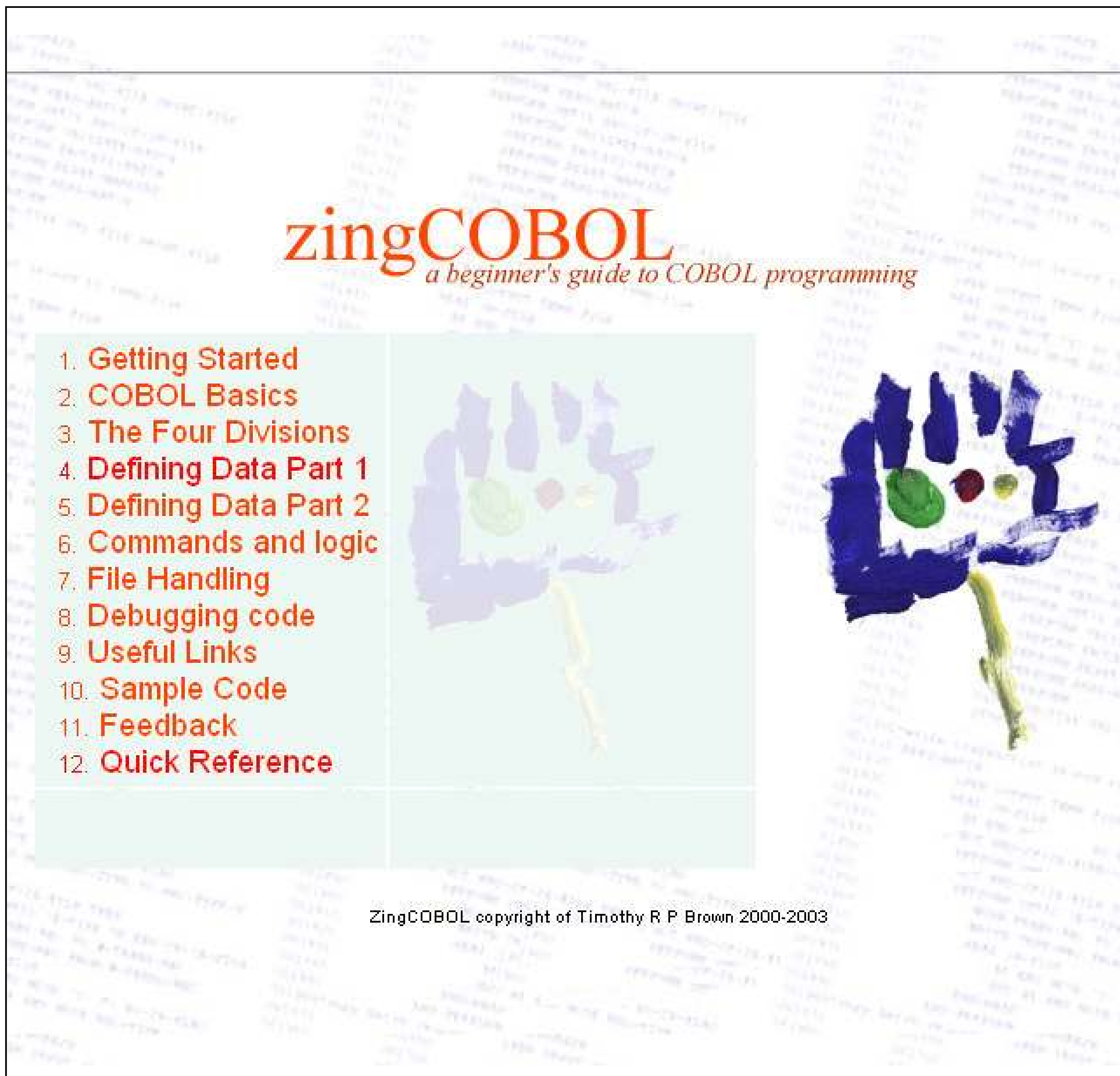
ZingCOBOL

A beginner's guide to COBOL programming



By Timothy R P Brown

Reproduced from zingcobol.tripod.com

1. Getting Started

1.1 Introduction

The aim of the ZingCOBOL is to give the basics of the COBOL programming language for anyone who knows a little bit about computers (not much) and preferably will at least have come across another procedural progamming language such as C, BASIC or Pascal. If you want to learn good structured programming then, although the basic COBOL syntax is provided here, other sources can provide more effective guidance (for example, see The Kasten COBOL Page).

The floating SITE MENU button can be clicked to bring up a temporary menu for navigating this site. If your browser doesn't support this feature (or the popup window that results) there is a table of contents at the bottom of every page to navigate with.

If you wish to find a specific item the Quick Reference page should take you to the desired section. This tutorial is by no means extensive but the basics should be covered here.

What's written here will hopefully be correct (tell me otherwise) and maybe even informative. However, I would strongly recommend buying a good book on COBOL programming, and/or have someone teach you it.

If you have any queries, comments, or suggestions you can either go to the zingCOBOL Forum (all levels of ability are welcome), use the Feedback form and/or sign the Guestbook.

I hope ZingCOBOL will prove useful to you.

1.2 COBOL - a brief overview

COBOL (*CO*mmon *B*usiness *O*rientated *L*anguage) has been around for yonks (since 1959), updated in 1968, 1977 and 1985. OO COBOL was developed in the 1990's. Well suited to business applications, i.e. used for large batch processes running on mini-computer and mainframes (medium to large platforms). About 65% of new critical applications use COBOL; several billion lines of COBOL code exist throughout the world, used by over a million companies. So it may be old but it remains one of the most important languages in commercial use today. (source: Computer Weekly, Dec 9th, 1999).

1.3 What you'll need

The best way to learn to programme/learn a new language is to actually be able to write code and run it on a computer. Consequently, you really need a computer (probably a PC), a text editor (Notepad or WordPad will do) to write the code into, and most importantly, a COBOL compiler which will check your code and then convert it into something the computer can understand and execute. I use the Fujitsu COBOL85 ver3.0 compiler which can be downloaded for free (see the Links page).

If you want to download a simple program for added/refreshing line numbers, go to the Links page.

A brief description of how to compile a program using Fujitsu COBOL85 version 3.0 can be read here

ZingCOBOLCopyright Timothy R P Brown 2003

2. COBOL Basics

2.1 Coding Areas
2.2 Syntax
2.3 The Hello World Program

2.1 Coding Areas

If you look at the COBOL coding in later sections (e.g. League Table program in the Sample code section) the specific positions of coding elements are important for the compiler to understand. Essentially, the first 6 spaces are ignored by the compiler and are usually used by the programmer for line numbers. These numbers are *not* the same as those in BASIC where the line number is used as part of the logic (e.g. GOTO 280, sending the logic to line 280).

The seventh position is called the **continuation area.** Only certain characters ever appear here, these being:

`*` (asterisk), `/` (solidus or forward slash), or `-` (hyphen).

The asterisk is used to precede a comment, i.e. all that follows is ignored by the compiler. The solidus is used to indicate a page break when printing coding from the compiler, but it too can be used as comment since the rest of the line is ignored by the compiler. The hyphen is used as a continuation marker, i.e. when a quoted literal needs to be extended over to the next line. It is *not* for continuing a statement onto the next line (this is unnecessary*) and also cannot be used to continue a COBOL word. (*You can write any COBOL statement over as many lines as you like, so long as you stay in the correct coding region and don't split strings.)

```
000200*Here is a comment.
000210/A new line for printing and a comment.
000220
  :
000340       DISPLAY 'This might be a very long string that
000350-     'needs to be continued onto the next line'
```

Positions 8 to 11 and 12 to 72 are called **area A** and **area B**, respectively. These are used in specific instances that will be detailed in later sections.

Summary	
Positions	Designation
1 to 6	line code
7	continuation area
8 to 11	area A
12 to 72	area B

2.2 Syntax

Identifier names

User-defined names must conform to the following rules:

- Must only consist of alphabetic and numeric characters and/or hyphens
- The name must contain at least one alphabetic character
- Must be no more than 30 characters
- When using hyphens, they must not appear at the beginning or end of the name

Some examples of legal names:

```
A123
RecordCount-1
WAGE-IN
Tot-2-out
```

Like all COBOL code, the compiler will not distinguish between upper and lower case letters (except within quotes).

Lastly, COBOL has a large list of **reserved words** that cannot be used as identifier names. A list of COBOL reserved words is given elsewhere.

Punctuation

The full stop (period) is the most important punctuation mark used, and its use will be detailed later (see Scope terminators). Generally, every line of the IDENTIFICATION, ENVIRONMENT, and DATA DIVISION end in a period.

Quotation marks, either single or double, are used to surround quoted literals (and when calling a sub-program). However, donât mix them when surrounding the literal, e.g.

" This is bad **â**

" but this is ok **"**

Commas and semi-colons are also used to separate lists of identifiers, e.g.
MOVE 2 TO DATA-ITEM-1, DATA-ITEM-2, DATA-ITEM-3
A space must follow the comma/semi-colon. They are optional however, and a space would suffice, but it does add to clarity.

Spelling

Since COBOL was developed in the USA, the spelling of words is American, e.g. INITIALIZE or ORGANIZATION (using Z rather than S). Brits be warned!

In many cases, abbreviations and alternative spellings are available (see reserved word list), e.g. ZERO ZEROS ZEROES all mean the same thing. Likewise, LINE and LINES, PICTURE and PIC, THROUGH and THRU.

2.3 The 'Hello World' Program

As is traditional for all introductory lessons for a programming language, here's a 'Hello World' program:

```
000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID.   HELLO-WORLD-PROG.
```

```
000030 AUTHOR.       TIMOTHY R P BROWN.
000040*The standard Hello world program
000050
000060 ENVIRONMENT DIVISION.
000070
000080 DATA DIVISION.
000090 WORKING-STORAGE SECTION.
000100 01 TEXT-OUT    PIC X(12) VALUE 'Hello World!'.
000110
000120 PROCEDURE DIVISION.
000130 MAIN-PARAGRAPH.
000140        DISPLAY TEXT-OUT
000150        STOP RUN.
```

ZingCOBOL Copyright Timothy R P Brown 2003

3. The Four Divisions

3.1 The IDENTIFICATION DIVISION
3.2 The ENVIRONMENT DIVISION
3.3 The DATA DIVISION
3.3 The PROCEDURE DIVISION

COBOL program code is divided into four basic division: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE divisions. The identification division is required, but in theory the others are not absolute (although you won't have much of a program without any procedures or data!).

The identification division tells the computer the name of the program and supplies other documentation concerning the program's author, when it was written, when it was compiled, who it is intended for...etc. In fact, only the program name is required by the compiler.

```
000100 INDENTIFICATION DIVISION.
000110 PROGRAM-ID.     EXAMPLE-1-PROG.
000120 AUTHOR.     ZINGMATTER.
000130 INSTALLATION.  XYZ GROUP.
000140 DATE-WRITTEN.  17/5/00.
000150 DATE-COMPILED.
000160 SECURITY.      LOCAL GROUP.
```

Note:

- The use of full stops is important. Throughout these pages, note where they are positioned
- The first words (PROGRAM-ID, AUTHOR etc..) are written in area A, the details are in area B
- The DATE-COMPILED detail is written automatically by the compiler

This division tells the computer what the program will be interacting with (i.e. its environment) such as printers, disk drives, other files etc... As such, there are two important sections: the CONFIGURATION SECTION (which defines the source and object computer) and the INPUT-OUTPUT SECTION (which defines printers, files that may by used and assigns identifier names to these external features).

```
000260 ENVIRONMENT DIVISION.
000270 CONFIGURATION SECTION.
000280 SOURCE-COMPUTER.    IBM PC.
000290 OBJECT-COMPUTER.    IBM PC.
000300 INPUT-OUTPUT SECTION.
000310 FILE-CONTROL.
000320    SELECT INPUT-FILE ASSIGN TO 'input.dat'
000330      ORGANIZATION IS LINE SEQUENTIAL.
000340    SELECT PRINT-FILE ASSIGN TO PRINTER.
```

Notes:

- You probably wouldn't need to bother with the configuration section (I think this is an oldie thing)
- The DIVISION and SECTION words are written into area A but the SELECT clause should be in area B.
- The full stop doesn't appear in the SELECT clause until after the ORGANIZATION has been specified.
- INPUT-FILE and PRINT-FILE are user-defined names that are used in the program to refer to 'input.dat' and the printer, respectively. If the input.dat file was on a different disk drive, within a directory structure, then you could write: ...ASSIGN TO 'D:datafiles/data/input.dat'.
- Line 000330 describes the structure or form of the data written in 'input.dat' file. In this case, each record is on a new line in the file (see File Handling section for details).
- The printer also is assigned but the organization doesn't have to be secified.
- For the SELECT clause, if no organization is defined the computer defaults to SEQUENTIAL organization (i.e. each record appears in a long string with no line breaks.

Things look clearer when you see a full program (see Sample Code section).

3.3 The DATA DIVISION

The data division is where memory space in the computer is allocated to data and identifiers that are to be used by the program. Two important sections of this division are the FILE SECTION and the WORKING-STORAGE SECTION. The file section is used to define the structure, size and type of the data that will be read from or written to a file.

Suppose the 'input.dat' file (described above) contains a series of records about a companies customers, giving details of name, address, and customer number. If you were to open 'input.dat' with a text editor you would see each record on a new line like this:

```
Joe Bloggs  20Shelly Road        Bigtown      023320
John Dow    15Keats Avenue       Nowheresville042101
Jock MacDoon05Elliot Drive       Midwich      100230
```

```
etc...
```

The different pieces of data need to be defined so that the program can read a record at a time, placing each piece of information into the right area of memory (which will be labelled by an identifier).

The file section for this may look like this:

```
000400 DATA DIVISION.
000410 FILE SECTION.
000420
000430 FD INPUT-FILE.
000440 01 CUSTOMER-DATA.
000450       03 NAME       PIC X(12).
000460       03 ADDRESS.
000470             05 HOUSE-NUMBER   PIC 99.
```

```
000480              05 STREET          PIC X(19).
000490              05 CITY            PIC X(13).
000500        03 CUST-NUMBER PIC 9(6).
```

Notes:

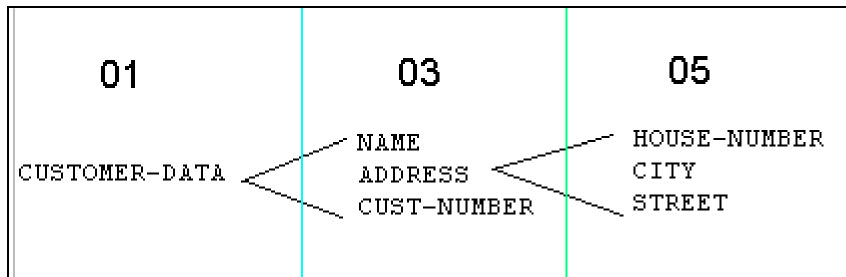
- 'FD' stands for File Descriptor, and names the file, INPUT-FILE (assigned in the environment division), and describes the exact structure of the data in each record. All records in this file MUST be of exactly the same structure.

- '01 CUSTOMER-DATA' is the group name and refers to all of the single record that is read into the computer memory from the file. The higher numbers (levels), 03.. and 05.. will contain the indivual *fields* of the record.

- Both FD and 01 are written in area A while higher levels are in area B.

- Level 01 is sub-grouped into level 03 fields. Notice that one of the level 03 sub-groups is itself sub-grouped into level 05. The sub-grouping could continue upwards as required to 07, 09 etc.. These numbers (except level 01) could as easily be 02, 03, 04 ...or any increasing number scale. There are some numbers (i.e. 66, 77 and 88) which actually have other uses but these will be discussed in the Defining Data section.

- The PIC (short for PICTURE) clause indicates the size and type of data that that field contains. For example, in line 000450, the data name (identifier) NAME has been defined as holding 12 characters of alphnumeric data. It could have been written as PIC XXXXXXXXXXXX be that's a pain. 'X' means alphnumeric and can contain any ASCII character. However, even if it contained '2' you could not do any calculations on this as the information is stored as the ASCII code for the character '2', rather than the actual number 2. Line 000470 defines HOUSE-NUMBER as PIC 9(2), which can hold a 2-digit number. You can do calculations with this since '9' is used to denote a numeric field.



- Notice how the group names (CUSTOMER-DATA and ADDRESS) do not have PIC descriptions. This is because the higher level field descriptions when added together will be the size of the group name, i.e. CUSTOMER-NUMBER will hold 46 characters which turns out to be the size of each record (spaces are included). You can refer to these group names but when doing so all data will be treated as alphanumeric and cannot be used for calculations, even if all of the higher group items are numeric.

The WORKING-STORAGE SECTION of the data division is for defining data that is to be stored in temporary memory, i.e. during program run-time. Effectively, this is where, for example, an identifier is defined that will hold the result of a calculation.

```
000500 DATA DIVISION.
000510 WORKING-STORAGE SECTION.
000520
000530 01 RECORD-COUNTER    PIC 9(5).
```

Also see the 'Hello World' program. In that case the string to be displayed on the screen is actually defined in working-storage using the VALUE clause (01 TEXT-OUT PIC X(12) VALUE 'Hello World!'). The same can be done for numeric data e.g.:

```
000800 01 TOTALS-IN.
000810       03 1ST-NO   PIC 99 VALUE ZERO.
000820       03 2ND-NO   PIC 999 VALUE 100.
```

The equivalent to filling an item such as 1ST-NO (above) with zeroes, is filling an alphanumeric (PIC X) item with spaces e.g. 01 MESSAGE PIC X(12) VALUE SPACES.

A further section of the data division is the LINKAGE SECTION, which is used to facilitate the sharing of data between programs (or sub-programs). See below.

3.4 The PROCEDURE DIVISION

The procedure division is where the logic of the program actually found. Here is where the various commands are written (see Commands and logic section).

COBOL is a modular language, in that a program is usually broken up into units described as paragraphs.

```
000900 PROCEDURE DIVISION.
000910 CONTROL-PARAGRAPH.
000920       PERFORM READ-DATA-FILE
000930       PERFORM CALULATE-PRICES
000940       PERFORM PRINT-PRICE-REPORT
000950       STOP RUN.
```

The PERFORM statement is used to 'call' other paragraphs to do each task. These paragraphs would appear in the same coding and are part of the same program. In the above example, the program would consist of four paragraphs: the CONTROL-PARAGRAPH and the three called from within it. All of the paragraph names are user-defined. Even if a program only has one paragraph, it must still have a name. The 'Hello World' program has a paragraph name MAIN-PARAGRAPH. Regarding punctuation, as a rule there should only be two full stops in any paragraph; one after the paragraph name and the other at the end of the paragraph.

