literals cannot be split across lines.

No support is provided for tape hardware.

P. H. Dirksen
J. W. Welch

April 1982

Acknowledgement

The design and implementation of the Waterloo microCOBOL processors is based upon ideas evolved over the past decade in a number of organizations. All members of the Computer Systems Group (University of Waterloo), the Waterloo Foundation for the Advancement of Computing, and Waterloo Computing Systems, Ltd. have made a substantial contribution to its development. The actual design and programming of the system directly involved the following people: James Bruyn, Keith Campbell, Martin Leistner, Lyle Resnick, Liz Ruest, Jack Schueler, David Till and Jim Welch. Sharon Haydamak was responsible for the production of this manuscript.

Acknowledgement: American National Standards Institute

Portions of this manual have been reproduced from "American National Standard Programming Language COBOL" (X3.23-1974). We are indebted to the unnamed authors for this excellent technical document. The following paragraphs provide acknowledgement, as requested in the standard.

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

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

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-1760, copyrighted 1960 by Minneapolis-Honeywell

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

Table of Contents

Waterloo microCOBOL

Tutorial Examples

Much of the material in the tutorial portion of this text is the result of experience accumulated over many years of presenting courses on the subject of file processing at University of Waterloo. In particular, the authors wish to acknowledge the work done in the following text.

An Introduction to COBOL with WATBOL,
A Structured Programming Approach
D.D.Cowan, P.H.Dirksen, and J.W.Graham,
WATFAC Publications,
Box 803,
Waterloo, Ontario, Canada.

Waterloo Computing Systems Newsletter

The software described in this manual was implemented by Waterloo Computing Systems Limited. From time-to-time enhancements to this system or completely new systems will become available.

A newsletter is published periodically to inform users of recent developments in Waterloo software. This publication is the most direct means of communicating up-to-date information to the various users. Details regarding subscriptions to this newsletter may be obtained by writing:

Waterloo Computing Systems Newsletter
Box 943,
Waterloo, Ontario, Canada
N2J 4C3

Chapter 1

Tutorial Examples

1.1 Introduction

The following tutorial is a sequence of examples meant to introduce the reader to the "flavour" of Waterloo microCOBOL. They do not present a complete or rigorous treatment of any topic, as this detailed information is available in the reference manual in the latter part of this document. This tutorial could be useful in the following situations:

(1) Someone already familiar with COBOL can determine some of the major differences between Waterloo microCOBOL and the dialect already known.

(2) Teachers may find the examples useful as a progressive introduction of the material to their students.

(3) People who already know some other language can get an appreciation for Waterloo microCOBOL before reading the reference manual.

(4) Complete novices could run the various programs, and possibly learn some of the material by exploring the various language features in conjunction with the reference material.

In order that the examples be fully appreciated, it is important that they be entered into the computer and executed.

1.2 Introductory Examples.

1.2.1 A Minimum Program.

Every COBOL program requires a certain number of basic statements. These are presented in this example.

```
identification division.
program-id. EXAMPLE-1.
environment division.
configuration section.
source-computer. CBM-SuperPET.
object-computer. CBM-SuperPET.
data division.
procedure division.
    stop run.
```

Notes

(1) COBOL is a programming language which was first developed in the early 1960's to be used to solve business data processing problems. COBOL in fact stands for COmmon Business Oriented Language.

(2) In order to run any COBOL program a number of statements are required. These statements must be present for the program to work properly.

(3) Example 1 contains these basic necessary statements. All future examples will also contain these statements with some possible slight modifications and additions.

Hint The reader should enter these lines exactly as they appear. This set of statements can then be saved in a file. Thus they need not be entered for each program but instead can be retrieved using the **get** command. In future examples, this file will be referred to as "texts". The following notes should be read before one enters the above lines.

(4) COBOL statements are usually entered beginning in either column 2 or column 6.

(5) The first seven statements in this example are entered in column 2 while the last statement is entered in column 6.

(6) Column 2 is called *margin* A and to column 6 is called *margin* B.

(7) Each line ends with a period.

(8) Users who have used COBOL before will notice that the programs are entered in lower case letters. Waterloo microCOBOL permits the use of both upper and lower case letters; the rules when upper and lower case letters are both used are described in the reference manual.

(9) If the reader wishes to know more about these statements, he should refer to the reference manual portion of this text. As more examples are presented, these statements will be described in more detail. However, the following notes about these statements are appropriate.

Notes

(1) A COBOL program consists of four divisions. These are the identification division, the environment division, the data division and the procedure division. Note that each of these appears once in this example.

(2) The use of 'EXAMPLE-1' as a **program-id** is for documentation purposes only.

(3) Users will recognize 'CBM-SuperPET' as the name of a computer. If programs are run on a different system, it is not necessary to change this name as it is only used for documentation purposes.

1.2.2 Display a name.

One of the first things we want to do in a program is to display information. This program demonstrates one simple way to accomplish this.

```
*
* Display a Name on the Terminal.
*
 identification division.
 program-id. EXAMPLE-2.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.
 procedure division.
      display 'James'.
      stop run.
```

Sample Program Execution

```
run
Execution begins...
James
...Execution ends.
```

Notes

(1) In this example and all future examples the output produced by the program is displayed following the program. A line appearing in *italics* represents a line entered by the user; non-italicized lines have been displayed by the COBOL processor. When a program is *run*, at least two lines are displayed, namely that the program has started and that the program has stopped. This is indicated by the lines

```
Execution begins...
```

and

```
...Execution ends.
```

(2) The line containing the name

```
James
```

is displayed between the above two lines. It is the output produced by the program. Any output will always appear between these two lines.

(3) Three lines have been inserted at the beginning of the program. These lines, containing an * (asterisk) in column 1, are called *comments*. Comments have no effect on the program; they are used for documentation purposes only. Comment lines may be entered anywhere in the program.

(4) The line

```
display 'James'.
```

has been inserted in the procedure division portion of the program. When the program is run, this causes the line containing the name James to be displayed.

(5) The characters to be displayed are enclosed by quotation marks.

(6) The procedure division now contains two statements which are more commonly referred to as *sentences*. Each sentence begins with a *verb* which indicates the desired action to be performed. Each sentence ends with a period.

(7) The **display** verb causes the string of characters to be displayed on the screen.

(8) The **stop** verb indicates that no more actions are required.

(9) The **program-id** line has been used to give a different name to this program. It is used for documentation purposes only.

(10) It is important to note that every program processed goes through two distinct phases, one following the other in time. First the program is read by the system to determine certain types of errors, in particular syntax or grammar errors are detected. Then the system actually begins executing the statements.

(11) This program could be easily entered by using the following steps:

i) Use the **get** command to load a copy of the previous program.

ii) Add the three **comment** lines at the top of the program.

iii) Add the **display** statement.

Use of this technique results in less time to enter the program and also reduces "entry" errors. As we will see in future examples, it is often easier to modify an existing program than to enter completely a new program.

1.2.3 Accept Data from the Terminal.

Another common requirement is to input some information into a program. This program accepts some data and then displays it on the screen.

```
*
* Accept Data from the Terminal.
*
 identification division.
 program-id. EXAMPLE-3.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 procedure division.
      display 'Enter a name of 5 characters'.
      accept name.
      display name.
      stop run.
```

Sample Program Execution

```
run
Execution begins...
Enter a name of 5 characters
James
James
...Execution ends.
```

Notes

(1) When this program is run, a line is displayed asking that a name containing exactly 5 characters be entered. The next line, which appears in italics, is the response entered by the user, namely the name James. The following line is the output produced by the program. In this and all future examples, all lines entered by the user are displayed in italics.

(2) The data read from the terminal is placed in **working-storage section** of the **data division.** Working storage can be thought of a large piece of paper within the computer. Information is "written" or placed into working storage.

(3) For this particular example, working storage consists of an area large enough to contain a 5-character string which will be called "name". It is defined as follows:

```
working-storage section.
01  name picture xxxxx.
```

The 01 is a *level number* and is entered in margin A. "Name" is the name of the area and is entered in margin B. The **picture** clause defines the characteristics of the area. In this case, an area capable of holding 5 consecutive characters is defined by using five x's.

(4) Level numbers will be explained more fully in future examples.

(5) "Name" is referred to as a *data-name* and is chosen by the programmer. The rules for choosing such names are described in a later note.

(6) The **accept** verb is used to input the desired string of characters, specifying it should be placed in working-storage in the area called "name".

(7) The **accept** verb takes the characters that are entered and places them in the area called "name". If more than 5 characters are entered the left-most 5 characters are placed in "name". If less than 5 characters are entered, the characters are placed left-justified in "name". The remaining characters are left unchanged. Thus the user should enter blanks or spaces if the name contains less than 5 characters. The next example shows another way of accepting variable length names.

(8) Blank lines are inserted before and after the working-storage section to make the program easier to read.

(9) A data-name consists of not more than thirty characters chosen from the letters, the digits, and the hyphen; it must contain at least one letter. The hyphen may be used anywhere except at the beginning or end.

(10) COBOL reserves a number of words for its own use. These are called *reserved words*. For example, all COBOL verbs are reserved words as are most of the words in "texts". COBOL's reserved words are listed in the reference section of this text (see RESERVED WORDS).

(11) A data-name *cannot* be a COBOL reserved word. For example, it is not possible to use the data-name "input" instead of "name" since "input" is a reserved word. However, it is possible to use "input-place".

1.2.4 The Perform Verb.

Programs can be written in such a way that they are easier to read and understand. This clarity is achieved by organizing the program into modules or parts.

```
*
* Introduce Perform Verb.
*
 identification division.
 program-id. EXAMPLE-4.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 procedure division.
      perform get-name.
      perform display-name.
      stop run.

 get-name.
      display 'Enter a name up to 5 characters'.
      move spaces to name.
      accept name.

 display-name.
      display name.
```

Sample Program Execution

```
run
Execution begins...
Enter a name up to 5 characters
Jim
Jim
...Execution ends.
```

Notes

(1) This program is a slight modification of the previous example. The procedure division is organized differently.

(2) The two actions of accepting the name to be read and displaying the name have been separated into two distinct parts or modules.

(3) In COBOL, these parts are called *paragraphs;* each paragraph is identified by a *paragraph-name*. The paragraph-name is entered in margin A and the paragraphs are placed following the **stop run.**

(4) Paragraph-names are formed in the same way as data-names. When a paragraph-name is used to signify the beginning of a paragraph, it must be followed by a period.

(5) Blank lines have been inserted to make it easier to identify the paragraphs.

(6) The **perform** verb in the sentence

```
perform get-name
```

acts exactly as one would expect, namely it causes the sentences in the paragraph named "get-name" to be executed.

(7) When the "get-name" paragraph is completed, control passes to the sentence following the

```
perform get-name
```

namely the

```
perform display-name.
```

(8) The two **performs** cause the two paragraphs to be executed in the appropriate sequence.

(9) The two paragraphs can be placed in any order following the **stop run.** Of course, the two **performs** must be placed in the correct sequence in order for the program to function properly.

(10) A new sentence

```
move spaces to name
```

has been inserted before the **accept** sentence. This will cause five spaces or blank characters to be placed in "name". We can now enter a name of any length up to 5 characters. The **move** verb will be described in more detail in a future example.

1.2.5 The Until Clause.

One of the most important features of computers is to perform certain tasks a number of times. This program introduces one method of doing such tasks repetitively until a signal is encountered to stop the process.

```
*
* Perform Verb with Until Clause.
*
 identification division.
 program-id. EXAMPLE-5.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 procedure division.
      perform get-name.
      perform process-name
           until name = 'stop '.
      stop run.

 get-name.
      display 'Enter a name up to 5 characters'.
      move spaces to name.
      accept name.

 process-name.
      display name.
      perform get-name.
```

Sample Program Execution

```
run
Execution begins...
Enter a name up to 5 characters
James
James
Enter a name up to 5 characters
Jim
Jim
Enter a name up to 5 characters
Mary
Mary
Enter a name up to 5 characters
stop
...Execution ends.
```

(1) This program asks the user to enter a name. After the name is displayed, the program asks that another name be entered. This process continues until the characters 'stop ' are entered at which time the program terminates.

(2) The program is written to accept and then display an unknown number of names. The two actions of displaying and reading the name, are placed in a paragraph called "process-name".

(3) The "process-name" paragraph makes use of the previously written paragraph "get-name", which displays the prompt message and then reads a name. Paragraphs can contain **performs** of other paragraphs.

(4) The program now works as follows:

i) An initial name is read using the "get-name" paragraph.

ii) The "process-name" paragraph is then performed. This causes the name to be displayed and another name to be read.

iii) Control returns to the **perform-until** sentence which determines if the newly entered string is 'stop '. If not, the "process-name" paragraph is executed again. If the name is 'stop ', control passes to the sentence following the **perform-until.**

(5) We refer to

```
name = 'stop '
```

as a *condition*, in this case the *equals* condition. The condition compares the value of "name" with the characters 'stop '. If they are equal the value of the condition is true; otherwise it is false. A more general form of condition is discussed in a later section.

(6) While the clause

```
until name = 'stop '
```

could have been entered on the same line as the **perform**, it has been entered as a separate line and indented to make the program more readable.

(7) A COBOL sentence can be written on more than one line. Sometimes this occurs because a line is too long but more often it is done to improve readability. The continued line is usually indented in order that the continuation can be clearly seen.

(8) The condition could have been written as

```
name = 'stop'
```

In this case, the blank character has been omitted from the end of the string. Before comparing two fields COBOL checks that the two strings have the same length. If one string is shorter, it is padded on the right with blank characters to make it the same length as the longer string. Thus 'stop' would be set to 'stop ' before the comparison is done.

1.2.6 Read a Number of Fields from the Terminal.

On many occasions we wish to enter a number of items of information about a particular person or thing. For example, we might wish to also enter such items as sex, age, etc.

```
*
* Read a Number of Fields from the Terminal.
*
 identification division.
 program-id. EXAMPLE-6.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 data division.

 working-storage section.

 01   student-no          pic xxxx.
 01   name                pic xxxxx.
 01   age                 pic xx.
 01   sex                 pic x.

 procedure division.
      perform get-id.
      perform process-student-data
           until student-no = '9999'.
      stop run.

 process-student-data.
      perform get-name.
      perform get-age.
      perform get-sex.
      display student-no name age sex.
      perform get-id.

 get-id.
     display 'Enter student number (9999 to stop)'.
      move spaces to student-no.
      accept student-no.
```

```
get-name.
    display 'Enter name'.
    move spaces to name.
    accept name.

get-age.
    display 'Enter age'.
    move spaces to age.
    accept age.

get-sex.
    display 'Enter sex (M or F)'.
    accept sex.
```

Sample Program Execution

```
run
Execution begins...
Enter student number (9999 to stop)
1234
Enter name
James
Enter age
15
Enter sex (M or F)
M
1234James15M
Enter student number (9999 to stop)
2345
Enter name
Marie
Enter age
15
Enter sex (M or F)
F
2345Marie15F
Enter student number (9999 to stop)
9999
...Execution ends.
```

Notes

(1) This program prompts the user to enter 4 quantities, namely a student number, name, age, and sex and then displays this data on the terminal. This sequence is repeated until the student number '9999' is entered.

(2) Three lines have been added to the working-storage section to define the areas for the three new items of data. The new data-names are "student-number", "age" and "sex" and they have a size of 4, 2 and 1 characters respectively.

(3) The reserved word **picture** is used quite frequently in COBOL programs. To save time and space, a short form, **pic** can be used.

(4) This program uses the student number '9999' to terminate processing instead of the name 'stop' as in the previous example. Thus

```
perform get-id
```

is used at the start of the procedure division instead of

```
perform get-name.
```

(5) The "process-name" paragraph has been replaced by the paragraph called "process-student-data". It causes the number, age, and sex to be read, displays the appropriate line and then reads the next student number. The sentence

```
perform process-student-data
    until student-number = '9999'
```

controls the reading and displaying of the student data.

