PL/I Programming

I0103

Textbook 1

Independent Study Program

First Edition (July 1980)

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available outside the United States.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. Address comments concerning the contents of this publication to IBM Corporation, Publications Services, Education Center, South Road, Poughkeepsie, New York 12602

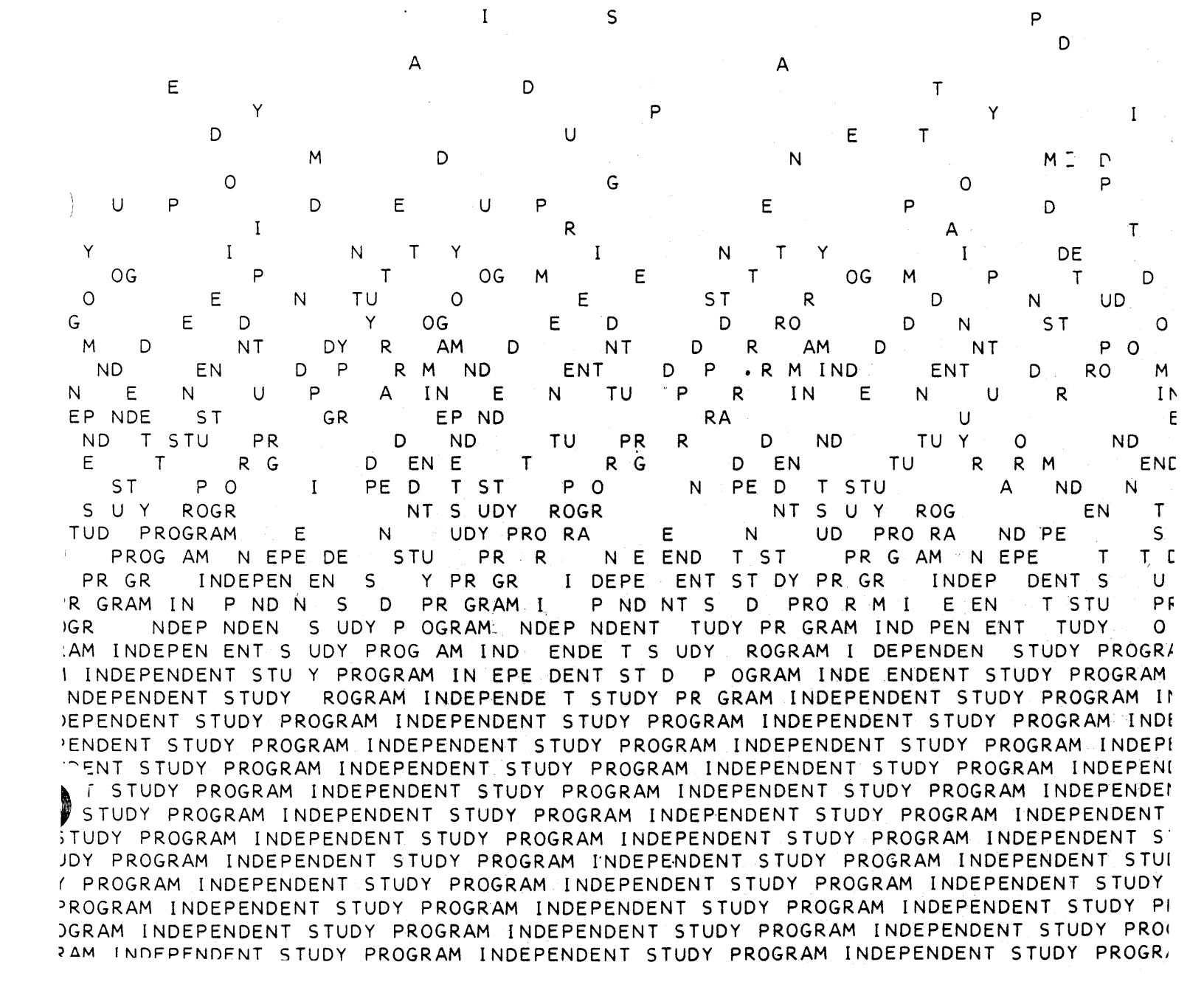
IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Contents

Textbook 1

Textbook 2

Topic **1**



Topic 1

Introduction to the PL/I Language

Since you are commencing a study of PL/I, it might be useful to see what PL/I is about, how it developed and where it fits into computing today.

PL/I is a high level programming language aimed at both scientific and commercial programming problems.

The purpose of all programming languages is to enable the programmer to communicate to the computer what work he requires it to do. High level languages enable him to do this in a way which is relatively easily read and understood by humans, compared with the low level languages, which require that problems be specified in terms which are more readily 'understood' by the computer, but are not so readily understood by humans without a considerable amount of special training. High level languages are 'problem-oriented', low level languages are 'machine-oriented'.

For instance, a pay-roll application may require that pay slips be printed showing basic pay, gross pay, tax and net pay. Having already stored all the information required for these calculations, the calculations could be specified in PL/I as:

```
BASIC_PAY = HOURLY_RATE * BASIC_HOURS;

GROSS_PAY = HOURLY_RATE * 1.5 * OVERTIME_HOURS + BASIC_PAY;

TAX = (GROSS_PAY - TAX_FREE) * TAX_RATE;

NET_PAY = GROSS_PAY - TAX - OTHER_DEDUCTIONS;
```

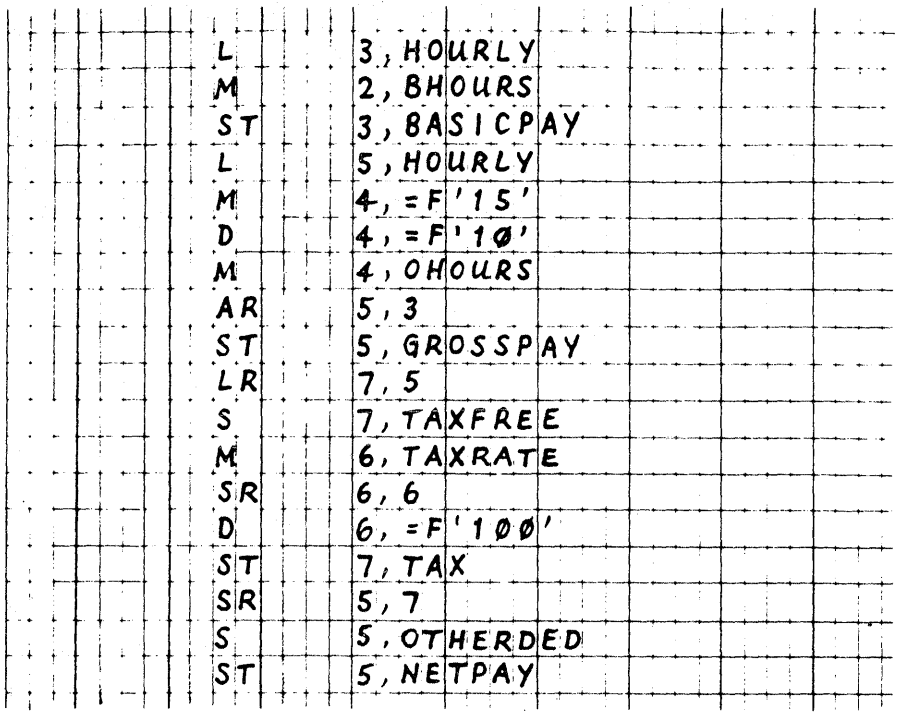
This looks fairly close to normal arithmetic, with a few differences.

The underscore or break character (_) is used in PL/I to indicate that the two groups of characters either side of it are both parts of one name.

Multiplication is indicated by *, instead of x.

A semi-colon is added to the end of each statement.

Given this information, an otherwise untrained person will probably be able to say how net pay is calculated. The same calculations coded in Assembler, a fairly low level language, could be written as follows:



Unless you have previously been writing in Assembler, I expect that you will find the PL/I easier to understand.

In fairness to both languages, it should be mentioned that in both it is possible to include comments to aid comprehension. These are blocks of text which are ignored by the computer, but which help to make a program more readable.

It was stated that PL/I is equally suitable for scientific and commercial programming. These are terms which cannot be formally defined, but are generally taken to indicate the type of processing performed on the data, rather than the origins of the data. Scientific programs are generally taken to be programs which read in and print out a small amount of data relative to the amount of calculation done in the program, whereas commercial programs read in and print out a lot of data, and do relatively little processing on it. In the earliest days of computing these were quite reasonable definitions and described fairly well the sort of problems which were solved using computers by the scientific and commercial worlds. Although the terms survived, the distinction is not so valid and more programs fall into the grey area between the extremes.

A program which might now be written in PL/I would, before PL/I had been developed, probably have been written in FORTRAN or COBOL, two other high level languages. COBOL, COmmon Business Oriented Language, was developed by a group of computer users and computer users and computer manufacturers in the United States in 1959 for programming business applications. As such, it is heavily biased towards handling character information and file processing. FORTRAN, FORmula TRANslation, was first developed in the United States by IBM in 1954 for mainly mathematical use, and as such has powerful computational capabilities. Both languages have developed considerably since their introductions, but both still have their biases.

Many installations have a need to write programs of scientific, commercial and in-between natures. For instance, a distribution company may use its computer mainly for accounting

purposes, but may also need to write programs for vehicle route scheduling. An engineering company may use its computer largely for design work, but may also wish to use it for accounting. To meet these needs, the companies would need programmers skilled in FORTRAN and programmers skilled in COBOL or alternatively, programmers skilled in both FORTRAN and COBOL, languages with considerable differences between them.

To overcome these difficulties, SHARE, the scientific users' organization, GUIDE, the commercial users' organization, and IBM set up a team in 1964 to develop a new, multipurpose language. The result was the first PL/I compiler, developed by the IBM Laboratory, Hursley, England in 1966. This was the F-level compiler for running on System/360 under Operating System (OS). A sub-set compiler, the D-level compiler, was later released for running under System/360 Disk Operating System (DOS). These compilers had further facilities added in later releases.

By 1970 Hursley Laboratory had developed the Optimizing compilers for System/360 and System/370 (OS and DOS). As the name suggests, these compilers produce programs which both occupy less space in storage and execute more quickly than programs compiled by the F-level or D-level compilers. They also provide many enhancements to the facilities of the F-level and, particularly, the D-level compilers. This text is concerned with the Optimizing compilers for DOS and OS in all their forms.

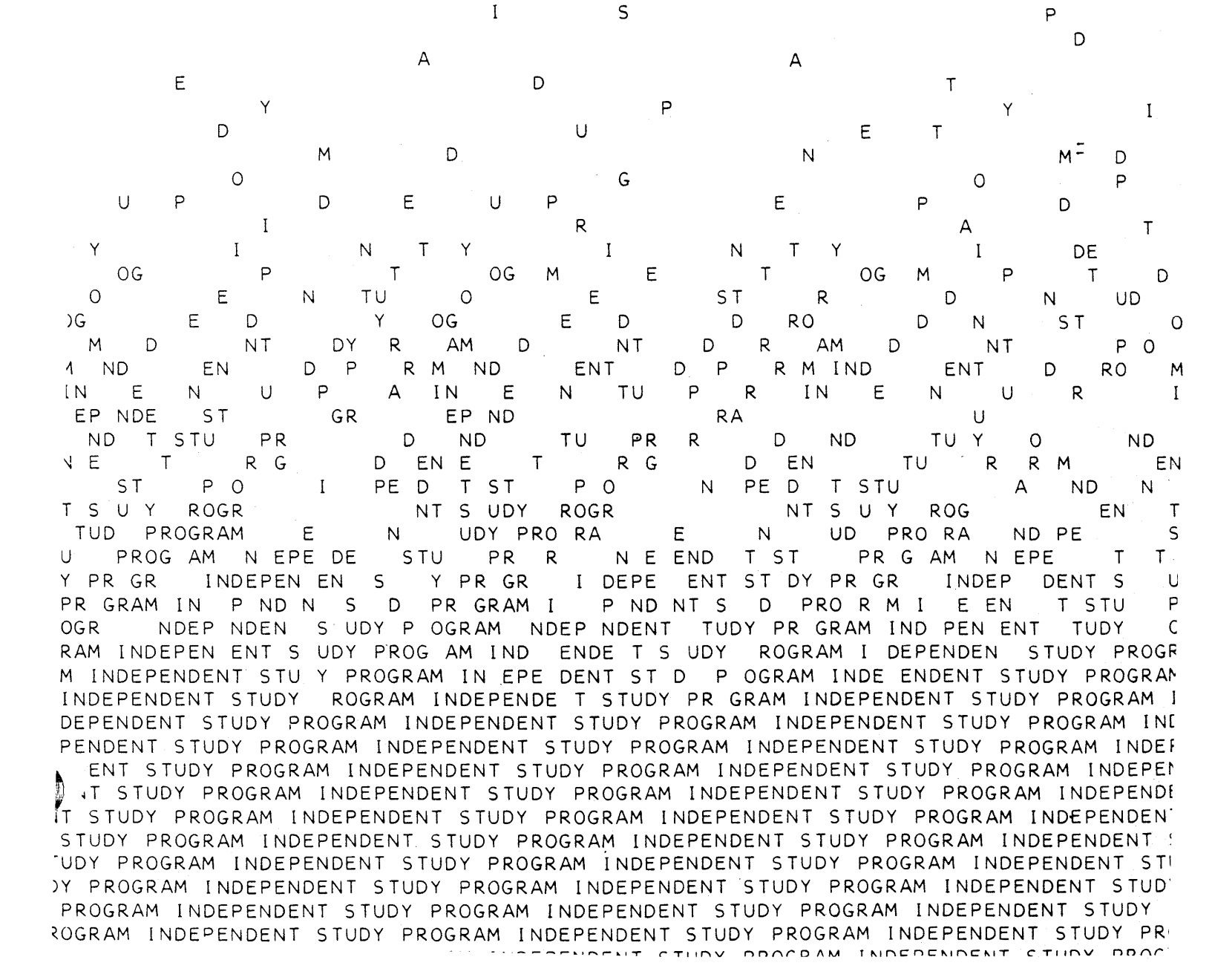
If a programming language contains facilities for a wide range of applications areas, no one programmer is likely to need all of its facilities, and it is undesirable if the programmer has to learn all about the language before he can successfully use it. With these thoughts in mind, PL/I was designed with a default capability and to be sub-settable. That is, where there are options available to the programmer, if he does not specify one, a workable option will be selected by default. As the programmer learns more he may override the defaults in the interests of greater efficiency or flexibility in a particular situation.

A commercial programmer may not have heard of a mathematical concept such as complex numbers and probably will not need to use them in his work. His programming efficiency will not be hindered if he knows nothing of the facilities for processing complex numbers in PL/I. If he later has a need to process complex numbers, he need only learn these particular PL/I facilities - and what complex numbers are The rest of what he knows of the structure of PL/I programs and statements, and the input/output facilities, will not be changed and can be used in his current work.

In this course you will be learning a fairly large sub-set of the PL/I language.

Topic

2



Topic 2

Elements of PL/I

In this topic you will learn the basic statements which constitute the PL/I language. This knowledge will be assumed in later topics.

Objectives

At the end of the topic you should be able to:

- describe the basic format of a PL/I program.
- code valid identifiers.

Programs and Procedures

In Topic 1, you saw some examples of PL/I statements. Statements are the basic elements of the language which cause work to be done. They are grouped together into larger logical blocks called procedures, which normally perform descrete logical functions. A program consists of one or more procedures.

For example, a payroll program might have four procedures. One procedure might control the processes of payroll calculations. Another might read clock cards and check that the information on them was reasonable. A third might calculate the gross pay, various deductions and the net pay. The last might print the pay slips and update the payroll file. This division of work between procedures is a reasonably logical division. It is up to the programmer to choose logical divisions of work to form procedures.

Main Procedures

Every program will always have one MAIN PROCEDURE. When the system passes control to the PL/I program, that is, when the PL/I program starts to execute, it will start at the beginning of the main procedure. If the program contains further procedures, they will only be executed if control is specifically passed to them. Procedures other than main procedures will not be dealt with further until a later topic.