Sub-programs

A program may also refer to a different program, called a sub-program. A sub-program is an entirely different program from the calling program, with its own divisions etc... with the exception that it does not end with STOP RUN (which would return you to the operating system), but with EXIT PROGRAM. The sub-program is a module, rather than a subroutine which is what a paragraph could be described as. The verb CALL is used to activate the sub-program:

```
000800 DATA DIVISION.
000810 WORKING-STORAGE SECTION.
000820 01 W-DATE-IN     PIC 9(6).
    :
000850 LINKAGE SECTION.
000860 01 L-DATE-IN.
000870       03  DAY     PIC 99.
000880       03  MONTH   PIC 99.
000890       03  YEAR    PIC 99.
    :
000900 PROCEDURE DIVISION.
000910 CONTROL-PARAGRAPH.
000920       PERFORM READ-FILE
000930       CALL "VALIDATE-DATE" USING L-DATE-IN
    :
001950       STOP RUN.
    :

003000 IDENTIFICATION DIVISION.
003010 PROGRAM-ID   VALIDATE-DATE.
003020 ...........
    :        ............etc.....
003500 PRODECURE DIVISION USING L-DATE-IN.
    :
004000 EXIT PROGRAM.
```

In the above code, a sub-program is called, named VALIDATE-DATE

In order to use data from the calling program in the sub-program the calling program uses a section in the data division called the LINKAGE SECTION. The item W-DATE-IN in the calling program occupies the same memory address as the sub-program's item L-DATE-IN, so the number placed in W-DATE-IN item using the VALUE clause is also in L-DATE-IN. Note: you cannot use VALUE in the linkage section.

The procedure division of the sub-program requiring the use of linkage section defined data must say so by: PROCEDURE DIVISION USING ...[linkage section items to be used] also refered to by the CALL ... USING. See lines 000930 and 3500 above.

In the above example, what is being called ("VALIDATE-DATE") is a literal. This means that you could use an identifier instead, allowing you a choice between sub-programs depending on what the literal had been previously defined as. For example, if a record was of type "A" then you may want to process that record using sub-program PROCESS-A-REC, but if a type "B" record the use PROCESS-B-REC.
The logic might be as follows:

```
     :
0003000       IF RECORD-TYPE = "A" THEN
0003010            MOVE "PROCESS-A-REC" TO SUB-PROG
0003020       ELSE MOVE "PROCESS-B-REC" TO SUB-PROG
0003030       CALL SUB-PROG USING L-REC-DATA
    :
```

Although I haven't described the various commands of the procedure division (see Commands and logic sections) the above code is fairly clear...if a marker called RECORD-TYPE has been set as "A" then place (i.e. MOVE) the string "PROCESS-A-REC" into the area of memory labelled as SUB-PROG (so now SUB-PROG *contains* this string). Otherwise (i.e. ELSE) it is assumed that the only other type there is can be "B" type and so "PROCESS-B-REC" is MOVEd into SUB-PROG. Depending on what the item SUB-PROG contains the desired sub-program will be called.

ZingCOBOL Copyright Timothy R P Brown 2003

4. Defining Data Part 1

4.1 Number Formats
4.2 Moving and Editing data
4.3 Initializing data

A large portion of any COBOL program consists of the data division and how the data is defined and manipulated. As already described in the previous section (The Four Divisions), each identifier used in the procedure division must be defined. How they are defined depends of what is to be performed on that data.

More on data definition for tables (arrays), Boolean data, and for writing printing data, is discussed in the following section (Defining Data Part 2).

The MOVE verb is used extensively in COBOL to manipulate data and so is introduced here. As the name suggests, MOVE simply tells the computer to place a certain item of data to a specified identifier. a typical statement would be of the form:

MOVE [identifier or literal] TO [identifier-1] [identifier-2]...

4.1 Number Formats

There are three types of number formats (that I'm aware of): DISPLAY, PACKED-DECIMAL (or COMPUTATIONAL-3 aka COMP-3), and BINARY. These are defined in the data division after the PIC clause (although DISPLAY format is default so you don't really have to define it). In DISPLAY format (aka character format) a single digit (i.e. PIC 9) would use 1 byte (8 bits) of memory. In order to save space and processing time, calculation can be performed in a more economical way using COMP-3 or BINARY formats. This is because they use less bytes:

```
e.g.

01 ITEM-1  PIC 9(3) VALUE 123 USAGE IS DISPLAY.

This uses 4 bytes of memory: one for each digit plus one for the sign (+ or -).

01 ITEM-1  PIC 9(3) VALUE 123 USAGE IS PACKED-DECIMAL.

This uses 2 bytes: each individual digit is converted to binary -
                          1    2    3    sign
                       0001 0010 0011    1111  (unsigned)
                                         1101  (negative)
                                         1100  (positve)
```

01 ITEM-1 PIC 9(3) VALUE 123 USAGE IS BINARY.

This uses 1 byte: 123 (one-hundred and twenty-three) is converted into binary: 01111011

These compressed formats can be used for calculations but not much use for displaying or printing the result. Hence, it is necessary to convert the result of such a calculation back into DISPLAY format:

```
    :
000100 01 NUMBER-1    PIC 9(3) USAGE IS BINARY.
```

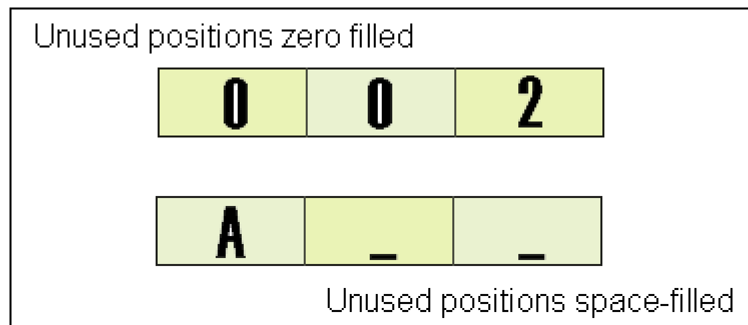
```
000110 01 NUMBER-2    PIC 9(3) USAGE IS BINARY.
000120 01 ANSWER-OUT  PIC 9(4) USAGE IS DISPLAY.
   :
000200       MULTIPLY NUMBER-1 BY NUMBER-2 GIVING ANSWER-OUT
000210       DISPLAY ANSWER-OUT
```

Both 'USAGE' and 'IS' are optional (as is 'DISPLAY').

4.2 Moving and Editing Data

Care must be taken when moving (using the MOVE verb) data to an item. In other languages, such as Pascal, when to assign a value to XYZ (defined at the beginning of the program as an integer for example), then that's it. In COBOL, you have to be sure that the value you are moving to XYZ item is not bigger than the defined size. If you moved 1234 to a PIC 999 defined item, then if you were to examine XYZ it would contain 234. For numeric data, the digits are truncated from the left. If item ABC were defined as PIC XXX and you moved "abcd" into it, on examination you would find it contained "abc". For alpha-numeric data characters are truncated from the right.

Conversely, moving data that is smaller than the PIC definition has certain effects. If 2 was moved to item XYZ above, then the number 2 is written to the right-most position and the leading positions are zero-filled (see figure below). Likewise, moving "A" to ABC above, the letter "A" would be written to the left-most position with the trialing positions being space-filled.



So what about data with decimal places?
To use decimal places, the letter 'V' is used in the PIC description:

01 COST-OF-ITEM PIC 9(4)**V**99.

Here the item COST-OF-ITEM can contain a number that has two decimal places. 'V' is called an *implied* decimal place in that the 'V' itself is not an area of memory, i.e. the above PIC description will hold 6 digits - 4 before the decimal point and two after. The computer will align the number being moved into the item around the decimal point. See the examples below:

```
number going into COST-OF-ITEM   contents of COST-OF-ITEM  [PIC 9(4)V99]

     1234.56                      1234.56
     1234                         1234.00  (zero-filled)
        1                         0001.00  (zero-filled)
        0.1                       0000.10  (zero-filled)
   654321.12                      4321.12  (digits are truncated)
   654321                         4321.00  (digits are truncated)
     1234.567                     1234.56  (figure is NOT rounded up to 1234.57)
```

If you were to display COST-OF-ITEM it would appear as 123456 since the decimal point is assumed, not actual. For the purposes of printing or displaying a number you would need to actually show where decimal point is. You may also wish to avoid having a long string of zeros in front of a number and have spaces used instead. So the number would first have to be moved into an outputing item...well have a look at the small program below (don't worry too much about any of the commands used in the procedure division, although they're pretty self explanatory):

```
000010 IDENTIFICATION DIVISION.
000020
000030 PROGRAM-ID.   DATA-MOVING-PROG.
000040 AUTHOR.       TRP BROWN.
000050
000070 DATA DIVISION.
000080
000090 WORKING-STORAGE SECTION.
000100 01 COST-OF-ITEM     PIC 9(4)V99.
000110 01 NUMBER-OF-ITEMS  PIC 99.
000130 01 TOTAL-COST       PIC 9(5)V99.
000140 01 TOTAL-COST-OUT   PIC Z(4)9.99.
000150
000160 PROCEDURE DIVISION.
000170 MAIN-PARAGRAPH.
000180     MOVE 4.56 TO COST-OF-ITEM
000190     MOVE 5 TO NUMBER-OF-ITEMS
000200     COMPUTE TOTAL-COST =
000205            COST-OF-ITEMS * NUMBER-OF-ITEMS
000210     MOVE TOTAL-COST TO TOTAL-COST-OUT
```

- the VDU will then diplay:
  £ 22.80 in total
- The program gives values to COST-OF-ITEM and NUMBER-OF-ITEMS (lines 180 & 190) and multiples them to give TOTAL-COST (line 200).
- This result of this calculation is then moved to TOTAL-COST-OUT (line 210).
- Line 140 has a PIC description 'ZZZZ9.99'. The Z means 'zero-supression' where spaces are added instead of zeros. This means that you would not be allowed to perform arithemic functions on TOTAL-COST-OUT since it now contains alphanumeric data (i.e. spaces). Also, an actual decimal point (i.e. a full stop) is used in place of the 'V'
- You could actually write in line 200: COMPUTE TOTAL-COST-OUT = COST-OF-ITEMS * NUMBER-OF-ITEMS. Here the result of the calculation is put straight into TOTAL-COST-OUT, but no further calculations could be performed on the result.

You'll notice that there is a gap (of 2 spaces) between the '£' sign and the actual number in the displayed output. To avoid this when using '£' or '$' signs (well COBOL is a business language), you can zero-suppress as follows:

```
000140 01 TOTAL-COST-OUT  PIC ££££9V99.
```

the VDU will then display:
£22.80 in total

If you want nothing in a field when printing a value that is zero then use **BLANK WHEN ZERO**:

```
000140 01 PRINT-VALUE  PIC Z(5)9V99 BLANK WHEN ZERO.
```

the VDU will then display: ...er nothing...

More Data Editing

Signed data needs to be able to indicate whether it is a positive or negative number. An item may have a definition:

01 DATA-VALUE-1 PIC S999.

'S' indicates that the data is signed and so allows for negative values to be stored. If data is being entered from a keyboard say, as -201, into DATA-ITEM-2, the computer needs to be told that the first character is a sign:

01 DATA-VALUE-2 PIC S999 SIGN LEADING SEPARATE.

This would be suitable for a item into which "-201" can be entered. Here 'SIGN LEADING SEPARATE' indicates that a + or - occurs immediately before the number (you can also have 'SIGN TRAILING SEPARATE'). The important feature is the 'S' prior to the 999 (irrespective of leading or trailing signs).

For output, the sign can be maniputed to show signs and zero-suppress using a 'floating sign'. Look at the following examples:

Sending field		**Receiving field**	
Description (PIC)	*Contents*	*Description (PIC)*	*Contents*
S999	-21	S999	-021
S999	-21	--99	-21
S999	+21	--99	21
S999	-21	++99	-21
S999	+21	++99	+21
S999 SIGN LEADING SEPARATE	-347	999+	347-
S999 SIGN TRAILING SEPARATE	347-	-999	-347

The last two examples in the table show how the sign can be moved to the other end of the number when SIGN LEADING/TRAILING SEPARATE is used.

Some characters can be inserted into numbers, these being SPACE, SOLIDUS, ZERO (using 'B' '/' and '0' respectively):

Sending field		**Receiving field**	
Description (PIC)	*Contents*	*Description (PIC)*	*Contents*
99999	12345	99B9B99	12 3 45
99999	12345	9909099	1203045
999999	170568	99/99/99	17/05/68

Adding a solidus can be useful for printing the date (which can be obtained directly from the computer in the form

of yymmdd [you have to switch the order around first]). I can only assume that adding zeros to a number is for fraudulent purposes.

Redefining Data

It is sometimes useful to be able to have data that can be defined as either numeric or alphanumeric. This is done by redefining the data. One way is *implicit redefinition*:

```
01 DATA-ITEM-X.
     03 DATA-ITEM-1  PIC 99.
```

Although DATA-ITEM-X and DATA-ITEM-1 refer to the same area of memory storage, the level 03 item is numeric. However, group items are always alphanumeric and as a result, if you moved the number 25 into DATA-ITEM-1, you could use DATA-ITEM-X as an alphanumeric item containing the literal "25".

Explicit redefinition uses the verb REDEFINES so that you could do this:

```
01 DATA-ITEM-X.
     03 DATA-ITEM-1  PIC 99.
     03 DATA-ITEM-2  REDEFINES DATA-ITEM-1 PIC XX.
```

REDEFINES cannot be used for level 01 items and can only redefine items on the same level.

Another use for REDEFINES is to offer an alternative PIC desciption for the same data group:

```
01 DATE-TODAY.
      03 UK-DATE.
           05 UK-DAY      PIC 99.
           05 UK-MONTH    PIC 99.
           05 UK-YEAR     PIC 99.
      03 JULIAN-DATE REDEFINES UK-DATE.
           05 JUL-YEAR    PIC 99.
           05 JUL-DAY     PIC 999.
```

- UK date format is ddmmyy while Julian date format is yyddd (i.e. nth day of the year)
- You could move (depending on the type of date given) the date into either UK-DATE or JULIAN-DATE and later in the program call the date using DATE-TODAY
- JULIAN-DATE has one less 9 than UK-DATE. The computer will simply space-fill the unused byte.

4.3 Initializing Data

During a program run it is often necessary to reset an item, or group of items, back to zero (or other value), or

back to a certain literal. Often the program requires data to be set at a certain value (or set literal) at the beginning of a run. For example, an item may be used to count the number of records that have been read by the program. each time this has occurred the line:

```
COMPUTE REC-COUNT = REC-COUNT + 1
```

Obviously, the first time REC-COUNT is encountered, it would need to have a value (probably zero). This could be acheived in the data division:

```
01 REC-COUNT  PIC 9(4) VALUE ZERO.
```

Alternatively, early in the procedure division, the command

```
MOVE ZERO TO REC-COUNT
```

would have the same effect. If, however, you wished to set a group of items to zero (to zeroize) and/or set other alphanumeric items in that group to spaces then you could use the INITIALIZE verb. For example:

```
000200 DATA DIVISION.
000210 WORKING-STORAGE SECTION
000220 01 DATA-GROUP.
000230    03 REC-COUNTER   PIC 9(4).
000240    03 REC-TYPE      PIC X(2).
000250    03 REC-DATE      PIC 9(6).
000260    03 FILLER        PIC X(14) VALUE 'Record details'.
```

And in the procedure division:

```
000400     INITIALIZE DATA-GROUP
```

The effect of this will be that whatever the contents of any of the level 03 items prior to the initialize statement REC-COUNTER will now contain zero, as will REC-DATE, and REC-TYPE will contain spaces. However, FILLER (the last item), is actually a reserved word and refers to an used area. The word 'FILLER' can actually be omitted (i.e. 01   PIC X(14) VALUE 'Record details'.). As you will see in the Printing/writing data part of the next section, a literal can be assigned to this. Following initialization the filler will remain unchanged (and not space-filled).

ZingCOBOL Copyright Timothy R P Brown 2003

5. Defining Data Part 2

5.1 Printing/writing data
5.2 Tables
5.3 Boolean Data
5.4 HIGH-VALUES and LOW-VALUES

How data is prepared for printing or for writing to a file is largely controlled by how it is defined in the data division. This section also describes how tables (aka arrays in other languages) are defined and used in COBOL. The definition and use of Boolean data (i.e. true or false) is discussed too.

5.1 Printing and Writing Data

The specific commands used for printing or writing data are given in the Commands and logic sections. Much of how that data will look, such as in a report, is defined in the data division.

The following code is taken from a full program given in the Sample code section should illustrate how a printed report is defined in the data division. If writing to a file it would be virtually identical (see Sample code section for an example of this).