(6) The sentence

```
display student-no name age sex.
```

displays the line on the terminal. The data-names in this sentence are separated by a blank.

(7) For each record read, a line is displayed. It is somewhat disturbing that there are no spaces between the four items of output. This can be remedied by using

```
display student-no ' ' name ' ' age ' ' sex.
```

which inserts the blank character between each item.

(8) The sentence which displays the student data could be replaced by the two sentences

```
display student-no.
display name ' ' age ' ' sex.
```

which would cause two lines to be displayed for each student.

(9) More blanks could be placed between any of the items on the output line by increasing the number of spaces between the quotes.

```
display student-no '   ' name ...
```

would place 4 spaces between the number and the name.

(10) The four data items are quite often referred to as *fields*.

1.2.7 Define a Simple Data Structure.

In many cases it is more convenient to represent and deal with a number of data items or fields as a single entity. This collection of information about a particular person or thing is called a *record*.

```
*
* Read a Number of Fields from the Terminal.
*
 identification division.
 program-id. EXAMPLE-7.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 data division.

 working-storage section.

 01   student-data.
      02 student-no          pic xxxx.
      02 name                pic xxxxx.
      02 age                 pic xx.
      02 sex                 pic x.

 procedure division.
      perform get-id.
      perform process-student-data
           until student-no = '9999'.
      stop run.

 process-student-data.
      perform get-name.
      perform get-age.
      perform get-sex.
      display student-data.
      perform get-id.

 get-id.
      display 'Enter student number (9999 to stop)'.
      move spaces to student-no.
      accept student-no.
```

```
get-name.
    display 'Enter name'.
    move spaces to name.
    accept name.

get-age.
    display 'Enter age'.
    move spaces to age.
    accept age.

get-sex.
    display 'Enter sex (M or F)'.
    accept sex.
```

Sample Program Execution

```
run
Execution begins...
Enter student number (9999 to stop)
4321
Enter name
Fred
Enter age
16
Enter sex (M or F)
M
4321Fred 16M
Enter student number (9999 to stop)
6543
Enter name
Bev
Enter age
14
Enter sex (M or F)
F
6543Bev  14F
Enter student number (9999 to stop)
9999
...Execution ends.
```

Notes

(1) This program is another version of the previous example which creates a *data-structure* containing the four fields.

(2) The data structure is defined as follows:

```
01  student-data.
    02 student-no     pic xxxx.
    02 name           pic xxxxx.
    02 age            pic xx.
    02 sex            pic x.
```

A new 01-level data-name is introduced, namely "student-data". The four 01-level items from the previous example have been changed to 02-level items and have been entered in margin B.

(3) The data structure can be described as follows:

i) The four data-items can be considered as a collection of information about a particular student. This collection is called a *record* and the 01-level data-name permits the program to refer to the entire record.

ii) The original four data-items have the same data-names and the same sizes as before but they now have been defined with 02-level numbers. They can be used in the same way as they were used in previous examples.

(4) The 01-level item ends in a period.

(5) The 01-level item does not have a **picture** clause if it is subdivided into 02-level items.

(6) The 01-level item in this example is considered to have 12 characters i.e. the sum of the sizes of the fields at the 02-level.

(7) The user should again note that some or all of the output fields are not separated by blanks. A future example will remedy this situation.

1.2.8 Create an Output Data Structure.

Often when displaying a number of fields, we want to setup or format the line with appropriate spacing and then to refer to the line as an output record. An output data structure is defined and used in this example to allow more flexibility on output.

```
*
* Create an Output Data Structure.
*
 identification division.
 program-id. EXAMPLE-8.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 data division.

 working-storage section.

 01   student-data.
      02 student-no          pic xxxx.
      02 name                pic xxxxx.
      02 age                 pic xx.
      02 sex                 pic x.

 01   display-record.
      02 out-student-no      pic xxxx.
      02 filler              pic xxx value is spaces.
      02 out-name            pic xxxxx.
      02 filler              pic xxx value is spaces.
      02 out-age             pic xx.
      02 filler              pic xxx value is spaces.
      02 out-sex             pic x.

 procedure division.
      perform get-id.
      perform process-student-data
           until student-no = '9999'.
      stop run.
```

```
process-student-data.
    perform get-name.
    perform get-age.
    perform get-sex.
    perform edit-and-display-record.
    perform get-id.

get-id.
    display 'Enter student number (9999 to stop)'.
    move spaces to student-no.
    accept student-no.

get-name.
    display 'Enter name'.
    move spaces to name.
    accept name.

get-age.
    display 'Enter age'.
    move spaces to age.
    accept age.

get-sex.
    display 'Enter sex (M or F)'.
    accept sex.

edit-and-display-record.
    move student-no to out-student-no.
    move name to out-name.
    move age to out-age.
    move sex to out-sex.
    display display-record.
```

Sample Program Execution

```
run
Execution begins...
Enter student number (9999 to stop)
5555
Enter name
Eliza
Enter age
15
Enter sex (M or F)
F
5555  Eliza  15  F
Enter student number (9999 to stop)
1111
Enter name
Paul
Enter age
14
Enter sex (M or F)
M
1111  Paul  14  M
Enter student number (9999 to stop)
9999
...Execution ends.
```

Notes

(1) Each student record is read and then displayed with each field separated by at least three spaces.

(2) A new data structure called "display-record" is defined. It contains four fields with the newly defined data-names "out-student-no", "out-name", "out-age", and "out-sex". These will be used to contain the four fields for output.

(3) Each of these fields is separated by a field which is called **filler. Filler** is a COBOL reserved word and is used to "fill" or insert space in a record. The **picture** clause indicates how much space is to be inserted.

(4) The **value is** clause specifies the particular character or characters we wish to insert in the field. In this case, the COBOL reserved word **spaces** indicates that the field is to contain spaces or blanks.

(5) If the **value is** clause is omitted in the **filler**, the field is said to be *undefined*. If such a field were displayed, it would contain one or more question marks (??).

(6) The **move** verb is used to move an item from one place in working-storage to another. Thus

```
move student-no to out-student-no.
```

causes the 4-character number to be moved from "student-data" to "out-student-no" in "display-record".

(7) There is no difficulty with moving data from one field to another if the fields are the same size. However, if the *receiving* field is larger than the sending field, the data is inserted left-justified and an appropriate number of blanks are inserted on the right. The first two lines of the display record could be replaced by

```
02 out-student-no  pic xxxxxxx.
```

and the output would be the same since the four character name would be moved to the left-most four positions and blanks would be inserted in the remaining three positions.

(8) If the receiving field is smaller than the sending field, the data again is inserted left-justified. However, the 'extra' characters on the right are truncated.

1.2.9 Produce a Simple Report.

Definition of **picture** clauses can made simpler especially when 'long' fields are required.

```
*
* Read a Number of Fields from the Terminal
* and Print a Small Report.
*
 identification division.
 program-id. EXAMPLE-9.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 data division.

 working-storage section.

 01  student-data.
     02 student-no          pic xxxx.
     02 name                pic xxxxx.
     02 age                 pic xx.
     02 sex                 pic x.

 01  heading-line.
     02 filler        pic x(12) value is 'Student Data'.

 01  first-line.
     02 filler              pic x(8) value is 'number '.
     02 out-student-no      pic xxxx.

 01  second-line.
     02 filler              pic x(5) value is 'name'.
     02 out-name            pic x(5).

 01  third-line.
     02 filler              pic x(4) value is 'age '.
     02 out-age             pic xx.
     02 filler              pic x(5) value is ' sex '.
     02 out-sex             pic x.
```

```
procedure division.
    perform get-id.
    perform process-student-data
        until student-no = '9999'.
    stop run.

process-student-data.
    perform get-name.
    perform get-age.
    perform get-sex.
    perform edit-and-display-record.
    perform get-id.

get-id.
    display 'Enter student number (9999 to stop)'.
    move spaces to student-no.
    accept student-no.

get-name.
    display 'Enter name'.
    move spaces to name.
    accept name.

get-age.
    display 'Enter age'.
    move spaces to age.
    accept age.

get-sex.
    display 'Enter sex (M or F)'.
    accept sex.

edit-and-display-record.
    move student-no to out-student-no.
    move name to out-name.
    move age to out-age.
    move sex to out-sex.
    display heading-line.
    display first-line.
    display second-line.
    display third-line.
```

Sample Program Execution

```
run
Execution begins...
Enter student number (9999 to stop)
9876
Enter name
Bob
Enter age
17
Enter sex (M or F)
M
Student Data
number  9876
name Bob
age 17 sex M
Enter student number (9999 to stop)
5786
Enter name
John
Enter age
16
Enter sex (M or F)
M
Student Data
number  5786
name John
age 16 sex M
Enter student number (9999 to stop)
9999
...Execution ends.
```

Notes

(1) The example contains most of the material that has been presented in the previous examples. The program prompts for a number, name, age, and sex and displays a small report for each student.

(2) A heading is placed at the beginning of each student report.

(3) The "heading-line" defines a field of 12 characters by using the clause

```
02 filler pic x(12) value is 'Student Data'.
```

which is equivalent to

```
02 filler pic xxxxxxxxxxxx value is 'Student Data'.
```

The former method of defining **pictures** is often more convenient than the latter.

(4) Both methods of defining **pictures** can be used in a particular data-structure definition.

1.3 Reading and Writing Files.

1.3.1 Getting Prepared To Use Files.

In each of the previous examples the user has been required to re-enter the student data - a somewhat tiring and boring situation. It would be preferable if the data could be entered once and then saved away for future use. (We have already done something similar when we saved our programs away for future use.) A collection of records, in this case the student records, is referred to as a *file*. This example shows how a file is defined. The next example uses the file.

```
*
* Define a File to Hold the Student Records.
*
 identification division.
 program-id. EXAMPLE-10.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'students'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 working-storage section.

 01   student-data.
      02 student-no              pic xxxx.
      02 name                    pic xxxxx.
      02 age                     pic xx.
      02 sex                     pic x.
```

```
procedure division.
    stop run.
```

Sample Program Execution

```
run
Execution begins...
...Execution ends.
```

Notes

(1) COBOL requires certain information in order to handle a file. A number of new statements are introduced in this example in order to define a file.

(2) The lines

```
input-output section.
file-control.
    select student-file
        assign to 'students'.
```

are placed in the environment division. They specify the *file-name* used by the program, "student-file", as well as the *system-name* of the file, "students".

(3) The data division has been modified to include the lines:

```
file section.
fd  student-file
    label records are standard.
01  student-record.
    02 pic x(60).
```

This section is used to describe some of the attributes of a particular file. It is also used to set-up an intermediate area into which data will be read or from which data will be written.

(4) The file definition, **fd**, defines the name of the file, namely "student-file". It is also necessary to tell the system how to deal with the *label* of the file. In this example, we indicate that labels will handled in a *standard* fashion. The reference manual expands on the concept of labels.

(5) The lines

```
01 student-record.
   02 pic x(60).
```

define a *record-name* for the file namely, "student-record" and specify that it contains 60 characters. The purpose of this area is to act as an *input-output* area to receive data from from the file or to send data to the file.

(6) When the record is read or written, 60 characters of information will be transmitted.

(7) Record-names and file-names are formed in the same way as data-names. System-names also consist of characters chosen from the letters and digits. The number of characters varies from system to system. Use of "short" system-names is usually safer, especially if one wants to use programs on a variety of systems.

Hint Since the student file will be used in many of the future examples, it is suggested that "texts" be modified to include the input-output and file section statements introduced in this example.

1.3.2 Create a Simple File.

Having defined the file in the previous example, we now accept student data and write it into the newly defined file.

```
*
* Write Student Records into a File.
*
 identification division.
 program-id. EXAMPLE-11.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'students'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 working-storage section.

 01   student-data.
      02  student-no              pic xxxx.
      02 name                    pic xxxxx.
      02 age                     pic xx.
      02 sex                     pic x.
```

```
procedure division.
    open output student-file.
    perform get-id.
    perform process-student-data
        until student-no = '9999'.
    move '9999' to student-no.
    write student-record from student-data.
    close student-file.
    stop run.

process-student-data.
    perform get-name.
    perform get-age.
    perform get-sex.
    write student-record from student-data.
    perform get-id.

get-id.
    display 'Enter student number (9999 to stop)'.
    move spaces to student-no.
    accept student-no.

get-name.
    display 'Enter name'.
    move spaces to name.
    accept name.

get-age.
    display 'Enter age'.
    move spaces to age.
    accept age.

get-sex.
    display 'Enter sex (M or F)'.
    accept sex.
```

Sample Program Execution

```
run
Execution begins...
Enter student number (9999 to stop)
4326
Enter name
Doug
Enter age
14
Enter sex (M or F)
M
Enter student number (9999 to stop)
3758
Enter name
Jane
Enter age
16
Enter sex (M or F)
F
Enter student number (9999 to stop)
6420
Enter name
Pat
Enter age
16
Enter sex (M or F)
F
Enter student number (9999 to stop)
9999
...Execution ends.
```

Notes

(1) The previous example defined the required file. This example will accept data as before and write it into the file.

(2) A file must be *opened* before it can be used. The statement

```
open output student-file
```

in the main paragraph of the procedure division specifies that the file is to be made available for output purposes.

(3) The program again requests the user to enter student number, name, age, and sex.

(4) The **display** verb cannot be used to write the data to the file. The statement

```
write student-record from student-data
```

causes the record to be written to the file. In effect this statementis saying, "Write the record as defined in the file definition and obtain the data from the area called student-data in working-storage." In fact, the data in working storage is moved to the file section and then it is written to the file.

(5) This process continues until the student number 9999 is entered.

(6) Future examples will want to read the file that has been created. In order to do this some means has to included in the file to recognize that we have read the last record i.e. we are at the end of the file. In our previous examples, entering 9999 accomplished this.

(7) A special *end-of-file* or *sentinel* record containing the student number 9999 is written after the last student record is accepted from the terminal.

(8) The statement

```
close student-file
```

releases the file indicating that the program no longer needs the file. It is not valid to specify "output" in a **close** sentence.

(9) The description of **open** and **close** is somewhat vague and omits many details about these two verbs. The description of the complete actions of **open** and **close** are described in the reference manual.

1.3.3 Read and Print a File.

In the previous example we created a student file. It would seem appropriate that we read this file and display the records to assure ourselves that they were written correctly.

```
*
* Read and Print the Student Records.
*
 identification division.
 program-id. EXAMPLE-12.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'students'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 working-storage section.

 01   student-data.
      02 student-no              pic xxxx.
      02 name                    pic xxxxx.
      02 age                     pic xx.
      02 sex                     pic x.
```

```
procedure division.
    open input student-file.
    perform read-student-record.
    perform process-student-data
        until student-no = '9999'.
    close student-file.
    stop run.

process-student-data.
    display student-data.
    perform read-student-record.

read-student-record.
    read student-file into student-data.
```

Sample Program Execution

```
run
Execution begins...
4326Doug    14M
3758Jane    16F
6420Pat     16F
...Execution ends.
```

Notes

(1) This program reads the file created in the previous example and displays each record as it appears in the file.

(2) The statement

```
open input student-file
```

specifies that the "student-file" is to be made available for input.

(3) The **accept** verb cannot be used to read a file. The statement

```
read student-file into student-data
```

causes a record to be read from the file and to be placed in working storage in the area called "student-data". In fact, the data is read into the file section and placed in the area called "student-record". It is then moved to working storage.

(4) Each record read is displayed exactly as it is contained in the file i.e. spaces may not be present between fields of the displayed record.

(5) When the last record containing a student number of 9999 is read, the reading and displaying process terminates.

(6) The **close** sentence releases the file.

(7) The file used in this and the previous example is referred to as a *sequential file*. Writing a sequential file means that each record is written immediately following the previous record. Reading a sequential file means that after a record has been read, the next record is then available to be read.