Procedures other than main procedures are an example of the sub-settability of PL/I. It is possible to write quite complicated programs using only a MAIN PROCEDURE. However, by the use of other procedures, the writing and program testing can often be done more quickly and more effectively, and the program will be easier to maintain and modify.

Basic Program Form

The great majority of computer programs have the basic form:

Read some data

Manipulate it

Write out some results

This basic form may vary considerably but there are few programs which do not conform to it.

Sample Program

As an example of a simple program, let us consider a requirement to copy a name and address, stored on a punched card, onto the line printer. There is no means of doing this directly, so we must instruct the computer to read the card into main storage and then to write out what it has just read. Note that this program does not conform to the basic pattern. Although it reads in some data, and writes out some results (i.e. the data which it has just read), it does not manipulate it in any way.

Here is a program which will do this, assuming that the data on the punched card is 80 characters long.

```
MYPROG: PROCEDURE OPTIONS (MAIN);
        DECLARE NAME_AND_ADDRESS CHARACTER (80);
        READ FILE (CARDSIN) INTO (NAME_AND_ADDRESS);
        WRITE FILE (PRNTOUT) FROM (NAME_AND_ADDRESS);
        END;
```

Note: This would not be a complete program for running on a DOS/VS system. The names CARDSIN and PRINTOUT would need DECLARE statements which are different from the DECLARE statements which could be used in OS/VS.

This will apply to all programs in the text which use this type of input and output except in some later topics where the differences will be explained.

The PROCEDURE Statement

The program is complete and consists of a main procedure only. The start of this is shown by the procedure statement:

```
MYPROG: PROCEDURE OPTIONS (MAIN);
```

The construction of this statement is:

```
MYPROG      the name of the program.
:           a colon to separate the name from the rest of
            the statement.
PROCEDURE OPTIONS (MAIN) to identify a main procedure
;           to terminate the statement
```

The end of the main procedure is indicated by the statement:

```
END;
```

where END indicates the type of statement and ; indicates the end of the statement. Every procedure must begin with a PROCEDURE statement and be terminated by an END statement. MYPROG is a statement label. Any statement may have a label attached to it. It is used so that the statement may be referred to uniquely within the program. Procedure statements must have labels attached to them. The label chosen for the main procedure is not important, as it is referred to only by the system, which notes what it is. Labels on other procedures are important as it is by reference to the label on the procedure statement that the programmer requests the execution of the statements in that procedure.

The DECLARE Statement

The name and address on the punched card can be assumed to be 80 characters long. If it is less than this, then the rest of the information will be blanks. Space will be needed to hold the 80 characters of information in main storage, and this is reserved and given a name by which it can be referred to by the statement:

```
DECLARE NAME_AND_ADDRESS CHARACTER (80);
```

The statement construction is:

DECLARE	identifies the statement type. One thing that a DECLARE statement can do is to request space in main storage, as here.
NAME__AND__ADDRESS	the name by which we choose to refer to that main storage. There is no connection, as far as PL/I is concerned, between the name which we use and what will be put into it. We could equally well have called it X or SQUAREROOT.
CHARACTER (80)	specifies that the identifier NAME AND ADDRESS is to refer to a part of main storage which will hold character information 80 characters long.
;	terminates the statement.

The processes of reading and writing involve transfer of information between the main storage areas of the computer and some external storage device. Typically, a System/370 will have many thousands, perhaps millions, of bytes of main storage and several external storage devices of various types. The instructions which we give must do more than tell the computer to read or write some information. They must specify an area of main storage and an external device to be used in this transfer.

In PL/I, we do not have to specify the particular area of main storage by a main storage address. The DECLARE statement requested an area of main storage 80 bytes long. We may now refer to that space, wherever it has been allocated, by the NAME__AND__ADDRESS. We will use this name in our READ and WRITE statements to identify the area of main storage to be used in the transfers.

Similarly, rather than naming specifically the card reader, a line printer or a particular magnetic disk in the statements, we will use a name which will be associated with a particular device elsewhere. In DOS/VS, it will be associated partly through a DECLARE statement, covered in Topics 5 and 11, and, in both DOS/VS and OS/VS, by Job Control Language, which is covered on a separate segment. In both DOS/VS and OS/VS the name used is called the file name.

The READ Statement

The statement which instructs the computer to read some information is:

```
READ FILE (CARDSIN) INTO (NAME_AND_ADDRESS);
```

The construction of the statement is:

READ	identifies the operation to be carried out - read in 1 record (in this case, 1 card).

FILE (CARDSIN)	specifies that CARDSIN is the name used for the file to be read from.
INTO(NAME__AND__ADDRESS)	specifies that the name of the main storage area to receive the data read is NAME__AND__ADDRESS.
;	terminates the statement.

The WRITE Statement

Having read the information into NAME__AND__ADDRESS, we may now request that it is written out again by the instruction:

```
WRITE FILE (PRNTOUT) FROM (NAME_AND_ADDRESS);
```

The construction of the statement is similar to that of the READ statement:

WRITE	identifies the operation to be carried out - write out 1 record (in this case, 1 line of print).
FILE(PRINTOUT)	specifies that PRNTOUT is the name of the file to be written to.
FROM(NAME__AND__ADDRESS)	specifies that the information to be written is held in the area of main storage identified by NAME__AND__ADDRESS.
;	terminates the statement.

Execution of the Sample Program

When the program is compiled, prior to execution, note will be taken of the DECLARE statement and, it will not, as such, form a part of the executable program. Similarly, the PROCEDURE statement is taken to indicate the point at which the program is to start executing, but does not, as such, do anything at execution.

Both the PROCEDURE and DECLARE statements are non-executable statements.

When the program is executed, execution will start at the first executable statement following the PROCEDURE statement, the READ statement. Having executed the first executable statement, the next will be executed, and so on until the END statement. This flow of control may be interrupted by statements which we have not yet studied. In this program, a card will be read in and then the contents will be written out to the printer. The program will then terminate.

When space was allocated for NAME__AND__ADDRESS, it was as if a request had been made for some work space on a dirty blackboard. A piece of space was allocated, but it contained some information already. This information is unlikely, except by coincidence, to be of any use. When the READ statement is executed, useful information will be put into NAME__AND__ADDRESS, and will completely replace what was in there. The old information will not be recoverable. Taking the blackboard analogy, the area of board was wiped clean and some new information was written in its place. This information will remain there until it is explicitly replaced by some new information. It may be copied to other areas of main storage

or to external devices, the value held there may be used in calculations, but it will not be changed until it is replaced by new information.

The Assignment Statement

performance of a math. eq or LET equal

Topic 1 contained some PL/I statements which performed calculations and which looked very similar to algebraic statements. These were assignment statements. For example:

```
BASIC_PAY = HOURLY_RATE * BASIC_HOURS;
```

Although this looks similar to an algebraic statement, the effect is considerably different. The effect of this statement is:

Take the value currently stored in the area of main storage known as HOURLY__RATE.

Take the value currently stored in the area of main storage known as BASIC__HOURS.

Multiply them together.

Store the result in the area of main storage known as BASIC__PAY.

The assignment statement is not a statement of fact. It is a statement of what we want to happen. Thus, a statement like:

```
NUMBER_OF_EVENTS = NUMBER_OF_EVENTS + 1;
```

is valid in PL/I. The effect is:

Take the value currently stored in the area of main storage known as NUMBER__OF__EVENTS.

Add 1 to this value.

Store the result in the area of main storage known as NUMBER__OF__EVENTS.

The variable on the left-hand side of the equals sign is called the target variable. This is the only variable which is altered by the assignment statement. So, HOURLY__RATE and BASIC__HOURS would not have their current values changed by the statement:

```
BASIC_PAY = HOURLY_RATE * BASIC_HOURS;
```

A statement:

```
NUMBER = NUMBER + NUMBER + NUMBER;
```

where the current value held in NUMBER before the statement was executed was 2, would put a value of 6(=2+2+2) into NUMBER.

The construction of the assignment statement is considerably more flexible than the construction of most other statements. It must consist of:

A target variable

An expression

;

The expression may be an arithmetic expression or a string expression. An arithmetic expression is an expression which has an arithmetic result. It may be an arithmetic constant, a variable or an expression of constants and variables which may involve addition, subtraction, multiplication, division and exponentiation. A string expression mainly generates a string of characters. It may be a character string constant or variable, like NAME__AND__ADDRESS, or it may be an expression which manipulates character strings. This statement will be discussed further in a later topic.

Construction of PL/I Statements

We have now seen examples of six statements:

PROCEDURE
DECLARE
READ
WRITE
Assignment
END

and a complete program. Let us now look at the general rules for the construction of PL/I statements.

Character Sets

PL/I programs are written using characters from either the 60-character set or the 48-character set. The character set which you use will depend on the reading/printing facilities available on the computer which you use. We will use the 60-character set in this text.

The following points about the character sets should be noted:

1) All alphabetic characters are in upper case. Lower case alphabetic characters may not be used in the writing of a PL/I program.

2) The characters $, @ and # are counted as being alphabetic.

3) Most of the characters of the 60-character set which do not appear in the 48-character set are replaced by combinations of characters which do appear in the 48-character set.

   These combinations of characters only have special meaning if it is specified that the 48-character set is being used.

The characters are used to form identifiers, constants, separators and comments.

Language Character Sets

60-CHARACTER	DESCRIPTION	48-CHARACTER
A through Z	Alphabetic	A through Z
$	Alphabetic	$
@ and =	Alphabetic	not available
0 through 9	Numeric	0 through 9
	Blank	
=	Equal	=
+	Plus	+
-	Minus	-
*	Asterisk or Multiply	*
/	Slash or Divide	/
(	Left Parenthesis	(
)	Right Parenthesis	)
,	Comma	,
.	Decimal Point or Period	.
'	Quote	'
%	Percent Symbol	//
:	Colon	..
;	Semicolon	..
¬	Not Symbol	NOT
&	And Symbol	AND
|	Or Symbol	OR
<	Less than	LT
¢	Break Character	not available
?	Question Mark	not available
>	Greater Than	GT
||	Concatenation (compound symbol)	CAT

Note:

1. The 48 character set // is 2 oblique lines and occupies 2 card columns.
2. The 60 character set | | is 2 vertical lines and occupies 2 card columns.
3. The presence of a blank is sometimes indicated by ƀ.

Identifiers

An identifier is a string of characters from one to 31 characters in length. The first character must be one of the 29 characters of the PL/I extended alphabet. **The rest of the characters may be alphabetic, numeric, and break characters.** They may not contain any other special characters, including blanks. **Some identifiers have special meanings in PL/I, like READ, others are used as variable names, i.e. names of areas of storage which may hold data, like NAME_AND_ADDRESS.**

Some identifiers have to be used outside the procedure in which they are declared. Examples are the labels on procedure statements and file names. These identifiers are called external names. They must not exceed seven characters in length, and they must not contain the break character. **If this length is exceeded, they will be shortened by taking the first four and last three characters of the name supplied.**

Identifiers like READ, DECLARE, FILE, which may have a special use in PL/I, are called keywords. Their special meaning is restricted to their use in context; out of context they may be used as normal identifiers. It would be quite proper to declare READ to have the attributes CHARACTER (80) and then to write:

```
READ FILE (MYFILE) INTO (READ);
```

The first appearance of READ is recognized as a keyword by the construction of the rest of the statement. Appearing between brackets after the keyword INTO, the second occurrence of READ must refer to a variable. All statements, except the assignment statement and the null statement, which has yet to be described, start with a keyword, and are identified by both the keyword and the construction of the statement.

There is no ambiguity in writing:

```
READ = NAME_AND_ADDRESS;
```

where READ is declared with attributes CHARACTER (80), because the appearance of the = character, immediately after it, shows that the statement is an assignment statement, and READ cannot be used as a keyword here.

Many keywords have convenient abbreviations. For keywords which you have seen so far, the following abbreviations exist:

PROCEDURE	PROC
DECLARE	DCL
CHARACTER	CHAR

A full list of keywords and their abbreviations is contained in the appropriate PL/I Optimizing Compiler: Language Reference Manuals for DOS/VS and OS/VS.

The strings of characters, which form the 48-character set equivalents of 60-character set special characters, are reserved words if the 48-character set is being used. That is, they have a special meaning, irrespective of the context in which they are used. They may not be used as variable names, file names or statement labels. If they are, then they will be treated as if they were the special characters. If the 60-character set is being used then they have no special significance.

Valid Identifiers

DATA
FIRST_NUMBER
INPUT_FILE
$_INCOME
FILE
A12345
A_VERY_VERY_VERY_VERY_LONG_NAME
X
CATS
CAT (for 60-character set only)

Invalid Identifiers

1_FILE	starts with a numeric character
_FILE	starts with the break character
FILE/NAME	contains a special character
CAT	if using the 48-character set, this is a reserved word.

Note: INPUT_FILE is shown as being a valid identifier. If it were used as a filename, as looks likely, then it would be truncated to INPUILE. It would be perfectly proper, if somewhat confusing, to use the identifier INPUT_FILE to identify an area which would hold numeric data.

Constants

PL/I has several data types. NAME_AND_ADDRESS was declared with the CHARACTER attribute, and variables such as HOURLY_RATE imply, by their use, that they should have some numeric attributes. Constants may be coded with any data type which may be used for variables. They are not declared. They are given their attributes by the way in which they are coded.

An example of a character constant is:

' PL/I IS '

It is identified as a character constant by beginning with a single quote and ending with a single quote. The quotes do not form a part of the character constant, they indicate that PL/I IS a character constant.

An example of a numeric constant is:

1.5

It is a fixed point decimal constant. It is also possible to code floating point decimal constants, and fixed and floating point binary constants. These will all be covered in Topic 3.

Separators

A PL/I statement may never contain two identifiers, constants, or an identifier and a constant without a separator between them.

If it did, they would be treated as one identifier, subject to the length restriction of 31 characters.

A separator may be any of the special characters, other than the break character, depending on the context. In the statement:

```
READ FILE(MYFILE)INTO(READ);
```

the identifiers used are READ, FILE, MYFILE, INTO and READ, of which the first READ, FILE and INTO are also keywords. The separators used are a blank, (,), (and).

In the statement:

```
GROSS_PAY=HOURLY_RATE*1.5*OVERTIME_HOURS+BASIC_PAY;
```

the identifiers used are GROSS__PAY, HOURLY__RATE, OVERTIME__HOURS and BASIC__PAY. The constant 1.5 is used and also the separators =, *, * and +.

The only separators needed in PL/I statements are those required by the syntax of the statement. Where no other separator is needed, as between READ and FILE, a blank must be inserted.

Blanks in PL/I and the Layout

Blanks are sometimes necessary in PL/I to act as separators, when, due to the syntax, identifiers would otherwise be adjacent. They need not otherwise be used at all.

The following shows the sample program written with the minimum number of blanks.

```
MYPROG:PROC OPTIONS(MAIN);DECLARE NAME_AND_ADDRESS CHARACTER(80);READ F
ILE(CARDSIN)INTO(NAME_AND_ADDRESS);WRITE FILE(PRNTOUT)FROM(NAME_AND_ADD
RESS);END;
```

The end of a PL/I statement is not a special case. One PL/I statement may follow another on the same line, if desired. Similarly, the end of a line is not special. PL/I statements are normally coded anywhere between columns 2 and 72 inclusive of a card. Column 72 of one card and column 2 of the next are treated as being adjacent, so that identifiers may run over from one card to the next with no errors.

Whenever a separator occurs in a program, one or more blanks may also be inserted. There is no limit to the number which may be inserted, and they incur no overhead. It is normal to use blanks to put each statement on a new line and to identify groups of statements by indenting them from other statements. Some installations have standards concerning the use of blanks and the layout of PL/I programs. You should check whether your installation does.

Due to the flexibility of layout in PL/I programs, it is not necessary to have special forms to code them on. They may be coded on any form that has 80 columns and is not pre-printed in any column.

Comments

Comments are text inserted in a program to explain to the programmer, or any other person reading the program, what it does and how it works. They have no effect on the working of the program and are ignored by the computer. Intelligent use of comments will make understanding a program much easier and speed debugging.