If you wished to print a report in the form of a table then you would first have to assign an identifier name to the printer in the environment division using the select clause.

```
000110 ENVIRONMENT DIVISION.
000120 INPUT-OUTPUT SECTION.
000130 FILE-CONTROL.
000140
   :
000210       SELECT PRINT-FILE ASSIGN TO PRINTER.
000220
000230
000240 DATA DIVISION.
000250 FILE SECTION.
   :
000580 FD PRINT-FILE.
000590 01 REPORT-OUT      PIC X(80).
   :
000630 WORKING-STORAGE SECTION.
000640
   :
001040 01 PRINT-HEADERS.
001050       03 P-TITLE.
001060             05 P-TITLE-TXT  PIC X(49)  VALUE
001070      '  Batch Control Program - Error Report.  Page:'.
001080             05 P-PAGE-NO    PIC Z9 VALUE ZERO.
001090       03 COL-HEAD-1 PIC X(31)
001100         VALUE ' PART    CUST/   DATE     QUANT'.
001110       03 COL-HEAD-2         PIC X(24)
001120         VALUE '  NO     SUP NO  SUP/REC'.
001130
001140 01 PRINT-LINE.
001150       03 P-PART-NO          PIC X(8).
001160       03                    PIC X VALUE SPACE.
001170       03 P-CUS-SUP-NO       PIC X(6).
001180       03                    PIC XX VALUE SPACES.
```

```
001190        03 P-DATE-S-D.
001200              05 P-DATE-1      PIC XX.
001210              05               PIC X VALUE '/'.
001220              05 P-DATE-2      PIC XX.
001230              05               PIC X VALUE '/'.
001240              05 P-DATE-3      PIC XX.
001250        03                     PIC X VALUE SPACE.
001260        03 P-QUANT             PIC Z(4)9.
001270
001280 01 P-FOOTER.
001290        03 TOT-REC-TXT         PIC X(21)
001300          VALUE 'Total record number: '.
001310       03 P-REC-COUNT          PIC ZZ9 VALUE ZERO.
001320
001330 01 P-BATCH-REC.
001340        03 BAT-TITLE           PIC X(38)
001350     VALUE '  HASH TOTALS IN BATCH CONTROL RECORD'.
001360        03 BATCH-SOURCE        PIC X(29) VALUE SPACES.
001370        03 P-BAT-CUS-SUPP.
001380            05 BAT-CUS-SUP         PIC X(25)
001390           VALUE '  CUSTOMER/SUPPLIER NOS: '.
001400            05 BAT-C-S-N-TOT       PIC Z(7)9.
001410        03 P-BAT-DATE.
001420            05 BAT-DATE            PIC X(9)
001430                  VALUE '  DATES: '.
001440            05 BAT-D-S-D-TOT       PIC Z(7)9.
001450        03 P-BAT-QUANT.
001460            05 BAT-QUANT           PIC X(14)
001470               VALUE '  QUANTITIES: '.
001480            05 BAT-Q-TOT           PIC Z(7)9.
001490        03 P-BAT-PART.
001500            05 BAT-PART            PIC X(12)
001510               VALUE '  PART NOS: '.
001520            05 BAT-P-N-TOT         PIC Z(7)9.
   :
```

- The printout would have the following format: [click here]
- The printer was assigned to PRINT-FILE (the FD level) with the level 01 called REPORT-OUT
- There are four groups used to define each main part of the printout: PRINT-HEADERS (for the title and column heads), PRINT-LINE (for the actual data from the records), P-FOOTER (for the totals at the end of the table), and P-BATCH which appears after the main table and lists various totals
- To define text, fillers are used with a VALUE of what the text is to be, e.g.

```
001090        03 COL-HEAD-1 PIC X(31)
001100            VALUE ' PART    CUST/   DATE     QUANT'.
```

  This is the first line of the column header. COL-HEAD-2 giving the next line.
- Spaces between the titles done by defining a PIC X size that is larger then the text since the extra spaces will be space-filled
- Spaces between data are acheived by the use of fillers with a VALUE SPACES for the

desired PIC X size.

- Data and strings to be printed are first moved to the appropriate item of the print group and then the entire group is written to REPORT-OUT, which is defined as PIC X(80). For example:

```
003220        MOVE PAGE-NO TO P-PAGE-NO
003230        WRITE REPORT-OUT FROM P-TITLE AFTER PAGE
```

  Here the page number is moved to the P-TITLE sub-group member (of PRINT-HEADERS) P-PAGE-NO. The following line effectively means:
  MOVE P-TITLE TO REPORT-OUT
  WRITE REPORT-OUT AFTER PAGE
  (AFTER PAGE instructs the printer to start a new page)
- It is in the data groups involved in printing (or writing to a file) that data editing (such as zero-supression) is performed
- By simply changing the ASSIGN PRINT-FILE TO 'PRINTER' to ASSIGN PRINT-FILE TO 'report.txt' would be all that was required to produce the same report in a file called 'report.txt' and add ORGANIZATION IS LINE SEQUENTIAL. Although, the AFTER PAGE and AFTER ... LINES would have no effect

5.2 Tables

Also known as an array in other languages, a table is a group of data associated with a single item name. To identify pieces data (*elements*) within that table the item is given a *subscript* number which follows the name.

W-NAME (elements)	Subscript
Smith	1
Jones	2
MacDoon	3
Walker	4
O'Leary	5

So, DISPLAY W-NAME (2) will give "Jones".

A 2-dimensional table uses two subscripts:

SALES-TABLE		BRANCH-NO 1	2	3	4
MONTH	1	3.2	3.2	1.9	8.3
	2	3.1	2.8	5.6	4.5
	3	2.3	5.5	2.0	1.2
	4	3.3	1.1	4.0	6.1

So element (2, 4) will contain "1.1".

To define the W-NAME (1-dimensional) table in the data division:

```
01 W-NAME  PIC X(10) OCCURS 5 TIMES.
```

The word TIMES is optional. Also, the PIC clause can also be written after the OCCURS ... clause.

To define the SALES-TABLE (2-dimensional) table, just add another level to the group. Hence:

```
01 SALES-TABLE.
       03 BRANCH-NO OCCURS 4.
            05 MONTHLY-SALES OCCURS 4 PIC 9V9.
```

Notice how only the top level 05 contains the PIC clause. Level 01 describes a whole table made up of 4 items (level 03) containing 4 elements (level 05).

Table can be multi-dimensional, but always the last level will be the identifier name that is associated with the subscripts anc will have the PIC clause.

For the use of tables, see the League Table Program in the Sample Code section.

Finally, don't try to refer to a table element that is of a greater value than that defined in the data division, i.e. W-NAME (6 will cause a runtime error and terminate the program. It should be obvious that a subscript should be a numeric literal or ar identifier that is numeric.

The use of identifiers as subscripts is where tables are of most use, i.e. MONTHLY-SALES (INPUT-BRANCH, INPUT MONTH).

5.3 Boolean Data

Boolean data is either TRUE or FALSE. These are data types are useful for flags for so-called condition-name conditions (see Commands and Logic section).

A simple example:

```
000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID.    NUMBER-SIZE-PROG.
000120 AUTHOR.        TRP BROWN.
000130
000140 DATA DIVISION.
000150 WORKING-STORAGE SECTION.
000160 01 NUMBER-SIZE PIC X.
000170      88 BIG-NUMBER VALUE 'Y'.
000180
000190 77 DATA-NUMBER  PIC 9(6).
000200
000210
000220 PROCEDURE DIVISION.
000230 INPUT-NUMBER-PARAGRAPH.
000240       MOVE 'N' TO NUMBER-SIZE
000250       ACCEPT DATA-NUMBER
000260       IF DATA-NUMBER > 1000
000270            THEN MOVE 'Y' TO NUMBER-SIZE
000280       END-IF
000290       IF BIG-NUMBER
000300            THEN DISPLAY 'Thats a big number'
000310       ELSE DISPLAY 'Thats a little number'
000320       END-IF
000330       STOP RUN.
```

- When then number entered (line 250) is greater than 1000 then a 'Y' character is moved to the level 01 item NUMBER-SIZE. The effect of this is to give the level 88 item BIG-NUMBER a TRUE condition. This is what level 88 is for in COBOL.
- Line 240 initially sets BIG-NUMBER to false by moving an 'N' character into NUMBER-SIZE, although any character (other than 'Y') would have the same effect.
- IF BIG-NUMBER THEN... is like saying "IF BIG-NUMBER is true THEN..."

Multiple level 88 can be set for a single group, or you can have more than one option that will set the condition to true.

```
01 THIRTY-DAY-MONTHS PIC X VALUE SPACE.
       88 SEPTEMBER   VALUE 'S'.
       88 APRIL       VALUE 'A'.
       88 JUNE        VALUE 'J'.
       88 NOVEMBER    VALUE 'N'.
```

```
01 MONTHS-CHECK PIC X.
        88 SHORT-MONTH VALUE 'S' 'A'
                             'J' 'N'
                             'F'.

01 GRADES-CHECK PIC 999.
       88 A-GRADE    VALUE 70 THRU 100.
       88 B-GRADE    VALUE 60 THRU 69.
       88 C-GRADE    VALUE 50 THRU 59.
       88 FAIL-GRADE VALUE 0 THRU 49.
```

GRADES-CHECK uses THRU (or THROUGH) to allow a range of numeric values to be tested.

SET

A useful verb to use is **SET**. Rather than having to use the line:

MOVE 'Y' TO NUMBER-SIZE

as in the code example above, you can simply set the boolean variable to true by coding:

SET BIG-NUMBER TO TRUE

This means that you don't have to worry about what the value of the level 01 item has to be in order to make the associated level 88 to be true (notice that it is the level 88 item name that is set to true and NOT the level 01 item). Of course, you might also code

SET BIG-NUMBER TO FALSE.

5.4 HIGH-VALUES and LOW-VALUES

There are occasions when you may wish to set a variable to an infinitely high or infinitely low number. For example, suppose you were merging two files on surnames as the primary key:

```
*in data division FILE SECTION

 FD FILE-1.
 01 RECORD-1.
     03 IN-NAME-1 PIC X(20).
     03 FILLER    PIC X(50).
```

```
 FD MERGE-FILE.
 01 RECORD-OUT    PIC X(70).

   :
   :

   PERFORM WITH TEST AFTER EOF-FLAG-1 AND EOF-FLAG-2
*loop until each file has been read to completion
*read each file
      READ FILE-1
          AT END SET EOF-FLAG-1 TO TRUE
          MOVE HIGH-VALUES TO IN-NAME-1
      END-READ
      READ FILE-2
          AT END SET EOF-FLAG-2 TO TRUE
          MOVE HIGH-VALUES TO IN-NAME-2
      END-READ

*sort the records (assuming no 2 names are the same)
*on ascending surname
      IF IN-NAME-1 IS < IN-NAME-2 THEN
          WRITE RECORD-OUT FROM RECORD-1
      ELSE
          WRITE RECORD-OUT FROM RECORD-2
      END-IF

   END-PERFORM
```

In this example, when IN-NAME-1 is less than IN-NAME-2 (based on their ASCII values e.g. A < B etc..) then the FILE-1 record (RECORD-1) is written to the merge file (RECORD-OUT). One of FILE-1 and FILE-2 will come to an end before the other so the completed file has its IN-NAME-_ value set to constant that will ALWAYS be greater than the IN-NAME-_ value still being read, ensuring all remain files are written to the merge file. This is done with the lines: **MOVE HIGH-VALUES TO IN-NAME-1** and **MOVE HIGH-VALUES TO IN-NAME-2**

It is important to note that HIGH-VALUES and LOW-VALUES are **ALPHANUMERIC** in type, so you can't set numerically defined variables to this type (you would have to implicitly redefine the variable first). This is an annoying quirk of COBOL.

ZingCOBOL Copyright Timothy R P Brown 2003

6. Commands and Logic

6.1 ACCEPT and DISPLAY
6.2 MOVE
6.3 PERFORM
6.4 IF..THEN..ELSE
6.5 Conditions
6.6 EVALUATE
6.7 Arithmetic
6.8 Strings
6.9 WRITE
6.10 Scope Terminators

Many of the commands described in this section have already been used in earlier sections but here their description will be shown alongside related commands, clauses and verbs. It should be noted that a command probably *is* a verb, while a clause is a collection of COBOL words without a verb...something like that...

6.1 ACCEPT and DISPLAY

To enter data via the console during a program run, use the ACCEPT verb, e.g:
ACCEPT W-DATA-IN

To display data on the console during a run use the DISPLAY verb, i.e:
DISPLAY W-DATA-OUT

To place text with the outputed data you would code:
DISPLAY 'Inputed data is ' W-DATA-OUT

6.2 MOVE

The format is:

MOVE [literal-1 or identifier-1] TO [identifier-2] ...

The MOVE statement has already been extensively used in the examples in the Defining Data section. A couple of features have not been described yet: CORRESPONDING (abreviation CORR) and the qualification OF or IN. The elipsis (**...**) means more of the same, i.e. above [identifier-2] [identifier-3] [identifier-4]...and so on.

To move a group of items from one field description to another:

```
     03 DATE-IN.
            05 W-DAY         PIC 99.
            05 W-MONTH       PIC 99.
            05 W-YEAR        PIC 99.
 :
     03 DATE-OUT.
            05 W-DAY          PIC 99.
            05                       PIC X VALUE '/'.
            05 W-MONTH        PIC 99.
            05                       PIC X VALUE '/'.
            05 W-YEAR         PIC 99.
```

If you were to code: MOVE DATE-IN TO DATE-OUT you would end up with the 6 characters of DATE-IN appearing in the first 6 positions of DATE-OUT, including over-written fillers. To get the contents of W-DAY of DATE-IN into W-DAY of DATE-OUT (and the same for the other two items) you could either move them individually, or you could simply code: MOVE CORRESPONDING DATE-IN TO DATE-OUT. To do this the items *must* have the same name spelling and *must* be of the same level (here they are both level 03). They don't have to be in the same level 01 group.

Of course, this does present the programmer with a potential problem, this being that if elsewhere in the program you were to code, say, ADD 12 to W-MONTH, the compiler would report a syntax error since it W-MONTH appears twice in the data division and doesn't know which one you mean. To remedy this, you have to qualify the item, i.e. state which group W-MONTH you mean, i.e. :

MOVE 12 TO W-MONTH **IN** DATE-OUT.

You could use the word **OF** instead of IN here to the same effect.

Reference modification

To access specific characters within a string you can use a reference modifier.
STRING-ITEM (*startPosition*:*Length*)
The start position is the nth character of the STRING-ITEM. For MicroFocus compilers at least, the length can be omitted if you want all characters to the end of the string. e.g.

```
      WORKING-STORAGE SECTION.

      01 STRING-1  PIC X(10) VALUE 'ABCDEFGHIJ'.
      01 STRING-2  PIC X(10) VALUE SPACES.
      01 STRING-3  PIC X(10) VALUE SPACES.
      01 STRING-4  PIC X(10) VALUE SPACES.
      01 STRING-5  PIC X(10) VALUE SPACES.
      01 STRING-6  PIC X(10) VALUE SPACES.
          :
          :

in procedure division:

          MOVE STRING-1(2:6)   TO STRING-2
          MOVE STRING-1(1:9)   TO STRING-3
          MOVE STRING-1(6)     TO STRING-4
          MOVE STRING-1(5:1)   TO STRING-5
          MOVE STRING-1(3:3)   TO STRING-6
```

Then:
STRING-2 will contain characters 2 to 6, i.e. : "BCDEFG "
STRING-3 will contain characters 1 to 9, i.e. : "ABCDEFGHI "
STRING-4 will contain characters 6 to the end of STRING-1, i.e. : "FGHIJ "
STRING-5 will contain character 5 only, i.e. : "E "
STRING-6 will contain characters 3 to 5, i.e. : "CDE "

6.3 PERFORM

The PERFORM verb is one of the most important in COBOL (alongside MOVE). PERFORM has already been encountered in the Four Divisions section, where it was used to call paragraphs from within a control paragraph. Of course, it doesn't have to be a control (or main) paragraph.

```
    000290 PROCEDURE DIVISION.
```

```
000300 XYZ-PARAGRAPH.
000310        PERFORM FIRST-PROCESS
000320        PERFORM SECOND-PARAGRAPH
000330        STOP RUN.
  :


002000 FIRST-PROCESS.
002010     [statements]
  :        [last statement].
```

In the above code, the paragraph FIRST-PROCESS is executed. When the full stop at the end of this paragraph is encountered the logic will return to XYZ-PARAGRAPH at the next line, i.e. line 320. This is called an Out-of-Line PERFORM.

The PERFORM verb can form the bases of a repetitive loop (or sub-routine) until a certin condition has been met. For Example:

```
  :
000290 PROCEDURE DIVISION.
000300 XYZ-PARAGRAPH.
000310        PERFORM COUNT-PROCESS UNTIL W-COUNTER > 10
000320        STOP RUN.
001000
002000 COUNT-PROCESS.
002010     COMPUTE W-COUNTER = W-COUNTER + 1
002020     DISPLAY 'Number of loops is ' W-COUNTER.
```

In the above code, COUNT-PROCESS is executed until the value of W-COUNT has reached 11.
The format for an Out-of-Line PERFORM is:

PERFORM [paragraph-name] UNTIL [condition]

An In-Line PERFORM, rather than execute a paragraph (aka procedure), allows for the repeated execution of a series of commands. The format for an In-Line PERFORM is:

PERFORM UNTIL
{action}...
END-PERFORM

Example:

```
  :
000290 PROCEDURE DIVISION.
000300 XYZ-PARAGRAPH.
000305        MOVE ZERO TO W-COUNTER
000310        PERFORM UNTIL W-COUNTER > 10
000320           COMPUTE W-COUNTER = W-COUNTER + 1
000330           DISPLAY 'This is loop number: ' W-COUNTER
```

```
000340        END-PERFORM
000350        DISPLAY 'Counter is now equal to: ' W-COUNTER
000360        STOP RUN.
```

END-PERFORM defines the *scope* of the PERFORM loop, and is a Scope terminator. Other such scope terminators exist for other commands that will be described further on. The code above will loop 11 times (showning numbers 1 to 11). This is because when W-COUNTER is equal to 10, the condition (W-COUNTER) is still false. 1 is then added, and W-COUNTER is displayed as 11, and *now* when W-COUNTER is tested the condition will be true and the logic will then jump to the statement that immediately follows END-PERFORM.

This type of PEFORM tests the condition before the following statements are allowed to proceed. Using WITH TEST can be used to define when the test is done:

```
   :
000290 PROCEDURE DIVISION.
000300 XYZ-PARAGRAPH.
000305        MOVE ZERO TO W-COUNTER
000310        PERFORM WITH TEST AFTER UNTIL W-COUNTER > 10
000320           COMPUTE W-COUNTER = W-COUNTER + 1
000330           DISPLAY 'This is loop number: ' W-COUNTER
000340        END-PERFORM
000350        DISPLAY 'Counter is now equal to: ' W-COUNTER
000360        STOP RUN.
```

Now the condition is tested after the commands within the PERFORM..END-PERFORM loop has be executed once. (WITH TEST **BEFORE** has same effect as initial example).

If you wanted to loop a desired number of times you could use TIMES

```
              PERFORM 5 TIMES
                     COMPUTE W-NUMBER = XYZ * 3
              END-PERFORM
```

The format is:

PERFORM {identifier or literal} TIMES
{action}...
END-PERFORM

To have a loop using an increment (such as a 'for..do' loop in Pascal or FOR in BASIC), the PERFORM VARYING statement is used.
The Format is:

PERFORM {paragraph-name if out-of-line} VARYING {identifier-1}

FROM {identifier-2 or literal} BY {identifier-3 or literal}
UNTIL {condition}
END-PERFORM

What does all that mean? Well look at the example:

```
  :
000290 PROCEDURE DIVISION.
000300 XYZ-PARAGRAPH.
000310       PERFORM VARYING W-COUNTER FROM 1 BY 2
000320               UNTIL W-COUNTER > 10
000330          DISPLAY 'This is loop number: ' W-COUNTER
000340       END-PERFORM
000350       DISPLAY 'Counter is now equal to: ' W-COUNTER
000360       STOP RUN.
```

This code will display:

```
This is loop number: 1
This is loop number: 3
This is loop number: 5
This is loop number: 7
This is loop number: 9
Counter is now equal to: 11
```

This because with each loop, W-COUNTER has increased from 1 by increments of 2. When W-COUNT was equal to 11 then the condition W-COUNTER > 10 is now true and so the loop is exited. If you wanted to count downwards you could code:

PERFORM VARYING W-COUNTER FROM 20 BY -1
UNTIL W-COUNTER < ZERO.