1.3.4 A Standard Method for Handling End of File.

In the example in which the file was created, we had to go to extra effort to create the sentinel record. Recognition of the end of a file is a common problem in file processing, and it should be no surprise that there is a standard method of dealing with the problem. If this were not the case, each program would require different and special tests to determine when the end of file was reached.

```
*
* At  End Clause.
*
 identification division.
 program-id. EXAMPLE-13.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'students'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 working-storage section.

 01   student-data.
      02 student-no              pic xxxx.
      02 name                    pic xxxxx.
      02 age                     pic xx.
      02 sex                     pic x.
```

```
procedure division.
    open input student-file.
    perform read-student-record.
    perform process-student-data
        until student-no = '9999'.
    close student-file.
    stop run.

process-student-data.
    display student-data.
    perform read-student-record.

read-student-record.
    read student-file into student-data
        at end move '9999' to student-no.
```

Sample Program Execution

run
Execution begins...
4326Doug 14M
3758Jane 16F
6420Pat 16F
...Execution ends.

Notes

(1) As a matter of course, whenever a file is created, a special end of file record is written. When examining a file, this record does not appear to be there as it is never displayed. The end-of-file record is automatically written when the file being created is closed using the **close** verb.

(2) This special record is recognized automatically whenever it is read by the system. This is accomplished by adding an extra clause in the **read** sentence which causes special processing when the end-of-file is read.

(3) When the end-of-file is encountered, no record is transferred to working storage. However, the **at end** clause is executed and in this case the constant 9999 is moved to the "student-no" field. This has same effect as reading the dummy or sentinel record. Recall that the **perform-until** checks this field to determine if there are any more records.

(4) The **at end** clause is executed only when the end-of-file record is read.

1.3.5 At End and High-values.

Using 9999 as the signal for an end-for-file might cause some potential problems. For example, someone might inadvertently assign a student the number 9999.

```
*
* At  End and High-Values.
*
 identification division.
 program-id. EXAMPLE-14.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'students'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 working-storage section.

 01   student-data.
      02 student-no              pic xxxx.
      02 name                    pic xxxxx.
      02 age                     pic xx.
      02 sex                     pic x.
```

```
procedure division.
    open input student-file.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file.
    stop run.

process-student-data.
    display student-data.
    perform read-student-record.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
4326Doug    14M
3758Jane    16F
6420Pat     16F
9999Pat     16F
...Execution ends.
```

Notes

(1) COBOL has a special constant known as **high-values** which can be used instead of 9999. In mathematical terms this quantity can be compared to *infinity*, in that no larger quantity can be assigned to the field.

(2) Both the **perform-until** and the **at end** clauses have been modified to use **high-values.**

(3) When the program is run an extra line is printed, namely the sentinel record with 9999 as the student number. Recall that this record was inserted by the program that created the file. Note also that the name, age, and sex are the same as the previous record. Recall that we only changed the student number before we wrote the sentinel record. This can be remedied by modifying the program that created the file.

(4) COBOL also has a constant called **low-values** which is the smallest possible value.

(5) **High-values** and **low-values** are called *figurative constants*. **Spaces** and **zero** are also figurative constants.

1.3.6 Use the File Provided with the System.

To save time and to provide some consistency, a version of the student file is provided with the system. This section describes the file and displays an unspaced listing of the file.

```
*
* Introduce the Student File.
*
 identification division.
 program-id. EXAMPLE-15.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 working-storage section.

 01   student-data.
      02 student-no              pic xxxx.
      02 name                    pic x(20).
      02 age                     pic xx.
      02 sex                     pic x.
      02 class                   pic x.
      02 school                  pic x.
      02 algebra                 pic xxx.
      02 geometry                pic xxx.
      02 physics                 pic xxx.
      02 chemistry               pic xxx.
      02 english                 pic xxx.
```

```
procedure division.
    open input student-file.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file.
    stop run.

process-student-data.
    display student-data.
    perform read-student-record.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution Begins...
1234Smith                SA  14m13075100075065084
1236Jones                TO  14m220760780550570 78
1238Winterbourne         MS  14m310780880560670 88
1239Harrison             K   14m420220870650870 68
1240Graham               JW  14m210000680750670 87
1242Welch                JW  14m310750750760750 75
1243Dirksen              PH  14m420740850540680 84
1245Cowan                DD  15f 330550660770880 99
1249Sullivan             J   15f 420440550660770 88
1256Kitchen              MP  14m430740491000970 36
1266Taylor               YO  13f 330950830720660 55
1268Allen                TT  13f 210980840730650 59
1270Xerxes               X   13f 130990880770660 55
1272Zimmerman            AB  13f 320950850780610 57
1375Quantas              FL  15m220660660660660 66
1388Beatle               RA  15f 110650620730760 87
1390Cruikshank           TR  15f 330550640770760 85
1393Hopper               BU  15f 230450690370260 35
...Execution ends.
```

Notes

(1) In order to make the file available, the user is requested to run the program called "CBL43". A copy of this program appears as the last example in the tutorial section of the text.

(2) The student file contains 18 records.

(3) Each record in the file is 60 characters long and is composed of the following fields:

Student Number
The student number field is 4 digits in length and contains a 4-digit number.

Name
The name field is 20 characters in length and contains both surname and initials. The surname occupies the first 17 positions and the initials occupy the last 3 positions of this field.

Age
The age field is 2 characters in length and contains numbers in the range 12 to 19.

Sex
The sex field is 1 character in length and contains either an M or an F.

Class
The class field is 1 character in length and contains numbers in the range 1 to 4.

School
The school field is 1 character in length and contains numbers in the range 1 to 3.

Algebra, *Geometry*, *Physics*, *Chemistry*, *and English*
These five fields are all 3 digits in length and contain marks or grades for each of the subjects. Possible grades range from 0 to 100.

Space Reserved for Future Use.
The student record is defined to have 60 characters. The above fields occupy 44 characters of the record; the remaining 16 characters are reserved for future use.

(4) The system-name for the student file is "textfile".

(5) If the user plans to use the student file, it is suggested that the definition for "student-data" be entered as a get file.

1.3.7 Print a Report Using the Student File.

In all the examples presented to this point, we have used the **display** verb for producing output to the screen. It is more traditional in COBOL to define a special file for screen output and to use the **write** verb for displaying records.

```
*
* Print a Report Using the Student File.
*
 identification division.
 program-id. EXAMPLE-16.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).
```

```
working-storage section.

01  student-data.
    02 student-no          pic xxxx.
    02 name.
       03surname           pic x(17).
       03initials          pic xxx.
    02 age                 pic xx.
    02 sex                 pic x.
    02 class               pic x.
    02 school              pic x.
    02 algebra             pic xxx.
    02 geometry            pic xxx.
    02 physics             pic xxx.
    02 chemistry           pic xxx.
    02 english             pic xxx.

01  report-heading.
    02 filler              pic x(20) value is spaces.
    02 filler              pic x(20) value is 'Student Reports'.
    02 filler              pic x(40) value is spaces.

01  first-line.
    02 out-student-no      pic x(4).
    02 filler              pic x(10) value is spaces.
    02 out-initials        pic xxx.
    02 filler              pic x value is spaces.
    02 out-surname         pic x(17).

01  second-line.
    02 filler              pic x(5) value is ' alg'.
    02 out-algebra         pic xxx.
    02 filler              pic x(5) value is ' grnt'.
    02 out-geometry        pic xxx.
    02 filler              pic x(5) value is ' phy'.
    02 out-physics         pic xxx.
    02 filler              pic x(5) value is ' chm'.
    02 out-chemistry       pic xxx.
    02 filler              pic x(5) value is ' eng'.
    02 out-english         pic xxx.

01  blank-line             pic x(80) value is spaces.
```

```
procedure division.
    open input student-file.
    open output screen.
    write display-record from report-heading.
    write display-record from blank-line.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file
          screen.
    stop run.

process-student-data.
    perform display-student-data.
    perform read-student-record.

display-student-data.
    move student-no to out-student-no.
    move initials to out-initials.
    move surname to out-surname.
    write display-record from first-line.
    move algebra to out-algebra.
    move geometry to out-geometry.
    move physics to out-physics.
    move chemistry to out-chemistry.
    move english to out-english.
    write display-record from second-line.
    write display-record from blank-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                    Student Reports

1234        SA  Smith
 alg 075 gmt 100 phy 075 chm 065 eng 084

1236        TO  Jones
 alg 076 gmt 078 phy 055 chm 057 eng 078

1238        MS  Winterbourne
 alg 078 gmt 088 phy 056 chm 067 eng 088

1239        K   Harrison
 alg 022 gmt 087 phy 065 chm 087 eng 068

1240        JW  Graham
 alg 000 gmt 068 phy 075 chm 067 eng 087

1242        JW  Welch
 alg 075 gmt 075 phy 076 chm 075 eng 075

1243        PH  Dirksen
 alg 074 gmt 085 phy 054 chm 068 eng 084

1245        DD  Cowan
 alg 055 gmt 066 phy 077 chm 088 eng 099

1249        J   Sullivan
 alg 044 gmt 055 phy 066 chm 077 eng 088

1256        MP  Kitchen
 alg 074 gmt 049 phy 100 chm 097 eng 036

1266        YO  Taylor
 alg 095 gmt 083 phy 072 chm 066 eng 055

1268        TT  Allen
 alg 098 gmt 084 phy 073 chm 065 eng 059

1270        X   Xerxes
```

```
 alg 099 gmt 088 phy 077 chm 066 eng 055

1272        AB  Zimmerman
 alg 095 gmt 085 phy 078 chm 061 eng 057

1375        FL  Quantas
 alg 066 gmt 066 phy 066 chm 066 eng 066

1388        RA  Beatle
 alg 065 gmt 062 phy 073 chm 076 eng 087

1390        TR  Cruikshank
 alg 055 gmt 064 phy 077 chm 076 eng 085

1393        BU  Hopper
 alg 045 gmt 069 phy 037 chm 026 eng 035

...Execution ends.
```

Notes

(1) This example displays a report using the student file described in the previous example. The report consists of a heading followed by two lines for each student containing selected fields of the file. A blank line is displayed between each student report.

(2) While use of the **display** verb is appropriate to display records, it is more traditional to use the **write** verb. This, of course, requires the proper file definition. The file-name is called "screen" and the record-name defined in the file section is called "display-record". An area of 80 characters is defined since most screens can contain an 80 character line. However, it should be noted that some systems because of their hardware design will cause an additional line containing blanks to be printed if the 80th character is not a blank.

(3) The file "screen" is opened and closed in the appropriate places in the program.

(4) Records are displayed using the **write** verb in the form

```
write display-record from ...
```

(5) A record containing spaces is defined in working storage and is used to display blank lines.

(6) The name field of the student record consists of a 17 character surname followed by 3 characters for the initials. In this example we wish to display the initials before the surname. In order to do this we must define two fields for the name.

(7) COBOL permits us to subdivide a field further by introducing new level numbers and by using new data-names. The field "name" is divided into two fields "surname" and "initials" as follows:

```
02 name.
   03 surname    pic x(17).
   03 initials   pic xxx.
```

The **picture** clause has been removed from the 02-level item and two new items are defined at the 03-level; these new items are elementary items. Now the name can be referred to by using "name" or by using "surname" and/or "initials".

(8) The 03-level number items have been indented to improve readability.

(9) COBOL permits us to use up to 49 levels; the reference manual describes the rules of how these can be used.

1.3.8 Inputting a File Name.

We often want to change the system-name in a program. For example instead of directing output to the terminal, we might wish to display it on the printer. This example shows how this could be accomplished under program control.

```
*
* Input a File Name.
*
 identification division.
 program-id. EXAMPLE-17.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to ''.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard
      value of '' is output-file-name.
 01   display-record.
      02 filler          pic x(80).

 working-storage section.

 01   output-file-name          pic x(12).
```

```
01  student-data.
    02 student-no          pic xxxx.
    02 name                pic x(20).
    02 age                 pic xx.
    02 sex                 pic x.
    02 class               pic x.
    02 school              pic x.
    02 algebra             pic xxx.
    02 geometry            pic xxx.
    02 physics             pic xxx.
    02 chemistry           pic xxx.
    02 english             pic xxx.

01  report-heading.
    02 filler              pic x(20) value is spaces.
    02 filler              pic x(20) value is 'Student Reports'.
    02 filler              pic x(40) value is spaces.

01  first-line.
    02 out-student-no      pic x(4).
    02 filler              pic x(10) value is spaces.
    02 out-name            pic x(20).

01  second-line.
    02 filler              pic x(5) value is ' alg'.
    02 out-algebra         pic xxx.
    02 filler              pic x(5) value is ' gmt'.
    02 out-geometry        pic xxx.
    02 filler              pic x(5) value is ' phy'.
    02 out-physics         pic xxx.
    02 filler              pic x(5) value is ' chm'.
    02 out-chemistry       pic xxx.
    02 filler              pic x(5) value is ' eng'.
    02 out-english         pic xxx.

01  blank-line             pic x(80) value is spaces.
```

```
procedure division.
    display 'enter output file name - terminal or printer'.
    move spaces to output-file-name.
    accept output-file-name.
    open input student-file.
    open output screen.
    write display-record from report-heading.
    write display-record from blank-line.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file
          screen.
    stop run.

process-student-data.
    perform display-student-data.
    perform read-student-record.

display-student-data.
    move student-no to out-student-no.
    move name to out-name.
    write display-record from first-line.
    move algebra to out-algebra.
    move geometry to out-geometry.
    move physics to out-physics.
    move chemistry to out-chemistry.
    move english to out-english.
    write display-record from second-line.
    write display-record from blank-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
enter output file name - terminal or printer
printer
                        Student Reports

234         Smith           SA
alg 075 gmt 100 phy 075 chm 065 eng 084

236         Jones           TO
alg 076 gmt 078 phy 055 chm 057 eng 078

238         Winterbourne    MS
alg 078 gmt 088 phy 056 chm 067 eng 088

239         Harrison        K
alg 022 gmt 087 phy 065 chm 087 eng 068

240         Graham          JW
alg 000 gmt 068 phy 075 chm 067 eng 087

242         Welch           JW
alg 075 gmt 075 phy 076 chm 075 eng 075

243         Dirksen         PH
alg 074 gmt 085 phy 054 chm 068 eng 084

245         Cowan           DD
alg 055 gmt 066 phy 077 chm 088 eng 099

249         Sullivan        J
alg 044 gmt 055 phy 066 chm 077 eng 088

256         Kitchen         MP
alg 074 gmt 049 phy 100 chm 097 eng 036

266         Taylor          YO
alg 095 gmt 083 phy 072 chm 066 eng 055

268         Allen           TT
alg 098 gmt 084 phy 073 chm 065 eng 059
```

```
270         Xerxes          X
alg 099 gmt 088 phy 077 chm 066 eng 055

272         Zimmerman       AB
alg 095 gmt 085 phy 078 chm 061 eng 057

375         Quantas         FL
alg 066 gmt 066 phy 066 chm 066 eng 066

388         Beatle          RA
alg 065 gmt 062 phy 073 chm 076 eng 087

390         Cruikshank      TR
alg 055 gmt 064 phy 077 chm 076 eng 085

393         Hopper          BU
alg 045 gmt 069 phy 037 chm 026 eng 035

...Execution ends.
```

Notes

(1) This example prompts the user to enter the system-name, either "terminal" or "printer", and then displays a report similar to the previous example on the chosen device.

(2) Before running this example, the user should check local installation rules of printing. These will vary from system to system and from location to location.

(3) The **assign** clause for the "screen" has been changed to

```
assign screen to ''
```

i.e. the system-name has been made a null string.

(4) A new clause has been added to the **fd** statement, namely

```
value of '' is output-file-name.
```

where "output-file-name" is a data-name which is defined in working storage. The period has been placed following the new clause.

(5) Before the file is opened, the user is prompted to enter the proper output system-name, either "printer" or "terminal". If "terminal" is entered, the program functions exactly the same as the previous example. However, if "printer" is entered, the output will appear on the line-printer.