The format of a comment is any text preceded by the characters /* and followed by */. There must be no blanks embedded in the delimiters, and the text must not contain the adjacent pair of characters */. A comment may be inserted in a program anywhere a blank may be inserted.

The following are valid comments:

```
/* CALCULATE AND PRINT PAY FOR ONE MAN */
/* ERROR ROUTINES */
/* READ FILE(CARDSIN) INTO(NAME_AND_ADDRESS); */
```

Although the contents of the last comment have the same syntax as a valid PL/I statement, it will not be treated as such because it is contained in a comment.

Statement Format

PL/I statements have a general form:

[statement label:] [keyword] [statement body] ;

where square brackets indicate that the contents of the brackets may or may not appear.

Note that the only part of the statement which is not optional is the semi-colon (;). A statement which has only a semi-colon

```
;
```

is called a null statement and indicates no operation. As you may expect, a statement which causes no operation has limited usefulness, but it is not without uses.

A statement label is an identifier attached to the beginning of a statement and separated from the rest of the statement by a colon (:). It is used to identify a statement so that control may be passed to that statement. For the statements which you have seen so far, it is optional on all but the PROCEDURE statement, on which it is obligatory.

The keyword appears in all statements except the null and assignment statements. It identifies what type of statement it is, e.g. READ, PROCEDURE, END.

The remainder of the statement makes up the statement body. Statements such as END; and the null statement have no statement body.

You should now do questions 1 to 5 in the exercises at the end of this topic. There are similar exercises at the end of all other topics. Their aim is to help you to asses how well you have understood the material so far. Try to answer them without referring to the text, only doing so if you are stuck. Answers are provided on the pages following the exercises.

More General Programs

Let's consider the program which read in a record, possibly containing a name and address, and wrote it out again. While it will perform the task specified, it would be more useful to be able to process more than one record. Immediately preceding the END statement, the program had read and written a record, and still held the contents of that record in NAME_AND_ADDRESS. By the definition of our problem, that information is no longer needed, and so we may place further information in NAME_AND_ADDRESS, by putting further PL/I statements before the END. The following coding takes advantage of this situation to add a further READ and WRITE statement to copy a second record.

```
TWOCARD: PROCEDURE OPTIONS (MAIN);
         DECLARE NAME_AND_ADDRESS CHARACTER (80);
      R1: READ FILE (CARDSIN) INTO(NAME_AND_ADDRESS);
      W1: WRITE FILE (PRNTOUT) FROM(NAME_AND_ADDRESS);
      R2: READ FILE (CARDSIN) INTO(NAME_AND_ADDRESS);
      W2: WRITE FILE (PRNTOUT) FROM(NAME_AND_ADDRESS);
         END;
```

The READ and WRITE statements have all had statement labels attached to them so that we may identify them. Assuming the input data is on punched cards and the output is to be printed, the process would be:

R1: Read the first card and hold the information on it in the area of main storage known as NAME_AND_ADDRESS.

W1: Copy the current contents of NAME_AND_ADDRESS to the line printer.

R2: Read the next card and store its contents in NAME_AND_ADDRESS. Until this is done, NAME_AND_ADDRESS will contain the value last put into it i.e. the contents of the first card.

W2: Copy the current contents of NAME_AND_ADDRESS to the line printer.

Stop executing.

Two points arise:

1) The statements to process the second card are exactly the same as those to process the first card.

2) The program would be more flexible, and hence more widely useful, if it could copy any number of cards.

If we are prepared to accept a restriction on the contents of the cards, say, that none of the addresses will be blank, and decide that we shall indicate the end of address cards by including a card which is completely blank, we may state the processes required to copy all the cards up to the blank card as:

Read a card.
If it is not blank, print it out and read another card.
If it is not blank, print it out and read another card.
If it is not blank, print it out and read another card.
etc.
When a blank card is found, stop executing.

In our language, we require a form of statement or statements which will enable us to keep repeating the block of instructions:

If it is not blank, print it out and read another card until the condition does not hold.

The DO Group

The DO group provides this facility. It begins with a DO statement and terminates with an END statement. The DO statement specifies a condition under which the statements up to the END statement are to be executed repeatedly.

The following shows the specified problem coded with a DO group.

```
NCARD:  PROCEDURE OPTIONS (MAIN);                                    /* 10 */
        DECLARE NAME_AND_ADDRESS CHARACTER (80);                     /* 20 */
        READ FILE (CARDSIN) INTO (NAME_AND_ADDRESS);                 /* 30 */
        DO WHILE (NAME_AND_ADDRESS ¬= ' ');                          /* 40 */
           WRITE FILE (PRNTOUT) FROM (NAME_AND_ADDRESS);             /* 50 */
           READ FILE (CARDSIN) INTO (NAME_AND_ADDRESS);              /* 60 */
        END;                                                         /* 70 */
        END;                                                         /* 80 */
```

One of the many possible formats for a DO group is

DO WHILE (condition);

Repeat the statements of the group while the condition is true. In this case the condition is 'NAME AND ADDRESS is not blank'. ¬= is a comparison operator standing for 'is not equal to'. There are comparison operators for all the other comparisons - is equal to, is greater than, is less than, is greater than or equal to, is less than or equal to, is not greater than, and is not less than. These will be covered further in a later topic, together with other statements which may cause the program to execute differently for different sets of input data.

The statements of the DO group are statements 40 to 70. They are indented here to make them stand out as a group. This is not essential, blanks may be used in the normal way, but it improves the readability of the program. If there are no blank cards in the file, then the program will carry on reading cards until there are no more left; this will cause an error. PL/I provides facilities for detecting and dealing with this error; these will be covered later.

There are other formats for a DO group which will be covered later.

Executing PL/I Programs

PL/I programs, as written, cannot be executed, they cannot control the working of the computer.

System/370 can only execute programs which are in machine code, so the PL/I source statements must be turned into machine code. This is done in two stages, and the procedure is illustrated below.

Compilation

The first stage uses the PL/I Optimizing compiler, which is a program which fits into our standard pattern. That is, it:

Reads data
Manipulates it
Produces output

The data which it reads is PL/I source statements. The manipulation which it does is to turn these into object modules, that is, machine code.

The amount of output which it produces is variable, and is controlled by a series of options which will be dealt with in Topic 16. Typically it will give tables which describe the program, and an object module.

Link Editing

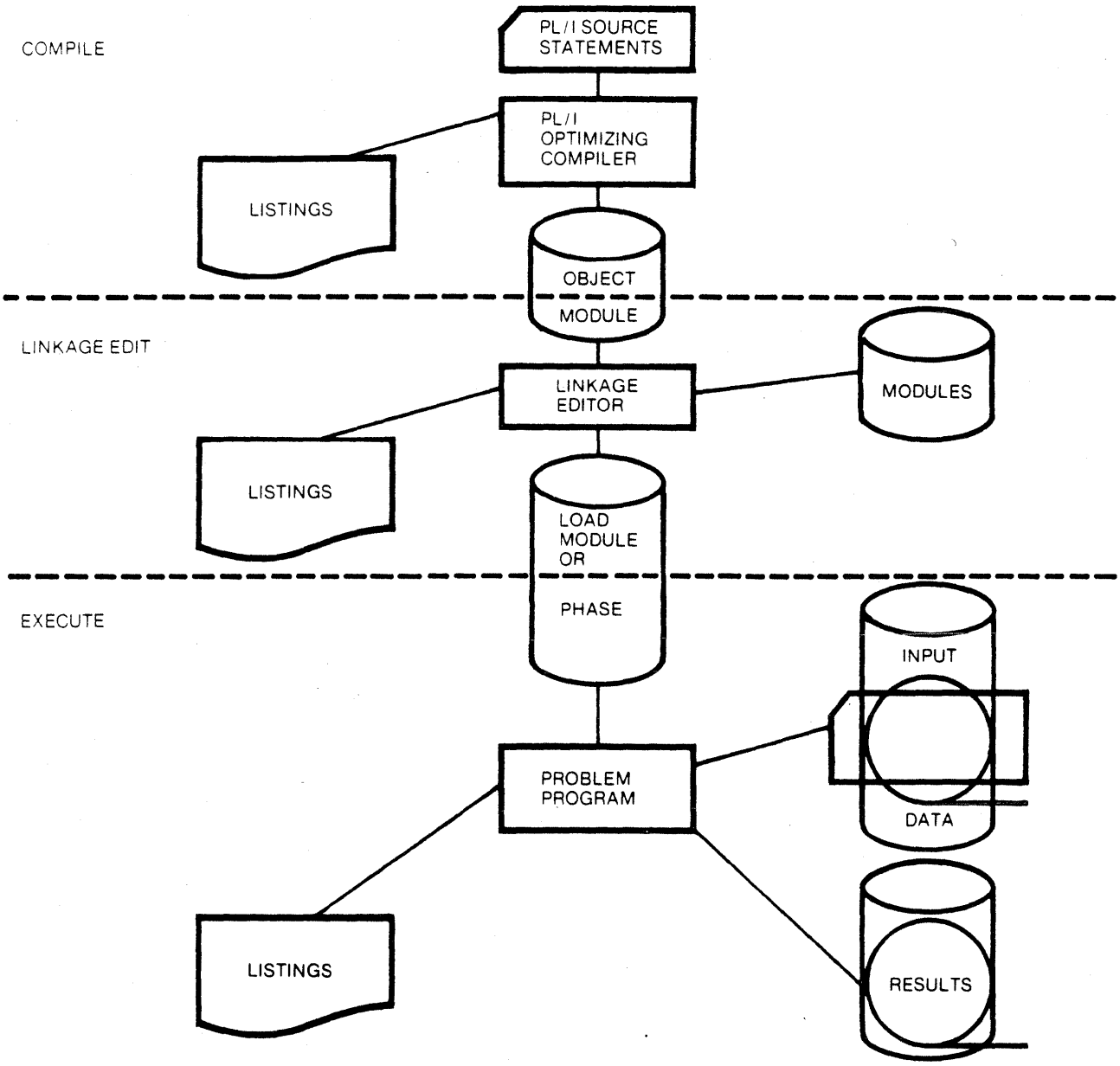
Although the object module consists of machine code, it is not yet in executable form. The program may consist of more than one procedure. Only one procedure may be compiled at a time, so there must be a means of linking the procedures together. Similarly, programs which use statements like READ implicitly request linkage to routines which are previously written and are supplied with the compiler and operating system. This linking is done by the Linkage Editor program.

The Linkage Editor program again conforms to the standard pattern. The data which it reads is an object module or modules produced by a compiler, which it links with other modules which may be user written or system supplied. The output which it produces is a number of printed reports and, hopefully, an executable program. This executable program is called a phase in DOS/VS terminology, or a load module in OS/VS terminology.

Execution

The load module or phase may now be executed, and normally will read input from one or more media, process it, and produce some printed output and possibly other output on other media, depending on what the programmer has coded.

The process is subject to errors at all stages. The programmer may write statements which are not legal PL/I. He may refer to routines which do not exist. His program may fail to process the data which it reads. The detection of these errors and their correction will be covered later in the segment.



You should now complete the exercises at the end of this topic.

Exercises

1. Which of the following are valid identifiers when using the 60-character set:

   a) IDENTIFIER

   b) VALID-1

   c) @_COST

   d) IN/OUT

   e) 4_SQUARE

   f) NOT

2. Of the identifiers in question 1 which are valid under the 60-character set, which could not be used under the 48-character set? A, F

3. Which of the following are valid identifiers to be used for a file name?

   a) EXTERNAL

   b) A37259

   c) PROC_1

   d) GET@

4. Write a statement to read a record from a file named STOCK into a variable named ITEM, using the minimum number of blanks. READ FILE (STOCK) INTO (ITEM)

5. Assuming that all identifiers used have been suitably declared, which of the following statements are valid?

```
/* A */  WRITEFILE(VARIABLE) FROM(FILE);
/* B */  TEN = 11;
/* C */  YOURS = MINE
/* D */  A /* THE BEGINNING */: PROC OPTIONS(MAIN);
```

6. What must be the first and last statements of a program? Procedure, END

7. What must be done to a PL/I program before it can be used to control the computer?

Answers

1. a) Valid

   b) Invalid. - is a special character and is not allowed in an identifier.

   c) Valid.

   d) Invalid. / is a special character and is not allowed in an identifier.

   e) Invalid. A numeric character may only appear in the body of an identifier.

   f) Valid.

2. c) @ and __ do not appear in the 48-character set and have no replacements.

   f) NOT is a reserved word in the 48-character set.

3. A file name is an external name. It may not exceed 7 characters and may not contain the break character.

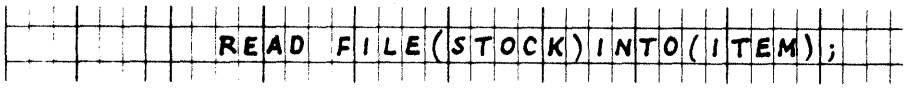
   a) Invalid. It is too long.

   b) Valid.

   c) Invalid. It contains the break character.

   d) Valid.

4.



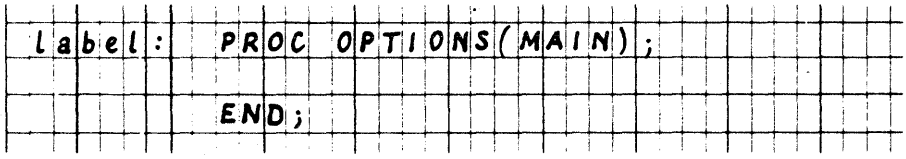
5. a) Invalid. There is no space between WRITE and FILE.

   b) Valid.

   c) Invalid. There is no semi-colon at the end of the statement.

   d) Valid. A comment may appear anywhere a blank can be used.

6.

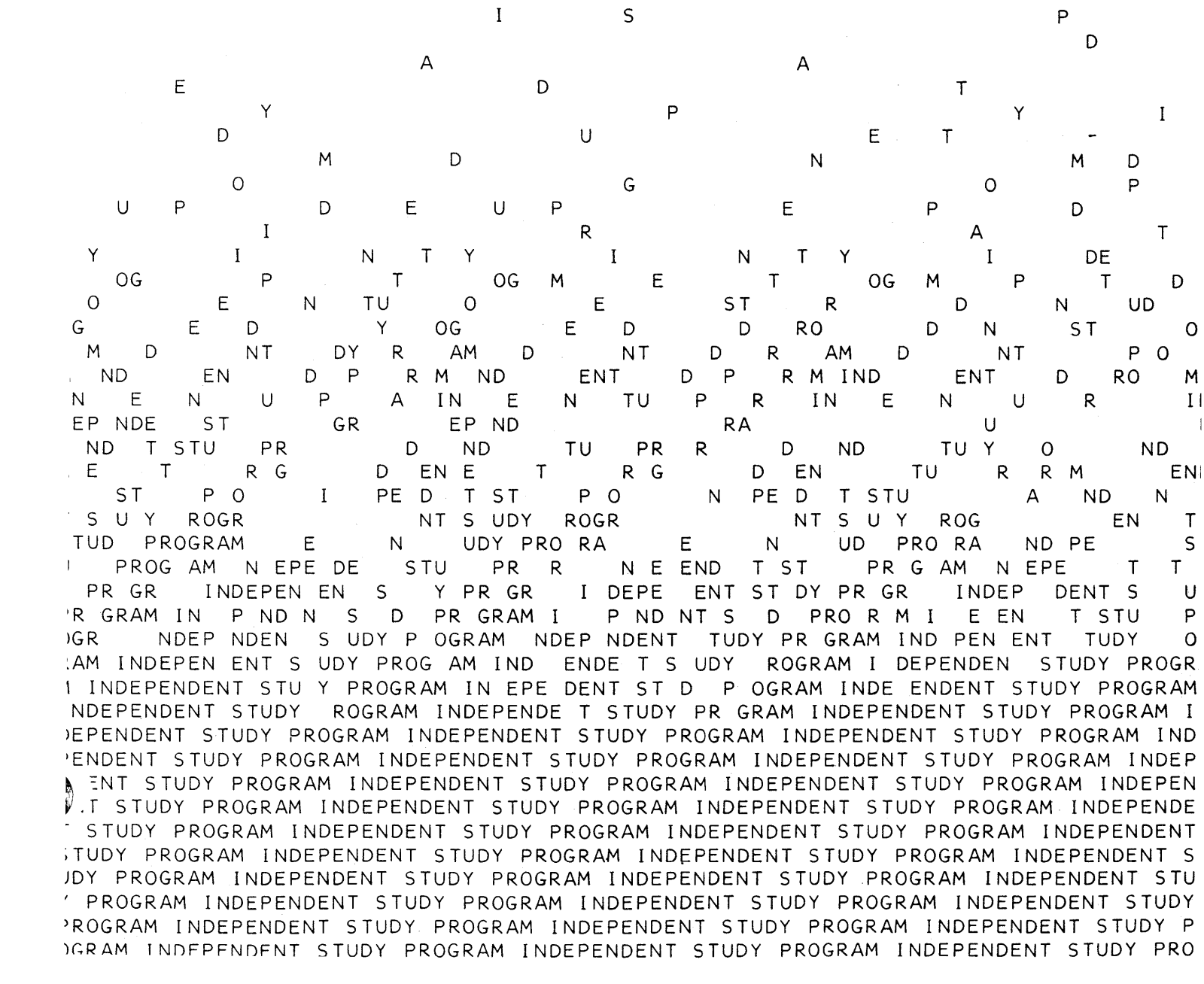


   The label on the PROCEDURE statement may be any identifier up to 7 characters long, not containing the break character.