The last thing to mention is **PERFORM..THRU**. If a program had a series of paragraphs, just for the sake of argument, called PROCESS-1, PROCESS-2, PROCESS-3 and PROCESS-4, then if you wished to execute these paragraphs in the order that they are written you could code: **PERFORM PROCESS-1 THRU PROCESS-4** with any out-of-line loops and conditions you might want. Seemingly, this is not good programming practise so is generally avoided.

6.4 IF..THEN..ELSE

Another fundamental part of programming logic is the ability to offer a choice of what to do that depends on the conditions asked of. (I'm not sure that makes any sense...). The format is:

IF {identifier-1} {condition} {identifier-2 or literal} ...

THEN {statements}
[ELSE {statements}]
END-IF

An example:

```
IF X = Y THEN
   MOVE 1 TO Y-COUNTER
ELSE
   MOVE 1 TO X-COUNTER
END-IF
```

ELSE is used if an alternative statement is to be executed if the first condition is false. If there was *only* to be action if X = Y then the ELSE would be ommitted.

The END-IF terminates the IF statement. All that lies between IF and END-IF will depend on the conditions being tested.

Multiple conditions can be tested, i.e. IF (X = Y) AND (Y < 100) THEN ..

The types of conditions available are described in the following section..

6.5 Conditions

There are four types of conditions that could be tested either in a PERFORM, IF..THEN, or EVALUATE (see next section), these being:

```
1. Class conditions
2. Relational conditions
3. Sign conditions
4. Condition-name conditions
```

Class conditions test where an item is NUMERIC, ALPHABETIC, ALPHABETIC-LOWER, or ALPHABETIC-HIGHER (as in lower or upper case).

Relational conditions allow comparisons, i.e: GREATER THAN, LESS THAN, EQUAL TO or their sign equivalent: **" > ", " < ", " = "**, respectively.

Sign conditions test whether an item IS POSITIVE, IS NEGATIVE, or IS NOT ZERO. (note 'IS' is optional)

Condition-name conditions are as described in the Defining data (part 2) section, where a named condition is defined in the data division using a level 88 description.

Conditions can be combined using AND, OR, AND NOT, OR NOT, and brackets. The most common combinations would probably be GREATER THAN OR EQUAL TO and LESS THAN OR EQUAL TO, which can simply be written **>=** and **<=** respectively. Also, NOT EQUAL TO would be **<>** although I find the Fujitsu compiler rejects '<>' so I just use '**NOT =** ' instead.

More complex combinations can be acheived with the use of brackets. eg.

```
   IF ( X > Y ) AND ( ( Y + 10 < Z ) OR ( X - 10 > Z ) ) THEN ...
```

```
Remember:
  [true] AND [false] = FALSE
  [true] AND [true]  = TRUE
  [true] OR [false]  = TRUE
  [true] OR [true]   = TRUE
         NOT [true]  = FALSE
         NOT [false] = TRUE
```

Alpha-numeric comparisons can also be made that relate to their ASCII character value, so 'A' < 'Z' etc...

The **SET** verb is quite useful when working with boolean items and has been discussed in the previous section.

6.6 EVALUATE

If there are a large number of conditional alternatives, then using a large number of nested IF statements can be messy:

```
IF A = 1 THEN PERFORM PARA-1
ELSE
        IF A = 2 THEN PERFORM PARA-2
        ELSE
                IF A = 3 THEN PERFORM PARA-3
                ELSE
                        IF A = 4 THEN PERFORM PARA-4
                        END-IF
                END-IF
        END-IF
END-IF
```

The above example only tested four possible values for 'A'. Suppose there were ten or twenty? This is where the EVALUATE statement is of great use. The format is:

```
                { identifier-1 }                 { identifier-2 }
                { literal-1    }                 { literal-2    }
EVALUATE        { expression-1 }   ALSO          { expression-2 }
                { TRUE         }                 { TRUE         }
                { FALSE        }                 { FALSE        }

        WHEN  {statement-1}...
        WHEN OTHER  {statement-2}...

END-EVALUATE
```

The best way to understand this is to look at the following examples:

Example 1.

```
EVALUATE W-NUM
      WHEN 1 MOVE 10 TO NEW-DATA
             DISPLAY 'NEW-DATA IS 10'
      WHEN 2 MOVE 20 TO NEW-DATA
             DISPLAY 'NEW-DATA IS 20'
      WHEN 3 MOVE 30 TO NEW-DATA
             DISPLAY 'NEW-DATA IS 30'
      WHEN OTHER MOVE ZERO TO NEW-DATA
             DISPLAY 'NEW-DATA IS 0'
END-EVALUATE
```

Example 2.

```
EVALUATE SCORE
      WHEN 0 THRU 49 MOVE 'FAIL' TO W-GRADE
      WHEN 50 THRU 59 MOVE 'C' TO W-GRADE
      WHEN 60 THRU 69 MOVE 'B' TO W-GRADE
      WHEN OTHER MOVE 'A' TO W-GRADE
END EVALUATE
```

Example 3.

```
EVALUATE (FUEL-TYPE = 'PETROL') ALSO (ENGINE-SIZE > 1.1)
      WHEN TRUE ALSO TRUE
                DISPLAY '20% PETEROL DUTY TO PAY'
      WHEN TRUE ALSO FALSE
                DISPLAY '10% PETROL DUTY TO PAY'
      WHEN FALSE ALSO TRUE
                DISPLAY '15% DIESEL DUTY TO PAY'
      WHEN FALSE ALSO FLASE
                DISPLAY '5% DIESEL DUTY TO PAY'
END-EVALUATE
```

Example 1 shows how an item (W-NUM) is compared to a set of possibilities, and when true, any number of statements can be executed. Example 2 shows how a range of values can be studied using the THRU clause. Care should be taken to ensure that these ranges do not overlap. Both of these examples use WHEN OTHER. Again, care should be taken: in example 2, as it is coded, a score of -1 would result in an A-grade being awarded. A better coded solution would include:

```
        :
      WHEN 70 THRU 100 MOVE 'A' TO W-GRADE
      WHEN OTHER DISPLAY 'ERROR. SCORE NOT VALID'
END EVALUATE
```

Now, when SCORE is less then zero (or greater than 100) an error message will be displayed.

6.7 Arithmetic

To perform arithmetic calculations there are two ways of going about doing this: using the ADD, SUBTRACT, MULITPLY, DIVIDE verbs, or using the COMPUTE verb as seen already. The formats for the first group are as follows**:** [square brackets indicate optional words]

ADD {identifier-1 or literal}... TO {identifier-2 or literal}...

[GIVING {identifier-3}]
[NOT] [ON SIZE ERROR {statements}]
[END-ADD]

Examples:

```
ADD NUM-A TO NUM-B GIVING NUM-TOTAL-1
ADD NUM-A, 20 TO NUM-B GIVING NUM-TOTAL-2
ADD 3 TO NUM-TOTAL-3
```

When the word GIVING is not used (as in the third example) the identifier that follows 'TO' is where the result of the addition. This also applies to SUBTRACT and MULTIPLY. ON SIZE ERROR is a conditional, whereby if the result

of the calculation is larger than the PIC description (i.e. the result is truncated either at the leading end or the decimal places). On such an occasion a series of statements can be executed. The use of ON SIZE ERROR means that a scope terminator is required (END-ADD). The second example adds both NUM-A *and* 20 to NUM-B.

```
SUBTRACT {identifier-1 or literal}... FROM {identifier-2 or literal}...

    [GIVING {identifier-3}]
    [NOT] [ON SIZE ERROR {statements}]
[END-SUBTRACT]
```

Examples:

```
SUBTRACT 200 FROM NUM-C GIVING NUM-D
     ON SIZE ERROR DISPLAY 'NUM-D is out of range'
END-SUBTRACT
SUBTRACT NUM-F FROM 20       ** this won't work! **
```

The second example is illegal because, in the absence of a receiving identifier after GIVING, the result of the subtraction has nowhere to go (20 is a literal). The same would apply to ADD and MULTIPLY.

```
MULTIPLY {identifier-1 or literal}... BY {identifier-2 or literal}...

    [GIVING {identifier-3}][ROUNDED]
    [NOT] [ON SIZE ERROR {statements}]
[END-MULTIPLY]
```

Examples:

```
MULTIPLY NUM-G BY 20 GIVING NUM-F
MULTIPLY 20 BY NUM-G
```

```
DIVIDE {identifier-1 or literal} BY {identifier-2 or literal}...
    GIVING {identifier-3} [ROUNDED] [REMAINDER {identifier-4}]
    [NOT] [ON SIZE ERROR {statements}]
[END-DIVIDE]
```

Examples:

```
DIVIDE NUM-H BY 3 GIVING NUM-I REMAINDER NUM-REMAIN
DIVIDE NUM-Y BY 3 GIVING NUM-K ROUNDED
```

The DIVIDE statement differs from the previous 3 in that GIVING is required. Also, the remainder of the division (e.g. 7 divided by 3 equals 3 remainder 1) can be stored in an identifier. The ROUNDED option, which is also available for the MULTIPLY statement, will round to the nearest significant decimal place, defined by the PIC clause. E.g.**:**

```
000100 77 NUM-A  PIC 99 VALUE 10.
000200 77 NUM-B  PIC 9V99.
  :
002000      DIVIDE NUM-A BY 3 GIVING NUM-B
002010           ON SIZE-ERROR DISPLAY 'RESULT IS TRUNCATED'
002020      END-DIVIDE
002030
002040      DIVIDE NUM-A BY 3 GIVING NUM-B ROUNDED
002050           ON SIZE-ERROR DISPLAY 'RESULT IS TRUNCATED'
002020      END-DIVIDE
```

The first DIVIDE statement will result in a size error (20 / 3 = 6.66666..) as NUM-B will contain 6.66 but will have truncated the rest. This does not apply to the second DIVIDE statement since it has been rounded to fit the pic description 9V99, and so in this case NUM-B will contain 6.67.

```
DIVIDE {identifier-1 or literal} INTO {identifier-2 or literal}...
     GIVING {identifier-3} [ROUNDED] [REMAINDER {identifier-4}]
     [NOT] [ON SIZE ERROR {statements}]
[END-DIVIDE]
```

Examples:

```
DIVIDE 3 INTO NUM-Y GIVING NUM-K ROUNDED
```

This differs from the previous DIVIDE statement only in the order of numerator and denominator (both mean NUM-Y / 3).

COMPUTE

As previously seen in earlier sections, COMPUTE can be used to do arithmetic calculations. The format is:

```
COMPUTE {identifier-1} [ROUNDED] = arithmetic expression
    [NOT] [ON SIZE ERROR {statements}] [END-COMPUTE]
```

with the operations:

```
+    add
-    subtract
*    multiply
/    divide
**   to the power of
```

Note that brackets need to be used for complex calculations where signs have presidence over each other, for example: 2 + 3 * 2 equals 8 (and not 10) since 3 * 2 is calculated before the addition. Remember your school maths lessons (BROMDAS or something).

6.8 Strings

STRING

```
STRING {identifier-1 or literal-1} DELIMITED BY {identifier-2 or literal-2 or
SIZE}...
     INTO {identifier-3}
     ON OVERFLOW [statements]
     NOT ON OVERFLOW [statements]
END-STRING
```

STRING will move a series of strings into a destination string (from left to right without space filling). If the destination string is not large enough to hold all the source strings then this can be detected and acted on by the ON OVERFLOW condition. The DELIMITED word specifies the source string characters to be used:

```
01 W-DAY       PIC XXX VALUE 'MON'.
01 W-MONTH     PIC XXX VALUE '5  '.
01 W-YEAR      PIC XXXX VALUE '2000;'.
  :

STRING W-DAY DELIMITED BY SIZE
       '/' DELIMITED BY SIZE
       W-MONTH DELIMITED BY SPACES
       '/' DELIMITED BY SIZE
       W-YEAR DELIMITED BY ';'
    INTO DATE-STRING
END-STRING
```

The item DATE-STRING will contain "MON/5/2000".

UNSTRING

UNSTRING {identifier-1 or literal-1} DELIMITED BY {identifier-2 or literal-2 or SIZE}...
INTO {identifier-3 COUNT IN identifier-4}...
TALLYING IN {identifier-5}
ON OVERFLOW [statements]
NOT ON OVERFLOW [statements]
END-UNSTRING

UNSTRING allows you to break up a string into small strings placed into new items:

```
01 W-LONG-STRING   PIC X(50) VALUE 'Name;Address;Post Code'.
  :

UNSTRING W-LONG-STRING DELIMITED BY ';'
      INTO W-NAME COUNT IN CHARS-NAME
```

```
                W-ADDRESS COUNT IN CHARS-ADDR
                W-POST-CODE COUNT IN CHARS-PCODE
           TALLYING IN NUM-STRINGS-OUT
     END-UNSTRING
```

Here then string 'Name' will be placed into W-NAME, containing 4 characters, thus CHARS-NAME will contain the value of 4. Likewise for W-ADDRESS ('Address') CHARS-ADDR (7) etc... Notice how the ; character has been lost. Any character, including spaces can be used as a delimiter. TALLYING IN will count the number of items that were filled by the UNSTRING operation, in this case NUM-STRINGS-OUT will contain the value 3. Lastly, the ON OVERFLOW detects when each target of the UNSTRING operation has been used but there remains unused characters in the source string, e.g. if W-LONG-STRING contained 'Name;Address;Post Code;Country'.

INSPECT

INSPECT {identifier-1} REPLACING CHARACTERS BY {identifier-2 or literal-1}
{BEFORE or AFTER} [INITIAL {identifier-3 or literal-2}]
{ALL or LEADING or FIRST} {identifier-4 or literal-3}
BY {identifier-5 or literal-4} {BEFORE or AFTER} INITIAL {identifier-6 or literal-5}

This form of INSPECT allows you to change characters within a string using the various options above.

INSPECT {identifier-1} TALLYING {identifier-2}
{BEFORE or AFTER} [INITIAL {identifier-3 or literal-2}]
{ALL or LEADING or FIRST} {identifier-4 or literal-3}
BY {identifier-5 or literal-4} {BEFORE or AFTER} INITIAL {identifier-6 or literal-5}

Here the source string is inspected and a tally of the number of characters defined (using the subsequent options) is held in {identifier-2}.

6.9 WRITE

To output data to the printer or to a file, the verb WRITE is used. It would be of the form:

WRITE {level 01 name of file/printer FD}

For example:

```
000100 ENVIRONMENT DIVISION.
000200 INPUT-OUTPUT SECTION.
000300 FILE-CONTROL.
000400      ASSIGN PRINT-FILE TO PRINTER.
   :
```

```
000500 DATA DIVISION.
000600 FILE SECTION.
000700 FD PRINT-FILE.
000800 01 P-DATA   PIC X(80).
  :

000900 WORKING-STORAGE SECTION.
001000 01 DATA-NUMBER PIC 9(6) VALUE 123456.
001100 01 PRINT-NUMBER PIC X(6).
  :

010900*in procedure division
011100         MOVE DATA-NUMBER TO PRINT-NUMBER
011200         MOVE PRINT-NUMBER TO P-DATA
011300         WRITE P-DATA
```

To simplify things the word **FROM** can be used to save always having to first MOVE the data (PRINT-NUMBER) into the printing item (P-DATA above). So, line 011200 and 011300 can simply be written as:

011100 WRITE P-DATA FROM PRINT-NUMBER

In addition to WRITE, the is also REWRITE and DELETE which are used to update records within files that have been opened in I-O mode (see the following section). When using DELETE you must first read the record that is to be deleted. Also, when deleting a record you refer to the FILE NAME rather than the record name:

```
000300 FD IN-FILE.
000400 01 CUST-RECORD.
000500     03 C-NAME   PIC X(20).
000600     03 C-NUMBER PIC 9(6).
  :

001000* in procedure division
001100         READ IN-FILE
001200           NOT AT END
001300             IF C-NUMBER = 123456 THEN
001400                 DELETE IN-FILE
001500             ELSE MOVE C-NUMBER TO W-DATA-STORE
001600             END-IF
001700         END-READ
```

For details on the READ statement, see the following section

6.10 Scope Terminators

In the section COBOL basics I mentioned the full stop (period). This is what can be described as a *scope terminator*. Many COBOL verbs have their own scope terminator, for example, END-IF, END-READ, END-PERFORM etc... The

purpose of a scope terminator is to define when a verb's scope (i.e. associated logic) is finished.

For example:

```
READ IN-FILE
  AT END MOVE 'Y' TO EOF FLAG
  NOT AT END
     IF REC-IN = 'Z' THEN
        PERFORM PROCEDURE-X
     END-IF
END-READ
[more code]
```

In the above example END-READ defines the scope of the READ statement since there is a condition involved (AT END of the file, or NOT AT END of the file), while END-IF defines then end of the IF condition, i.e. END-READ and END-IF define their scope. Any code that follows the read statement will apply regardless of the READ condition (which is what you would want in the above example). Without END-READ the subsequent code would only be performed while NOT AT END is true: some nasty bugs could ensue! Things become even more scary if you forget to use END-IF or END-PERFORM (especially when looping). There's a good chance the compiler might pick up the error.

However, a period is also a scope terminator. You might also code:

```
READ IN-FILE
  AT END MOVE 'Y' TO EOF FLAG
  NOT AT END
     PERFORM UNTIL REC-IN = 'A'
       IF REC-IN = 'Z' THEN
          PERFORM PROCEDURE-X.
     END-PERFORM
[more code]
```

This would have the same effect as the first example (assuming the compiler doesn't complain). Some people do use periods in place of END-IF etc (note: I'm not sure you allowed to replace END-PERFORM however). Problems may arise when you forget to use a scope terminator somewhere and there's a period somewhere further down the code then the compiler might just get confused.

It is important to realise that the period will terminate *all* ongoing conditions. So in the above example, the period will act as both an END-IF, END-PERFORM and END-READ.