(6) On examining the output produced on the printer, the user may be somewhat surprised at the results. The style and format of the output will depend on the type of system and printer that are used. One possible result will be that the first character of each line will not be printed. A second result may be that the vertical spacing of the output seems somewhat bizarre. The next example will try to correct these unusual results.

1.3.9 Printer Control Characters.

In the previous example, the first character of each line was not displayed when the output was directed to the printer. Engineers have designed many printers so that the first print position is a code which provides the printer with information about vertical spacing. This position is not printed and must be supplied by the programmer. This special character is often referred to as the *print-control-character*.

```
*
* Print a Report on the Printer.
*
 identification division.
 program-id. EXAMPLE-18.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to ''.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard
      value of '' is output-file-name.
 01   display-record.
      02 filler          pic x(80).

 working-storage section.

 01   output-file-name          pic x(12).
```

```
01  student-data.
    02 student-no          pic xxxx.
    02 name                pic x(20).
    02 age                 pic xx.
    02 sex                 pic x.
    02 class               pic x.
    02 school              pic x.
    02 algebra             pic xxx.
    02 geometry            pic xxx.
    02 physics             pic xxx.
    02 chemistry           pic xxx.
    02 english             pic xxx.

01  report-heading.
    02 filler              pic x value is spaces.
    02 filler              pic x(20) value is spaces.
    02 filler              pic x(20) value is 'Student Reports'.
    02 filler              pic x(39) value is spaces.

01  first-line.
    02 filler              pic x     value is spaces.
    02 out-student-no      pic x(4).
    02 filler              pic x(10) value is spaces.
    02 out-name            pic x(20).

01  second-line.
    02 filler              pic x     value is spaces.
    02 filler              pic x(5) value is ' alg'.
    02 out-algebra         pic xxx.
    02 filler              pic x(5) value is ' gmt'.
    02 out-geometry        pic xxx.
    02 filler              pic x(5) value is ' phy'.
    02 out-physics         pic xxx.
    02 filler              pic x(5) value is ' chm'.
    02 out-chemistry       pic xxx.
    02 filler              pic x(5) value is ' eng'.
    02 out-english         pic xxx.

01  blank-line             pic x(80) value is spaces.
```

```
procedure division.
     display 'enter output file name - terminal or printer'.
     move spaces to output-file-name.
     accept output-file-name.
     open input student-file.
     open output screen.
     write display-record from report-heading
          after advancing 1 lines.
     write display-record from blank-line
          after advancing 1 lines.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
           screen.
     stop run.

process-student-data.
     perform display-student-data.
     perform read-student-record.

display-student-data.
     move student-no to out-student-no.
     move name to out-name.
     write display-record from first-line
          after advancing 1 lines.
     move algebra to out-algebra.
     move geometry to out-geometry.
     move physics to out-physics.
     move chemistry to out-chemistry.
     move english to out-english.
     write display-record from second-line
          after advancing 1 lines
     write display-record from blank-line
          after advancing 1 lines.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
enter output file name - terminal or printer
printer
                              Student Reports

1234          Smith           SA
 alg 075 gmt 100 phy 075 chm 065 eng 084

1236          Jones           TO
 alg 076 gmt 078 phy 055 chm 057 eng 078

1238          Winterbourne    MS
 alg 078 gmt 088 phy 056 chm 067 eng 088

1239          Harrison        K
 alg 022 gmt 087 phy 065 chm 087 eng 068

1240          Graham          JW
 alg 000 gmt 068 phy 075 chm 067 eng 087

1242          Welch           JW
 alg 075 gmt 075 phy 076 chm 075 eng 075

1243          Dirksen         PH
 alg 074 gmt 085 phy 054 chm 068 eng 084

1245          Cowan           DD
 alg 055 gmt 066 phy 077 chm 088 eng 099

1249          Sullivan        J
 alg 044 gmt 055 phy 066 chm 077 eng 088

1256          Kitchen         MP
 alg 074 gmt 049 phy 100 chm 097 eng 036

1266          Taylor          YO
 alg 095 gmt 083 phy 072 chm 066 eng 055

1268          Allen           TT
 alg 098 gmt 084 phy 073 chm 065 eng 059
```

```
1270        Xerxes          X
 alg 099 gmt 088 phy 077 chm 066 eng 055

1272        Zimmerman       AB
 alg 095 gmt 085 phy 078 chm 061 eng 057

1375        Quantas         FL
 alg 066 gmt 066 phy 066 chm 066 eng 066

1388        Beatle          RA
 alg 065 gmt 062 phy 073 chm 076 eng 087

1390        Cruikshank      TR
 alg 055 gmt 064 phy 077 chm 076 eng 085

1393        Hopper          BU
 alg 045 gmt 069 phy 037 chm 026 eng 035

...Execution ends.
```

Notes

(1) COBOL provides us with a number of ways of handling the print-control-character. This example shows one method; others are described in the reference manual.

(2) Another field is added to each of the record definitions for lines to be displayed. In each case, a field of one character is defined as the first character in the record. It is initialized to contain the blank character.

(3) Now when the program is run, the report is printed with "proper" spacing and containing all the desired characters.

(4) The use of the "blank" as the print-control-character indicates that we wish to do "single" spacing. Other characters are used for "double" and "triple" spacing. Another character is used to space the printer to the top of the page. These characters depend on the type of system and printer being used and are described in the reference manual.

(5) When this program is run with output directed to the screen, a blank character may be displayed at the start of each line. This means that for some screens, we will be limited to 79 characters.

(6) If the user plans to use both the terminal and the printer interchangeably in programs, it will be wise to plan for the use of the print-control-character. The remainder of the examples in this tutorial deal with output directed to the terminal.

1.4 Selection.

1.4.1 Selection Using the If Verb.

In previous examples which used the student file, we always displayed a line for each record in the file. Clearly on occasion we wish to display a sub-set or selection of lines from the file. The **if** verb gives us a way to make decisions in our COBOL programs and in particular to select and display certain records.

```
*
* If   Sentence.
*
 identification division.
 program-id. EXAMPLE-19.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).
```

```
working-storage section.

01   student-data.
     02 student-no          pic xxxx.
     02 name                pic x(20).
     02 age                 pic xx.
     02 sex                 pic x.
     02 class               pic x.
     02 school              pic x.
     02 algebra             pic xxx.
     02 geometry            pic xxx.
     02 physics             pic xxx.
     02 chemistry           pic xxx.
     02 english             pic xxx.

01   report-heading.
     02 filler        pic x(20) value is spaces.
     02 filler        pic x(25) value is 'Algebra Report'.
     02 filler        pic x(40) value is spaces.

01   display-line.
     02 out-student-no      pic x(4).
     02 filler              pic x(10) value is spaces.
     02 out-name            pic x(20).
     02 filler              pic x(5) value is spaces.
     02 out-algebra         pic xxx.

01   blank-line            pic x(80) value is spaces.

procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
           screen.
     stop run.
```

```
process-student-data.
    if algebra > '049'
        perform display-student-data.
    perform read-student-record.

display-student-data.
    move student-no to out-student-no.
    move name to out-name.
    move algebra to out-algebra.
    write display-record from display-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

run
Execution begins...

Algebra Report

1234	Smith	SA	075
1236	Jones	TO	076
1238	Winterbourne	MS	078
1242	Welch	JW	075
1243	Dirksen	PH	074
1245	Cowan	DD	055
1256	Kitchen	MP	074
1266	Taylor	YO	095
1268	Allen	TT	098
1270	Xerxes	X	099
1272	Zimmerman	AB	095
1375	Quantas	FL	066
1388	Beatle	RA	065
1390	Cruikshank	TR	055

...Execution ends.

Notes

(1) This example produces a report containing the student number, name, and algebra mark of those students whose algebra mark is 50 or greater.

(2) This can be accomplished by changing the "process-student-data" paragraph as follows:

```
process-student-data.
    if algebra > '049'
        perform display-student-data.
    perform read-student-record.
```

Note that we have introduced a new COBOL verb, namely **if**. When the **if** sentence is encountered during execution, the condition

```
algebra > '049'
```

is evaluated. If the condition is *true*, the paragraph "display-student-data" is executed. If the condition is *false*, control proceeds to the next sentence, and another record is read.

(3) The **if** sentence always contains a condition. These conditions are similar to those used with the **perform-until** sentence.

(4) The symbols '>' and '=' used in conditions in this example are called *relational operators*. Actually there is a third one namely, '<'. The word **not** can be included with each of the three conditions as follows:

```
not =
not <
not >
```

giving a total of six relational operators. It is also possible to use **equal** or even **equals** instead of '='. A complete list of alternatives can be found in the reference manual.

(5) The **if** sentence ends with a period. It is important to note that there is only one period, and this period terminates the sentence. The importance of this will become evident in the next example.

(6) The reader might be tempted to omit the 0(zero) from the '049' portion of the condition and write it as '49' or even ' 49'. If the program were run, it would be unlikely that one would obtain the correct results. Thus, it is usually necessary to include the 0(zero) in the '049'. The reader is referred to the reference manual to determine the reason for this.

(7) The **perform** portion of the **if** is indented for easier readability.

1.4.2 Another Version of If.

It is sometimes convenient to group a number of COBOL statements as a single entity. This example demonstrates how this can be done.

```
*
* If   Sentence (Another way).
*
 identification division.
 program-id. EXAMPLE-20.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).
```

```
working-storage section.

01   student-data.
     02 student-no          pic xxxx.
     02 name                pic x(20).
     02 age                 pic xx.
     02 sex                 pic x.
     02 class               pic x.
     02 school              pic x.
     02 algebra             pic xxx.
     02 geometry            pic xxx.
     02 physics             pic xxx.
     02 chemistry           pic xxx.
     02 english             pic xxx.

01   report-heading.
     02 filler          pic x(20) value is spaces.
     02 filler          pic x(25) value is 'Algebra Report'.
     02 filler          pic x(40) value is spaces.

01   display-line.
     02 out-student-no       pic x(4).
     02 filler               pic x(10) value is spaces.
     02 out-name             pic x(20).
     02 filler               pic x(5) value is spaces.
     02 out-algebra          pic xxx.

01   blank-line              pic x(80) value is spaces.

procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
           screen.
     stop run.
```

```
process-student-data.
    perform display-student-data.
    perform read-student-record.

display-student-data.
    if algebra > '049'
        move student-no to out-student-no
        move name to out-name
        move algebra to out-algebra
        write display-record from display-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                    Algebra Report

1234    Smith          SA    075
1236    Jones          TO    076
1238    Winterbourne   MS    078
1242    Welch          JW    075
1243    Dirksen        PH    074
1245    Cowan          DD    055
1256    Kitchen        MP    074
1266    Taylor         YO    095
1268    Allen          TT    098
1270    Xerxes         X     099
1272    Zimmerman      AB    095
1375    Quantas        FL    066
1388    Beatle         RA    065
1390    Cruikshank     TR    055
...Execution ends.
```

Notes

(1) This example is another version of the previous example. It presents the concept of the *range* of the **if** and shows how a series of COBOL statements can be executed when the condition is true.

(2) The **if** sentence has been moved to the "display-student-data" paragraph. The periods have been removed from each of the sentences except the last in this paragraph. The **if** is followed by the four statements and is terminated by a period to form the **if** sentence. The statements in the *range* of the **if** are executed if the condition is true.

(3) All statements in the range of the **if** are indented for readability.

(4) The importance of the period cannot be over-emphasized. It terminates the **if** and no other periods should be placed in the range of the **if**.

(5) The word *statement* has been and will be used to refer to a COBOL sentence without a period.

(6) While the versions of the **if** in this and the previous example both function properly, it is suggested that one avoid, if possible, "large" **if** sentences.

1.4.3 The Else Option.

In the previous two examples, we did not require a line to be displayed if the algebra mark was less than 50. It is sometimes the case that we wish to perform one action if a condition is true and an alternative action if the condition is false; for example, one action if the mark is less than 50 and another if the mark is greater or equal to 50.

```
*
* Else Option.
*
 identification division.
 program-id. EXAMPLE-21.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).
```

```
working-storage section.

01   student-data.
     02 student-no          pic xxxx.
     02 name                pic x(20).
     02 age                 pic xx.
     02 sex                 pic x.
     02 class               pic x.
     02 school              pic x.
     02 algebra             pic xxx.
     02 geometry            pic xxx.
     02 physics             pic xxx.
     02 chemistry           pic xxx.
     02 english             pic xxx.

01   report-heading.
     02 filler          pic x(20) value is spaces.
     02 filler          pic x(20) value is 'Pass - Fail Report'.
     02 filler          pic x(40) value is spaces.

01   display-line.
     02 out-student-no      pic x(4).
     02 filler              pic x(10) value is spaces.
     02 out-name            pic x(20).
     02 filler              pic x(5) value is spaces.
     02 out-algebra         pic xxx.
     02 filler              pic x(5) value is spaces.
     02 pass-fail           pic x(10).

01   blank-line             pic x(80) value is spaces.

procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
           screen.
     stop run.
```

```
process-student-data.
    perform display-student-data.
    perform read-student-record.

display-student-data.
    if algebra < '050'
        move 'failed' to pass-fail
    else
        move 'passed' to pass-fail.
    move student-no to out-student-no.
    move name to out-name.
    move algebra to out-algebra.
    write display-record from display-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
```

Pass - Fail Report

1234	Smith	SA	075	passed
1236	Jones	TO	076	passed
1238	Winterbourne	MS	078	passed
1239	Harrison	K	022	failed
1240	Graham	JW	000	failed
1242	Welch	JW	075	passed
1243	Dirksen	PH	074	passed
1245	Cowan	DD	055	passed
1249	Sullivan	J	044	failed
1256	Kitchen	MP	074	passed
1266	Taylor	YO	095	passed
1268	Allen	TT	098	passed
1270	Xerxes	X	099	passed
1272	Zimmerman	AB	095	passed
1375	Quantas	FL	066	passed
1388	Beatle	RA	065	passed
1390	Cruikshank	TR	055	passed
1393	Hopper	BU	045	failed

```
...Execution ends.
```

Notes

(1) In this example a report is produced displaying student number, name, and algebra mark for all students as well as a field indicating if the student passed or failed algebra.

(2) A new field has been included in "display-line" to contain the pass-fail indication.

(3) The "display-student-data" paragraph has been altered to include the sentence

```
if algebra < '050'
    move 'failed' to pass-fail
else
    move 'passed' to pass-fail.
```

(4) The condition is tested and if it is true, 'failed' is moved to the display record; if it is false, 'passed' is moved. We refer to the two actions as being contained in the *true range* and *false range* of the **if** sentence. The true range ends with the **else** and the false range ends with the period.

(5) The **else** is placed on a separate line and both the true and false ranges are indented for readability.

1.4.4 Multiple Choice.

In the previous example we caused either "passed" or "failed" to be displayed in the record. This can be thought of as two *cases*. However, many situations arise where more than two cases are involved.

```
*
* Multiple Choice.
*
 identification division.
 program-id. EXAMPLE-22.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).
```

```
working-storage section.

01   student-data.
     02 student-no          pic xxxx.
     02 name                pic x(20).
     02 age                 pic xx.
     02 sex                 pic x.
     02 class               pic x.
     02 school              pic x.
     02 algebra             pic xxx.
     02 geometry            pic xxx.
     02 physics             pic xxx.
     02 chemistry           pic xxx.
     02 english             pic xxx.

01   report-heading.
     02 filler      pic x(20) value is spaces.
     02 filler      pic x(25) value is 'Letter - Grade Report'.
     02 filler      pic x(35) value is spaces.

01   first-line.
     02 out-student-no      pic x(4).
     02 filler              pic x(10) value is spaces.
     02 out-name            pic x(20).
     02 filler              pic x(5) value is spaces.
     02 out-algebra         pic xxx.
     02 filler              pic x(5) value is spaces.
     02 grade               pic x.