7. Before a PL/I program can be executed, it must be turned into machine code by the PL/I Optimizing Compiler, and all necessary system supplied or user-written routines must be linked to it by the Linkage Editor program.

Topic

3



Topic 3

The DECLARE Statement and Data Elements

This topic covers the DECLARE statement and also the various forms of data elements.

Objectives

On completion of this topic you should be able to:

- write valid DECLARE statements for character and numeric data variables
- calculate the amount of space occupied by data variables
- state the full attributes of partially declared variables
- initialize data variables using the INITIAL attribute.

Introduction

If data is to be manipulated within a program then storage needs to be reserved for this data within the computer. This is the main function of the DECLARE statement, which was first met in Topic 2. The other necessary function of the statement is to give a name to this storage in order that it can be referred to within the program and also to describe the format of the data so that it can be manipulated. In this topic we shall be covering the data elements, i.e. the data variables which hold numeric and character data. We shall not be covering all possible data types nor shall we cover all of the facilities of the DECLARE statement. However, we shall cover enough of both to enable you to do a considerable amount of PL/I programming.-

The DECLARE Statement

The DECLARE statement is the statement which is used to attach attributes to PL/I identifiers.

DECLARE statements seen so far have all been for element variables, simple variables which hold information. They are also used to give attributes to groups of variables, and to other identifiers which cannot hold information, such as file names. These will be covered in later topics.

A DECLARE statement may be used to give attributes to one or more identifiers. Let us look at the statement to declare one identifier:

DECLARE IDENTIFIER [attribute(s)];

The square brackets are a part of the standard IBM method for demonstrating syntax. They indicate that the contents, in this case the attributes, are optional and may be left out if not needed.

DECLARE, which may be abbreviated to DCL, is the keyword which identifies the statement type. It must be the first part of the statement and separated from the identifier by one or more spaces.

The identifier is that which is being given attributes, and must be the second part of the statement. It must be separated from the attributes, if any, by one or more spaces.

The attributes which follow the identifier are keywords, some of which have abbreviations. They may be in any order, and must be separated from each other by a blank or blanks. Some attributes, such as CHAR, may have further information following them in brackets, e.g. CHAR(20). Where this is so, blanks between the open bracket, (, and the preceding attribute, and between the close bracket,), and the following attribute are optional.

In the example below, declarations 1 and 2 show alternative spacings for the same declaration. There is no ambiguity because CHAR(10) could not mean anything but character, of length 10.

```
DCL NAME CHAR(10);                                  /* 1  */
DCL NAME CHAR (10);                                 /* 2  */
DCL IN FILE INPUT SEQL;                             /* 3  */
DCL INFILE INPUT SEQL;                              /* 4  VALID BUT MAY NOT
                                                       GIVE WHAT WAS INTENDED */
DCL IN FILEINPUT SEQL;                              /* 5  */

DCL NAME   CHAR(10),                                /* 6  */
    STR    CHAR(10),
    STRIN  CHAR(20),
    IN     FILE INPUT SEQL,
    OUT    FILE OUTPUT SEQL;
DCL NAME   CHAR(10);                                /* 7  */
DCL STR    CHAR(10);                                /* 8  */
DCL STRIN  CHAR(20);                                /* 9  */
DCL IN     FILE INPUT SEQL;                         /* 10 */
DCL OUT    FILE OUTPUT SEQL;                        /* 11 */
DCL (NAME, STR) CHAR(10);                           /* 12 */
DCL (IN INPUT, OUT OUTPUT) FILE SEQL;               /* 13 */
DCL (IN INPUT, OUT) FILE OUTPUT SEQL;               /* 14 */
```

Declaration 3 shows a declaration for a file name. It would not be complete for DOS. The attribute FILE says that this is a file name. The attribute INPUT says that it may only be read from, and not written to (its opposite is OUTPUT). The attribute SEQL, an abbreviation for SEQUENTIAL, says that the file is to be read in sequence, one record after another. These attributes will be dealt with further in Topic 5.

In declaration 4, the blank has been missed out between IN and FILE. The rules of the syntax will be followed and the statement will be taken as a declaration of the identifier INFILE with the attributes INPUT and SEQL. This will be quite legal as the attributes INPUT and SEQL both imply FILE. That is, if either of these attributes is coded, then it is assumed that FILE is required also, so coding the attribute FILE is optional.

Declaration 5 has no spaces between FILE and INPUT, and so FILEINPUT will be taken as an attribute keyword. It will not be recognized as a valid one, and will cause an error.

A declaration statement may declare more than one identifier:

DCL identifier [attribute(s)],

identifier [attribute(s)].....;

The list of attributes, if any, for the first identifier is followed by a comma, then the name of the next identifier and its attributes and so on. The last attribute of the last identifier is followed by a semi-colon. The effect is exactly as if each identifier were declared in a separate statement. Declaring several identifiers in the same declaration does not link them together in any way. An example is shown as declaration 6 above. This would have exactly the same effect as declarations 7 to 11.

Note the use of spaces and the comma as separators. Spaces are used to separate the attributes of an identifier, and to separate the identifier from its attribute. The comma separates one identifier and its attributes from another identifier and its attributes.

IN and OUT, in declaration 6, have the attributes FILE and SEQL in common, but IN has the attribute INPUT, and OUT has the attribute OUTPUT, to say that it may only be written to. In this situation, the common attributes may still be factored, as in declaration 13, but the attributes which are exclusive to any identifier are written with that identifier, inside the brackets.

The identifier IN takes the attribute INPUT within the brackets and the attributes FILE and SEQL from outside the brackets. The identifier OUT takes the attribute OUTPUT inside the brackets and the attributes FILE and SEQL from outside the brackets.

The general format for a declaration with factored attributes becomes:

DCL (identifier [attribute(s)],

identifier [attribute(s)],...) attribute(s);

Attributes occurring within the brackets and attributes occurring outside the brackets must be additive attributes and not alternative attributes. That is, they must be attributes which can occur together as attributes for one identifier. Declaration 14 is invalid because INPUT and OUTPUT, which would both be applied to IN are alternative attributes.

It should be noted that the attributes of STR and STRIN may not be factored. Although both have the attribute CHAR, the length must be coded immediately after the CHAR attribute and cannot be put before it. The same applies to precision attributes, which are covered later in this topic.

The position and order of declarations has no significance in a PL/I procedure. A DECLARE statement may occur anywhere between the PROCEDURE and END statements of a procedure. It does not matter if the statement which declares an identifier occurs after statements which use that identifier. Similarly, it is not significant whether identifiers are declared in any particular order, or whether they are declared in separate statements or the same statement, or whether they are declared with factored attributes or not. The relative positioning of PL/I variables in storage does not depend on how they are declared.

We have now covered the general form of the DECLARE statement. The attributes which may be attached to identifiers will be covered as they arise.

Please attempt questions 1 and 2 in the exercises at the end of this topic before continuing.

Character Data

In Topic 2 use was made of the variable NAME AND ADDRESS, which was declared:

The declare statement specifies

a) what sort of data is to be held in the variable

b) how much data is to be held

c) how the data is to be held.

NAME AND ADDRESS will hold 80 characters of data. Using Extended Binary Coded Decimal Interchange Code (EBCDIC), it will hold any characters which can be stored in System/370 and System/360. EBCDIC is a coding system which stores one character of data per byte. As a byte has 8 data bits, there are 2^8=256 possible combinations. Not all combinations are allocated in EBCDIC. The full list of characters represented, and their internal representations, is shown on System/370 Reference Summary card (Form GX20-1850). It includes all of the 60 character set characters.

The EBCDIC system considers the byte in two halves - the left hand 4 bits are called the zone and indicate a group of characters - the numeric characters, the upper case characters A-I etc. The right hand 4 bits are called the numeric and indicate which character of the group is to be represented. Below are shown some examples.

Character	Binary	Hexadecimal
A	1100 0001	C1
B	1100 0010	C2
C	1100 0011	C3
I	1100 1011	C9
1	1111 0001	F1
2	1111 0010	F2
3	1111 0011	F3

All the normal keyboard data input devices and printing output devices transmit data in EBCDIC.

Character String Variables

Variables with the CHARACTER attribute are called character string variables. It is also possible to use character string constants in PL/I programs. The maximum length of the character string variables is 32767, and the minimum is 1. If a length is not given in the declaration, a default of 1 is taken.

The following are declarations of valid character string variables:

```
DCL TEXT      CHAR(30);
DCL SHORT_1   CHAR(1);
DCL LONG_1    CHAR(32767);
DCL NUMERIC   CHAR(256);      /* THE NAME USED HAS NO SIGNIFICANCE */
DCL CHAR      CHAR;           /* DEFAULT LENGTH = 1 */
```

The following are not valid declarations of character string variables:

```
DCL INVALID_1  CHAR(0);       /* THE LENGTH MUST BE > 0 */
DCL INVALID_2  CHAR(32768);   /*      BUT <= 32767      */
DCL INVALID_3  CHARS(100);    /* KEYWORD IS CHARACTER OR CHAR */
```

Character String Constants

A character string constant consists of a single quote character (') followed by the EBCDIC characters which make up the constant, and terminated by a further single quote ('). The quotes do not form a part of the constant and do not contribute to its length. Blanks within the quotes do count as part of the constant.

Below are some valid character string constants:

Constant	Length
'TEXT'	4
' TEXT AND BLANKS '	17
' A = B+C; '	10
'/* NOT A COMMENT */'	19
'12.67'	5

Note that, although the third constant appears to contain a PL/I statement and the fourth appears to contain a PL/I comment, both lose their significance and are simply character string constants.

The maximum length of a character string constant is not fixed. It depends on how much space is available for the PL/I compiler, but will never be less than 512 characters.

A single quote is the character which terminates a character string constant, special arrangements must be made to include one in the constant. Every single quote character which is to be included in a constant must be represented as two adjacent single quote characters. They will then be treated as a single quote character in the constant. Hence 'SARAH''S' is 7 characters - long and represents SARAH'S.

The following are valid character string constants:

Constant	Contents	Length
''''	'	1
''''''	''	2
'SHAKESPEARE''S ''''HAMLET'''''	SHAKESPEARE'S ''HAMLET''	24

A character string which consists of a string of characters repeated a number of times may be coded using a repetition factor. The string of characters is coded as a character string constant. The repetition factor is coded before it as a decimal integer constant, enclosed in parentheses.

Thus:

(2) 'HA' is equivalent to 'HAHA'
(3) 'HA' is equivalent to 'HAHAHA'

The string which is repeated may be any character string constant which is legal for that compiler. The repetition factor may be in the range 1 to 32767. Character string constants with repetition factors may be used anywhere where ordinary character string constants may be used.

The following are invalid character string constants:

`'MANDY'S MISTAKE'`	The embedded quote will be taken as terminating the constant, and the final quote as starting a new constant.
`'SPIRAL STAIRS`	Missing end quote.
`2'STROKE'`	No parentheses around the repetition factor.

You should now attempt questions 3, 4 and 5 at the end of this topic.

Numeric Data

It is possible, in the extreme, to process numeric data in character string variables, but PL/I provides other data types which can do so more efficiently in terms of speed and storage requirements. There are several data types for numeric data, each of which has advantages in certain circumstances.

Numeric variables in PL/I are declared with attributes of base, scale and precision. The base states to what base the number will be held and is DECIMAL or BINARY. DECIMAL variables will hold numbers as a series of decimal digits. BINARY variables will hold numbers as a series of binary digits. The scale may be FIXED or FLOAT. A variable with the FIXED attribute holds data in a fixed point form, that is a fixed number of digits and the binary or decimal point at a fixed position within them. Variables with the FLOAT attribute hold data in a floating point form. Each number is held in two parts. The exponent indicates the magnitude of the number - the position of the point, while the mantissa shows the value in a standardized form. As both the exponent and mantissa are variable parts of the data, the range of magnitudes of numbers which may be held in floating point variables is very wide.

The precision of a variable states how many digits of precision a variable may hold. If a variable has a BINARY base, it will be in binary digits. If it has a DECIMAL base, it will be in decimal digits. For floating point point variables, the precision indicates the number of digits in the mantissa. The exponent has a fixed precision. For fixed point variables, the precision must show the total number of digits to be held, and also the position of the decimal or the binary point within them. The precision is not identified by a keyword, but must be enclosed in brackets.

When declaring numeric variables, the following rules apply:

The base and the scale may be in either order.

The precision must immediately follow the base or the scale. It may come between them, or after the last, but not precede both of them.

Adjacent identifiers and keywords must be separated by at least one blank.

As the precision is enclosed in brackets, it need not be separated from adjacent keywords by blanks, but it may be.

The keyword DECIMAL may be abbreviated to DEC.

The keyword BINARY may be abbreviated to BIN.

In the example below the variables BIN__1 to BIN__6 all have identical attributes. The different layouts have no effect in PL/I terms, but you may find some easier to read than others, or easier to change later.

```
DCL BIN_1 BINARY FIXED (15,Ø);
DCL BIN_2 BIN    FIXED (15,Ø);
DCL BIN_3 FIXED  BINARY(15,Ø);
DCL BIN_4 BINARY (15,Ø) FIXED;
DCL BIN_5 BINARY (15,Ø)FIXED;
DCL BIN_6 FIXED  (15,Ø) BINARY;
```

We will now look at the characteristics of various combinations of attributes.

Binary Fixed

Binary fixed point variables and constants are the data items which can be processed most rapidly in the computer. They are most commonly used for processing integer data. Numbers held in fixed point binary variables or constants are held in binary form, and occupy either a half-word (2 bytes) or a full word (4 bytes), depending on the precision specified.

Below are some typical declarations of binary fixed point variables.

```
DCL BIN_1 BINARY FIXED(15,Ø);
DCL BIN_2 BIN FIXED(15);
DCL BIN_3 BIN FIXED(31);
DCL BIN_4 BIN FIXED;
DCL BIN_5 BIN FIXED(6,3);

   BIN_1 = 1Ø1B;
```

The precision described for fixed point binary variables is in binary digits, and specifies how many binary digits are to be held, irrespective of the sign. Thus BIN__1 has, effectively, 15 bits for a value, plus one bit for a sign. The numbers it can hold range from 32767 to -32768 (2^{15} -1 to 2^{15}), and will have no fractional part.

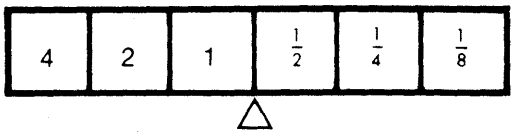
When the precision of a fixed point variable does not show a position for the point, then it is assumed to be at the end. BIN__2 has identical attributes to BIN__1.