Look at this paragraph:

```
000090*this works, using period scope terminators
000100 PARAGRAPH-ABC.
000200     MOVE 0 TO N
000300     PERFORM UNTIL N > 10
000400        COMPUTE N = N + 1
000500        DISPLAY N.
000600
000700     PERFORM PROCEDURE-Y  N TIMES
000800     PERFORM UNTIL END-OF-FILE
000900       READ IN-FILE
001000         AT END MOVE 'Y' TO EOF-FLAG
001100         NOT AT END
001200           ADD VALUE-FROM-RECORD TO N GIVING X.
001300
```

```
001400     END-PERFORM
001500     DISPLAY X.
```

In the first example, the code will display numbers 1 to 10. It will then perform PROCEDURE-Y 11 times. Finally, the numbers coming from the IN-FILE (VALUE-FROM-RECORD) will be added to 11 giving X, which is then displayed.

But what if we were to forget to put a period at the end of line 500?

```
000090*this has a syntax error
000100 PARAGRAPH-ABC.
000200     MOVE 0 TO N.
000300     PERFORM UNTIL N > 10
000400        COMPUTE N = N + 1
000500        DISPLAY N
000600
000700     PERFORM PROCEDURE-Y  N TIMES.
000800     PERFORM UNTIL END-OF-FILE
000900       READ IN-FILE
001000         AT END MOVE 'Y' TO EOF-FLAG
001100         NOT AT END
001200           ADD VALUE-FROM-RECORD TO N GIVING X.
001300
001400     END-PERFORM
001500     DISPLAY X.
```

Now, the period on line 700 will terminate the scope of the PERFORM statement on line 300. This means that PROCEDURE-Y gets performed 1+2+3+4+5+6+7+8+9+10+11 times (that's 66 times!). Oh dear.

In fact, when I tried to test these code fragments by compiling [on the Fujitsu COBOL v3] it complained bitterly! The compiler was particularly bothered by the lack of END-PERFORMS.

I was taught to *only* use 2 periods in any paragraph: the first after the paragraph name, the second (and last) at the end of the paragraph. So always use the verb's own scope terminator. More typing but less headaches in my humble opinion. Here's what the above code would look like when following this advice:

```
000090*using just 2 periods
000100 PARAGRAPH-ABC.
000200     MOVE 0 TO N
000300     PERFORM UNTIL N > 10
000400        COMPUTE N = N + 1
000500        DISPLAY N
000600     END-PERFORM
000700     PERFORM PROCEDURE-Y  N TIMES
000800     PERFORM UNTIL END-OF-FILE
000900       READ IN-FILE
001000         AT END MOVE 'Y' TO EOF-FLAG
001100         NOT AT END
001200           ADD VALUE-FROM-RECORD TO N GIVING X
001300       END-READ
001400     END-PERFORM
001500     DISPLAY X.
```

Ahh...that's better...

ZingCOBOL Copyright Timothy R P Brown 2003

7. File Handling

7.1 Reading and writing
7.2 REWRITE, DELETE and EXTEND
7.3 File organization
7.4 SORT and MERGE
7.5 Input and output procedures
7.6 File Status (error handling)

This section outlines how data can read from and written to files, how records are organized within a file and how records can be manipulated (e.g. sorting, merging).

7.1 Reading and Writing

In order to either read, alter or create a new file, we must first open it (even if it doesn't even exist yet). In doing so, a open mode must be defined. To simply read data from an existing file it would be opened in INPUT mode. In this mode, the file is read-only and cannot be altered in any way.
If writing to new file, i.e. creating one (or overwriting an existing file so be careful) the new file would be opened in OUTPUT mode. You cannot read data from a file opened in OUTPUT mode.
EXTEND mode allows for records to be added to the end of an existing file.
I-O mode is for input and output access to the file, such as when you wish to update a record, or delete a record.

When a file is no longer required, the file needs to be closed again (using CLOSE). You can open and close a file as often as you like during a program run, although bear in mind that each time you open a file the computer will read from the first record onwards (in INPUT and I-O mode) or will overwrite in OUTPUT mode.

```
OPEN {INPUT or OUTPUT or I-O or EXTEND} {filename-1}...
     {INPUT or OUTPUT or I-O or EXTEND} {filename-2}...
```

e.g.

```
OPEN INPUT DATA-1-FILE DATA-2-FILE
     OUTPUT NEW-DATA-FILE
            :
            :
CLOSE DATA-1-FILE DATA-2-FILE NEW-DATA-FILE
```

READ

The READ statement will read the data from a file, taking precisely the data that is defined in the file descriptor (FD) in the data division (file section) (see The Four Divisions section).
The format is:

```
READ {FD filename}
      AT END {statements}
      NOT AT END {statements}
END-READ
```

Since a file would likely contain more than one record, the READ statement is often contained within a PERFORM loop:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.      READ-EXAMPLE.
 AUTHOR           ZINGMATTER.

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.

     ASSIGN IN-FILE TO 'A:CUSTOMER.DAT'
         ORGANIZATION IS LINE SEQUENTIAL.
     ASSIGN PRINT-FILE TO PRINTER.
       :
 DATA DIVISION.
 FILE SECTION.
 FD IN-FILE.
 01 CUSTOMER-DETAILS.
     03 CUS-NAME      PIC X(20).
     03 CUS-NUM       PIC 9(6).

 FD PRINT-FILE.
 01 PRINT-REC         PIC X(60).

 WORKING-STORAGE SECTION.
 01 EOF-FLAG PIC X.
     88 END-OF-IN-FILE VALUE 'Y'.

 01 P-CUS-DETAILS
     03               PIC X(5) VALUE SPACES.
     03 P-NAME        PIC X(25).
     03 P-NUM         PIC Z(5)9.
       :
 PROCEDURE DIVISION.
 MAIN-PARAGRAPH.
    OPEN INPUT IN-FILE
*"Prime" read
    READ IN-FILE
      AT END MOVE 'Y' TO EOF-FLAG
      NOT AT END PERFORM PRINT-DETAILS
    END-READ

*Main reading loop
    PERFORM UNTIL END-OF-IN-FILE
       READ IN-FILE
          AT END MOVE 'Y' TO EOF-FLAG
          NOT AT END PERFORM PRINT-DETAILS
       END-READ
    END-PERFORM

    STOP RUN.

 PRINT-DETAILS.
    MOVE CUS-NAME TO P-NAME
    MOVE CUS-NUM TO P-NUM
    WRITE PRINT-REC FROM P-CUS-DETAILS AFTER 1 LINE.
```

- A record containing a customer name (CUS-NAME) and the customer number (CUS-NUM) are read from a file *customer.dat* assign to IN-FILE.
- The file is opened for INPUT (i.e. read-only).
- The "prime read" refered to in the comment is the initial read of IN-FILE that allows for the possibility that the file contains no records.
- The AT END clause tests for the end of file condition. When true, a series of statements can then be executed. Likewise, the NOT AT END clause allows for a series of statements to be executed when this condition is true. In the above example, when the file contains no more records (i.e. is at the end of the file) 'Y' is moved to EOF-FLAG, thereby making the condition name condition (END-OF-IN-FILE) true. When not at the end of the file, a record is read into memory and the paragraph PRINT-DETAILS is executed.
- The statements between PERFORM UNTIL... and END-PERFORM are executed until the END-OF-IN-FILE condition is true (when the AT END of the read statement is true).

If you want to place data from a record into an item in WORKING-STORAGE (in addition to the memory space already allocated to the same data defined in the data division - so not much call for it), then use **READ ... INTO**. i.e:

```
READ IN-FILE INTO W-RECORD-IN
```

7.2 REWRITE, DELETE, and EXTEND

In order to ammend a record in a file, such as to update data (see League Table Program in sample programs section), to delete a record altogther, or to add a record to the end of a file, you can use **REWRITE**, **DELETE** or **EXTEND**, respectively. However, to use REWRITE or DELETE you must open the file using I-O mode. Also, DELETE can only be used on files with RELATIVE or INDEXED organization (see example below).

RELATIVE and INDEXED files are discussed in the following section (File Organization).

The format of the DELETE statement is:

DELETE filename
ON INVALID KEY
{statements}
NOT ON INVALID KEY
{statements}
END-DELETE

ON INVALID KEY means the record was not found, so you might want to display an error message e.g. DISPLAY 'RECORD NOT FOUND'

To REWRITE you can refer to the level 01 name to change the record with the ammended field:

```
FD IN-FILE
01 RECORD-IN.
     03 IN-NAME    PIC X(20).
     03 IN-ADDRESS PIC X(60).

PROCEDURE DIVISION.
MAIN-PARAGRAPH.
```

```
  :

OPEN I-O IN-FILE
  :

READ IN-FILE

   IF IN-NAME = 'BILLY NOMATES' THEN
     MOVE 'JIMMY MOREPALS' TO IN-NAME
     REWRITE RECORD-IN
   ELSE
     DISPLAY IN-NAME
   END-IF
  :
```

To EXTEND you must open the file in EXTEND mode:

```
OPEN EXTEND IN-FILE
  :


  DISPLAY 'Type in new name'
  ACCEPT NEW-NAME
  MOVE NEW-NAME TO IN-NAME
  EXTEND IN-FILE

  DISPLAY 'Type in new address'
  ACCEPT NEW-ADDRESS
  MOVE NEW-ADDRESS TO IN-ADDRESS
  EXTEND IN-FILE
  :
```

Here is a sample program that deletes a record from an INDEXED file using the DELETE statement, followed by deletion of a record that does not use the DELETE statement but writes the whole file (less the record to be deleted) to a temporary file. The program asks for a six digit code that identifies the record to be removed from the file. If you want to try this program then you'll need to create a couple of test files: TESTDATA1.DAT and TESTDATA2.TXT.

TESTDATA1.DAT needs to be an indexed file. To create this you'll need to compile and run the Create INDEXED file program and Read INDEXED file program (both in the Sample Code section).

TESTDATA2.TXT should be LINE SEQUENTIAL and of the form:

```
CODE--SOME ENTRY OF 43 CHARACTERS
123456abc----------*********----------*********
:
:
```

```
000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID.  DELETION-EXAMPLE.
000030 AUTHOR. TIM-R-P-BROWN.
```

```
000040* Program that deletes a record from a
000050* file where the specified record ID code is entered
000060* by the user. 2 differing methods are used.
000070
000080 ENVIRONMENT DIVISION.
000090
000100 INPUT-OUTPUT SECTION.
000110 FILE-CONTROL.
000120
000130       SELECT IN-FILE-1 ASSIGN TO 'TESTDATA1.DAT'
000140         ORGANIZATION IS INDEXED
000150         ACCESS MODE IS DYNAMIC
000160         RECORD KEY IS RECORD-CODE-1.
000170       SELECT IN-FILE-2 ASSIGN TO 'TESTDATA2.TXT'
000180         ORGANIZATION IS LINE SEQUENTIAL.
000190       SELECT TEMP-FILE ASSIGN TO 'TEMP.TXT'
000200         ORGANIZATION IS LINE SEQUENTIAL.
000210
000220 DATA DIVISION.
000230 FILE SECTION.
000240
000250 FD IN-FILE-1.
000260 01 RECORD-1.
000270      03 RECORD-CODE-1    PIC X(6).
000280      03 RECORD-DETAILS-1 PIC X(43).
000290
000300 FD IN-FILE-2.
000310 01 RECORD-2.
000320      03 RECORD-CODE-2    PIC X(6).
000330      03 RECORD-DETAILS-2 PIC X(43).
000340
000350 FD TEMP-FILE.
000360 01 TEMP-RECORD.
000370      03 TEMP-CODE        PIC X(6).
000380      03 TEMP-DETAILS     PIC X(43).
000390
000400
000410
000420 WORKING-STORAGE SECTION.
000430
000440 01 END-OF-FILE-FLAG PIC X VALUE 'N'.
000450      88 EOF  VALUE 'Y'.
000460
000470 01 REC-DELETE-FLAG  PIC X VALUE 'N'.
000480      88 RECORD-DELETED VALUE 'Y'.
000490
000500 01 DEL-CODE  PIC X(6) VALUE SPACES.
000510
000520
000530
000540 PROCEDURE DIVISION.
000550
000560 MAIN-PARAGRAPH.
000570
000580     PERFORM FIRST-METHOD
000590     MOVE 'Y' TO END-OF-FILE-FLAG
000600     PERFORM SECOND-METHOD
000610     STOP RUN.
```

```
000620
000630***********************************************************
000640
000650 FIRST-METHOD.
000660* Paragraph that uses the DELETE to remove a record
000670
000680      DISPLAY 'Enter 6 digit code of record to be deleted'
000690      ACCEPT RECORD-CODE-1
000700      OPEN I-O IN-FILE-1
000710
000720
000730         DELETE IN-FILE-1
000740           INVALID KEY DISPLAY 'RECORD NOT FOUND'
000750
000760         END-DELETE
000770
000780
000790      CLOSE IN-FILE-1.
000800
000810***********************************************************
000820
000830 SECOND-METHOD.
000840* Paragraph that writes to a temporary file without
000850* including the record to be deleted
000860
000870      DISPLAY 'Enter 6 digit code of record to be deleted'
000880      ACCEPT DEL-CODE
000890      OPEN INPUT IN-FILE-2
000900           OUTPUT TEMP-FILE
000910
000920      MOVE 'N' TO REC-DELETE-FLAG
000930      MOVE 'N' TO END-OF-FILE-FLAG
000940
000950*----first write all records (except the selected one) to
000960*----the temporary file
000970      PERFORM UNTIL EOF
000980         READ IN-FILE-2
000990           AT END SET EOF TO TRUE
001000           NOT AT END
001010              IF RECORD-CODE-2 = DEL-CODE THEN
001020                   SET RECORD-DELETED TO TRUE
001030              ELSE
001040                   WRITE TEMP-RECORD FROM RECORD-2
001050              END-IF
001060         END-READ
001070      END-PERFORM
001080
001090
001100
001110      IF NOT RECORD-DELETED THEN
001120         DISPLAY 'Record not found'
001130      END-IF
001140
001150      CLOSE IN-FILE-2 TEMP-FILE
001160
001170      MOVE 'N' TO END-OF-FILE-FLAG
001180
001190*----now read all records from temp-file to a new 'TESTDATA-2.TXT'
```

```
001200*----This is virtually the same as just renaming the temporary file
001210*----when you think about it, just done the COBOL way!
001220      OPEN INPUT TEMP-FILE
001230           OUTPUT IN-FILE-2
001240*---------the original 'TESTDATA-2.TXT' will be overwritten-----*
001250
001260      PERFORM UNTIL EOF
001270         READ TEMP-FILE
001280           AT END SET EOF TO TRUE
001290           NOT AT END
001300              WRITE RECORD-2 FROM TEMP-RECORD
001310         END-READ
001320      END-PERFORM
001330
001340      CLOSE TEMP-FILE IN-FILE-2.
001350
001360***********************************************************
001370***********************************************************
```

7.3 File Organization

There are at least four ways in which the records on a file may be organised: SEQUENTIAL, LINE SEQUENTIAL, RELATIVE, AND INDEXED. When a file contains several records (hundreds or even thousands) if you only wanted to access one or two of them, it would waste processor time having to search an entire file in order to read them if stored in sequential or line sequential formats. Hence, relative and indexed files are of particular advantage.

Relative files

These files are organised so that a record can be accessed by referring to its position within the file, i.e. relative to other records. This is acheived by calculating the size (in characters, defined in the FD description) of each record and multiplying it by the required nth record....eh?? you ask. Consider the following program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.      RELATIVE-EXAMPLE.
AUTHOR           TRP BROWN.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

    ASSIGN IN-FILE TO 'A:CUSTOMER.DAT'
        ORGANIZATION IS RELATIVE
        ACCESS MODE IS DYNAMIC
        RELATIVE KEY IS ENTER-NUM.

DATA DIVISION.
FILE SECTION.
FD IN-FILE.
01 CUSTOMER-DETAILS.
    03 CUS-NAME      PIC X(20).
    03 CUS-CODE      PIC X(6).


WORKING-STORAGE SECTION.

01 ENTER-NUM         PIC 9(4).
```

```
PROCEDURE DIVISION.
MAIN-PARAGRAPH.
   OPEN INPUT IN-FILE
   DISPLAY 'ENTER CUSTOMER NUMBER'
   ACCEPT ENTER-NUM

   READ IN-FILE
      INVALID KEY DISPLAY 'RECORD NOT FOUND'
      NOT INVALID KEY
         DISPLAY 'CUSTOMER NAME: ' CUS-NAME
         DISPLAY 'CODE:          ' CUS-CODE
   END-READ

   CLOSE IN-FILE
   STOP RUN.
```

- In the environment division, the assign clause contains a number of extra words. The organization is RELATIVE. This is followed by **ACCESS MODE IS DYNAMIC**. This means that the file can be read sequentially or RANDOMLY, i.e. direct access whereby the computer can calculate where to look for the required record. You alternatively use **ACCESS MODE IS RANDOM** but this doesn't allow for a sequential access option (so what's the point using it...?).
- The next line **RELATIVE KEY IS ENTER-NUM** refers to this item defined in working storage that will contain the record number required. When the number is entered into the keyboard (ACCEPT ENTER-NUM), the computer will multiply this number (minus 1) by the size of the record (CUSTOMER-DETAILS containing 26 characters):
  e.g.
  (102-1) * 26 = 2626 characters into the file will be immediately followed by the 102nd record.
- The read statement, rather than using AT END and NOT AT END, uses **INVALID KEY** and **NOT INVALID KEY**. Here these depend on whether the file has been found or not.
- The PIC size of ENTER-NUM is 9(4), which limits the file to 9,999 records but you could increase this if you wanted.
- It should be noted that you are not allowed to use an item defined in the FD as a relative key.

Indexed files

An indexed file contains records that, unlike relative files, do not require the key to be numeric. Look at the following code (similar to the above code):

```
IDENTIFICATION DIVISION.
PROGRAM-ID.      INDEXED-EXAMPLE.
AUTHOR           TRP BROWN.
```

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

    ASSIGN IN-FILE TO 'A:CUSTOMER.DAT'
        ORGANIZATION IS INDEXED
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS CUS-CODE.

DATA DIVISION.
FILE SECTION.
FD IN-FILE.
01 CUSTOMER-DETAILS.
    03 CUS-CODE      PIC X(6).
    03 CUS-NAME      PIC X(20).


WORKING-STORAGE SECTION.