01   blank-line             pic x(80) value is spaces.

procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
           screen.
     stop run.
```

```
process-student-data.
     perform display-student-data.
     perform read-student-record.

display-student-data.
     if      algebra < '050'
          move 'F' to grade
     else if algebra < '060'
          move 'D' to grade
     else if algebra < '066'
          move 'C' to grade
     else if algebra < '075'
          move 'B' to grade
     else
          move 'A' to grade.
     move student-no to out-student-no.
     move name to out-name.
     move algebra to out-algebra.
     write display-record from first-line.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

run
Execution begins...

Letter - Grade Report

1234	Smith	SA	075	A
1236	Jones	TO	076	A
1238	Winterbourne	MS	078	A
1239	Harrison	K	022	F
1240	Graham	JW	000	F
1242	Welch	JW	075	A
1243	Dirksen	PH	074	B
1245	Cowan	DD	055	D
1249	Sullivan	J	044	F
1256	Kitchen	MP	074	B
1266	Taylor	YO	095	A
1268	Allen	TT	098	A
1270	Xerxes	X	099	A
1272	Zimmerman	AB	095	A
1375	Quantas	FL	066	B
1388	Beatle	RA	065	C
1390	Cruikshank	TR	055	D
1393	Hopper	BU	045	F

...Execution ends.

Notes

(1) The example produces a report which includes letter grades as well as the numeric values. The letter A, B, C, D, or F is displayed in the appropriate situation.

(2) The "display-student-data" paragraph now contains the more complicated **if** which handles the necessary cases.

```
if      algebra < '050'
        move 'F' to grade
else if algebra < '060'
        move 'D' to grade
else if algebra < '066'
        move 'C' to grade
else if algebra < '075'
        move 'B' to grade
else
        move 'A' to grade.
```

(3) Here we have a number of **if**'s with the entire **if** sentence ending with a single period. If the algebra mark is less than 50, an F is moved to the display line and then control passes to the next sentence. If the mark is not less than 50 control passes to the clause

```
else if algebra < '060'
```

Here the program asks is the mark is less than 60 and if that is the case, a D is moved to the display line and then control passes to the next sentence. If not control passes to the next **else if** clause. This process continues until the correct condition is found.

(4) The **else if** clause

```
else if algebra < '060'
```

determines if the mark is in the range 50 to 59 since the previous condition

```
algebra < '050'
```

eliminated the case of all marks less than 50. Similarly, at each of the other **else if** clauses, the program checks for the correct range of marks.

(5) The if sentence could have been written as follows:

```
if algebra < '050'
    move 'F' to grade
else
    if algebra < '060'
        move 'D' to grade
    else
        if algebra < '066'
            move 'C' to grade
        else
            if algebra < '075'
                move 'B' to grade
            else
                move 'A' to grade.
```

Either style of the if sentence is acceptable. However, the authors prefer the style used in the program.

1.4.5 Logical Operators - And and Or.

On many occasions we wish to test more than one condition. The *logical operators* **and** and **or** can be used to accomplish this.

```
*
* And and Or.
*
 identification division.
 program-id. EXAMPLE-23.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).
```

```
working-storage section.

01   student-data.
     02 student-no          pic xxxx.
     02 name                pic x(20).
     02 age                 pic xx.
     02 sex                 pic x.
     02 class               pic x.
     02 school              pic x.
     02 algebra             pic xxx.
     02 geometry            pic xxx.
     02 physics             pic xxx.
     02 chemistry           pic xxx.
     02 english             pic xxx.

01   report-heading.
     02 filler          pic x(20) value is spaces.
     02 filler          pic x(30)
          value is 'Class 2 - Algebra Report'.
     02 filler          pic x(40) value is spaces.

01   first-line.
     02 out-student-no      pic x(4).
     02 filler              pic x(10) value is spaces.
     02 out-name            pic x(20).
     02 filler              pic x(5) value is spaces.
     02 out-algebra         pic xxxxx.

01   blank-line            pic x(80) value is spaces.

procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
           screen.
     stop run.
```

```
process-student-data.
    if algebra > '074' and class = '2'
        perform display-student-data.
    perform read-student-record.

display-student-data.
    move student-no to out-student-no.
    move name to out-name.
    move algebra to out-algebra.
    write display-record from first-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                    Class 2 - Algebra Report

1236        Jones                   TO      076
1268        Allen                   TT      098
...Execution ends.
```

Notes

(1) This example produces a report of students in the second class whose algebra mark is 75 or greater.

(2) In this case the **if** sentence

```
if algebra > '074' and class = '2'
    perform display-student-data.
```

uses a *compound condition* with the logical operator **and**. If both conditions are true the record is displayed. If either or both of the conditions are false, the record is not displayed.

(3) If the logical operator **or** were used instead of **and** in this example, records would be displayed for all students in the second class as well as all those in other classes whose algebra mark was 75 or greater.

1.4.6 Combined Use of And and Or

On occasion we wish to combine the logical operators **and** and **or** in a compound condition.

```
*
* Compound Conditions.
*
 identification division.
 program-id. EXAMPLE-24.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).
```

```
working-storage section.

01  student-data.
    02 student-no          pic xxxx.
    02 name                pic x(20).
    02 age                 pic xx.
    02 sex                 pic x.
    02 class               pic x.
    02 school              pic x.
    02 algebra             pic xxx.
    02 geometry            pic xxx.
    02 physics             pic xxx.
    02 chemistry           pic xxx.
    02 english             pic xxx.

01  report-heading.
    02 filler              pic x(20) value is spaces.
    02 filler              pic x(30)
          value is 'Class 2 & 4 - Algebra Report'.
    02 filler              pic x(40) value is spaces.

01  first-line.
    02 out-student-no      pic x(4).
    02 filler              pic x(10) value is spaces.
    02 out-name            pic x(20).
    02 filler              pic x(5) value is spaces.
    02 out-class           pic x.
    02 filler              pic xx value is spaces.
    02 out-algebra         pic xxxxx.

01  blank-line             pic x(80) value is spaces.

procedure division.
    open input student-file.
    open output screen.
    write display-record from report-heading.
    write display-record from blank-line.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file
          screen.
    stop run.
```

```
process-student-data.
     if   (algebra > '074' and class = '2')
       or
            (algebra < '050' and class = '4')
          perform display-student-data.
     perform read-student-record.

display-student-data.
     move student-no to out-student-no.
     move name to out-name.
     move class to out-class.
     move algebra to out-algebra.
     write display-record from first-line.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                    Class 2 & 4 - Algebra Report

1236        Jones              TO      2    076
1239        Harrison           K       4    022
1249        Sullivan           J       4    044
1268        Allen              TT      2    098
...Execution ends.
```

Notes

(1) This example produces a report for students in the second class whose algebra mark is 75 or greater as well as students in the fourth class whose algebra mark is less than 50.

(2) The compound condition

```
     (algebra > '074' and class = '2')
  or
     (algebra < '050' and class = '4')
```

performs the required test. It determines if the class 2 - algebra condition is true and then if the class 4- algebra condition is true. If either is true the record is displayed.

(3) Parentheses have been introduced in order that the conditions are evaluated in the desired order and to make the compound condition easier to understand. Quantities enclosed in parentheses are evaluated first.

(4) If parentheses are omitted, and's are evaluated first, followed by or's. Thus in this example the parentheses could have been omitted. However, they were included to reduce possible ambiguity.

(5) The following compound condition illustrates the use of parentheses.

```
    (class = '2' or class = '4')
and
    (algebra < '050' or algebra > '075')
```

In this case, the report would contain students in the second or fourth class who had marks less than 50 or greater than 75. If parentheses were omitted the report would contain all students in class 2, students in class 4 with algebra marks less than 50, and students whose algebra mark was greater than 75.

(6) Finally, this new compound condition could be written somewhat more compactly as

```
    (class = '2' or '4')
and
    (algebra > '075' or < '050')
```

The reader is referred to the reference manual for a more complete presentation of *implied subjects* in compound conditions.

1.5 Arithmetic.

1.5.1 Integer Arithmetic

One of the major uses of computers is to perform arithmetic. This example introduces the verbs used for arithmetic operations. It also presents a new field definition for defining numbers to be used in arithmetic operations. The program itself has little meaning; it is used to demonstrate arithmetic operations.

```
*
* Simple Arithmetic (Integer Numbers).
*
 identification division.
 program-id. EXAMPLE-25.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01  a     pic 9(4) value is 1234.
 01  b     pic 9(6) value is 123456.

 01  c     pic 9(7).
 01  d     pic 9(6).
 01  e     pic 9(10).
 01  f     pic 9(3).
```

```
procedure division.
    add a b giving c.
    subtract a from b giving d.
    multiply a by b giving e.
    divide a into b giving f.
    display 'a ' a.
    display 'b ' b.
    display ' '.
    display 'c ' c.
    display 'd ' d.
    display 'e ' e.
    display 'f ' f.
    stop run.
```

Sample Program Execution

```
run
Execution begins...
a   1234
b   123456

c   0124690
d   122222
e   0152344704
f   100
...Execution ends.
```

Notes

(1) This example defines two fields "a" and "b", assigns the values 1234 and 123456, and then finds the sum, difference, product, and quotient of the two values. The answers of the four operations are stored in four fields and are then displayed.

(2) A different **picture** clause is required for fields which are used to store numbers which will be used to enter into arithmetic operations. The two definitions for "a" and "b" are written as

```
01  a pic 9(4) value is 1234.
01  b pic 9(6) value is 123456.
```

Here "a" holds a 4-digit number with the value 1234 and "b" holds a 6-digit number with the value 123456. The two definitions could have been written as:

```
01  a pic 9999 value is 1234.
01  b pic 999999 value is 123456.
```

(3) It is a basic rule of COBOL that fields that are to enter into arithmetic operations *must* be declared using **picture**'s with 9's instead of x's. These new **picture**'s are referred to as *numeric pictures*.

(4) The integer values, 1234 and 123456, are not enclosed by quotation marks. They are referred to as *numeric literals*.

(5) The four basic sentences used in this example for doing arithmetic operations are:

```
add a b giving c.
subtract a from b giving d.
multiply a by b giving e.
divide a into b giving f.
```

The meaning of each of these sentences is fairly obvious. In each case the contents of the fields "a" and "b" enter into an arithmetic operation, and the answers are stored in "c", "d", "e", and "f" respectively.

(6) The fields "c", "d", "e", and "f" are the receiving fields for the four computations. When these fields have more positions than needed to hold the answer, zeros are padded on the left.

(7) When division is performed the result is stored in "f" giving a value 100. The remainder, namely 56, is lost. Later we will see how the remainder can be retained.

(8) A number of other forms of the arithmetic verbs exist in COBOL. These are described in the reference manual. Several are presented in future examples.

1.5.2 Decimal Places

In the previous example, the decimal point was assumed to be to the right of the right-most digit. This example introduces decimal values in arithmetic operations.

```
*
* Simple Arithmetic (With Decimal Places).
*
 identification division.
 program-id. EXAMPLE-26.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01   a      pic 99v99 value is 12.34.
 01   b      pic 999v999 value is 123.456.

 01   c      pic 9(7).
 01   d      pic 999v999.
 01   e      pic 9(5)v999.
 01   f      pic 999v999.

 procedure division.
      add a b giving c.
      subtract a from b giving d.
      multiply a by b giving e.
      divide a into b giving f.
      display 'a ' a.
      display 'b ' b.
      display ' '.
      display 'c ' c.
      display 'd ' d.
      display 'e ' e.
      display 'f ' f.
      stop run.
```

Sample Program Execution

```
run
Execution begins...
a  1234
b  123456

c  0000135
d  111116
e  01523447
f  010004
...Execution ends.
```

Notes

(1) To indicate a decimal point in the field, we use a 'v' in the appropriate place in the **picture** clause. For example,

```
01  a pic 99v99 value is 12.34.
01  b pic 999v999 value is 123.456.
```

defines a 4-digit field which has two decimal places, and a 6-digit field which has 3 decimal places. The values 12.34 and 123.456 are assigned to the two fields.

(2) Consider the statement

```
add a b giving c.
```

Here the computer lines up the decimal points and performs the addition as follows:

```
 12.34
123.456
-------
135.796
```

The result is stored in "c". However, "c" has a **picture** of 9(7) meaning it can hold a 7-digit integer. Hence, the portion of the result to the right of the decimal is dropped or truncated. Only the 135 is stored with four zeroes inserted on the left to fill out the field.

(3) In the case of the subtract operation, we were not satisfied with the truncation of the result to an integer. The **picture** clause for "d" is defined as

```
01  d pic 999v999.
```

and the appropriate value is stored in "d" namely 11116. Note that no decimal point is displayed i.e. we have to remember where it is. In a similar way, we have included decimal places for "e" and "f".

(4) There is no decimal point physically recorded in working storage. The symbol 'v' is used to indicate its position.

(5) Numeric literals can contain decimal points, as illustrated by the values 12.34 and 123.456.

(6) Consider the statement

```
add a b giving c rounded
```

and assume that the **picture** clause for "c" has been defined as

```
01  c  pic 99999v99.
```

Here the value displayed would be 0013580. Since the **picture** clause is defined to have 2 digits to the right of the decimal, the third digit after the decimal is examined and if it is five or greater, the result is *rounded*. In this example, the third digit after the decimal is a 6 and hence the value is rounded. The **rounded** option can be used with the other arithmetic verbs.

(7) Assume that the **picture** clause for "e" had been written as

```
01  e  pic 999v999.
```

In this case the result for "e" would have been 523447 i.e. the 1 has been dropped from the result. COBOL does *not* check if the result is too large for the receiving field; it merely truncates the result to fit in the receiving field. This can be remedied by using the **on size error** option. For example,

```
multiply a by b giving e
    on size error display 'arithmetic overflow'.
```

would tell the user that an error had occurred and an appropriate action could be taken. This option can also be used with all the arithmetic verbs.

(8) The reader is referred to the reference manual for a more complete description of **rounded** and **on size error**.

1.5.3 Negative Numbers

The previous two examples used only positive values. This example shows how negative values can be defined and used.

```
*
* Simple Arithmetic (With Decimal Places and Negative Numbers).
*
 identification division.
 program-id. EXAMPLE-27.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01  a      pic s99v99 value is -12.34.
 01  b      pic s999v999 value is 123.456.

 01  c      pic s9(4)v999.
 01  d      pic s9(4)v99.
 01  e      pic s9(6)v99999.
 01  f      pic s9(3)v999.

 procedure division.
      add a b giving c.
      subtract a from b giving d.
      multiply a by b giving e.
      divide a into b giving f.
      display 'a ' a.
      display 'b ' b.
      display ' '.
      display 'c ' c.
      display 'd ' d.
      display 'e ' e.
      display 'f ' f.
      stop run.
```

Sample Program Execution

```
run
Execution begins...
a   123M
b   12345F

c   011111F
d   01357I
e   0015234470M
f   01000M
...Execution ends.
```

Notes

(1) We wish to assign the value -12.34 to "a" rather than 12.34. This is done by changing the **picture** for "a" to

```
02 a pic s99v99 value is -12.34.
```

(2) The numeric literal is written as one would expect, namely -12.34.

(3) The **picture** has been changed to s99v99 from 99v99. The character 's' is included to indicate that the field may contain a negative value and that provision must be made to store a sign.

(4) The other **picture** clauses have also been changed in anticipation that the respective fields may contain negative values. Clearly the definition for "b" need not have been changed since "b" contains a positive value. However, without doing the actual arithmetic operations, we cannot be sure if "c", "d", "e", or "f" will contain positive or negative results.

(5) If the 's' is omitted and the result of an arithmetic operation is negative, the result will be stored as a positive value. Hence it is safer to include the 's' for all fields unless one is certain that the value to be stored is positive.

(6) Upon examination of the output, the user is probably somewhat disconcerted to find that the right-most character of each of the results may be a letter instead of a digit. In fact the output produced by your system may differ from that shown in this example. This depends on the particular coding or *collating sequence* used by your system. You should refer to the reference manual if your output differs.