The maximum precision which may be specified is 31. A fixed point binary variable whose precision is (31,0) may hold integers in the range 2147483647 to -2147483648 (2^{31}-1 to -2^{31}). The minimum precision is (1,0). This will allow the variable to hold the integers 1,0,-1 and -2 (2^{1}-1 to -2^{1}).

Fixed point binary variables which are declared with a precision of 1 to 15 use half-word binary and occupy 2 bytes. Those with a precision of 16 to 31 use full word binary and occupy 4 bytes. Variables like BIN__4, which are declared with the attributes BINARY and FIXED, but no precision, take a default precision of (15,0).

Fixed point binary variables may also be declared to have a fractional part. BIN__5 in the preceeding example will hold six binary digits, of which the last three will be considered as being fractional or after the binary point.

The effective significance of the bits is



The symbol Δ indicates the position of the implied binary point. It does not take up any space. The first bit effectively indicates the sign. If we look only at positive numbers, we may say that the first bit is the sign and will be set to 0 to indicate positive. Moving left from the binary point, the bits indicate the values of 1, 2 and 4 respectively. The value +2 will be held as 0010 000. As the bits to the left of the binary point indicate ascending positive powers of 2(2^{0}, 2^{1}

and 2^2), so the bits to the right indicate increasing negative powers of 2 (2^{-1} or 1/2, 2^{-2} or 1/4, and 2^{-3} or 1/8). Thus, +2 1/4 will be held as 0010 010.

BIN__5 can hold numbers correct to the nearest eighth. However, fractional decimal numbers cannot in general be converted exactly to fractional binary numbers. Each fractional decimal digit will need approximately 3.32 fractional binary digits to hold it, but even then it will not hold it exactly. Because of this, decimal numbers with fractional parts are often held in other types of variables.

Fixed point binary constants may also be coded in programs. A fixed point binary constant is defined as:

an optional sign

a string of binary digits containing an optional binary point

a character B.

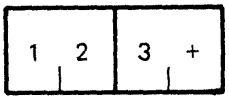
There may be no blanks in the constant. If no sign is included, it is assumed that the number is positive. If no binary point is included, it is assumed that the number is a binary integer. Moving left from the binary point, the 1's mean 1, 2, 4, 8, 16, 32, 64 etc. Thus 101B is 4 + 0*2 + 1 = 5. Moving right from the binary point, the 1's mean 1/2, 1/4, 1/8, 1/16, 1/32 etc. Thus 11.101B is 2 + 1 + 1/2 + 0/4 + 1/8 = 3 5/8.

Fixed point binary constants have precisions implied by the number of digits coded. The total precision is the number of digits coded both before and after the binary point, but not including the sign, the point or the B. The implied precision of the fractional part is the number of digits coded after the binary point. Thus 101B has an implied precision of (3,0) and 11.101B has an implied precision of (5,3). In Fig. 3.3, the value assigned to BIN__1 has an implied precision of (3,0) while BIN__1 has a precision of (15.0). The significance of this will be discussed in Topic 4. If it were desired to code 101B with an implied precision of, say, (8,3), this could be done by adding 0's to force that precision i.e. 00101.000B.

You should now attempt questions 6 and 7 in the exercises at the end of this topic.

Decimal Fixed

Variables with the attributes DECIMAL FIXED hold numeric information in S/370 packed format. In this format, a code to indicate a sign is held in the right-most half-byte (4 bits) of the field, and every other half-byte holds one decimal digit. Thus, 123 would be held as:



Below are some declarations of fixed point decimal variables.

```
DCL DEC_1  DECIMAL      FIXED(7);
DCL DEC_2  FIXED        DECIMAL(7);
DCL DEC_3  DEC          FIXED(7);
DCL DEC_4  DEC(7)       FIXED;
DCL DEC_5  FIXED(7)     DEC;
DCL DEC_6  DEC          FIXED(7,0);
DCL DEC_7  DEC          FIXED(7,7);
DCL DEC_8  DEC          FIXED(2,1);
DCL DEC_9  DEC          FIXED(15);
DCL DEC_10 DEC          FIXED(1);


     DEC_1  =   123;
     DEC_7  =  -0.356;
```

The declarations follow the same rules as for fixed point binary variables, with the following differences:

DECIMAL, which may be abbreviated to DEC, replaces BINARY.

The precision denotes the total number of decimal digits to be held and the number of those digits which are after the decimal point. The sign is not included.

The maximum precision is 15.

If the precision is not specified, a default of (5,0) is taken.

Thus, DEC_1 with a precision of (7,0), can hold integers in the range 9999999 to -9999999, DEC_7 with a precision of (7,7), can hold numbers in the range 0.9999999 to -0.9999999, to correct to 7 places of decimals.

For holding integers, DECIMAL FIXED and BINARY FIXED variables are largely interchangeable. The following differences exist:

BINARY FIXED variables are processed more quickly than DECIMAL FIXED variables.

Punched card input and printer output is in character format. Conversion between DECIMAL FIXED and CHARACTER is quicker than conversion between BINARY FIXED and CHARACTER.

DECIMAL FIXED variables, with a maximum precision of (15,0), may hold much larger numbers than BINARY FIXED variables.

Non-integer decimal numbers cannot always be held exactly in BINARY FIXED variables. If they can be held exactly in 15 decimal digits, then they can be held exactly in a DECIMAL FIXED variable. Hence, sums of money in dollars and cents, up to $99.99 could be held exactly in a variable:

```
DCL MONEY DEC FIXED(4,2);
```

DECIMAL FIXED variables are therefore recommended for usage in commercial applications involving computations with money amounts.

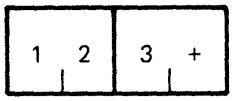
DECIMAL FIXED constants are coded as one would normally write decimal constants outside of a computing environment. That is, an optional sign, a series of decimal digits, followed by an optional decimal point and further decimal digits. There are no special identifying characters and there must be no embedded blanks. Thus, the two assignment statements above assign a value of 123 to DEC 1 and -0.356 to DEC 7. If a constant is coded as 101, it means one hundred and one. It is not a binary representation of decimal 5 as it is not followed by B.

The implied precision of a decimal fixed point constant is the number of digits in the constant, ignoring the sign and decimal point, if present. The number of fractional digits is the number of digits coded after the decimal point, including all 0's.

The precision of 123 is (3,0).

The precision of 00123.00 is (7,2).

The layout of the constant 123 was stated to be



From this, it can be seen that each decimal digit requires a half-byte, and the sign requires a further half-byte. Fixed point decimal variables may be of any length from 1 byte to 8 bytes, in steps of 1 byte. The following table shows the space required for variables declared with the stated precision, or for constants with that precision implied.

Precision	Number of Bytes
(3,0)	2
(1,0)	1
(1,1)	1
(15,0)	8
(15,8)	8
(6,0)	4

If the total precision is even, the left-most half-byte is not used.

Fixed point decimal variables may, by suitable choice of precisions, be used to process a wide range of values with complete accuracy. Before going on to read about the attributes and characteristics of floating point variables, please attempt questions 8 to 10 at the end of this topic.

Decimal Float and Binary Float

Fixed point variables are declared to have a number of decimal or binary digits with a decimal or binary point positioned at a fixed position in those digits.

A variable declared:

```
DCL CASH DEC FIXED(5,2);
```

could hold dollar money amounts with complete accuracy from $999.99 to $0.00 to $-999.99. The minimum increment is one cent and amounts are always recorded correct to a cent, so a DECIMAL FIXED variable is quite suitable. However, consider a program which calculates the average of a series of measurements in meters. If a DECIMAL FIXED variable were used to hold the measurement, what would be an appropriate precision? What would the measurements be? If the measurements were the lengths of modern oil tankers, they might be 400 meters. It is unlikely that they would be measured more accurately than the nearest meter, and so:

```
DCL LENGTH_1 FIXED DEC(3,0);
```

would give a suitable variable.

However, if the measurements were the widths of fine wires or textile filaments, they might be about 0.00002 meters, possibly measured to three significant figures e.g. 0.0000254, 0.0000326 etc. A suitable DECIMAL FIXED variable to hold these would be:

```
DCL LENGTH_2 FIXED DEC(7,7);
```

If we wish our program to be a general purpose program to deal with both oil tanker lengths and wire diameters, we will have to declare the variable:

```
DCL LENGTH_3 FIXED DEC(10,7);
```

but we will still encounter trouble if we wish to average the distances between towns, or the diameters of tissue cells.

The length of a ship, say 432 meters, could be written as $4.32 * 10^2$ (or 4.32 * 100). The diameter of a wire, say 0.0000254 meters, could be written as $2.54 * 10^{-5}$ (or 2.54/100000). This is called normalized floating point notation.

PL/I can process data held in this manner by using variables with the FLOAT attribute, instead of the FIXED attribute.

Decimal Float

Variables with the attributes DECIMAL FLOAT hold information in S/370 hexadecimal floating point format. They are declared with a precision attribute containing one constant. This does not reflect the magnitude of numbers which can be stored, only the precision with which they will be stored. All floating point variables can hold numbers in the range 16^{63} to -16^{63} (approximately 10^{75} to -10^{75}). The smallest number which can be distinguished from 0 is 16^{-64} (approximately 10^{-78}).

The precision specifies the number of significant decimal digits to be held. As the values will be held in hexadecimal, the number actually held may be greater. If the values to be held are integers then, assuming that the precision is adequate, they will be held exactly. However, as the values are held in a hexadecimal format, values with fractional parts will not normally by held exactly in floating point variables. The precision implies that, if the value actually held were rounded to that number of significant decimal digits, it would be correct. Expressed to a greater number of significant decimal digits, it may not appear correct.

When DECIMAL FLOAT variables are declared:

They have the attributes DECIMAL (or DEC) and FLOAT.

They have a precision consisting of a single constant in the range 1 to 33.

If a precision is not given, a default of 6 applies.

Below are some examples.

```
DCL DEC_FL_1  FLOAT     DECIMAL(3);
DCL DEC_FL_2  FLOAT     DEC(3);
DCL DEC_FL_3  DEC       FLOAT(3);
DCL DEC_FL_4  FLOAT(3)  DEC;
DCL DEC_FL_5  DEC(3)    FLOAT;
DCL DEC_FL_6  DEC       FLOAT(6);
DCL DEC_FL_7  DEC       FLOAT(16);
DCL DEC_FL_8  DEC       FLOAT(33);

    DEC_FL_1 = 1.23E1;          /* ASSIGNS 12.3 TO DEC_FL_1 */
    DEC_FL_1 = 123E-1;          /* ASSIGNS 12.3 TO DEC_FL_1 */
    DEC_FL_6 = -0.0123E3;       /* ASSIGNS -12.3 TO DEC_FL_6 */
```

Floating point decimal constants are coded as a decimal fixed point constant, followed by an E and an optionally signed decimal integer exponent of not more than two digits (the power of ten by which the fixed point decimal constant must be multiplied to give the true value). The implied precision of a floating point decimal constant is the number of decimal digits before the E.

The first two assignments in the example above have exactly the same effect - to set up a floating point decimal constant with a precision of 3 containing 12.3, and to assign that to DEC__FL__1. The last sets up a floating point decimal constant with a precision of 6, containing -12.3, and assigns that to DEC__FL__6.

Decimal floating point variables and constants use S/370 short, long or extended floating point form, depending on the precision used. The amount of space used is:

Precision	Number of Bytes
1-6	4
7-16	8
17-33	16

Binary Float

Variables with the attributes BINARY FLOAT hold and process data in exactly the same way as variables with the attributes DECIMAL FLOAT. The effective differences are in the declaration, and are:

The attribute BINARY (BIN) replaces the attribute DECIMAL.

Precisions are declared in the number of binary digits of precision required, instead of the number of decimal digits.

The range of precisions allowed is 1 to 109.

If a precision is not specified, a default of 21 applies.

Some examples are shown below.

```
DCL BIN_FL_1  FLOAT       BINARY(12);
DCL BIN_FL_2  FLOAT       BIN(12);
DCL BIN_FL_3  BIN         FLOAT(12);
DCL BIN_FL_4  FLOAT(12)BIN;
DCL BIN_FL_5  BIN(12)FLOAT;
DCL BIN_FL_6  BIN         FLOAT(21);
DCL BIN_FL_7  BIN         FLOAT(53);
DCL BIN_FL_8  BIN         FLOAT(109);

   BIN_FL_1 = 101E0B;        /* PUT 101B (5) INTO BIN_FL_1      */
   BIN_FL_1 = 10.1E1B;       /* PUT 101B (5) INTO BIN_FL_1      */
   BIN_FL_1 = 101E-1B;       /* PUT 101.0B (5) INTO BIN_FL_1    */
   BIN_FL_1 = 0.101010E3B;   /* PUT 101.01B (5,25) INTO BIN_FL_1*/
```

Floating point binary constants are coded in a program by writing:

an optional sign, which is the sign of the whole number

a string of binary digits, with or without a binary point

a letter E

an optionally signed decimal integer constant, which is the binary exponent, that is, the power of 2 by which the binary constant preceding the E must be multiplied to give the value intended

a letter B.

They have a precision implied by the number of binary digits before the E. The constants used in the assignments above have precisions of 3, 3, 4 and 6.

Binary floating point variables and constants use the same forms as decimal floating point variables and constants. The amount of space used is:

Precision	Number of Bytes
1-21	4
22-53	8
54-109	16

Floating point binary variables and constants are likely to be used rather than decimal floating point variables and constants when information about source data is available in binary form.

Between then, binary and decimal floating point variables provide an effective means of processing widely varying values, especially if, when writing the program, full information is not available on the magnitude of the numbers to be processed.

Before continuing with the next section, please do questions 11 to 13 at the end of this topic.

Undeclared and Partially Declared Variables

Mention has been made, in the description of the various attributes of omitting the precision or length from declarations.

PL/I will also take default action if other attributes are omitted, or if a variable is not declared.

The following rules describe the full situation:

If a variable is declared with base and scale, but no precision, a preciation will be given, as follows:

Attributes given BASE	SCALE	Attributes attached PRECISION
BINARY	FIXED	(15,0)
DECIMAL	FIXED	(5,0)
BINARY	FLOAT	(21)
DECIMAL	FLOAT	(6)

If a variable is declared with base, but no scale, the scale defaults to FLOAT. If a variable is declared with scale, but no base, the base defaults to DECIMAL. If the precision is also omitted, the normal default for that base and scale will be applied.

Attribute given	Attributes attached
BINARY	FLOAT (21)
DECIMAL	FLOAT (6)
FIXED	DECIMAL (5,0)
FLOAT	DECIMAL (6)

If a variable is not declared, or is declared but no given attributes, then it takes attributes depending on the first letter of its name. A variable which is undeclared is sometimes referred to as an **implicitly** declared variable.

First letter of name	Attributes attached
I – N	FIXED BINARY (15,0)
Others (A–H, O–Z, $, @, #)	FLOAT DECIMAL (6)

The use of these default attributes can save coding effort. However, coding full declarations may contribute towards the documentation of a program.

Before continuing with this topic, please attempt question 14 at the end of the topic.

Initialization

When a program starts executing, the value which will be in any variable is undefined and unpredictable. It may vary from run to run, and may not even be valid for that data type. To overcome this we may use the INITIAL attribute to specify a value. This is commonly used when setting up headings, or setting initial values in numeric variables which are to be used as counters or accumulators. There are many other used for this.

The initial attribute takes the general form:

INITIAL (constant)

INITIAL may be abbreviated to INIT.

The constant should be written according to the rules described earlier in this topic for constants of various types. That is, a character constant must have quotes surrounding it etc. Blanks may be inserted between the constant and the brackets.

The constant does not have to have identical attributes to the variable, as long as the value written can be held in the variable. The commonest use of this facility is that numeric variables are normally initialized with constants written with a fixed decimal format, whether the variable being initialized has a decimal or binary base, and a floating or fixed scale.

A character string constant used to initialize a character string-variable need not be as long as the variable. If it is shorter, then it will be padded on the right with blanks.

The rules which govern the conversion of constants to the attributes of the variables are the same as those which apply to assignments, and will be covered in Topic 4.