01 ENTER-NUM         PIC 9(4).

PROCEDURE DIVISION.
MAIN-PARAGRAPH.
   OPEN INPUT IN-FILE
   DISPLAY 'ENTER CUSTOMER NUMBER'
   ACCEPT ENTER-NUM

   READ IN-FILE
      INVALID KEY DISPLAY 'RECORD NOT FOUND'
      NOT INVALID KEY
         DISPLAY 'CUSTOMER NAME: ' CUS-NAME
         DISPLAY 'CODE:          ' CUS-CODE
   END-READ

   CLOSE IN-FILE
   STOP RUN.
```

The main differences between this example and the relative file example are that (1) the term RECORD KEY is used, rather than RELATIVE KEY; (2) any field can be used from the record. However, the field must be unique otherwise a *duplicate key* error would occur. Rather than directly access the file, as in relative file access, the computer searches a separate index file that contains pointers to the position of the actual record on the indexed (data) file. The field in the index, whether numeric or alphanumeric, must be in strict ascending order (ASCII characters are ordered according to their ASCII value, e.g. A < B etc...).

In order to read the indexed file sequentially, but in ascending order on the key field, the verb **START** is used. For the above example:

```
OPEN INPUT IN-FILE
START IN-FILE
KEY GREATER THAN 'D23301'
  INVALID KEY
     DISPLAY 'NO MORE RECORDS BEYOND THIS POINT'
```

```
   NOT INVALID KEY
      ....statements to process rest of file e.g. READ within a loop
 END-START
```

To differentiate between a sequential READ and a random READ when using DYNAMIC access mode, you would use the statements READ (with INVALID KEY) for random read, and **READ...NEXT** for sequential read (with AT END) e.g. :

```
*Random read
     MOVE 'E11323' TO CODE-NUM
     READ IN-FILE
        INVALID KEY DISPLAY 'CODE NOT FOUND'
     END-READ
*Sequential read
     READ IN-FILE NEXT
        AT END MOVE 'Y' TO EOF-FLAG
     END-READ
```

7.4 SORT and MERGE

If you wished to take a file of unordered records and produce a new file of these records sorted into ascending or descending order of a field you would use **SORT**. The League table program in the Sample code section uses this utility to generate a league table from updated records from a data file, sorted principally by descending points.

Consider this segment of code from this program:

```
000050 ENVIRONMENT DIVISION.
000060 INPUT-OUTPUT SECTION.
000070 FILE-CONTROL.
000080     SELECT TEAM-REC-IN ASSIGN TO "INPUT.REC"
000090     ORGANIZATION IS SEQUENTIAL.
000100     SELECT WORK-FILE ASSIGN TO SORTWK01.
000110     SELECT SORT-OUT ASSIGN TO "SORTED.REC"
000120     ORGANIZATION IS SEQUENTIAL.
000130     SELECT PRINT-FILE ASSIGN TO "PRINTOUT.TXT".
000140
000150
000160 DATA DIVISION.
000170 FILE SECTION.
000180 FD TEAM-REC-IN.
000190 01 TEAM-REC.
000200     03  TEAM-CODE   PIC XXX.
000210     03  TEAM-NAME   PIC X(20).
000220     03  PLAYED  PIC 99.
000230     03  GOALS-FOR   PIC 99.
000240     03  GOALS-AGST  PIC 99.
000250     03  G-WON   PIC 99.
000260     03  G-LOST  PIC 99.
000270     03  G-DRAWN PIC 99.
000280     03  GOAL-DIFF   PIC S99 SIGN LEADING SEPARATE.
000290     03  POINTS  PIC 99.
```

```
000300
000310 SD WORK-FILE.
000320 01 WORK-REC.
000330     03  TEAM-CODE-KEY PIC XXX.
000340     03                PIC X(22).
000350     03  GF-KEY        PIC 99.
000360     03                PIC X(8).
000370     03  GD-KEY        PIC S99  SIGN LEADING SEPARATE.
000380     03  POINTS-KEY    PIC 99.
000390
```

In addition to the FD for the TEAM-REC-IN (the main data file) there is also a WORK-FILE that the computer uses for sorting. Here it is assigned to SORTWK01, required for the Fujitsu COBOL compiler, but for MicroFocus you might code ASSIGN TO DISK or even ASSIGN TO "B:TEMPFILE".

The WORK-FILE does not have a FD descriptor, but rather, has a sort descriptor **SD**.

```
003310 SORT-TABLE.
003320     SORT WORK-FILE
003330         ON DESCENDING KEY POINTS-KEY GD-KEY GF-KEY
003340         USING TEAM-REC-IN
003350         GIVING SORT-OUT.
```

The SORT-TABLE paragraph then sorts the data file TEAM-REC-IN as shown above. Note that the SORT verb is followed by WORK-FILE and that TEAM-REC-IN is refered to with USING...

Since it is common for two teams to have the same number of points then, the DESCENDING KEY first attempts to sort by points (POINTS-KEY) but if these match then they are than sorted by goal difference (GD-KEY) and then by goals scored (GF-KEY). If these all match then the teams will be placed as the appear from the data file (for TEAM-REC-IN I placed them in alphabetical order).

SORT-OUT is the destination of the sorted data where the new league table would appear.

Note, a file that is to be sorted if already open, must be closed prior to sorting. THE SORT STATEMENT WILL AUTOMATICALLY OPEN THE UNSORTED FILE WHEN EXECUTED.

Merge

To merge two sorted files into a single sorted file, the **MERGE** statement is used:

```
MERGE WORK-FILE
   ON ASCENDING KEY CUS-CODE-KEY
   USING FILE-A
         FILE-B
   GIVING MERGED-FILE
```

You can merge more than 2 files if you wish. An **SD** would be required as used with a SORT.

7.5 INPUT and OUTPUT PROCEDURE

The SORT statement above sorted all the records in the file into a new file. But if you wanted to produce a sorted file that only contained, for example, product numbers which begin with a '1', you would use an **INPUT PROCEDURE**.

The record FD might be:

```
FD UNSORTED-FILE.
01 UNSORTED-RECORD.
     03 1ST-DIGIT-OF-CODE  PIC 9.
     03                    PIC X(20).
```

The description gives the minimum detail required. Now some procedure division:

```
PROCEDURE DIVISION.
SORT-SELECT.
     SORT WORK-FILE
         ON DESCENDING KEY PRODUCT-NO
         INPUT PROCEDURE SELECT-PROD-CODE
         GIVING SORTED-CODES-FILE
     STOP RUN.
```

The INPUT PROCEDURE clause acts like a PERFORM, indicating the logic to go to a different paragraph (i.e. procedure).

So the paragraph SELECT-PROD-CODE might be like this:

```
SELECT-PROD-CODE.
    OPEN INPUT UNSORTED-DATA-FILE
    PERFORM UNTIL END-OF-FILE
      READ UNSORTED-DATA-FILE
        AT END MOVE 'Y' TO EOF-FLAG
        NOT AT END
          IF 1ST-DIGIT-OF-CODE = 1 THEN
            MOVE UNSORTED-RECORD TO WORK-REC
            RELEASE WORK-REC
          END-IF
      END-READ
    END-PERFORM
    CLOSE UNSORTED-DATA-FILE
```

When the if condition is true, the record is moved to the work-file (WORK-REC is the level 01 name) by the RELEASE verb, even though the MOVE verb appears first (I dunno why..!). Unlike a simple SORT, you DO have to OPEN the unsorted file prior to an input procedure.

OUTPUT PROCEDURE

If you just want to print specific sorted fields you would use an **OUTPUT PROCEDURE**. Based on the above

example:

```
PROCEDURE DIVISION.
PRINT-SORT-REC.
     SORT WORK-FILE
         ON DESCENDING KEY PRODUCT-NO
         USING UNSORTED-RECORD
         OUTPUT PROCEDURE PRINT-SELECT-PROD-CODE
     STOP RUN.
```

The INPUT PROCEDURE clause acts like a PERFORM, indicating the logic to go to a different paragraph (i.e. procedure).

So the paragraph SELECT-PROD-CODE might be like this:

```
SELECT-PROD-CODE.
    OPEN OUTPUT PRINT-FILE
    PERFORM UNTIL END-OF-FILE
      RETURN UNSORTED-DATA-FILE
        AT END MOVE 'Y' TO EOF-FLAG
        NOT AT END
          {move fields in SD sort group to print fields}...
          WRITE PRINT-RECORD FROM {print group}
      END-RETURN
    END-PERFORM
    CLOSE PRINT-FILE.
```

Instead of READ you use RETURN and then WRITE the record to the printer rather than RELEASE the record to a file.

You can combine INPUT and OUTPUT procedures into the same sort statement by replacing both the USING and GIVING statements:

```
SORT WORK-FILE
   ON DESCENDING KEY PRODUCT-NO
   INPUT PROCEDURE SELECT-PROD-CODE
   OUTPUT PROCEDURE PRINT-SELECT-PROD-CODE
STOP RUN.
```

7.6 FILE STATUS (error handling)

A number of errors can occur that result from file input/output that programmer may wish to be able to deal with in order to avoid unexpected program termination.

Run time errors can arise quite easily from a file not being available to open, or if present the data is corrupted. Furthermore, what if there is no more disk space available or not enough space has been allocated to allow for addtition of new data. Other errors, such as attempting to close a file that isn't open, or to read a file opened for output only, may well derive from logical errors (that is, programming mistakes) but can be dealt with nonetheless when debugging. These kinds of errors will normally result in termination of the program run, whereas using File Status can allow the programmer to deal with any such problems without the

program run stopping and returning to the operating system.

File Status Codes are made of two digits, the first indicates one one of 5 classes:

0	Input/output operation successful
1	File "at end" condition
2	Invalid key
3	Permanent I/O error
4	Logic error

The second digit refers to the particular case within the class. Here are examples common to both Microfocus and Fujitsu compilers (although there are more besides). I would check your compiler documentation.

Code	Meaning
00	Input/output operation successful
02	Duplicate record key found (READ ok)
04	Length of record too large (READ ok)
10	File AT END
14	"The valid digits of a read relative record number are greater than the size of the relative key item of the file." *from Fujitsu manual - I'm not sure I what that means!*
16	Program tries to read file already AT END *note: Fujitsu compiler returns code "46" in this case*
22	Program attempts to write a record with a key that already exists
23	Record not found
24	Program attempts to write record to a disk that is full
30	Input/output operation unsuccessful, no further information available
34	Program attempts to write record to a disk that is full

35	Program tries to open non-existant file for INPUT, I-O or EXTEND
37	Program tries to open line sequential file in I-O mode
41	Program tries to open file that is already open
42	Program tries to close file that is not open
43	Program tries to delete or rewrite a record that has not been read
44	Program tries to write or rewrite a record of incorrect length
46	Program tries to read a record where the previous read or START has failed or the AT END condition has occurred
47	Program tries to read a record from a file opened in the incorrect mode
48	Program tries to write a record from a file opened in the incorrect mode
49	Program tries to delete or rewrite a record from a file opened in the incorrect mode

To use these codes you need to include the FILE STATUS clause in the SELECT statement of the environment division:

```
     SELECT TEST-FILE ASSIGN TO 'TEST-DATA.DAT'
       ORGANIZATION IS SEQUENTIAL
       FILE STATUS IS W-STATUS.
```

Of course W-STATUS could any user name you like. It *must* however be defined in working storage as **PIC XX**, i.e. as alpha numeric and not numeric. So, if during a program run a certain input/output error occurs, rather than the program terminate, the program will simply produce an error status.
You might code:

```
* Here a possible danger of too big a record being moved into W-RECORD
     READ RECORD-IN INTO W-RECORD
       IF W-STATUS = "04" THEN
          DISPLAY "Over-sized record has been read"
          SET REC-XS-FLAG TO TRUE
       END-IF
```

Another example might be, when reading from an indexed file:

```
READ IN-FILE
  IF W-STATUS = "23" THEN
     DISPLAY "Record not found"
  ELSE PERFORM MAIN-PROCESS
```

You could have easily have written:

```
READ IN-FILE
  INVALID KEY
     DISPLAY "Record not found"
  NOT INVALID KEY PERFORM MAIN-PROCESS
END-READ
```

So consider which is the best option and remember not to try and do both.

For Fujitsu compilers at least, although the program run is not terminated, the Fujitsu WINEXE enviroment will still produce a prompt indicating the error (with more detailed error codes). I'm not sure, but I suspect that this facility can be disabled. Check the user manual.

ZingCOBOL Copyright Timothy R P Brown 2003

---

8. Debugging COBOL code

So you've written your program, finally got it to compile after sorting out all those syntax errors and undefined variables and the rest. So you execute the program and Hey Presto! ... nothing happens, you get a runtime error, or worst of all, your computer locks up and you're reaching for CTRL+ALT+DEL. So what went wrong?

Don't worry. The next thing to do after writing your wonderfully crafted program is to fix all the bugs, that is, all the errors in the code that lie hidden in the logic beyond the reach of the compiler. Here are a few personal hints and tips of mine to set about debugging your program (and avoiding errors in the first place), or at least how I go about getting my code to do what I want it to do.

i. Before a line of code is written...preparation
ii. Commenting
iii. Variable names
iv. Break it up
v. "Stubs"
vi. Watching variables
vii. Debugging tools

(i) Before a line of code is written...preparation

The best way to avoid spending hours trying to untangle a mass of complex code (that you brilliantly typed into the computer straight from the top of your head) is PREPARATION. By that I mean: (1) be absolutely clear about what you want the program to do - know what the inputs and outputs are, (2) write a very broad algorithm and gradually refine as far as you can using pseudo-code, (3) draw a flow chart or structure chart that matches the pseudo-code, and (4) translate the flow chart into actual COBOL.

One issue is: Do I write the PROCEDURE DIVISION first and then go back and write the DATA DIVISION? This would seem a fairly sensible thing to do except that in practise you find you will forget to declare a whole slew of variables. A further point is that a good deal of COBOL involves doing things to the data that rely on what's been declared in the DATA DIVISION. Again, I would suggest preparing a fair proportion of the data definitions on paper first. When I write a program I write as much DATA DIVISION as possible before starting on the PROCEDURE DIVISION. Then, as I proceed through the code I keep going back to the DATA DIVISION to update it as soon as possible.

(ii) Commenting

Liberally sprinkle you code with comments that explain exactly what each fragment of code is meant to do. This really helps when trying figure out what's going on.

(iii) Variable names

Use variable names that are meaningful and stick to a standard format. For example, some people use a prefix before variable names to indicated the general function of the variable, such as printing variables:

```
01   PRINT-OUTPUT.
    03  P-NAME      PIC X(20).
    03  P-ADDRESS   PIC X(50).
```

```
       03  P-CUS-CODE PIC 9(6).
       03  P-PAGE-COUNT PIC 999.
```

This example uses "P-" before each name to indicate a member of the print output group. Sometimes the prefix "WS-" is used to indicate WORKING-STORAGE, "L-" for LINKAGE SECTION variables. Notice that the names are also meaningful. While you may spend longer typing out longer names you'll thank youself when it comes to fixing bugs.

Care should be taken when deciding names that you spell them correctly (PAGE-COUNTRE) and/or consistantly (RECORD-NUM and RECORD-NO) and that you don't try to use singular and plural names (CUSTOMER-TOTAL and CUSTOMER-TOTALS).

(iv) Break it up

Breaking your code into smaller procedures (i.e. paragraphs) not only makes the program easier to read, but easier spot where problems are arising.

(v) "Stubs"

One way to monitor what is going on when you run your program is to place "stubs" at important points in the logic. By stubs I mean a DISPLAY statement that tells you the that certain position in the logic has been executed:

```
   MAIN-PARAGRAPH.
        DISPLAY 'IN MAIN PARAGRAPH'
        PERFORM INIT-PARAGRAPH.
        PERFORM RECORD-READ-PARAGRAPH
            UNTIL NO-MORE-RECORDS
        PEFORM TERMINATE-RUN
        DISPLAY 'PROGRAM ENDING'
        STOP RUN.
```

If you have a DISPLAY at the beginning of each paragraph then the console window might look something like this:

```
IN MAIN PARAGRAPH
IN INIT-PARAGRAPH
IN RECORD-READ-PARAGRAPH
IN RECORD-READ-PARAGRAPH
IN RECORD-READ-PARAGRAPH
IN RECORD-READ-PARAGRAPH
IN RECORD-READ-PARAGRAPH
IN RECORD-READ-PARAGRAPH
IN RECORD-READ-PARAGRAPH
IN RECORD-READ-PARAGRAPH
IN RECORD-READ-PARAGRAPH
IN RECORD-READ-PARAGRAPH
IN RECORD-READ-PARAGRAPH
IN TERMINATE-PARAGRAPH
PROGRAM ENDING
```

You must remember to remove all of the stubs when the program is fully debugged.

(vi) Watching variables
Of course, just putting little flags say "IN PARAGRAPH XYZ" can be extended further to display the value of certain important variables:

```
DISPLAY "P-COUNTER = " P-COUNTER
```

Again, don't forget to remove them (or comment them out) when your done.

(vii) Debugging tools
Both the Fujitsu COBOL85 and Microfocus Personal COBOL compilers (and presumably other too) have debugging utilities. Most significant are the ability to *animate* the program and set *breakpoints* throughout your code. Animating your code allows you to view each line of code as the debugger steps through the program. You can pause the run at any point and check the value of variables. During this process you can specify variables that you want to watch throughout the run. Animating a program run can prove a bit tedious if large amounts of iterations are involved.

An alternative is to set breakpoints. By doing so the program run will pause at defined breakpoints (wherever you want them) to allow you to check the value of variables.

You should check your compiler documentation to find out how to use debugging utilites: for large programs they are well worth the effort.

* * *

When it comes to actually fixing errors try to avoid "hacks", that is, adding bits of code to correct erroneous data values rather than trying to find out why the data was wrong in the first place. You may find yourself getting bound up in ever more complex arrays of Boolean flags to allow certain conditions: e.g.