(7) In COBOL the sign is stored as part of the right-most digit in the field. This results in unexpected letters being displayed. The letters A - I represent the positive integers 1 - 9 and the letters J - R represent the negative integers 1 - 9. Thus, 123M is -12.34 (the 'v' places the decimal) and 12345F is 123.456. Positive zero is represented by { and negative zero is }. A future example will show how to display negative values in a more appropriate fashion.

1.5.4 Expressions and the Compute Verb.

In the previous examples, we were limited to one arithmetic operation for each verb. Mathematicians have provided us with a language for expressing arithmetic computations, namely algebra. COBOL has a facility to incorporate certain aspects of this language by permitting one to define a mathematical expression and to then compute the required value.

```
*
* Compute Verb.
*
 identification division.
 program-id. EXAMPLE-28.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01  a      pic s99v99 value is -12.34.
 01  b      pic s999v999 value is 123.456.

 01  c      pic s999v999.
 01  d      pic s9(6)v9(6).
 01  e      pic s99v99.
 01  f      pic s9(3).

 procedure division.
      compute c = a + b.
      compute d = a + b * c + 425.2.
      compute e = b / a * (b - a).
      compute f = 4 ** 3.
      display 'a ' a.
      display 'b ' b.
      display ' '.
      display 'c ' c.
      display 'd ' d.
      display 'e ' e.
      display 'f ' f.
      stop run.
```

Sample Program Execution

```
run
Execution begins...
a   123M
b   12345F

c   11111F
d   01413079689F
e   585P
f   06D
...Execution ends.
```

Notes

(1) This example contains a number of arithmetic expressions listed below. A few others are included for completeness.

```
a + b
a + b * c + 425.2
b / a * (b - a)
4 ** 3
a
- a + b
7.9
b ** - 2
```

Each example consists of one or more numeric quantities, combined with the symbols +, -, *, /, and **, which represent addition, subtraction, multiplication, division, and exponentiation respectively. The numeric quantities are data-names representing numeric fields or numeric literals. Parentheses are used to denote operations which are to be evaluated first.

(2) Just as in algebra, priority rules are used to determine the order of computation of the items in an expression. Quantities in parentheses are considered sub-expressions and are evaluated first, inner-most parentheses receive priority over outer parentheses. The priority of operators is as follows, arranged in descending order.

-	unary minus
**	exponentiation
* /	multiply and divide
+ -	add and subtract

Whenever any ambiguity exists, such as in the third example, the left-most operation is performed first.

(3) The expressions are entered with a space on either side of the operator. If spaces were not the rule, it would be difficult to determine if a-b were a data-name or an expression.

(4) The **compute** verb in the statement

```
compute d = a + b * c + 452.2
```

causes the expression to be evaluated and its value is assigned to "d".

(5) Spaces must appear on either side of the equal sign.

(6) The data-name to the left of the equal sign may appear in the arithmetic expression. Consider the example

```
compute a = a + 1.
```

Here the expression "a + 1" is evaluated first. Then the result is assigned to "a".

(7) Expressions may also be used in conditions. For example in the condition

```
a + 4 > b - 234
```

both arithmetic expressions are evaluated and then the comparison is made.

1.6 Printing and Editing Numeric Values.

1.6.1 Decimals in Output.

The previous several examples introduced how one can do arithmetic operations. However, the output produced left much to be desired in that it contained high-order zeroes, no decimal points, and even displayed letters instead of digits in the case of signed values. This example shows how decimal points can be included as part of the output. Future examples will show various ways of *editing* output values to have a more reasonable appearence.

```
*
* Displaying the Decimal Point.
*
 identification division.
 program-id. EXAMPLE-29.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01  a     pic 99V99 value is 12.34.
 01  b     pic 999V999 value is 123.456.

 01  c     pic 9(5).99.
 01  d     pic 9(5).999.
 01  e     pic 9(7).99999.
 01  f     pic 9(3).999.
```

```
procedure division.
    add a b giving c.
    subtract a from b giving d.
    multiply a by b giving e.
    divide a into b giving f.
    display 'a ' a.
    display 'b ' b.
    display ' '.
    display 'c ' c.
    display 'd ' d.
    display 'e ' e.
    display 'f ' f.
    stop run.
```

Sample Program Execution

```
run
Execution begins...
a   1234
b   123456

c   00135.79
d   00111.116
e   0001523.44704
f   010.004
...Execution ends.
```

Notes

(1) The four lines of output for "c", "d", "e", and "f" now include the decimal point in the correct position. This is accomplished by changing the four **picture** clauses to contain a '.' instead of a 'v'.

(2) Because the decimal points are inserted, an extra character appears in each of the fields that are displayed.

(3) By writing '.' instead of 'v' the decimal point actually appears in working storage. (Recall that in the case of 'v' only the position was recorded and no physical decimal point was inserted.) Since the decimal point is present, the field is *not* a legitimate *numeric* field, and cannot therefore enter into arithmetic operations. Thus it would be incorrect to change the definition of "a" as follows:

```
01 a pic 99.99 value is 12.34.
```

(4) **Picture** clauses containing the '.' are called *output pictures* and the data stored in them are called *numeric edited* data. They cannot be used in arithmetic operations. They can be used only to receive results from arithmetic operations and are used solely for display purposes.

1.6.2 Suppress Leading Zeros and Printing Minus Signs.

Leading zeros are usually not considered necessary when displaying numeric results. This example shows how they can be eliminated.

```
*
* Suppress Leading Zeroes.
*
 identification division.
 program-id. EXAMPLE-30.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01  a      pic 99V99 value is 12.34.
 01  b      pic 999V999 value is 123.456.

 01  c      pic z(5).99.
 01  d      pic z(5).99-.
 01  e      pic z(7).99999.
 01  f      pic z(3)9.999.

 procedure division.
      add a b giving c.
      subtract a from b giving d.
      multiply a by b giving e.
      divide a into b giving f.
      display 'a ' a.
      display 'b ' b.
      display ' '.
      display 'c ' c.
      display 'd ' d.
      display 'e ' e.
      display 'f ' f.
      stop run.
```

Sample Program Execution

```
run
Execution begins...
a   1234
b   123456

c       135.79
d       111.11
e        1523.44704
f       10.004
...Execution ends.
```

Notes

(1) The output no longer has leading zeroes. The symbol 'z' acts precisely like a '9' except that if the digit to be displayed is a '0', and no non-zero digit appears to the left of it, it is replaced by a blank character.

(2) It is common practice to have '9's' after the decimal place in output pictures. If "c" had been defined to have 2 z's after the decimal and the value to be displayed were 00000.00, all that would be displayed would be spaces. By using 9's we are assured that .00 would be displayed.

(3) If a value to be displayed is negative, we will wish to display a minus sign to the left of the value.

(4) The '-' character behaves as a 'z' with one extra feature. Should the number be negative, a minus sign is placed immediately to the left of the first non-blank character to be displayed. If the **picture** clauses for were written as

```
01  a    pic s99v99 value is -12.34.
01  b    pic 999v999 value is 123.456.

01  c    pic -(5).99.
01  d    pic -(5).99.
01  e    pic -(7).99999.
01  f    pic -(3).999.
```

and the output displayed would be

```
a  123M
b  123456

c  111.11
d  135.79
e  -1523.44704
f  -10.004
```

1.6.3 Dollar Signs, Commas, and CR.

On many occasions the values we wish to display represent some form of money, usually dollars. In this case we may wish to display the $ sign as well as inserting commas to make the output more readable. Finally, accountants rarely display a minus sign if a value is negative. They usually use the CR symbol. These features are incorporated in this example.

```
*
* Printing Dollar Signs, Commas, and CR.
*
 identification division.
 program-id. EXAMPLE-31.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01   a      pic s9999V99 value is -123.40.
 01   b      pic s999999V99 value is 12345.60.

 01   c      pic $(7).99.
 01   d      pic $$$,$$$,$$$.99CR.
 01   e      pic $$$,$$$,$$$.99CR.
 01   f      pic $$$,$$$,$$$.99CR.

 procedure division.
      add a b giving c.
      subtract a from b giving d.
      multiply a by b giving e.
      divide a into b giving f.
      display 'a ' a.
      display 'b ' b.
      display ' '.
      display 'c ' c.
      display 'd ' d.
      display 'e ' e.
      display 'f ' f.
      stop run.
```

Sample Program Execution

```
run
Execution begins...
a   01234}
b   0123456{

c     $12222.20
d          $12,469.00
e      $1,523,447.04CR
f            $100.04CR
...Execution ends.
```

Notes

(1) The '$' character also behaves like a 'z' with one extra feature. A '$' is inserted immediately to the left of the left-most non-blank character. Examine the value displayed for "c".

(2) In the last three lines of output, commas are inserted in the **picture** clauses. This causes commas to be displayed whenever there are significant digits to the left of it.

(3) The values to be displayed for "e" and "f" are negative and instead of displaying a minus sign to the left of the value, the symbol CR is displayed to the right of the value.

(4) Finally, note that as extra characters such as comma, CR, '.' etc. are included in the **picture** clauses, extra characters appear in the displayed values. Note also that when displaying "d" no CR is present but it should be noted that two blank spaces have been inserted.

1.6.4 Combining Edit Characters

In order to achieve the desired form of output, it is often necessary to combine some of the edit characters. This example shows a few examples of how this can be done.

```
*
* Combining Edit Symbols.
*
 identification division.
 program-id. EXAMPLE-32.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.
 data division.

 working-storage section.

 01  a     pic s9999v99 value is -123.40.
 01  b     pic s999999v99 value is 12345.60.

 01  c     pic $z(6).99.
 01  d     pic $zz,zzz,zzz.99CR.
 01  e     pic $zz,zzz,zz9.99CR.
 01  f     pic $zz,zzz,zz9.99CR.

 procedure division.
      add a b giving c.
      subtract a from b giving d.
      multiply a by b giving e.
      divide a into b giving f.
      display 'a ' a.
      display 'b ' b.
      display ' '.
      display 'c ' c.
      display 'd ' d.
      display 'e ' e.
      display 'f ' f.
      stop run.
```

Sample Program Execution

```
run
Execution Begins...
a  01234}
b  0123456{

c  $ 12222.20
d  $      12,469.00
e  $  1,523,447.04CR
f  $         100.04CR
...Execution ends.
```

Notes

(1) The '$' and 'z' symbol can be combined in one **picture** clause as in the above cases. In these cases the '$' signs all occur in column one of the output. In previous example the '$' signs *floated* to be immediately left of the most significant digit.

(2) In the last two cases a '9' has been inserted to the left of the decimal. This would cause a value, say .25, to be displayed as 0.25 instead of .25.

(3) The use of various edit characters to produce different forms of output depends very much on the user's particular preferences. It may also depend on installation standards.

1.7 Two Examples Using Files and Arithmetic.

1.7.1 Student Averages

This and the next example use some of the material presented in the previous examples to demonstrate how arithmetic can be used.

```
*
* Calculate Student Averages.
*
 identification division.
 program-id. EXAMPLE-33.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).

 working-storage section.

 01   student-total              pic 9(5).
```

```
01  student-data.
    02 student-no          pic xxxx.
    02 name                pic x(20).
    02 age                 pic xx.
    02 sex                 pic x.
    02 class               pic x.
    02 school              pic x.
    02 algebra             pic 999.
    02 geometry            pic 999.
    02 physics             pic 999.
    02 chemistry           pic 999.
    02 english             pic 999.

01  report-heading.
    02 filler              pic x(20) value is spaces.
    02 filler              pic x(30) value is 'Student Averages'.
    02 filler              pic x(30) value is spaces.

01  first-line.
    02 out-student-no      pic x(4).
    02 filler              pic x(5) value is spaces.
    02 out-name            pic x(20).
    02 out-algebra         pic z(4)9.
    02 out-geometry        pic z(4)9.
    02 out-physics         pic z(4)9.
    02 out-chemistry       pic z(4)9.
    02 out-english         pic z(4)9.
    02 out-average         pic z(5).9.

01  blank-line             pic x(80) value is spaces.

procedure division.
    open input student-file.
    open output screen.
    write display-record from report-heading.
    write display-record from blank-line.
    perform read-student-record.
    perform process-student-data
        until student-no = high-values.
    close student-file
          screen.
    stop run.
```

```
process-student-data.
      perform process-student-marks.
      perform display-student-data.
      perform read-student-record.

process-student-marks.
      move zero to student-total.
      add algebra to student-total.
      add geometry to student-total.
      add physics to student-total.
      add chemistry to student-total.
      add english to student-total.
      divide student-total by 5 giving out-average.

display-student-data.
      move student-no to out-student-no.
      move name to out-name.
      move algebra to out-algebra.
      move geometry to out-geometry.
      move physics to out-physics.
      move chemistry to out-chemistry.
      move english to out-english.
      write display-record from first-line.
      write display-record from blank-line.

read-student-record.
      read student-file into student-data
            at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                    Student Averages

1234  Smith          SA   75  100   75   65   84   79.8
1236  Jones          TO   76   78   55   57   78   68.8
1238  Winterbourne   MS   78   88   56   67   88   75.4
1239  Harrison       K    22   87   65   87   68   65.8
1240  Graham         JW    0   68   75   67   87   59.4
1242  Welch          JW   75   75   76   75   75   75.2
1243  Dirksen        PH   74   85   54   68   84   73.0
1245  Cowan          DD   55   66   77   88   99   77.0
1249  Sullivan       J    44   55   66   77   88   66.0
1256  Kitchen        MP   74   49  100   97   36   71.2
1266  Taylor         YO   95   83   72   66   55   74.2
1268  Allen          TT   98   84   73   65   59   75.8
1270  Xerxes         X    99   88   77   66   55   77.0
1272  Zimmerman      AB   95   85   78   61   57   75.2
1375  Quantas        FL   66   66   66   66   66   66.0
1388  Beatle         RA   65   62   73   76   87   72.6
1390  Cruikshank     TR   55   64   77   76   85   71.4
1393  Hopper         BU   45   69   37   26   35   42.4

...Execution ends.
```

Notes

(1) This example calculates the average mark of the five courses for each student and displays a report of this information.

(2) A heading is displayed for the report and then one line is displayed for each student containing the marks and the calculated average.

(3) An area called "student-total" is defined in working storage to contain the total of the five marks.

(4) The definitions for the five marks in "student-data" have been changed from pic xxx to pic 999. These values will be used in arithmetic operations and hence must be defined with 9's instead x's.

(5) The calculation of the student average is done in the "process-student-marks" paragraph using a number of sentences each with one verb. A slightly different version of the add is used, namely

```
add algebra to student-total
```

instead of the longer and more complicated

```
add algebra student-total giving student-total.
```

(6) The required computation could have been done in the following manner

```
compute out-average = (algebra + geometry + physics
                       + chemistry + english) / 5.
```

or alternatively as

```
add algebra geometry physics chemistry english
        giving student-total.
divide student-total by 5 giving out-average.
```

(7) If for some reason any particular mark in the file contained non-numeric characters, an error would occur when the value was read and the program would terminate.