Below are some valid uses of the INITIAL attribute.

```
DCL HEADING      CHAR(30)          INIT('  QUARTERLY SALES REPORT');
DCL ACCUMULATOR  FIXED DEC(8,3)    INIT(0);
DCL PRODUCT      FLOAT DEC(6)      INIT(1);
DCL PI           FLOAT DEC(6)      INIT(3.14159);
DCL BILLION      FIXED DEC(10)     INIT(1.0E9);
```

HEADING has a length of 30, while the constant is only 24 characters long. The constant will have 6 blanks added to it on the right. HEADING will contain these characters when execution commences, and will retain them until they are over-ridden by an assignment to HEADING, or a read into HEADING.

The constant put into ACCUMULATOR has the same base and scale as ACCUMULATOR, so its precision only will be changed.

The constants used for PRODUCT, PI and BILLION all have different scales from the variables. In each case the constant has been written in the most convenient form. The initialization of PI and BILLION allows these constants to be written out once, and then the corresponding variable names can be used in calculations.

Below is an extension of the listing program of Topic 2, which prints a heading, lists some names and addresses and then prints the number of names and addresses listed, with a message. The heading is put into NAME_AND_ADDRESS by the initial attribute, and is printed in statement number 40. Statement number 50 over-writes this by its READ operation.

```
COUNT:  PROC OPTIONS (MAIN);                                          /* 10 */
        DCL NAME_AND_ADDRESS CHAR(80)                                 /* 20 */
            INIT('LIST OF NAMES AND ADDRESSES');
        DCL ICOUNT FIXED BIN(15) INIT(0);                             /* 30 */

        WRITE FILE (PRNTOUT) FROM (NAME_AND_ADDRESS);                 /* 40 */
        READ  FILE (CARDSIN) INTO (NAME_AND_ADDRESS);                 /* 50 */

        DO WHILE (NAME_AND_ADDRESS ¬= ' ');                           /* 60 */
           ICOUNT = ICOUNT + 1;                                       /* 70 */
           WRITE FILE (PRNTOUT) FROM (NAME_AND_ADDRESS);              /* 80 */
           READ  FILE (CARDSIN) INTO (NAME_AND_ADDRESS);              /* 90 */
        END;                                                          /* 100 */

        NAME_AND_ADDRESS = 'NUMBER OF NAMES AND ADDRESSES';           /* 110 */
        WRITE FILE (PRNTOUT) FROM (NAME_AND_ADDRESS);                 /* 120 */
        NAME_AND_ADDRESS = ICOUNT;                                    /* 130 */
        WRITE FILE (PRNTOUT) FROM (NAME_AND_ADDRESS);                 /* 140 */
        END;  /* COUNT */                                             /* 150 */
```

In statement number 70, 1 is added to ICOUNT for each name and address read. To make this correct for the first name and address read, ICOUNT is initialized to 0. The program will thus work correctly even if no names and addresses are read.

The printing of the final line - the count of the number of names and addresses, raises a problem. The printer expects that information transmitted to it shall be in EBCDIC code - PL/I CHARACTER data. ICOUNT will hold the number in binary form, and so must be converted. Statement number 130 will do this by its assignment. This type of conversion will be discussed in Topic 4.

The layout of the output may leave a little to be desired. Each WRITE produces a line of print. It would be more elegant to have some blank lines between the heading and the first name and address, and to have the number of names and addresses on the same line as the message. Techniques for doing this will be discussed in Topics 5, 6 and 15.

You have now covered most of the major data types used in a processing program. In later topics you will meet one further major data type which is principally used for the input and output of numeric data, and ways of grouping together sets of associated variables - perhaps the entries in a tax table, or the items that make up a record of some transaction.

In the next topic, we look at how to use these data types in calculations and to move data about in a program. Before looking at Topic 4, please attempt questions 15 and 16 at the end of this topic.

Exercises

1. Which of the following declarations are valid?

```
/* A */    DCL VAR_1 CHAR(20);
/* B */    DCL VAR-2 CHAR(3);
/* C */    DCLVAR_3  CHAR(3);
/* D */    DCL VAR_4, CHAR;
/* E */    DCL VAR_5 CHAR(2), VAR_6 CHAR(3);
/* F */    DCL (VAR_7, VAR_8) CHAR(4);
/* G */    DCL VAR_9, VAR_10  CHAR(5);
```

2. Rewrite the following group of declarations using the minimum number of characters - discounting blanks. Note that, for DOS/VS, the ENVIRONMENT attribute would also be required for the file declarations.

```
DCL VAR_1 CHAR(3);
DCL VAR_2 CHAR(4);
DCL INFIL FILE INPUT;
DCL OUTFIL FILE OUTPUT;
DCL VAR-3 CHAR(4);
```

```
DCL VAR_1 Char(3),
(VAR-2, VAR-3) Char(
(INFIL INPUT, OUTFI
                OUTPUT
```

3. Which of the following declarations are valid?

```
/* A */    DCL CHAR_1 CHAR(32767);
/* B */    DCL CHAR_2 CHAR(1);
/* C */    DCL CHAR_3 CHAR    (30);
```

4. Which of the following assignments are valid?

```
           DCL CHAR_4 CHAR(10);

/* A */    CHAR_4  = 2'ABCDE';
/* B */    CHAR_4  = 'ROB'S COAT';
/* C */    CHAR_4  = '_123.45678';
```

5. Declare a variable called JOKE which is large enough to hold the characters HAHA and assign HAHA to it. DCL JOKE char (4)
JOKE = (2)'HA'
6. Which of the following declarations are valid and what are the attributes of the validly declared variables?

```
/* A */     DCL (BIN_1 FIXED(15),
                 BIN_2 FIXED(31),
                 BIN_3 FIXED(31)) BIN;
/* B */     DCL  BIN_4 FIXED BINARY;
/* C */     DCL  BIN_5 FIXED BIN(32);
/* D */     DCL  BIN_6 BINARY FIX(15);
```

7. a) Write a declaration for a fixed binary variable called INT, just capable of holding integers up to 255. (256 is 2^8).

   b) Write a declaration for a fixed binary variable called FRACT, capable of holding numbers up to 15, correct to the nearest eighth. (16 is 2^4).

8. Which of the following declarations are valid?

```
/* A */     DCL FIXED_DEC_1 DEC FIXED(5);
/* B */     DCL FIXED_DEC_2 DEC FIXED;
/* C */     DCL FIXED_DEC_3 DEC FIXED(16,3);
/* D */     DCL FIXED_DEC_4 DEC FIXED(10,5);
```

9. a) Declare a fixed decimal variable called MONEY capable of holding dollar amounts less than $100 to the nearest penny.

   b) Declare a fixed decimal variable called WEIGHT capable of holding weights in kilograms up to 500 Kg., correct to the nearest gram.

10. a) State one advantage of fixed decimal over fixed binary.

    b) State one advantage of fixed binary over fixed decimal.

11. Which of the following declarations are valid?

```
/* A */     DCL FL_1 DEC FLOAT(6);
/* B */     DCL FL_2 FLOAT DEC(6,2);
/* C */     DCL FL_3 DEC FLOAT;
/* D */     DCL FL_4 FLOAT BIN(109);
```

12. a) Declare a variable called FL__5 to hold information to at least 10 significant decimal digits.

    b) Declare a variable called FL__6 to hold information to 40 significant binary digits.

13. a) State one advantage of floating point variables over fixed point variables.

    b) When might you use a BINARY FLOAT variable rather than a DECIMAL FLOAT variable?

14. What are the base, scale and precision, and the length of the following variables?

```
/* A */    DCL VAR_1 FLOAT;
/* B */    DCL VAR_2 FIXED;
/* C */    DCL VAR_3 BIN(15);
/* D */    DCL VAR_4;
/* E */    DCL VAR_5 CHAR;
/* F */    DCL VAR_6 DEC FIXED;
/* G */    DCL IVAR_7 FIXED(15);
```

15. Which of the following declarations are valid?

```
/* A */    DCL VAR_1 CHAR(20) INIT ;
/* B */    DCL VAR_2 CHAR(20) INIT( ' ');
/* C */    DCL VAR_3 FIXED BIN(15,0) INIT(1001B);
/* D */    DCL VAR_4 FIXED INIT(1001B) BIN(15,0);
/* E */    DCL VAR_5 CHAR(7) INIT(HEADING);
```

16. Write a declaration, writing the minimum amount, for a floating point decimal variable called CONV, which will hold data correct to 6 significant decimal digits and which will contain 2.20462 when the program starts executing.

Answers

1. a) Valid.

   b) Invalid. VAR-2 contains a special character, and so is not a valid identifier.

   c) Invalid. There should be a space between DCL and VAR__3.

   d) Valid. The list of attributes is optional, so this will be taken as a declaration of the identifiers VAR__4 and CHAR, both with default attributes.

   e) Valid.

   f) Valid.

   g) Valid. VAR__9 will take default attributes.

```
DCL VAR_1 CHAR(3),
    (VAR_2, VAR_3) CHAR(4),
    (INFIL INPUT, OUTFIL OUTPUT);
```

2. The order is not significant. Note that INPUT and OUTPUT imply the FILE attribute. Also, in DOS/VS, the ENVIRONMENT attribute would be required in the file declarations.

3. a) Valid. This is the largest character string variable that can be declared.

   b) Valid.

   c) Valid.

4. a) Invalid. If the 2 is meant to be a repetition factor, it should be parenthesized.

```
CHAR_4 = (2)'ABCDE';
```

   b) Invalid. The ' between B and S will be taken as the end of the character string. It should be doubled.

```
CHAR_4 = 'ROB''S COAT';
```

   c) Valid. It is quite valid to have a character string constant where all the characters are numeric.

5.

```
DCL JOKE CHAR(4);

JOKE = 'HAHA';    /* EITHER THIS STATEMENT */
JOKE = (2)'HA';   /* OR THIS               */
```

Even a simple problem may have more than one solution.

6. a) Valid. The declarations expand to:

```
DCL BIN_1 FIXED BIN(15);
DCL BIN_2 FIXED BIN(31);
DCL BIN_3 FIXED BIN(31);
```

Note that, although all three are BINARY FIXED, the precision must follow BINARY or FIXED, and so one of these must come inside the brackets. However, BIN_2 and BIN_3 have identical attributes and it is possible to nest the factoring of attributes, so:

```
DCL (BIN_1 FIXED(15), (BIN_2, BIN_3) FIXED(31)) BIN;
```

has an identical meaning.

b) Valid. As IBIN 4 starts with a letter I and is not given any precision, it defaults to FIXED BINARY (15).

c) Invalid. The maximum permitted precision is 31.

d) Invalid. There is no abbreviation for FIXED.

7.

```
/* A */    DCL INT FIXED BIN(8);
/* B */    DCL FRACT FIXED BIN(7,3);
```

8. a) Valid.

b) Valid. The precision defaults to (5).

c) Invalid. The maximum precision is 15.

d) Valid.

9.

```
/* A */    DCL MONEY FIXED DEC(4,2);
```

If the amount were held in cents, it could be held in a fixed binary field:

```
DCL CENTS FIXED BIN(10);
```

which would have the advantages that you are about to list for question 10 b).

```
/* B */        DCL WEIGHT FIXED DEC(6,3);
```

10. a) Fixed decimal variables may hold fractional decimal numbers exactly.

    They may hold larger numbers than fixed binary variables.

    Their contents can be converted to character format more quickly.

    b) Fixed binary variables may be processed more quickly than fixed decimal variables.

    They also hold data in a more compact form.

11. a) Valid.

    b) Invalid. The precision should be one number only - the number of decimal digits to hold.

    c) Valid. It will default to DECIMAL FLOAT (6).

    d) Valid. This is the maximum precision.

12.

```
/* A */        DCL FL_5 DECIMAL FLOAT(10);
/* B */        DCL FL_6 BINARY  FLOAT(40);
```

13. a) Floating point variables hold a constant number of significant digits, irrespective of the magnitude of the number. They are more flexible.

    b) If the initial information came in binary form, and you knew how many binary digits were to be held, then the variable used might be declared as BINARY FLOAT. The two data types are inter-changeable.

14.

```
/* A */        DCL VAR_1 FLOAT DEC(6);
/* B */        DCL VAR_2 FIXED DEC(5);
/* C */        DCL VAR_3 FLOAT BIN(15);
/* D */        DCL VAR_4 FLOAT DEC(6);
/* E */        DCL VAR_5 CHAR(1);
/* F */        DCL VAR_6 FIXED DEC(5);
/* G */        DCL IVAR_7 FIXED DEC(15);
```

The letter I at the beginning of IVAR 7 has no influence as there are some attributes declared.

15. a) Invalid. A value, in brackets, must appear after INIT.

    b) Valid. The blanks between the open bracket and the first quote have no significance. The string of 2 blanks will be padded up to 20 with blanks.

c) Valid. This would put an initial value of 9 into VAR_3. It may well be easier to write 9 as the constant, and will be easier to read.

d) Valid. The order of attributes does not matter, as long as the bracketed constant immediately follows INIT.

e) Invalid. HEADING is not a valid constant, it should be 'HEADING'.

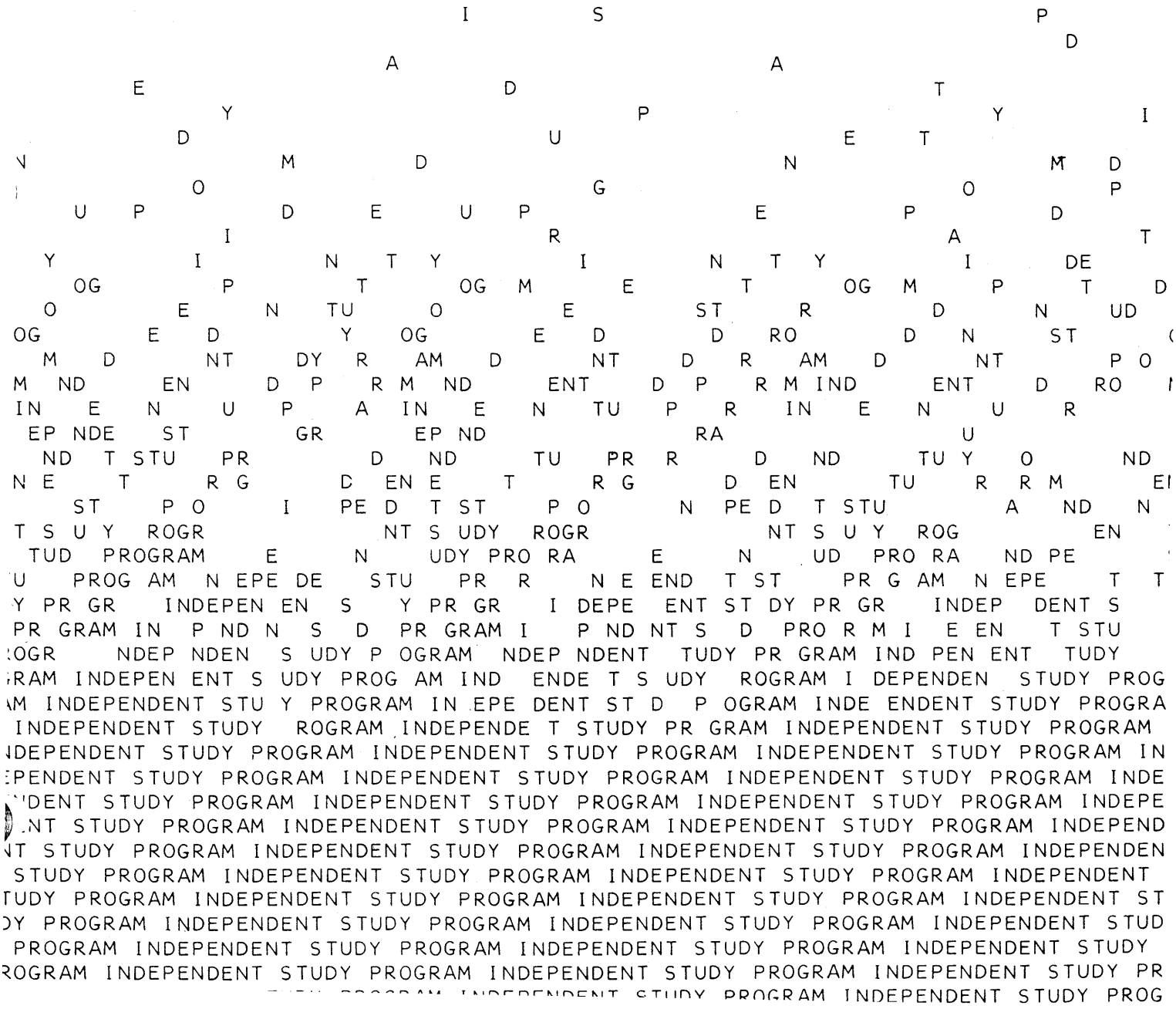
16.

```
DCL CONV INIT(2.2Ø462);
```

CONV will take the attributes FLOAT DEC(6) by default.