```
IF (X = Y) AND (Z >= W) AND ((A = B) OR (A <> C)) AND (D < F) THEN...scream..?
```

This being the case, see if your logic couldn't be better designed. Sometimes going back to the drawing board (more than once) is the best strategy in the long run. Like a famous chess grandmaster once said (I don't know who) "If you see a good move, look for a better one" : if you think of a good way of coding something, look for a better alternative (not as snappy ?!).

Something to keep in mind at all times is that one day someone other than yourself may have to read and understand (and perhaps modify) your code. Whether this is true or not it is a good habit to get into because it makes you write better code. And, as a software professional, this will almost certainly be the case.

ZingCOBOL Copyright Timothy R P Brown 2003

10. Sample COBOL code

10.1 Add line number program
10.2 Refresh line numbers program
10.3 League table program
10.4 Calculate prime numbers program
10.5 Create INDEXED file program
10.6 Read INDEXED file program

The sample code here was written while learning COBOL so they aren't particularly well structured. Also, they are not the usual type of COBOL program that you would normally come across. COBOL is more likely written for business applications such as payroll programs or stock control etc... Hopefully they might give an indication of how COBOL works.

10.1 Add line numbers program

This program is designed to add line numbers to COBOL code that has been typed into a text editor (e.g. Notepad) in the following format:

```
   :
PROCEDURE DIVISION.

MAIN-PARAGRAPH.
      MOVE X TO Y
*the comment asterisk will be placed in position 7
/as will the page break solidus
      IF Y > Z THEN
          ADD Z TO X
          MOVE X TO Z
      ELSE DISPLAY 'The hypen for continuing a string
-              'onto the next line also goes into position 7'
      END-IF
*all other text is placed from position 8
*so you still need to indent where required
STOP RUN.

*lastly, there is a limit of about
*70 characters per line (from position 8)
```

The text file containing COBOL code as above should be call named **input.txt**. Following execution, the program will produce a new file called **output.cob** although it will still be a simply text file, but can be compiled. The output.cob file for the above code would be:

```
   :
000010 PROCEDURE DIVISION.
000020
000030 MAIN-PARAGRAPH.
000040     MOVE X TO Y
000050*the comment asterisk will be placed in position 7
000060/as will the page break solidus
000070     IF Y > Z THEN
000080        ADD Z TO X
000090        MOVE X TO Z
000100     ELSE DISPLAY 'The hypen for continuing a string
```

```
000110-              'onto the next line also goes into position 7'
000120     END-IF
000130*all other text is placed from position 8
000140*so you still need to indent where required
000150     STOP RUN.
000160
000170*lastly, there is a limit of about
000180*70 characters per line (from position 8)
```

```
000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. LINE-NO-PROG.
000030 AUTHOR.     TIM R P BROWN.
000040****************************************************
000050* Program to add line numbers to typed code        *
000060* Allows for comment asterisk, solidus, or hyphen ,*
000070* moving it into position 7.                       *
000080*                                                  *
000090****************************************************
000100
000110 ENVIRONMENT DIVISION.
000120 INPUT-OUTPUT SECTION.
000130 FILE-CONTROL.
000140     SELECT IN-FILE ASSIGN TO 'INPUT.TXT'
000150        ORGANIZATION IS LINE SEQUENTIAL.
000160     SELECT OUT-FILE ASSIGN TO 'OUTPUT.COB'
000170        ORGANIZATION IS LINE SEQUENTIAL.
000180
000185*****************************************************
000187
000190 DATA DIVISION.
000200 FILE SECTION.
000210
000220 FD IN-FILE.
000230 01 LINE-CODE-IN.
000240     03 CHAR-1       PIC X.
000250     03 CODE-LINE    PIC X(110).
000260
000270 FD OUT-FILE.
000280 01 LINE-CODE-OUT    PIC X(120).
000290
000300
000310 WORKING-STORAGE SECTION.
000320
000330 01 EOF-FLAG     PIC X VALUE 'N'.
000340     88 END-OF-FILE        VALUE 'Y'.
000350
000360 01 NUMBER-CODE.
000370     03 L-NUM-CODE    PIC 9(6) VALUE ZEROS.
000380     03 B-SPACE   PIC X VALUE SPACE.
000390     03 L-CODE    PIC X(100) VALUE SPACES.
000400
000410 01 NUMBER-COMMENT.
000420     03 L-NUM-COM     PIC 9(6) VALUE ZEROS.
000430     03 L-COMMENT     PIC X(100) VALUE SPACES.
000440
000450 01 LINE-NUMBER      PIC 9(6) VALUE ZEROS.
```

```
000460
000470
000480*****************************************************
000490
000500 PROCEDURE DIVISION.
000510
000510 MAIN-PARA.
000520     OPEN INPUT IN-FILE
000530          OUTPUT OUT-FILE
000535
000540     PERFORM UNTIL END-OF-FILE
000550       ADD 10 TO LINE-NUMBER
000560       READ IN-FILE AT END
000570         MOVE 'Y' TO EOF-FLAG
000580       NOT AT END
000590         IF      (CHAR-1 = '*')
000600                  OR (CHAR-1 = '/')
000610                  OR (CHAR-1 = '-') THEN
000620            MOVE LINE-CODE-IN TO L-COMMENT
000630            MOVE LINE-NUMBER TO L-NUM-COM
000640            WRITE LINE-CODE-OUT FROM NUMBER-COMMENT
000660         ELSE
000670            MOVE LINE-CODE-IN TO L-CODE
000680            MOVE LINE-NUMBER TO L-NUM-CODE
000690            WRITE LINE-CODE-OUT FROM NUMBER-CODE
000720         END-IF
000730       END-READ
000740       INITIALIZE NUMBER-CODE NUMBER-COMMENT
000750     END-PERFORM
000760
000770     CLOSE IN-FILE OUT-FILE
000780     STOP RUN.
```

10.2 Refresh line numbers program

This program is designed to refresh COBOL code line numbers following editing that would result in uneven line number increases (or even no line number at all) where lines have been inserted or deleted.

```
00010 IDENTIFICATION DIVISION.
00020 PROGRAM-ID.    RENUMBER-PROG.
00030 AUTHOR.        TIMOTHY R P BROWN.
00040
00045*****************************************************
00050* Program to refresh numbers to typed code          *
00060* Allows for comment all characters at position 7    *
00065*****************************************************
00070
00080
00090 ENVIRONMENT DIVISION.
00100 INPUT-OUTPUT SECTION.
00110 FILE-CONTROL.
00120        SELECT IN-FILE ASSIGN TO 'INPUT.COB'
00130        ORGANIZATION IS LINE SEQUENTIAL.
00140        SELECT OUT-FILE ASSIGN TO 'RENUM.COB'
00150        ORGANIZATION IS LINE SEQUENTIAL.
00160
00170 DATA DIVISION.
```

```
00180 FILE SECTION.
00190
00200 FD IN-FILE.
00210 01 CODE-IN.
00230        03 OLD-NUM  PIC 9(6).
00240        03 IN-CODE  PIC X(150).
00250
00260 FD OUT-FILE.
00270 01 CODE-OUT        PIC X(91).
00280
00290
00300 WORKING-STORAGE SECTION.
00310
00320 01 EOF-FLAG      PIC X VALUE 'N'.
00330        88 END-OF-FILE  VALUE 'Y'.
00340
00350
00360 01 W-RENUMBER-CODE.
00370        03 W-NUM   PIC 9(6) VALUE ZEROS.
00380    03 W-CODE      PIC X(85) VALUE SPACES.
00390
00400 01 LINE-NUMBER    PIC 9(6) VALUE ZEROS.
00403
00407*****************************************************
00410
00420 PROCEDURE DIVISION.
00430 MAIN-PARA.
00440        OPEN INPUT IN-FILE
00450             OUTPUT OUT-FILE
00460
00470        PERFORM UNTIL END-OF-FILE
00480           ADD 10 TO LINE-NUMBER
00490           READ IN-FILE
00495              AT END MOVE 'Y' TO EOF-FLAG
00500              NOT AT END
00510                MOVE IN-CODE TO W-CODE
00520                MOVE LINE-NUMBER TO W-NUM
00530                WRITE CODE-OUT FROM W-RENUMBER-CODE
00550           END-READ
00570        END-PERFORM
00580
00590        CLOSE IN-FILE OUT-FILE
00600        STOP RUN.
```

10.3 League table program

This program is designed to update a football league table and print out a table when any scores have been added. The diplay prompts the user to input the score from a game. The points for each team involved are updated, as are the goals for, against and difference. The program will search the data file and update the relevant team record. When score input is complete, the program then sorts the data into a temporary file before printing out an updated league table. An OUTPUT PROCEDURE could have been used instead of producing a temporary sorted file.

This program would probably benefit from using an indexed file for the team records rather than searching the sequentail file, as done here.

This code is written for the 1999-2000 season of the English FA Premiership. The team data is stored on a sequential file in alphabetical order. If you wish to download a copy of this data file (with mostly fictional scores etc..) click here and a better program description click here.

```
000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID. TABLE-PROG.
000030 AUTHOR.     TIMOTHY R P BROWN.
000035
000037*************************************************
000040* Program to update a football league table         *
000045* and output a new updated table                    *
000046* Based on English Premiership season 1999-2000     *
000047*************************************************
000048
000050 ENVIRONMENT DIVISION.
000060 INPUT-OUTPUT SECTION.
000070 FILE-CONTROL.
000080     SELECT TEAM-REC-IN ASSIGN TO "INPUT.REC"
000090        ORGANIZATION IS SEQUENTIAL.
000100     SELECT WORK-FILE ASSIGN TO SORTWK01.
000105* for MicroFocus compiler
000107* replace SORTWK01 with 'WORKFILE.DAT'
000110     SELECT SORT-OUT ASSIGN TO "SORTED.REC"
000120        ORGANIZATION IS SEQUENTIAL.
000130     SELECT PRINT-FILE ASSIGN TO PRINTER.
000140
000150
000160 DATA DIVISION.
000170 FILE SECTION.
000180 FD TEAM-REC-IN.
000190 01 TEAM-REC.
000200     03  TEAM-CODE   PIC XXX.
000210     03  TEAM-NAME   PIC X(20).
000220     03  PLAYED      PIC 99.
000230     03  GOALS-FOR   PIC 99.
000240     03  GOALS-AGST  PIC 99.
000250     03  G-WON       PIC 99.
000260     03  G-LOST      PIC 99.
000270     03  G-DRAWN     PIC 99.
000280     03  GOAL-DIFF   PIC S99 SIGN LEADING SEPARATE.
000290     03  POINTS      PIC 99.
000300
000310 SD WORK-FILE.
000320 01 WORK-REC.
000330     03  TEAM-CODE-KEY PIC XXX.
000340     03                PIC X(22).
000350     03  GF-KEY        PIC 99.
000360     03                PIC X(8).
000370     03  GD-KEY        PIC S99  SIGN LEADING SEPARATE.
000380     03  POINTS-KEY    PIC 99.
000390
000400
000410 FD PRINT-FILE.
000420 01 TEXT-OUT     PIC X(60).
000430
000440 FD SORT-OUT.
000450 01 TEAM-REC-OUT.
000460     03  STEAM-CODE  PIC XXX.
000470     03  STEAM-NAME  PIC X(20).
000480     03  SPLAYED     PIC 99.
000490     03  SGOALS-FOR  PIC 99.
000500     03  SGOALS-AGST PIC 99.
```

```
000510     03  SG-WON      PIC 99.
000520     03  SG-LOST     PIC 99.
000530     03  SG-DRAWN    PIC 99.
000540     03  SGOAL-DIFF  PIC S999.
000550     03  SPOINTS     PIC 99.
000560
000570
000580
000590
000600 WORKING-STORAGE SECTION.
000610
000620 01 M              PIC 99.
000630 01 REAL-GOAL-DIFF PIC S999.
000640
000650 01 W-DATE.
000660     03  W-YEAR    PIC 99.
000670     03  W-MON     PIC 99.
000680     03  W-DAY     PIC 99.
000690
000700
000710 01 SCORE.
000720     03 W-H-SCR    PIC 9.
000730     03            PIC X VALUE "-".
000740     03 W-A-SCR    PIC 9.
000750
000760 01 P-TITLE.
000770     03            PIC X(5) VALUE SPACES.
000780     03 TAB-TITLE  PIC X(34)
000790         VALUE "The English FA Premier League".
000800
000810     03 P-DATE.
000820         05 P-DAY  PIC XX.
000830         05        PIC X VALUE "/".
000840         05 P-MON  PIC XX.
000850         05        PIC X VALUE "/".
000860         05 P-YEAR PIC XX.
000870
000880 01 P-UNDERLINE    PIC X(45) VALUE ALL "-".
000890 01 P-GAP          PIC X VALUE SPACE.
000900
000910 01 P-HEADER.
000920     03          PIC X(6) VALUE SPACES.
000930     03 TAB-TEAM PIC X(4) VALUE "TEAM".
000940     03          PIC X(11) VALUE SPACES.
000950     03 PLY      PIC X(5) VALUE "Playd".
000960     03          PIC X VALUE SPACE.
000970     03 WO       PIC XXX VALUE "Won".
000980     03          PIC X VALUE SPACE.
000990     03 DR       PIC XXXX VALUE "Drwn".
001000     03          PIC X VALUE SPACE.
001010     03 LO       PIC XXXX VALUE "Lost".
001020     03          PIC X VALUE SPACE.
001030     03 GF       PIC XXX VALUE "For".
001040     03          PIC X VALUE SPACE.
001050     03 GA       PIC X(5) VALUE "Agnst".
001060     03          PIC X VALUE SPACE.
001070     03 GD       PIC XX VALUE "GD".
001080     03          PIC X VALUE SPACE.
```

```
001090     03 PTS      PIC XXX VALUE "PTS".
001100
001110 01 W-TEXT-OUT.
001120     03  P-TAB-POS   PIC 99.
001130     03              PIC X VALUE SPACE.
001140     03  P-TEAM      PIC X(20).
001150     03  P-PLAYED    PIC 99.
001160     03              PIC XXX VALUE SPACES.
001170     03  P-G-WON     PIC Z9.
001180     03              PIC XX VALUE SPACES.
001190     03  P-G-DRAWN   PIC Z9.
001200     03              PIC XXX VALUE SPACES.
001210     03  P-G-LOST    PIC Z9.
001220     03              PIC XXX VALUE SPACES.
001230     03  P-GOALS-FOR PIC 99.
001240     03              PIC XX VALUE SPACES.
001250     03  P-GOALS-AGST  PIC 99.
001260     03              PIC XX VALUE SPACES.
001270     03  P-GOAL-DIFF PIC ZZ9.
001280     03              PIC XX VALUE SPACES.
001290     03  P-POINTS    PIC Z9.
001300
001310
001320 01 SCORE-TAB.
001330     03 TAB-SCORE    PIC 9 OCCURS 2.
001340 01 T-POINTS-TAB.
001350     03 T-POINTS     PIC 99 OCCURS 20.
001360 01 POINTS-TAB.
001370     03  TAB-POINTS  PIC 9 OCCURS 2.
001380 01 T-G-FOR-TAB.
001390     03 T-G-FOR      PIC 99 OCCURS 20.
001400 01 T-G-AGST-TAB.
001410     03 T-G-AGST     PIC 99 OCCURS 20.
001420 01 T-G-DIFF-TAB.
001430     03 T-G-DIFF     PIC 99 OCCURS 20.
001440 01 TAB-TEAM-NAME.
001450     03  TEAM        PIC XXX OCCURS 2.
001460
001470
001480 01 V-TEAM-FLAG      PIC X.
001490     88  V-TEAM      VALUE "Y".
001500 01 V-SCORE-FLAG     PIC X.
001510     88  V-SCORE     VALUE "Y".
001520 01 SORT-ONLY-FLAG   PIC X.
001530     88 SORT-ONLY    VALUE "Y".
001540
001550
001560 01 ENDING-KEY       PIC X VALUE SPACE.
001570 01 SWITCH           PIC 9.
001580 01 EOF-FLAG         PIC X VALUE "N".
001590 01 COUNTER          PIC 99.
001600 01 W-GOAL-DIFF      PIC 99.
001610 01 LAST-SCORE       PIC X.
001620 01 N                PIC 99.
001630
001640***********************************************************
001650
001660 PROCEDURE DIVISION.
```

```
001670
001680 MAIN-PARAGRAPH.
001690
001700     PERFORM DISPLAY-INSTRUCTIONS
001710     PERFORM INPUT-DATA
001720     PERFORM SORT-TABLE
001730     PERFORM PRINT-TABLE
001740     DISPLAY " Type Q or X to exit program."
001750     ACCEPT ENDING-KEY
001760     STOP RUN.
001770*****************************************************************
001780 DISPLAY-INSTRUCTIONS.
001790     DISPLAY "   Instructions"
001800     DISPLAY " "
001810     DISPLAY " Following prompts, enter the first "
001820     DISPLAY "3 letters of the team in lower case. "
001830     DISPLAY " Then enter the score (home team score first)."
001840     DISPLAY " To perform SORT ONLY function, type 'xxx' "
001850     DISPLAY "at both team prompts. ".
001860*****************************************************************
001870 INPUT-DATA.
001880     MOVE "n" TO LAST-SCORE
001890     MOVE "N" TO SORT-ONLY-FLAG
001900     PERFORM UNTIL LAST-SCORE = "y" OR "Y"
001910        MOVE "N" TO V-SCORE-FLAG
001920        MOVE "N" TO V-TEAM-FLAG
001930        PERFORM UNTIL V-TEAM
001940            DISPLAY "INPUT HOME TEAM >"
001950            ACCEPT TEAM (1)
001960            DISPLAY "INPUT AWAY TEAM >"
001970            ACCEPT TEAM (2)
001980            PERFORM VAL-TEAM
001990         END-PERFORM
002000         IF TEAM (1) = "XXX" OR "xxx" THEN
002010             MOVE "Y" TO LAST-SCORE
002020             PERFORM SORT-TABLE
002030         ELSE
002040         PERFORM UNTIL V-SCORE or SORT-ONLY
002050            DISPLAY "INPUT RESULT AS 'X-Y'"
002060            ACCEPT SCORE
002070            MOVE W-H-SCR TO TAB-SCORE (1)
002080            MOVE W-A-SCR TO TAB-SCORE (2)
002090            PERFORM VAL-SCORE
002100         END-PERFORM
002110         DISPLAY "LAST RESULT? Y/N"
002120         ACCEPT LAST-SCORE
002130         PERFORM CALC-POINTS
002140         PERFORM UPDATE-RECORD
002150         END-IF
002160     END-PERFORM.
002170*****************************************************************
002180 VAL-TEAM.
002190     PERFORM VARYING COUNTER FROM 1 BY 1
002200                 UNTIL COUNTER > 2
002210
002220        EVALUATE TRUE
002230           WHEN TEAM (COUNTER) = "ars" or "ast" or "bra" or
002240                     "che" or "cov" or "der" or
```