1.7.2 School Algebra Averages.

This example calculates school algebra averages.

```
*
* Calculate Class Algebra Averages.
*
 identification division.
 program-id. EXAMPLE-34.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).

 working-storage section.

 01   totals.
      02 total-1                  pic 9(5).
      02 total-2                  pic 9(5).
      02 total-3                  pic 9(5).
      02 count-1                  pic 9(5).
      02 count-2                  pic 9(5).
      02 count-3                  pic 9(5).
```

```
01   student-data.
     02 student-no          pic xxxx.
     02 name                pic x(20).
     02 age                 pic xx.
     02 sex                 pic x.
     02 class               pic x.
     02 school              pic x.
     02 algebra             pic 999.
     02 geometry            pic 999.
     02 physics             pic 999.
     02 chemistry           pic 999.
     02 english             pic 999.

01   report-heading.
     02 filler          pic x(20) value is spaces.
     02 filler          pic x(30) value is 'Algebra Averages'.
     02 filler          pic x(30) value is spaces.

01   school-line.
     02 filler              pic x(8) value is 'School'.
     02 out-school          pic 9.
     02 filler              pic x(12) value is ' Average is '.
     02 out-average         pic z(5).9.

01   blank-line             pic x(80) value is spaces.

procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     move zeros to totals.
     perform process-student-data
          until student-no = high-values.
     perform display-averages
     close student-file
           screen.
     stop run.

process-student-data.
     perform process-student-marks.
     perform read-student-record.
```

```
process-student-marks.
    if school = '1'
        add algebra to total-1
        add 1 to count-1
    else if school = '2'
        add algebra to total-2
        add 1 to count-2
    else
        add algebra to total-3
        add 1 to count-3.

display-averages.
    move 1 to out-school.
    divide total-1 by count-1 giving out-average.
    write display-record from school-line.
    move 2 to out-school.
    divide total-2 by count-2 giving out-average.
    write display-record from school-line.
    move 3 to out-school.
    divide total-3 by count-3 giving out-average.
    write display-record from school-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                        Algebra Averages

School  1 Average is    63.2
School  2 Average is    62.8
School  3 Average is    71.1
...Execution ends.
```

Notes

(1) This example calculates the algebra average for each of the three schools.

(2) In this case, three areas are defined to contain the total of the algebra marks for each school. Three other areas are defined to contain the number of students in each school.

(3) The six totals have been defined as a data structure called "totals". This permits us to store zeros in each of the six areas with the one sentence

```
move zeros to totals
```

instead of requiring six separate **move** sentences.

(4) The "process-student-marks" paragraph determines the school of the current record and then adds the algebra mark to the appropriate total. It also adds 1 to the appropriate "count" total.

(5) After all the records are read, the averages are calculated and displayed in the "display-averages" paragraph.

(6) The reader might be tempted to use **value is** clauses to set the six totals to zero. This would work correctly in this example. However, it not considered good programming practice to initialize values in working storage unless they are to remain unchanged by the program. In this case the totals were changed and hence were zeroed by the appropriate **move**.

1.8 Subscripted Data-names.

1.8.1 Subscripted Data-names.

Most data processing applications in some way involve the use of *tables* of information. This section introduces features of COBOL which permit the easy handling of table data, and in particular *subscripted data-names* are introduced.

```
*
* Number of Students at each School.
*
 identification division.
 program-id. EXAMPLE-35.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).

 working-storage section.

 01   i          pic 9.
```

```
01  count-table.
    02 counts        pic 9(5) occurs 3 times.

01  student-data.
    02  student-no          pic xxxx.
    02 name                 pic x(20).
    02 age                  pic xx.
    02 sex                  pic x.
    02 class                pic x.
    02 school               pic x.
    02 algebra              pic 999.
    02 geometry             pic 999.
    02 physics              pic 999.
    02 chemistry            pic 999.
    02 english              pic 999.

01  report-heading.
    02 filler          pic x(20) value is spaces.
    02 filler          pic x(30) value is 'Enrollment Numbers'.
    02 filler          pic x(30) value is spaces.

01  school-line.
    02 filler              pic x(8) value is 'School '.
    02 out-school          pic 9.
    02 filler              pic x(5) value is ' has '.
    02 out-count           pic zz9.
    02 filler              pic x(9) value is ' Students'.

01  blank-line             pic x(80) value is spaces.
```

```
procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     move zeros to count-table.
     perform process-student-data
          until student-no = high-values.
     move 1 to i.
     perform display-totals 3 times.
     close student-file
           screen.
     stop run.

process-student-data.
     perform process-student-marks.
     perform read-student-record.

process-student-marks.
     move school to i.
     add 1 to counts(i).

display-totals.
     move i to out-school.
     move counts(i) to out-count.
     write display-record from school-line.
     add 1 to i.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                         Enrollment Numbers

School  1 has   5 Students
School  2 has   6 Students
School  3 has   7 Students
...Execution ends.
```

Notes

(1) This program reads the student file and determines the number of students in each of the three schools. Note that a previous example needed this information in order to calculate school algebra averages. In that case, three areas were defined to hold the three required counters. In this example we introduce a new way of defining the three counters.

(2) A new definition appears in working storage, namely,

```
01  count-table.
    02 counts pic 9(5) occurs 3 times.
```

We have introduced the **occurs** clause on the 02-level table entry item. This means that the data-name "counts" is defined 3 times, each time with the same **picture** clause of 9(5). The 3 entries in the table are referred to as follows:

```
counts(1)
counts(2)
counts(3)
```

The integers contained in parentheses are known as *subscripts*.

(3) The data-name "counts" must be associated with one of the valid subscripts to be meaningful. The value of the subscript must be in the range 1-3 i.e. by using the clause

```
occurs 3 times
```

we promise that we will not use a subscript greater than 3. COBOL does not permit a subscript of zero or any negative value.

(4) The data-name "count-table" can be used to refer to all the items in the table. Thus the statement

```
move zeros to count-table
```

sets each item in the table to zero.

(5) The paragraph

```
process-student-marks.
    move school to i.
    add 1 to counts(i).
```

causes 1 to be added to the appropriate table item. The data-name "i" is defined as a numeric item and the value of "school" is moved to "i". Recall that the school field of the student record contains either a 1, 2, or 3. Subscripts must be defined as numeric fields and must contain numeric values.

(6) We could have defined "school" with a **picture** of '9' instead of 'x' and then the paragraph could have been written as

```
process-student-marks.
    add 1 to counts(school).
```

(7) In order to display the calculated values, "i" is initialized to 1 and the "display-totals" paragraph is executed 3 times; each time the appropriate line is displayed and "i" is incremented by 1.

(8) The power of the use of subscripts should be self-evident. For example if the number of schools were increased from 3 to 9, the "process-student-marks" paragraph requires no modification. Of course, the **occurs** clause would have to be changed to define nine items. We would also have to modify the part of the program that displays the results.

(9) The user may have been tempted to use the **value is** clause to set the initial table values to zero. This is not possible as COBOL does *not* permit one to use the **value is** clause to initialize subscripted data-names.

1.8.2 Perform Varying.

When using subscripted data-names, we quite often wish to execute a paragraph repetitively; the only difference is that we wish to change the subscript's value. This was the case in the previous example when we were displaying the results. The **perform** verb with the **varying** option gives us this capability.

```
*
* School Algebra Averages using Subscripted Data-names
* and the Perform Varying.
*
 identification division.
 program-id. EXAMPLE-36.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).

 working-storage section.

 01   i         pic 9.
```

```
01  totals-table.
    02 total          pic 9(5) occurs 3 times.
    02 counts         pic 9(5) occurs 3 times.

01  school-table.
    02 school-data        pic x(10) occurs 3 times.

01  student-data.
    02  student-no            pic xxxx.
    02 name                   pic x(20).
    02  age                   pic xx.
    02  sex                   pic x.
    02  class                 pic x.
    02  school                pic x.
    02  algebra               pic 999.
    02  geometry              pic 999.
    02  physics               pic 999.
    02  chemistry             pic 999.
    02  english               pic 999.

01  report-heading.
    02 filler         pic x(20) value is spaces.
    02 filler         pic x(30) value is 'Algebra Averages'.
    02 filler         pic x(30) value is spaces.

01  school-line.
    02 filler              pic x(13) value is ' Average for '.
    02  out-school         pic x(10).
    02 filler              pic x(4) value is ' is '.
    02  out-average        pic z(5).9.

01  blank-line             pic x(80) value is spaces.
```

```
procedure division.
    open input student-file.
    open output screen.
    write display-record from report-heading.
    write display-record from blank-line.
    move 'Central' to school-data(1).
    move 'Western' to school-data(2).
    move 'Southern' to school-data(3).
    perform read-student-record.
    move zeros to totals-table.
    perform process-student-data
        until student-no = high-values.
    perform display-averages
        varying i from 1 by 1
        until i > 3.
    close student-file
          screen.
    stop run.

process-student-data.
    perform process-student-marks.
    perform read-student-record.

process-student-marks.
    move school to i.
    add algebra to total(i).
    add 1 to counts(i).

display-averages.
    move school-data(i) to out-school.
    divide total(i) by counts(i) giving out-average.
    write display-record from school-line.

read-student-record.
    read student-file into student-data
        at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                    Algebra Averages

 Average for Central        is   63.2
 Average for Western        is   62.8
 Average for Southern       is   71.1
...Execution ends.
```

Notes

(1) This example calculates average algebra mark for each of the three schools. It also displays a school name instead of a school number.

(2) Another table is introduced to hold the sum of the grades for each school. While it could have been defined as a separate table, it has been defined as part of the totals-table. This permits us to zero both tables with one move sentence.

(3) The program reads each record and accumulates the algebra mark for the appropriate school as well as incrementing the appropriate counter.

(4) The "display-averages" paragraph calculates the average for each school and displays the desired values.

(5) The sentence

```
perform display-averages
    varying i from 1 by 1
    until i > 3.
```

causes the "display-averages" paragraph to be executed repeatedly as long as i is not greater than 3. Before the paragraph is executed, the value of "i" is set to 1 and the condition is evaluated to determine if it is true or false. If it is true the paragraph is executed; if false control passes to the next sentence. Each successive time, "i" is incremented by 1 and the condition is tested to determine if the paragraph should be executed.

(6) We could have displayed the results in the reverse order by changing the **perform** as follows:

```
perform display-averages
    varying i from 3 by -1
    until i = 0.
```

In this case, "i" will start with the value 3, and each time will be decremented by 1 until "i" is zero.

(7) A complete description of the **perform** verb can be found in the reference manual.

(8) Another table has been defined to contain the names of the three schools. This table is initialized with three **move**s of 'Central', 'Western', and 'Southern' respectively. These names are **move**d to the display line in the "display-averages" paragraph. The next example will show how to define these initial values in working storage.

(9) Note that tables can be defined using x's as well as 9's in the **picture** clauses.

1.8.3 The Redefines Clause with Subscripted Data-names.

While COBOL does not permit the initialization of tables directly using the **value is** clause, an indirect method is available.

```
*
* Redefines Clause.
*
 identification division.
 program-id. EXAMPLE-37.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).

 working-storage section.

 01   i          pic 9.

 01   totals-table.
      02 total          pic 9(5) occurs 3 times.
      02 counts         pic 9(5) occurs 3 times.
```

```
01  school-table-data.
    02 filler          pic x(10) value is 'Central'.
    02 filler          pic x(10) value is 'Western'.
    02 filler          pic x(10) value is 'Southern'.

01  school-table redefines school-table-data.
    02 school-data          pic x(10) occurs 3 times.

01  student-data.
    02  student-no              pic xxxx.
    02 name                     pic x(20).
    02  age                     pic xx.
    02  sex                     pic x.
    02  class                   pic x.
    02 school                   pic x.
    02  algebra                 pic 999.
    02  geometry                pic 999.
    02 physics                  pic 999.
    02 chemistry                pic 999.
    02 english                  pic 999.

01  report-heading.
    02 filler              pic x(20) value is spaces.
    02 filler              pic x(30) value is 'Algebra Averages'.
    02 filler              pic x(30) value is spaces.

01  school-line.
    02 filler                   pic x(13) value is ' Average for '.
    02  out-school             pic x(10).
    02 filler                   pic x(4) value is ' is '.
    02  out-average            pic z(5).9.

01  blank-line                 pic x(80) value is spaces.
```

```
procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     move zeros to totals-table.
     perform process-student-data
          until student-no = high-values.
     perform display-averages
          varying i from 1 by 1
          until i > 3.
     close student-file
           screen.
     stop run.

process-student-data.
     perform process-student-marks.
     perform read-student-record.

process-student-marks.
     move school to i.
     add algebra to total(i).
     add 1 to counts(i).

display-averages.
     move school-data(i) to out-school.
     divide total(i) by counts(i) giving out-average.
     write display-record from school-line.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
                    Algebra Averages

  Average for Central      is   63.2
  Average for Western      is   62.8
  Average for Southern     is   71.1
...Execution ends.
```

Notes

(1) This example also calculates and displays school algebra averages.

(2) An area called "school-table-data" is initialized to contain the three names of the schools.

(3) Immediately following the definition of the "school-table-data", we have inserted the following description of "school-table".

```
01 school-table redefines school-table-data.
    02 school-data pic x(10) occurs 3 times.
```

This differs from the previous example in that the **redefines** clause is introduced. This means that the item "school-table" occupies the *same* area in working storage as the item "school-table-data".

(4) Both areas are defined to have 30 characters; COBOL does not require that they same length if both are 01-level items.

(5) In essence, the **redefines** permits the user to give two or more definitions to the same area in working storage and then to refer to the area with the appropriate data-name.

(6) There are many other applications of this feature, but the assignment of initial values in a table of subscripted data-names is an important one.

1.8.4 Tables with Two Subscripts.

Consider the problem of finding the number of students in each of the four classes at each of the three schools. In this case it is more convenient to set up a table containing three rows representing the schools and four columns representing the classes. This example introduces tables with two subscripts.

```
*
* School Course Averages
* Data-names with Two Subscripts.
*
 identification division.
 program-id. EXAMPLE-38.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select screen
           assign to 'terminal'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 fd   screen
      label records are standard.
 01   display-record.
      02 filler          pic x(80).

 working-storage section.

 01   i         pic 9.
 01   j         pic 9.
```

```
01  school-table-data.
    02 filler          pic x(10) value is 'Central'.
    02 filler          pic x(10) value is 'Western'.
    02 filler          pic x(10) value is 'Southern'.

01  school-table redefines school-table-data.
    02 school-data     pic x(10) occurs 3 times.

01  summary-table.
    02 count-by-school occurs 3 times.
       03count-by-class          pic 9(5) occurs 4 times.

01  student-data.
    02 student-no              pic xxxx.
    02 name                    pic x(20).
    02 age                     pic xx.
    02 sex                     pic x.
    02 class                   pic x.
    02 school                  pic x.
    02 algebra                 pic 999.
    02 geometry                pic 999.
    02 physics                 pic 999.
    02 chemistry               pic 999.
    02 english                 pic 999.

01  report-heading.
    02 filler          pic x(20) value is spaces.
    02 filler          pic x(30) value is 'Enrollment Table'.
    02 filler          pic x(30) value is spaces.

01  school-line.
    02 out-school              pic x(10).
    02 out-counts              pic z(4)9 occurs 4 times.

01  blank-line                 pic x(80) value is spaces.
```

```
procedure division.
     open input student-file.
     open output screen.
     write display-record from report-heading.
     write display-record from blank-line.
     perform read-student-record.
     move zeros to summary-table.
     perform process-student-data
          until student-no = high-values.
     perform display-averages
          varying i from 1 by 1
          until i > 3.
     close student-file
           screen.
     stop run.

process-student-data.
     perform process-student-marks.
     perform read-student-record.

process-student-marks.
     move school to i.
     move class to j.
     add 1 to count-by-class(i, j).

display-averages.
     move school-data(i) to out-school.
     perform move-counts
          varying j from 1 by 1
          until j > 4.
     write display-record from school-line.

move-counts.
     move count-by-class(i, j) to out-counts(j).

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

run
Execution begins...

Enrollment Table

Central	1	2	2	0
Western	0	2	1	3
Southern	2	1	3	1

...Execution ends.

Notes

(1) A more involved definition is required to set up a table with two subscripts, namely

```
01 summary-table.
   02 count-by-school occurs 3 times.
      03 count-by-class pic 9(5) occurs 4 times.
```

(2) Note that "count-by-school" is defined to occur 3 times because there are 3 schools. These three entries are referred to as:

```
count-by school(1)
count-by-school(2)
count-by-school(3)
```

(3) For each class, we have "count-by-class" which occurs 4 times, once for each class. A problem arises when we wish to reference the data-names "count-by-class". If we consider the example

```
count-by-class(3)
```

we have a vague reference; we know it means the third entry, but for which class? The problem is solved by introducing a *second subscript as follows*

```
count-by-class(3, 2)
```

The first subscript refers to school 3, and the second one refers to class 2.

(4) Every reference to the data-name "count-by-class" *must* have two subscripts and is referred to as a *doubly-subscripted* data-name. "Summary-table" is often referred to as a *two-dimensional table* or an *array*.

(5) The above definition can be thought to have created 12 items each with a picture 9(5). The twelve items can be zeroed with the sentence

```
move zeros to summary-table.
```

(6) The paragraph "process-student-marks" assigns the school to "i" and the class to "j". Then "i" and "j" are used as row and column subscripts respectively and one is added to the correct table entry.

(7) After all the records are processed, the table is displayed. The area "school-line" was modified to contain a table to hold the four class counts. The values from a row are obtained from the "summary-table and then the record is displayed. This action is repeated for each row.

(8) COBOL allows up to three subscripts to be used.

1.9 Relative Files

1.9.1 Create a Relative File.

Suppose that a teacher wished to display the record for a particular student from the student file in order to look at the student's marks. It would not be difficult to write a program to accomplish this. However, consider now that the teacher wished to examine a second student's record. If this record occurred after the one just examined, there would be no major problem to retrieve it and display it. However, if it occurred before, we would have to go to some extra effort to retrieve the new record. The difficulty lies in the fact that the student file is a sequential file. *Relative files* permit us to access any record as easily as any other record. This section introduces relative files.