Topic

4



Topic 4

The Assignment Statement

This topic examines the processes of assignment, expression evaluation and data conversion for element variables. It covers both numeric and string data.

Objectives

On completion of this topic you should be able to:

- write valid assignment statements
- calculate the precision of arithmetic expression evaluation
- use the concatenation symbol to perform character expression evaluation.

Introduction

The assignment statement is an important statement in PL/I. It is the statement by which data is manipulated in main storage and by which calculations are performed. During the processes of assignment and expression evaluation, the variables involved may have different attributes and/or precision and therefore some conversion of the attributes may be necessary.

The Assignment Statement

General

The name of the assignment statement does not do it full justice. The statement does cause assignment - the copying of information into a data variable. It is also the statement by which calculations are performed in PL/I by the process of expression evaluation.

The basic form of the assignment statement is:

variable = expression;

The assignment statement is the only PL/I statement which does not have a keyword to identify it. It is recognized by the = character and the general construction. Like all PL/I statements, it terminates with a semi-colon. The expression on the right hand side of the = character - the source expression, is evaluated and the variable on the left hand side of the = character - the target variable, is given the value of the result of the expression evaluation. The target must be a variable name. It is the only variable whose value is changed by the execution of the statement, and it is changed as the last step of executing the statement.

The range of complexity allowed in the expression on the right hand side is considerable. At its simplest, it may be a constant or a variable. It may also be a complicated arithmetic or character string expression.

Arithmetic Assignments

The expression in an arithmetic assignment statement is similar to an algebraic expression, consisting of operands and operators. The operands are variables and constants, as discussed in Topic 3. Where they are constants, the value is used. Where they are variable names, the contents of the variables at the point at which the statement is executed are used. The operators and their meanings are:

OPERATOR	MEANING
+	add
-	subtract
*	multiply
/	divide
**	exponentiate (raise to the power of)

The following examples demonstrate the use of the arithmetic operators.

```
SUM                =  A + B;          /* 10 */
DIFFERENCE         =  A - B;          /* 20 */
PRODUCT            =  A * B;          /* 30 */
QUOTIENT           =  A / B;          /* 40 */
EXPONENTIAL        =  A**B;           /* 50 */

C  =  -A;                             /* 60 */
C  =  B + -A;                         /* 70 */
C  =  -B + A;                         /* 80 */
C  =  B - A;                          /* 90 */
```

In statement 10, the current contents of A and B will be added, and the sum will be put into SUM.

In statement 20, the current contents of B will be subtracted from the current contents of A and the result will be put into DIFFERENCE.

In statement 30, the current contents of A and B will be multiplied together and the result will be put into PRODUCT.

In statement 40, the current contents of A will be divided by the current contents of B, and the result will be put into QUOTIENT.

In statement 50, the current contents of A will be raised to the power of the current contents of B (A^B) and the result will be put into EXPONENTIAL. Note that the exponentiation operator must be coded as two * characters in adjacent columns.

The + and - operators as used in statements 10 and 20 are called infix operators. An infix operator operates on two operands and is identified by having an operand on either side. Thus the infix operator + in statement 10 has the operands A and B on either side of it and causes them to be added together. They may also be used as prefix operators. A prefix operator is one which operates on one operand only. It is identified by not having an operand to the left of it. Either it is the first character of the expression, as in statements 60 and 80, or the item to the left of it is an operator, as in statement 70. If two or more operators in an expression are not separated by operands, then all but the left-most are prefix operators and must be + or -. The effect of a - prefix is to multiply the operand by -1, effectively to change its sign. Thus, statements 70 and 90 have identical effect. The + prefix operator has no effect. Later in this topic you will see that it is important to be able to differentiate between prefix and infix operators.

Below in statement 10, the variable V1 is used five times. At the beginning of this topic it was stated that the only variable to have its contents changed is the target variable, and this is only done as the last step in the execution of the assignment statement. Hence, statement 10 and 20 have identical meanings.

```
V1 = V1 + V1 + V1 + V1;                                    /* 10 */
V1 = 4 * V1;                                               /* 20 */
V1 = V2 + V3 + V4;                                         /* 30 */
V1 = V2 * V3 + V4 * V5;                                    /* 40 */
AREA = PI * R ** 2;                                        /* 50 */
PERCENT = VARIABLE / BASE * 100;                           /* 60 */
X = -Y ** -2;                                              /* 70 */
A = -B ** -C ** -D;                                        /* 80 */
A = -D;                                                    /* 90 */
A = C ** A;                                                /* 100 */
A = -A;                                                    /* 110 */
A = B ** A;                                                /* 120 */
A = -A;                                                    /* 130 */
A = -(B ** -(C ** (-D)));                                  /* 140 */
RESULT = (A + B) / (C + D);                                /* 150 */
VALUE = (A + B) / (C + D)**(E + F);                        /* 160 */
```

The value of the result assigned to V1 in statement 30 does not depend on the order in which the two additions are carried out. However, the order in which the operations are carried out in statement 40 onwards is critical.

For example, the expression in statement 40 could be evaluated as:

(V2*V3) + (V4*V5)

or V2* (V3+V4) * V5

If we let the current contents of the variables be:

V2 = 2

V3 = 3

V4 = 4

V5 = 5

The result of evaluating the expression in the first way will be:

(V2*V3) + (V4*V5)

= (2*3) + (4*5)

= 6 + 20

= 26

The result of evaluating it the second way will be:

V2 * (V3+V4) * V5

= 2 * (3+4) * 5

= 2 * 7 * 5

= 70

To remove this ambiguity, PL/I has a hierarchy of operators which is shown below. The hierarchy determines the order in which any expression is evaluated.

Operators	Priority	Order of Processing
**		
*		

In statement 40 above, there are two operators, * and +. In the priority table, the prefix + is shown as having the highest priority, together with **. The infix + and - have the lowest priority. In this statement, the + has an operand either side of it, and so is an infix + which has a lower priority than *.

The expression will be evaluated as:

(V2*V3) + (V4*V5)

The order in which the two parenthesized expressions are evaluated is not significant.

The ** operator of statement 50 has a higher priority than the * operator, and so is processed first. This means that the expression does give the result desired. The current value in R will be squared, and that value will be multiplied by the current contents of PI, which hopefully will have been set to 3.1416 - probably by initializing it.

Statement 60 has two operators with the same level of priority. The order in which they are processed is critical, and will cause either:

VARIABLE/(BASE * 100)

or (VARIABLE/BASE) * 100

The last column of the priority table shows the order in which they will be processed, from left to right. VARIABLE will be divided by BASE, and the result of that will be multiplied by 100. This is the second option above, and is presumably what is desired.

Addition and subtraction are on the same level of priority, and a series of these will be processed from left to right also.

Statement 70 shows a statement containing three operators, all with the highest possible priority, since the two - signs are prefix -s. Operators at the highest priority are processed from right to left, and thus the expression means:

$-(Y^{-2})$

Similarly, the expression in statement 80 will be evaluated as the sequence of statements 90 to 130.

Although the order in which the expression in statement 80 will be evaluated is unambiguously defined, the exact meaning of it may not be immediately clear. It is not difficult to make a mistake in interpreting the order in which a complicated expression will be evaluated. Also, due to the priorities of operators, some expressions cannot be written as a single PL/I expression using only the operators and operands. To overcome these problems, parentheses may be used in PL/I statements, as in algebraic expressions, and their use will over-ride the normal priorities, if necessary.

Thus, statement 80 and statement 140 have the same effect. Where brackets are nested, the expression in the inner-most parentheses will be evaluated first, following the normal priority of operators. When this has been brought to a single result, this result will be used as an operand in evaluating the next level of parentheses etc.

The parentheses in statement 140 merely emphasize the order in which the expression is to be evaluated. If the parentheses are removed, the result will be unchanged. However, the expression:

$$\frac{A+B}{C+D}$$

could not be coded in PL/I in one statement without the use of parentheses. Statement 150 shows it incorporated in a PL/I statement.

In statements like 150 and 160, where there are several pairs of parentheses at the same level of nesting, the order in which the expressions in them are evaluated is not defined, and is not significant. When all parentheses at the lowest level have been evaluated, then the rules of priority of operators will be applied at the next level of nesting, so that the expression in statement 160 is:

$$\frac{A+B}{(C+D)^{E+F}}$$

You should now attempt questions 1 and 2 in the exercises at the end of this topic.

Precision in Expression Evaluation

As we have seen, when an arithmetic expression is evaluated, it is as a series of steps. Each step involves one operator and two operands, and produces a single result.

This result will be held in a work field, which is supplied by the system, and will be used as an operand in the next step, if any.

Thus, statement 30 involves three steps:

1) Add the contents of V2 and V3 and hold the result.

2) Add this result and the contents of V4.

   Hold that result.

3) Assign it to V1.

The intermediate work fields, like all other fields, must have attributes of base, scale and precision. These attributes depend on the attributes of the operands and on the operation which generated the result. In turn, the attributes of an intermediate result will influence the attributes of further intermediate results, and the final result.

The purpose of this section is to define the rules which determine the attributes of intermediate results.

We shall be considering two operands with precisions (p_1,q_1) and (p_2,q_2) which will produce a result with precision (p,q). If an operand or the result has a binary base, the relevant p and q will be in binary digits. If it has a decimal base, they will be in decimal digits. If an operand or the result is of floating point scale, the relevant q will not apply.

Attributes of result	Precision of the result		
	ADDITION or SUBTRACTION	MULTIPLICATION	DIVISION
FIXED DECIMAL (p, q)	$p = 1 + MAX(p_1 - q_1, p_2 - q_2) + q$ $q = MAX(q_1, q_2)$	$p = p_1 + p_2 + 1$ $q = q_1 + q_2$	$p = 15$ $q = 15 - ((p_1 - q_1) + q_2)$
FIXED BINARY (p, q)			$p = 31$ $q = 31 - ((p_1 - q_1) + q_2)$
FLOAT DECIMAL (p)	$p = MAX(p_1, p_2)$		
FLOAT BINARY (p)			

Note: For FIXED DECIMAL results, if the calculated value of p is greater than 15, it will be reduced to 15. positions

For FIXED BINARY results, if the calculated value of p is greater than 31, it will be reduced to 31.

Base and Scale the Same

The precision of the result of any operation, where the bases and scales of both operands are the same, is shown in the preceeding table.

Floating point operands are dealt with in a completely different manner from fixed point operands. If the operands are floating point, then the precision of the result is the maximum of the precisions of the two operands. This applies whatever operation is performed.

```
DCL FLD_1 FLOAT(4) INIT(100.0);      /* 10 */
DCL FLD_2 FLOAT(6) INIT(999.999);    /* 20 */
DCL FLD_3 FLOAT(14);                 /* 30 */

    FLD_3 = FLD_1 * FLD_2;           /* 40 */
```

When statement 40 is executed, FLD__1 and FLD__2 will be multiplied together to give a result whose precision is MAX(4,6) = 6. This result will be assigned to FLD__3, and will be padded to the precision of FLD__3 with non-significant zeros.

When dealing with fixed point operands, the precision of the result is dependent on the operation. In all cases, the total precision, p, cannot exceed the maximum for that data type - 15 for FIXED DECIMAL, and 31 for FIXED BINARY. If the calculated value exceeds these limits, it will be reduced to the limiting value.

Addition and Subtraction

```
DCL FD_1 FIXED(5,0) INIT(99999);                    /* 10 */
DCL FD_2 FIXED(8,2) INIT(999999.99);                /* 20 */
DCL FD_3 FIXED(9,2);                                /* 30 */

    FD_3 = FD_1 + FD_2;                             /* 40 */

    FD_4 = FD_1 + FD_2;                             /* 50 */
```

When two fixed point variables are added, the extreme situation is when both variables hold the largest numbers that they can. e.g.:

```
FD_1      99999.
FD_2    999999.99
       1099998.99
```

The number of fractional digits in the result will be the higher of the numbers of fractional digits in the operands. The number of integer digits will be one more than the higher of the numbers of integer digits in the operands.

This is the precision given by the formula for the addition and subtraction of fixed point variables and constants.

For this example:

$q=\max(q_1,q_2)$
$= \max(0,2)$
$= 2$

$p=1+\max(p_1-q_1,p_2-q_2)+q$
$= 1+\max(5,6)+2$
$= 9$

Where the application of this formula gives a value of p greater than the permitted maximum for that base, p will be reduced to that maximum. This will cause the number of integer digit positions to be reduced.

```
DCL FD_1 FIXED(13,3) INIT(5555555555.555);
DCL FD_2 FIXED(13,6) INIT(   5555555.555555);
DCL FD_3 FIXED(15,6);


    FD_3 = FD_1 + FD_2;
```

The precision needed for the result will be:

$q=\max(q_1,q_2)$
$= \max(3,6)$
$= 6$

$p=1+\max(p_1-q_1,p_2-q_2)+q$
$=1+\max(13-3,13-6)+6$
$= 17$

As a precision of (17,6) is not allowed, it will be reduced to (15,6). In this situation, the sum will be:

```
FD_1    555,555,555.555
FD_2      5,555,555.555,555
        561,111,111.110,555
```

which can be held in a fixed point decimal field with precision (15,6).

However, if the value in FD__1 were 5,555,555,555.555, the sum would become:

```
FD_1    5,555,555,555.555
FD_2        5,555,555.555,555
        5,561,111,111.110,555
```

This value could not be held in a precision of less than (16,6). In this situation, the result obtained is not defined.

It is the programmers responsibility to ensure that the result of the calculation can be held in the result field which will be provided.

Similar considerations apply to subtractions. The extreme situation here is when one operand holds the maximum positive value and the other holds the maximum negative value.

Multiplication

In multiplication, the extreme situation is again when both operands hold their maximum value, positive or negative. The formula which gives the precision of the result allows for this situation.

Example:

```
DCL FD_1 FIXED(2,1) INIT(9.9);
DCL FD_2 FIXED(3,2) INIT(9.99);
DCL FD_3 FIXED(6,3);

FD_3 = FD_1 * FD_2;
```

The calculation performed is:

```
9.99 * 9.9 = 98.901
```

The precision of the result will be:

$$p = p_1 + p_2 + 1 = 2 + 3 + 1 = 6$$

$$q = q_1 + q_2 = 1 + 2 = 3$$

A field with a precision of (6,3) will hold any result which can be produced by multiplying together these two variables.

As with addition and subtraction, when the required precision is not available due to limits on maximum precision, the programmer must ensure that the result field given can hold the result obtained.

Division

In division the total number of digits needed to hold the quotient with complete accuracy is not predictable from the numbers of digits in the dividend and divisor. Thus, 1/3 would need an infinite number of digits for the quotient: 0.333....

Because of this, the formula for calculating the precision of the result of a division always sets p to the maximum for the base of the operands. What can be predicted from the precision of the operands is the maximum number of integer digits in the quotient, which is the difference between p and q. The value calculated for q does not indicate how many fractional digits are needed to hold the result with complete accuracy, but how many digits are left for fractional places after allocating all that are needed to hold the integer part of the result.

Example:

```
DCL DIVIDEND FIXED(6,3) INIT(999.999);
DCL DIVISOR  FIXED(3,2) INIT(  0.01);
DCL QUOTIENT FLOAT(15);

   QUOTIENT = DIVIDEND / DIVISOR;
```

The extreme situation for division is when the dividend holds the largest value it can and the divisor holds the smallest, (see example above). The result of dividing 999.999 by 0.01 is 99999.9, needing 5 integer places.