```
002250                     "eve" or "lee" or "lei" or
002260                     "liv" or "man" or "mid" or
002270                     "new" or "she" or "sou" or
002280                     "sun" or "tot" or "wat" or
002290                     "wes" or "wim"
002300               MOVE "Y" TO V-TEAM-FLAG
002310           WHEN OTHER MOVE "N" TO V-TEAM-FLAG
002320        END-EVALUATE
002340     END-PERFORM
002350     IF NOT V-TEAM THEN DISPLAY
002360             "INVALID TEAM CODE ENTERED-"
002370             "RE-ENTER BOTH TEAM CODES AGAIN."
002380     END-IF.
002390*****************************************************************
002400 VAL-SCORE.
002410     IF ( W-H-SCR > 9 ) OR ( W-A-SCR > 9 )
002420         THEN PERFORM BIG-SCORE
002430     END-IF
002440     IF ( W-H-SCR NOT NUMERIC) OR ( W-H-SCR NOT NUMERIC)
002450         THEN MOVE "N" TO V-SCORE-FLAG
002460     ELSE MOVE "Y" TO V-SCORE-FLAG
002470     END-IF
002480     IF NOT V-SCORE THEN
002490        DISPLAY "INVALID SCORE ENTRY. PLEASE RE-ENTER SCORE."
002500     END-IF.
002510*****************************************************************
002520 BIG-SCORE.
002525* Putting a STOP RUN in this paragraph is probably
002527* very bad programming practise. Better logic could be used!
002530     DISPLAY "A team has scored more than 10 goals. "
002540     DISPLAY "This program will terminate now. "
002550     DISPLAY "Following this, the record in Input.rec "
002560     DISPLAY "will have to be ammended manually"
002580     DISPLAY " Following this perform SORT ONLY procedure."
002600     ACCEPT ENDING-KEY
002610     STOP RUN.
002620
002630*****************************************************************
002640 CALC-POINTS.
002650     IF TAB-SCORE (1) > TAB-SCORE (2) THEN
002660         MOVE 3 TO TAB-POINTS (1)
002670     ELSE
002680         IF TAB-SCORE (2) > TAB-SCORE (1) THEN
002690             ADD 3 TO TAB-POINTS (2)
002700         ELSE
002710             MOVE 1 TO TAB-POINTS (1)
002720             MOVE 1 TO TAB-POINTS (2)
002730         END-IF
002740     END-IF.
002750
002760*****************************************************************
002770 UPDATE-RECORD.
002790     MOVE 1 TO N
002800     MOVE 1 TO M
002810     OPEN I-O TEAM-REC-IN
002820     PERFORM UNTIL M > 20
002830     READ TEAM-REC-IN
002840     AT END
```

```
002850         DISPLAY TEAM (1) " has details ammended"
002860     NOT AT END
002870     IF TEAM (1) = TEAM-CODE THEN
002880         PERFORM ADJUST-DATA
002890
002900     END-IF
002910     ADD 1 TO M
002920     END-READ
002930     END-PERFORM
002940
002950     CLOSE TEAM-REC-IN
002955
002970     MOVE 2 TO N
002980     MOVE 1 TO M
002980
002990     OPEN I-O TEAM-REC-IN
003000     PERFORM UNTIL M > 20
003010        READ TEAM-REC-IN
003020           AT END
003030               DISPLAY TEAM (2) " has details ammended"
003040           NOT AT END
003050             IF TEAM (2) = TEAM-CODE THEN
003060                PERFORM ADJUST-DATA
003080             END-IF
003090             ADD 1 TO M
003100        END-READ
003110     END-PERFORM
003120     CLOSE TEAM-REC-IN
003130     DISPLAY "Table has been updated".
003140*************************************************************
003150 ADJUST-DATA.
003160     IF N = 1 THEN MOVE 2 TO SWITCH
003170     ELSE MOVE 1 TO SWITCH
003180     END-IF
003190         ADD TAB-SCORE (N) TO GOALS-FOR
003200         ADD TAB-SCORE (SWITCH) TO GOALS-AGST
003210         SUBTRACT GOALS-AGST FROM GOALS-FOR GIVING GOAL-DIFF
003220         ADD TAB-POINTS (N) TO POINTS
003230         ADD 1 TO PLAYED
003240         EVALUATE TAB-POINTS (N)
003250             WHEN 3 ADD 1 TO G-WON
003260             WHEN ZERO ADD 1 TO G-LOST
003270             WHEN 1 ADD 1 TO G-DRAWN
003280         END-EVALUATE
003290         REWRITE TEAM-REC.
003300*************************************************************
003310 SORT-TABLE.
003320     SORT WORK-FILE
003330         ON DESCENDING KEY POINTS-KEY GD-KEY GF-KEY
003340         USING TEAM-REC-IN
003350         GIVING SORT-OUT.
003360
003370*************************************************************
003380 PRINT-TABLE.
003390     ACCEPT W-DATE FROM DATE
003400     MOVE W-DAY TO P-DAY
003410     MOVE W-MON TO P-MON
003420     MOVE W-YEAR TO P-YEAR
```

```
003430
003440     OPEN INPUT SORT-OUT
003450          OUTPUT PRINT-FILE
003460
003470     WRITE TEXT-OUT FROM P-TITLE AFTER 1 LINE
003480     WRITE TEXT-OUT FROM P-UNDERLINE AFTER 1 LINE
003490     WRITE TEXT-OUT FROM P-GAP AFTER 1 LINE
003500     WRITE TEXT-OUT FROM P-HEADER AFTER 1 LINE
003510     MOVE 1 TO N
003520     PERFORM UNTIL N > 20
003530        READ SORT-OUT
003540           AT END MOVE "Y" TO EOF-FLAG
003550           NOT AT END
003560              MOVE N TO P-TAB-POS
003570              MOVE STEAM-NAME TO P-TEAM
003580              MOVE SPLAYED TO P-PLAYED
003590              MOVE SG-WON TO P-G-WON
003600              MOVE SG-LOST TO P-G-LOST
003610              MOVE SG-DRAWN TO P-G-DRAWN
003620              MOVE SGOALS-FOR TO P-GOALS-FOR
003630              MOVE SGOALS-AGST TO P-GOALS-AGST
003650              MOVE SGOAL-DIFF TO P-GOAL-DIFF
003660              MOVE SPOINTS TO P-POINTS
003670
003680              WRITE TEXT-OUT FROM W-TEXT-OUT
003700        END-READ
003710        ADD 1 TO N
003720     END-PERFORM
003730     CLOSE SORT-OUT PRINT-FILE
003740     DISPLAY "Table is now written to the printer".
003750*****************************************************************
```

10.4 Calculate prime numbers program

This is a little program that calulates prime numbers. You are prompted to enter a number (up to 1999) and the program will produce a file, 'PRIME-NO.TXT, which contains a table of all prime numbers up to the value entered.

```
000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID.    PRIME-NO-PROG.
000030 AUTHOR.        TIMOTHY R P BROWN.
000040
000050****************************************
000060* PROGRAM TO CALCULATE PRIME NUMBERS     *
000070****************************************
000080
000090 ENVIRONMENT DIVISION.
000100 INPUT-OUTPUT SECTION.
000110 FILE-CONTROL.
000120
000130       SELECT OUT-FILE ASSIGN TO 'PRIME-NO.TXT'
000140           ORGANIZATION IS LINE SEQUENTIAL.
000150******************************************
000160 DATA DIVISION.
000170 FILE SECTION.
000180
000190 FD OUT-FILE.
000200 01 NO-OUT      PIC X(80).
```

```
000210******************************************
000220 WORKING-STORAGE SECTION.
000230
000240 01 EVEN-FLAG PIC X.
000250      88 NUM-EVEN VALUE 'Y'.
000260 01 PRIME-FLAG PIC X .
000270      88 IS-PRIME VALUE 'Y'.
000280
000290 01 TOP-VALUE  PIC 9(7) VALUE ZERO.
000300
000310 01 COUNTERS.
000320       03 Y-COUNT PIC 9(6) OCCURS 1000.
000330
000340 01 CALC-NO       PIC 9(6) VALUE ZERO.
000350
000360 01 SUBS.
000370      03 X-SUB    PIC 9(6) VALUE 3.
000380 01 PRINT-SUBS.
000390      03 P-COUNT-X  PIC 9(6) VALUE 1.
000400
000410 01 A       PIC 9(6) VALUE ZERO.
000420 01 B       PIC 9(6) VALUE ZERO.
000430 01 C       PIC 9(6) VALUE ZERO.
000440 01 D       PIC 9(6) VALUE ZERO.
000450 01 Z       PIC 9(6) VALUE ZERO.
000460 01 PRIME-NO-COUNT PIC 9(6) VALUE 2.
000465
000470 01 PRINT-LINE.
000480      03 P-NUM1  PIC Z(5)9 VALUE ZERO.
000490      03 P-NUM2  PIC Z(5)9 VALUE ZERO.
000500      03 P-NUM3  PIC Z(5)9 VALUE ZERO.
000510      03 P-NUM4  PIC Z(5)9 VALUE ZERO.
000520      03 P-NUM5  PIC Z(5)9 VALUE ZERO.
000530
000540 01 EXIT-KEY PIC X VALUE SPACE.
000545
000550*******************************************************
000560 PROCEDURE DIVISION.
000570 MAIN-PARA.
000580      OPEN OUTPUT OUT-FILE
000590      DISPLAY 'ENTER VALUE TO WHICH PRIME NUMBERS '
000600      DISPLAY 'ARE TO BE CALCULATED BETWEEN 1 AND 999,999'
000602      MOVE 1 TO Y-COUNT (1)
000605      MOVE 2 TO Y-COUNT (2)
000610
000620*ENTER VALUE
000630      PERFORM UNTIL TOP-VALUE > 0
000640      ACCEPT TOP-VALUE
000650      END-PERFORM
000660
000670*ZEROISE TABLE
000680      MOVE ZEROS TO COUNTERS
000690
000700*DETERMINE PRIME NUMBERS AND PLACE IN TABLE
000710
000720      PERFORM VARYING CALC-NO FROM 3 BY 1
000730    UNTIL CALC-NO > TOP-VALUE
000740         DISPLAY  CALC-NO
```

```
000750         MOVE 1 TO C
000760         MOVE 'N' TO PRIME-FLAG
000770
000780*IS NUMBER EVEN (BUT NOT 2)?
000790
000800         DIVIDE CALC-NO BY 2 GIVING A REMAINDER Z
000830         IF Z = 0 THEN MOVE 'Y' TO EVEN-FLAG
000840         ELSE MOVE 'N' TO EVEN-FLAG
000850         END-IF
000860
000865*********************************************************
000870*DIVIDE EACH ODD NUMBER BY NUMBERS UP TO HALF THE CALC-NO
000880*LOOP EXITED WHEN A NUMBER DIVIDES IT WITH NO REMAINDER
000890*OR WHEN ALL NUMBERS CHECKED
000895*********************************************************
000900         IF NOT NUM-EVEN THEN
000910            PERFORM VARYING D FROM 3 BY 1
000920    UNTIL (C = 0) OR (D > ((CALC-NO + 1) / 2))
000930  DIVIDE CALC-NO BY D GIVING A REMAINDER C
000940            END-PERFORM
000950         END-IF
000960
000970         IF C = 0 THEN MOVE 'N' TO PRIME-FLAG
000980         ELSE MOVE 'Y' TO PRIME-FLAG
000990         END-IF
001000
001010*WHEN PRIME NUMBER DEFINED, MOVE IT INTO TABLE
001020         IF IS-PRIME THEN
001030            MOVE CALC-NO TO Y-COUNT (X-SUB)
001040            ADD 1 TO X-SUB PRIME-NO-COUNT
001050         END-IF
001060      END-PERFORM
001070
001080*STORE THE FINAL VALUE OF X-SUB BEFORE RE-USING IT
001090      MOVE X-SUB TO P-COUNT-X
001100      MOVE ZERO TO X-SUB
001110**************************************************
001120*WRITE TABLE
001130      PERFORM VARYING X-SUB FROM 1 BY 5
001140            UNTIL X-SUB > P-COUNT-X
001150         MOVE Y-COUNT (X-SUB) TO P-NUM1
001160         MOVE Y-COUNT (X-SUB + 1) TO P-NUM2
001170         MOVE Y-COUNT (X-SUB + 2) TO P-NUM3
001180         MOVE Y-COUNT (X-SUB + 3) TO P-NUM4
001190         MOVE Y-COUNT (X-SUB + 4) TO P-NUM5
001200         WRITE NO-OUT FROM PRINT-LINE AFTER 2 LINE
001230      END-PERFORM
001240
001250      DISPLAY 'CALCULATIONS COMPLETE - ' PRIME-NO-COUNT
001260            ' PRIME NUMBERS CALCULATED'
001270      CLOSE OUT-FILE
001280      STOP RUN.
```

10.5 Create INDEXED file program

This program takes a line sequential record file and converts it to an indexed file. The records must contain a unique key field that is in strict ascending order. The input file (from a text editor) should be called 'LINESEQFILE.TXT'. The

program output will be 'INDEXEDFILE.DAT'. You can change these in the ENVIRONMENT DIVISION if you want.

```
000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID.    CREATE-INDEX-PROG.
000030 AUTHOR.        TIMOTHY R P BROWN.
000040
000045****************************************************
000050* Program to convert a sorted (ascending)          *
000060* line sequential file ('LINESEQFILE.TXT') to      *
000070* an indexed file (output 'INDEXEDFILE.DAT').      *
000075****************************************************
000080
000090 ENVIRONMENT DIVISION.
000100 INPUT-OUTPUT SECTION.
000110 FILE-CONTROL.
000120
000130        SELECT OUT-FILE ASSIGN TO 'INDEXEDFILE.DAT'
000140           ORGANIZATION IS INDEXED
000150           ACCESS MODE IS SEQUENTIAL
000160           RECORD KEY IS INDEX-KEY.
000170        SELECT IN-FILE ASSIGN TO 'LINESEQFILE.TXT'
000180           ORGANIZATION IS LINE SEQUENTIAL.
000190
000200 DATA DIVISION.
000210 FILE SECTION.
000220
000230 FD OUT-FILE.
000240 01 MAKE-OUT.
000250        03 INDEX-KEY   PIC X(6).
000260        03             PIC X(120).
000270
000280 FD IN-FILE.
000290 01 IN-REC             PIC X(126).
000300
000310
000320 WORKING-STORAGE SECTION.
000340
000350 01 EOF-FLAG PIC X VALUE 'N'.
000360     88 END-OF-FILE VALUE 'Y'.
000370
000375****************************************************
000377
000380 PROCEDURE DIVISION.
000390 MAIN-PARA.
000400       OPEN INPUT IN-FILE
000410            OUTPUT OUT-FILE
000420
000430       PERFORM UNTIL END-OF-FILE
000440         READ IN-FILE
000450            AT END MOVE 'Y' TO EOF-FLAG
000460            NOT AT END
000470               MOVE IN-REC TO MAKE-OUT
000480               WRITE MAKE-OUT
000490         END-READ
000500       END-PERFORM
000510
000520       CLOSE OUT-FILE IN-FILE
000530       STOP RUN.
```

10.6 Read INDEXED file program

This program allows you to view the contents of an indexed file by generating a line sequential file of the original indexed file. If you tried to open an indexed file with a text editor you would just see gibberish. The input file for this program is 'INDEXEDFILE.DAT' giving an output text file called READFILE.TXT. Again, you can change these in the ENVIRONMENT DIVISION if you wish.

```
000010 IDENTIFICATION DIVISION.
000020 PROGRAM-ID.    INDEXED-TO-READ-PROG.
000030 AUTHOR.        TIMOTHY R P BROWN.
000040
000045*****************************************************
000050* Program to convert indexed file 'INDEXEDFILE.DAT' *
000060* to line sequential (output called 'READFILE.TXT') *
000070* for viewing with text editor.                     *
000075*****************************************************
000080
000090 ENVIRONMENT DIVISION.
000100 INPUT-OUTPUT SECTION.
000110 FILE-CONTROL.
000120
000130        SELECT IN-FILE ASSIGN TO 'INDEXEDFILE.DAT'
000140           ORGANIZATION IS INDEXED
000150           ACCESS MODE IS DYNAMIC
000160           RECORD KEY IS S-KEY-NO.
000170        SELECT OUT-FILE ASSIGN TO 'READFILE.TXT'
000180           ORGANIZATION IS LINE SEQUENTIAL.
000190
000200 DATA DIVISION.
000210 FILE SECTION.
000220
000230 FD IN-FILE.
000240 01 IN-REC.
000250     03 S-KEY-NO        PIC X(6).
000260     03                 PIC X(43).
000270
000280 FD OUT-FILE.
000290 01 OUT-REC             PIC X(49).
000300
000310
000320 WORKING-STORAGE SECTION.
000340
000350 01 EOF-FLAG PIC X VALUE 'N'.
000360     88 END-OF-FILE VALUE 'Y'.
000370
000373*****************************************************
000377
000380 PROCEDURE DIVISION.
000390 MAIN-PARA.
000400        OPEN INPUT IN-FILE
000410             OUTPUT OUT-FILE
000420
000430        PERFORM UNTIL END-OF-FILE
000440           READ IN-FILE NEXT
000450             AT END MOVE 'Y' TO EOF-FLAG
000460             NOT AT END
000470                WRITE OUT-REC FROM IN-REC
```

```
000480             END-READ
000490          END-PERFORM
000500
000510          CLOSE IN-FILE OUT-FILE
000520          STOP RUN.
```

ZingCOBOL Copyright Timothy R P Brown 2003

About ZingCOBOL

This book has been generated from the web site ZingCOBOL, found at one of the following locations:

http://members.lycos.co.uk/zingcobol
http://zingcobol.tripod.com
http://homepage.ntlworld.com/zingmatter/zingcobol

As a consequence of this, there may appear textual references that do not appear to make much sense since it was a hyperlink on the original site.

ZingCOBOL was developed, beginning in 1999, and gradually improved over time with corrections, additions, and some minor design changes.

The text, graphics, and COBOL source code are copyright of Timothy R P Brown and Zingmatter Web Design 1999 onwards. If you wish to reproduce this text for self learning then feel free to print it off. Likewise, for use within a teaching environment it is free to use in it an unaltered form. If you wish to publish any, or part of this book, or alter significant parts within, permission should be obtained from the author.

The flower image on the front cover was by my daughter Emily.

Timothy R P Brown
Glasgow, UK

August 2003

ZingCOBOL Copyright Timothy R P Brown 2003