```
*
* Create a Relative File.
*
 identification division.
 program-id. EXAMPLE-39.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.
      select student-inquiry
           assign to 'dirfile,rel'
           organization is relative
           access is sequential.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).
```

```
fd   student-inquiry
     label records are standard.
01   direct-record.
     02 filler          pic x(60).

working-storage section.

01   student-data.
     02 student-no      pic xxxx.
     02 name            pic x(20).
     02 age             pic xx.
     02 sex             pic x.
     02 class           pic x.
     02 school          pic x.
     02 algebra         pic 999.
     02 geometry        pic 999.
     02 physics         pic 999.
     02 chemistry       pic 999.
     02 english         pic 999.

procedure division.
     open input student-file
          output student-inquiry.
     perform read-student-record.
     perform process-student-data
          until student-no = high-values.
     close student-file
           student-inquiry.
     stop run.

process-student-data.
     write direct-record from student-data.
     perform read-student-record.

read-student-record.
     read student-file into student-data
          at end move high-values to student-no.
```

Sample Program Execution

```
run
Execution begins...
...Execution ends.
```

Notes

(1) This program reads the student file and creates a new version of the file as a *relative file*.

(2) A new file called "student-inquiry" is defined as a relative file. Its system-name is "dirfile,rel". The *rel* is required by the CBM-SuperPET to indicate that the file is relative. The chapter containing system dependent information should be consulted regarding the format of file-names on other systems.

(3) Two extra clauses

```
organization is relative
access is sequential
```

are added to the **select** statement. The first indicates that the file will be used in the future as a relative file. The second indicates that for this program we will be writing the records in a sequential fashion. This may seem somewhat confusing but we wish to read the records from the sequential file and write them in the same order, sequentially, as a new relative file. The next example will use this new file as a relative file.

(4) The **fd** for the new file is the same as the **fd** for the student file, except it has a new file and record name.

(5) Opening and closing the relative file is done in the same manner as opening and closing sequential files.

(6) Writing records to a relative file is also done in the same manner as writing records to a sequential file.

1.9.2 Read a Relative File.

Having written a relative file, we now wish to assure ourselves that the file exists and further that we can use it in a 'relative' fashion.

```
*
* Read a Relative File.
*
 identification division.
 program-id. EXAMPLE-40.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-inquiry
           assign to 'dirfile,rel'
           organization is relative
           access is random
           relative key is student-key.

 data division.

 file section.

 fd   student-inquiry
      label records are standard.
 01   direct-record.
      02 filler          pic x(60).

 working-storage section.

 01   student-key        pic 999.
```

```
01   student-data.
     02 student-no          pic xxxx.
     02 name                pic x(20).
     02 age                 pic xx.
     02 sex                 pic x.
     02 class               pic x.
     02 school              pic x.
     02 algebra             pic 999.
     02 geometry            pic 999.
     02 physics             pic 999.
     02 chemistry           pic 999.
     02 english             pic 999.

procedure division.
     open input student-inquiry.
     move 18 to student-key.
     perform read-student-record.
     perform process-student-data
          until student-key = 999.
     close student-inquiry.
     stop run.

process-student-data.
     display student-data.
     subtract 1 from student-key.
     perform read-student-record.

read-student-record.
     read student-inquiry into student-data
          invalid key move 999 to student-key.
```

Sample Program Execution

```
run
Execution begins...
1393Hopper          BU  15f23045069037026035
1390Cruikshank      TR  15f33055064077076085
1388Beatle          RA  15f11065062073076087
1375Quantas         FL  15m220660660660660 66
1272Zimmerman       AB  13f32095085078061057
1270Xerxes          X   13f13099088077066055
1268Allen           TT  13f21098084073065059
1266Taylor          YO  13f33095083072066055
1256Kitchen         MP  14m43074049100097036
1249Sullivan        J   15f42044055066077088
1245Cowan           DD  15f33055066077088099
1243Dirksen         PH  14m42074085054068084
1242Welch           JW  14m31075075076075075
1240Graham          JW  14m21000068075067087
1239Harrison        K   14m42022087065087068
1238Winterbourne    MS  14m31078088056067088
1236Jones           TO  14m22076078055057078
1234Smith           SA  14m13075100075065084
...Execution ends.
```

Notes

(1) This example reads and displays the 18 records of the relative file in reverse order i.e. the last one first, and the first one last.

(2) The **select** clause is modified to indicate that

```
access is random
```

instead of sequential i.e. we can now access the records in any order.

(3) Another clause

```
relative key is student-key
```

is added to the **select** clause. "Student-key" is a data-name which will be used to indicate which record we wish to read. Note that is defined in working storage and must be defined using a **picture** with 9's. The value of "student-key" must be an integer value lying in the range 1 to the 'number' of records in the relative file, in our case 18.

(4) This value is used to read the desired record in the file. If the value of "student-key" were 7, then the seventh record would be read.

(5) The **read** sentence has also been changed to reflect that we are reading a relative file.

```
read student-inquiry into student-data
    invalid key move 999 to student-key.
```

The **at end** clause is not used when reading a relative file. The clause

```
invalid key move 999 to student-key
```

is executed only if an invalid key is used, in our case outside the 1-18 range.

(6) When the program is executed, the value of 18 is assigned to "student-key" which will result in the last record to being read. Then "student-key" is decremented by one and the second-last record is read. This process continues until "student-key" is one and the first record is read. "Student-key is again decremented and now has a value of zero. When we attempt to read the zero'th record the **invalid key** clause is executed and 999 is assigned to "student-key" which in turn causes the "process-student-data" paragraph to terminate.

(7) Earlier we suggested that one should use **high-values** instead of a constant such as 999 to signal that there were no more records or in this case that an **invalid key** was encountered. **High-values** can only be assigned to a field whose **picture** clause contains x's; "student-key" must be defined using 9's. Hence, we use 999 to signal that the use of an invalid key. It is suggested that the "student-key" field be made at least one position larger than the number of records in the file.

1.9.3 Create a Relative File with an Index.

In order to use a relative file, we have to know the position of the record in the file. It would seem more appropriate to use the student number to identify records. In order to do this we must create an *index* to the file.

```
*
* Read a Relative File.
*
 identification division.
 program-id. EXAMPLE-41.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-inquiry
           assign to 'dirfile,rel'
           organization is relative
           access is random
           relative key is student-key.
      select index-file
           assign to 'indfile'.

 data division.

 file section.

 fd   student-inquiry
      label records are standard.
 01   direct-record.
      02 filler          pic x(60).

 fd   index-file
      label records are standard.
 01   index-record.
      02 filler          pic xxxx.

 working-storage section.

 01   student-key        pic 999.
```

```
01   student-data.
       02 student-no          pic xxxx.
       02 name                pic x(20).
       02 age                 pic xx.
       02 sex                 pic x.
       02 class               pic x.
       02 school              pic x.
       02 algebra             pic 999.
       02 geometry            pic 999.
       02 physics             pic 999.
       02 chemistry           pic 999.
       02 english             pic 999.

procedure division.
     open input student-inquiry
          output index-file.
     move 1 to student-key.
     perform read-student-record.
     perform create-student-index
          until student-key = 999.
     close student-inquiry
           index-file.
     stop run.

create-student-index.
     write index-record from student-no.
     add 1 to student-key.
     perform read-student-record.

read-student-record.
     read student-inquiry into student-data
          invalid key move 999 to student-key.
```

Sample Program Execution

```
run
Execution begins...
...Execution ends.
```

Notes

(1) This example reads the "student-inquiry" file and creates a new file called "index-file". This new file will also contain 18 records with each record containing only the student number.

(2) The program causes each record starting with the first to be read, and a new record containing the student number is written.

1.9.4 Extract Records from a Relative File.

Having created an *index*, we now wish to use it retrieve records randomly from the student file using the student number as the key.

```
*
* Read the Index and Selective Records.
*
 identification division.
 program-id. EXAMPLE-42.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select index-file
           assign to 'indfile'.
      select student-inquiry
           assign to 'dirfile,rel'
           organization is relative
           access is random
           relative key is student-key.

 data division.

 file section.

 fd   student-inquiry
      label records are standard.
 01   direct-record.
      02 filler          pic x(60).

 fd   index-file
      label records are standard.
 01   index-record.
      02 filler          pic xxxx.
```

```
working-storage section.

01   student-key              pic 999.
01   student-found            pic xxx.
01   student-id               pic xxxx.
01   index-exists             pic xxx.
01   number-of-keys           pic 99.
01   i                        pic 999.

01   student-index.
     02 index-key        pic xxxx occurs 18 times.

01   student-data.
     02  student-no          pic xxxx.
     02 name                pic x(20).
     02 age                 pic xx.
     02 sex                 pic x.
     02 class               pic x.
     02 school              pic x.
     02 algebra             pic 999.
     02  geometry           pic 999.
     02 physics             pic 999.
     02 chemistry           pic 999.
     02 english             pic 999.

procedure division.
     perform read-index.
     if index-exists = 'yes'
          perform display-student-records.
     stop run.

display-student-records.
     open input student-inquiry.
     perform get-student-id.
     perform process-student-records
          until student-id = 'stop'.
     close student-inquiry.
```

```
process-student-records.
     perform find-student-key.
     if student-found = 'yes'
          read student-inquiry into student-data
          display student-data
     else
          display 'invalid student id. ' student-id.
     perform get-student-id.

get-student-id.
     display 'enter student id - stop to stop'.
     accept student-id.

find-student-key.
     move 'no' to student-found.
     perform find-key
          varying i from 1 by 1
          until i > number-of-keys or student-found = 'yes'.

find-key.
     if student-id =index-key(i)
          move i to student-key
          move 'yes' to student-found.

read-index.
     open input index-file.
     move 'no' to index-exists.
     move 0 to i.
     perform read-index-record.
     perform store-and-read-index
          until student-id = high-values.
     if i > 0
          move i to number-of-keys
          move 'yes' to index-exists.
     close index-file.

store-and-read-index.
     add 1 to i.
     move student-id to index-key(i)
     perform read-index-record.
```

```
    read-index-record.
        read index-file into student-id
            at end move high-values to student-id.
```

Sample Program Execution

```
run
Execution begins...
enter student id - stop to stop
* 1234
1234Smith          SA 14m130751000750650B4
enter student id - stop to stop
1393
1393Hopper         BU 15f230450690370260B5
enter student id - stop to stop
5427
invalid student id. 5427
enter student id - stop to stop
stop
...Execution ends.
```

Notes

(1) This example asks the user to enter a student number. Then the record for that student is displayed. The program terminates when a value of 'stop' is entered. If a student number which is not in the file is entered, a message indicating this is displayed and the program requests another student number to be entered.

(2) The program initially reads the "index-file" and creates a table with one entry for each record in the file. If no records exist in the "index-file", the program terminates.

(3) The user is then requested to enter a student number and this number is successively compared to each value in the table. When an equal comparison is found, the position of the item in the table is used as the position of the record in the relative file. If the number is not found in the table, a signal is set and a message will be displayed.

(4) The method of searching the table is called a *sequential search*. If the table were quite large, it would probably be better to use some other searching method or algorithm. These methods are described in many computer science textbooks.

(5) If the number of records were quite large, there might not be enough memory to hold the entire index table. In this case, one would have to set up a scheme to possibly have an index to the index.

(6) It would be possible in this example to re-write the student record if, for example, we wished to change a mark or even add a field with a new mark. Thus we could read the record, make any changes, and then re-write the record in the same place using the same value of "student-key"; the value is not changed by the read. In this case, the file should be opened using the **I-O** option instead of the **input** option.

(7) It is also possible to add new records to the file. However, the method to do this is 'system dependent'. The user is referred to the reference manual to determine how this can be done.

1.10 Miscellaneous.

1.10.1 Create the Student File.

This program creates the student file used in many of the examples in the tutorial section of this text.

```
*
* Create Demonstration File.
*
 identification division.
 program-id. EXAMPLE-43.
 environment division.
 configuration section.
 source-computer. CBM-SuperPET.
 object-computer. CBM-SuperPET.

 input-output section.
 file-control.
      select student-file
           assign to 'textfile'.

 data division.

 file section.
 fd   student-file
      label records are standard.
 01   student-record.
      02 filler          pic x(60).

 working-storage section.

 01   rec-number              pic 999.

 01   student-data.
      02 filler          pic x(60).
```

```
01  demo-file.
    02  demo-data.
        03 filler  pic x(60) value is
           '1234Smith                    SA 14m13075100075065084'.
        03 filler  pic x(60) value is
           '1236Jones                    TO 14m220760780550570 78'.
        03 filler  pic x(60) value is
           '1238Winterbourne             MS 14m310780880560670 88'.
        03 filler  pic x(60) value is
           '1239Harrison                 K  14m420220870650870 68'.
        03 filler  pic x(60) value is
           '1240Graham                   JW 14m210000680750670 87'.
        03 filler  pic x(60) value is
           '1242Welch                    JW 14m310750750760750 75'.
        03 filler  pic x(60) value is
           '1243Dirksen                  PH 14m420740850540680 84'.
        03 filler  pic x(60) value is
           '1245Cowan                    DD 15f330550660770880 99'.
        03 filler  pic x(60) value is
           '1249Sullivan                 J  15f420440550660770 88'.
        03 filler  pic x(60) value is
           '1256Kitchen                  MP 14m430740491000970 36'.
        03 filler  pic x(60) value is
           '1266Taylor                   YO 13f330950830720660 55'.
        03 filler  pic x(60) value is
           '1268Allen                    TT 13f210980840730650 59'.
        03 filler  pic x(60) value is
           '1270Xerxes                   X  13f130990880770660 55'.
        03 filler  pic x(60) value is
           '1272Zimmerman                AB 13f320950850780610 57'.
        03 filler  pic x(60) value is
           '1375Quantas                  FL 15m220660660660660 66'.
        03 filler  pic x(60) value is
           '1388Beatle                   RA 15f110650620730760 87'.
        03 filler  pic x(60) value is
           '1390Cruikshank               TR 15f330550640770760 85'.
        03 filler  pic x(60) value is
           '1393Hopper                   BU 15f230450690370260 35'.
    02  demo-info redefines demo-data.
        03demo-rec          pic x(60) occurs 18 times.
```

```
    procedure division.
        open output student-file.
        perform write-demo
            varying rec-number from 1 by 1
            until rec-number > 18.
        close student-file.
        stop run.

    write-demo.
        move demo-rec(rec-number) to student-data.
        write student-record from student-data.
```

Sample Program Execution

```
run
Execution begins...
...Execution ends.
```