Application of the appropriate formula for division of fixed point decimal operands shows that the precision given for the result will be:

$$p = 15$$
$$q = 15-((p_1-q_1 + q_2)$$
$$= 15 - 5$$
$$= 10$$

Thus, the precision will be (15,10), which will give 5 integer digits and 10 fractional digits - the minimum number of integer digits required to hold the extreme situation.

If the values in the dividend and divisor are taken to the other extreme - the minimum value in the dividend and the maximum value in the divisor, the result will be:

```
0.001/9.99 = 0.0001 approximately
```

This will still be held in a result field with precision (15,10), and so will only be held to 7 significant figures, the first 8 digits being leading zeroes.

When selecting the precisions of fixed point variables which may be used in division, selecting the smallest precisions that are adequate for the data to be processed will give the maximum number of significant digits in the result. Using unnecessarily large variables will cause a loss of precision in the result.

Problems of this nature may be avoided by the use of floating point variables. With these, the number of significant digits held in the result will be the same as the precision of the result.

✓ You should now attempt question 3 in the exercises at the end of this Topic.

Bases and/or Scales Differ

All calculations in PL/I are done on data items which have the same base and scale, although the precisions need not be the same. If the programmer codes operands whose bases or scales differ, then the operands will be automatically converted to a common base and scale according to the following rules:

a) If the bases differ, the operand with a DECIMAL base will be converted to BINARY.

b) If the scales differ, the operand with a FIXED scale will be converted to FLOAT.

Examples:

```
DCL FIX_BIN   FIXED BIN(15,Ø) INIT(1Ø);          /* 1Ø */
DCL FIX_DEC   FIXED DEC(4,1)  INIT(2Ø);          /* 2Ø */
DCL FLO_BIN   FLOAT BIN(21)   INIT(3Ø);          /* 3Ø */
DCL FLO_DEC   FLOAT DEC(6)    INIT(4Ø);          /* 4Ø */
DCL ANSWER    FLOAT DEC(16);                     /* 5Ø */

   ANSWER = FIX_BIN + FIX_DEC;                   /* 6Ø */
   ANSWER = FIX_BIN + FLO_BIN;                   /* 7Ø */
   ANSWER = FIX_BIN + FLO_DEC * FIX_DEC;         /* 8Ø */
```

In statement 60, both have the same scale but the bases differ, so rule a) will apply and FIX__DEC will be converted to FIXED BINARY.

In statement 70, both have the same base, but the scales differ, so rule b) will be followed and FIX__BIN will be changed to FLOAT BINARY.

In both of the examples above, one operand was converted so that its base and scale were the same as those of the other. In statement 80, FIX__BIN has the dominant base - BINARY, but FLO__DEC has the dominant scale - FLOAT. In this case both operands are converted, so that the operation is carried out on operands which do not have the attributes of either of those coded. FIX__BIN and FLO__DEC will both be converted to FLOAT BINARY.

When operands are converted, the precision, as well as the base or scale will have to be changed.

If the precision before conversion is (p_1,q_1), then the precision after the conversion, (p_2,q_2), will be given by:

a) If the scale is changed from FIXED to FLOAT, the precision will be:

$$p_2 = p_1$$

b) If the base changes from DECIMAL to BINARY, the number of binary digits needed to represent each decimal digit is approximately 3.32.

A variable cannot have a fractional number of binary digits, so the precision will be:

$$p_2 = 3.32 * p_1$$
$$q_2 = 3.32 * q_1$$

where p_2 and q_2 are both rounded to the next higher integer.

c) If the base changes from DECIMAL to BINARY, and the scale changes from FIXED to FLOAT, then the precision of the converted operand is:

$$p_2 = 3.32 * p_1$$

rounded up to the next integer.

Applying these rules to statement 60, FIX__DEC will be converted to FIXED BINARY, with a precision:

$p_2 = 3.32*p_1$
$= 3.32* 4$
$= 13.28$
$= 14$

$q_2 = 3.32*q_1$
$= 3.32* 1$
$= 3.32$
$= 4$

The result of the addition will be in FIXED BINARY, and its precision will be that resulting from adding two FIXED BINARY variables with precisions (15,0) and 14,4):

$q = \max (q_1, q_2)$
$= \max (0,4)$
$= 4$

$p = 1 + \max (p_1-q_1, p_2-q_2) + q$
$= 1 + \max (15-0, 14-4) + 4$
$= 1 + 15 + 4$
$= 20$

In statement 70, FIX__BIN will be converted to FLOAT BINARY, with a precision of (15).

Statement 80 has two operations. The order in which they are carried out will depend on the relative priorities of the operators. The multiplication has the higher priority, and will be performed first. Both operands have the same base, but their scales differ, so FIX__DEC will be converted to FLOAT DECIMAL, with a precision (4). The product will also be in FLOAT DECIMAL, and its precision will be given by:

$p = \max (p_1, p_2)$
$= \max (6,4)$
$= 6$

This result will now form the second operand in the addition, giving an addition of a FIXED BINARY (15) to a FLOAT DECIMAL (6). Both operands will be converted to FLOAT BINARY. The converted precision of FIX__BIN will be (15). The converted precision of the product of FLO__DEC and FIX__DEC will be:

$p = 3.32 * p_1$
$= 3.32 * 6$
$= 19.92$
$= 20$

The precision of the sum will be:

$p = \max (p_1, p_2)$
$= \max (15,20)$
$= 20$

In all cases where conversion is done, the contents of the original variable will not be changed. A work field with the converted attributes is supplied by the system and the converted value is held there.

Conversions are all done automatically. They become important to the programmer when conversion is done from FIXED DECIMAL to FIXED BINARY. Fixed binary fields may not hold as large a number as fixed decimal fields, and fractional decimal numbers are not held exactly.

You should now attempt question 4 in the exercise at the end of this topic.

Conversion of Assignments

In each of the examples the attributes of the result did not match the attributes of the target variable, ANSWER. In each case, conversion will be done to the attributes of ANSWER.

When converting the attributes of the result to the attributes of the target, the base and scale will first be changed, if necessary, and their precision will be adjusted.

Base and Scale

Apart from the conversions which may occur during expression evaluation, conversions may occur from FLOAT to FIXED and from BINARY to DECIMAL.

When data is converted from FLOAT to FIXED, the precision of the converted value will be:

```
p = p1
q = the number of fractional digits in the value
    held in the FLOAT field.
```

When the data is converted from BINARY to DECIMAL, similar considerations apply as when converting from DECIMAL to BINARY; each decimal digit may represent approximately 3.32 binary digits, and the precision of the converted value will reflect this. Thus:

```
p = (p1/3.32) rounded up to the next integer
q = (q1/3.32) rounded up to the next integer
```

Precision

If the precisions of the converted value and the target do not match, the following will occur:

For floating point values, if the precision of the target is greater than that of the source, the source will be padded with lower order (right hand) zeroes.

If the precision of the target is less than that of the source, the source will have low order digits truncated.

For fixed point values, assignment will take place with decimal point alignment preserved. The fractional part will have the low order end padded or truncated, as appropriate. The integer part may be padded with high order (left hand) zeroes. If the integer part of the target is smaller than the integer part of the source, and the value in the source is such that significant digits would be lost, then the result is undefined.

You should now attempt question 5 in the exercises at the end of this topic.

Character Data in Arithmetic Expressions

It is possible to use character string variables and constants in arithmetic expressions, providing that the character variables contain valid arithmetic values. A valid arithmetic value in a character string variable consists of characters which obey the rules for numeric constants of one type or another, optionally preceded and followed by blank characters. There must be no alphabetic or special characters, other than those allowed in the format of numeric constants. The following are character string constants which could be used in an arithmetic expression.

Constant	Value
' 123.45'	123.45
' 123.45'	123.45
' 0123.45'	123.45
' 1.2345E+2'	123.45
' 101B'	5

The following character string constants could not be used in arithmetic expressions:

Constant	Reason
'APPLE PIE'	Alphabetic data
'12 34'	Embedded blank in the numeric characters
'12/34'	Arithmetic expressions are not allowed.

If character string variables or constants are used in simple assignment statements, then the attributes of the value they represent will be those implied by the way in which the value is written e.g.

Constant	Attributes
' 123.45'	FIXED DECIMAL (5,2)
' 123.45'	FIXED DECIMAL (5,2)
' 0123.450'	FIXED DECIMAL (7,3)
' 1.2345E+2'	FLOAT DECIMAL (5)
' 101B'	FIXED BINARY (3)

A simple assignment statement in this context has the form:

variable = variable;
or variable = constant;

If a character string variable or constant is used in an arithmetic expression, then the value contained in it will be converted to FIXED DECIMAL (15,0), whatever the implied attributes may be.

All conversion takes processing time, but the conversion of character string data to numeric is *more time consuming* than most and should be avoided where possible.

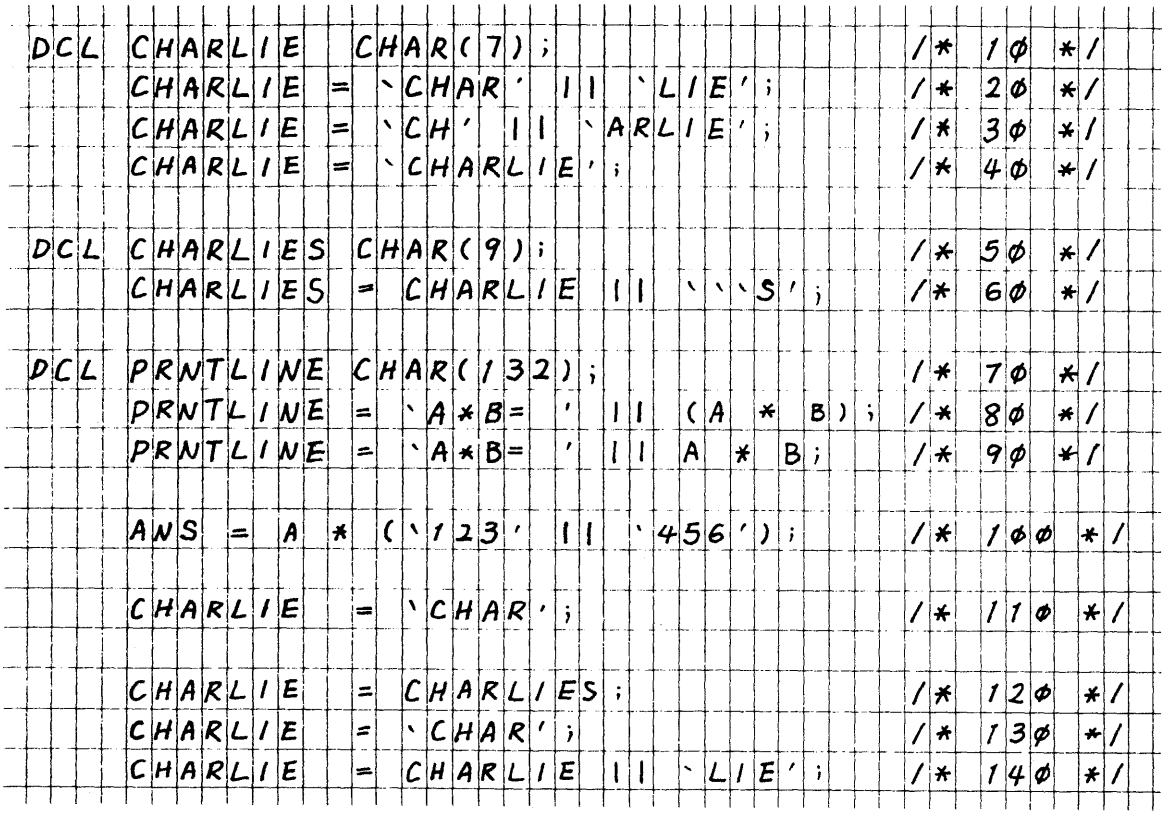
Numeric data which comes from punched cards or other keyboard input forms in character format. Ways of processing this data without doing slow character to numeric conversions will be discussed in Topic 6.

Character Expressions

A character expression is an expression whose result is a character string. There is only one operation which gives a character result - the concatenation operation. Concatenation is the process of joining together strings of characters to form a larger string. The length of the character string produced by a concatenation is equal to the sum of the lengths of the operands.

The operator used is, like the exponentiation operator, made up of two characters, in adjacent positions. It is two vertical lines, ++ in adjacent columns. Care should be taken when coding it to write full length lines, to avoid confusion with the number 1.

Below are some examples. In each case, the character string after the concatenation operator is joined to the character string before the operator with no intervening blanks. Thus, statements 20, 30 and 40 have identical meanings.



```
DCL  CHARLIE  CHAR(7);                                  /* 10 */
     CHARLIE = `CHAR' || `LIE';                        /* 20 */
     CHARLIE = `CH' || `ARLIE';                        /* 30 */
     CHARLIE = `CHARLIE';                              /* 40 */

DCL  CHARLIES CHAR(9);                                 /* 50 */
     CHARLIES = CHARLIE || ```S';                      /* 60 */

DCL  PRNTLINE CHAR(132);                               /* 70 */
     PRNTLINE = `A*B=' || (A * B);                     /* 80 */
     PRNTLINE = `A*B=' || A * B;                       /* 90 */

     ANS = A * (`123' || `456');                       /* 100 */

     CHARLIE = `CHAR';                                 /* 110 */

     CHARLIE = CHARLIES;                               /* 120 */
     CHARLIE = `CHAR';                                 /* 130 */
     CHARLIE = CHARLIE || `LIE';                       /* 140 */
```

Those statements all show both operands as character string constants, but either or both may be variables. Statement 60 shows the first operand as a variable and the second as a constant.

Just as character data may be used in arithmetic expressions, so numeric data and expressions may be used as operands of the concatenation operation.

Statement 80 shows a typical example of building up a printline to contain some text and numeric information. The variables A and B will be multiplied together. The result will be converted to a character representation of that value, and this will be concatenated to 'A*B=' and the whole assigned to PRINTLINE.

Statement 90 is the same as statement 80, except that the brackets around A*B have been removed. What will be the effect? Will the concatenation be performed before or after the multiplication? The situation is defined by an extension of the hierarchy of operators, shown below.

Operators	Priority	Order of Processing
** prefix + prefix -	highest	right to left
* /		left to right
+ -		left to right
||	lowest	left to right

From this it can be seen that the concatenation operator has a lower priority than any of the arithmetic operators, and so will be performed after all arithmetic operations in the expression. These priorities can be over-ridded by brackets, as with the arithmetic operators on their own.

Statement 100, however unlikely to be written, is quite legal. Because of the brackets, the priority will be over-ridden. The concatenation of '123' and '456' will be done first to give '123456', a character string of length 6. This will then be converted to FIXED DECIMAL (15) and multiplied by the contents of A, with further conversion, if necessary.

This expression is said to be an arithmetic expression, because the last operation carried out is an arithmetic operation, giving an arithmetic result.

Assignment of Character Expressions

When the result of a character expression is assigned to a character string, the following rules apply.

1) If the lengths of the target and source are the same, then direct assignment takes place.

2) If the target is longer than the source, then the source will be padded with blanks on the right up to the length of the target.

3) If the target is shorter than the source, the source will be truncated on the right.

This means that statements 20, 30, 40 and 60 will give direct assignment with no padding or truncation. In statement 110, 'CHAR' will be padded to 'CHARbbb' before any assignment, and in statement 120, 'CHARLIE''S' will be truncated to 'CHARLIE' before assignment.

It should be noted that the sequence of statements 130 and 140 will not have the effect which appears to be intended. Statement 130 will cause 'CHARbbb' to be assigned to CHARLIE. The expression in statement 140 will cause the string 'CHARbbbLIE' to be generated and this will be truncated to 'CHARbbb' on assignment to CHARLIE, so that statement 140 has no effect on the contents of CHARLIE.

As was noted earlier in this topic, if a character string constant, variable or expression is assigned to a numeric variable, then it must contain characters which represent a numeric constant of some form, optionally surrounded by blanks. The attributes of the constant will be those implied by the format of the constant.

The assignment statement may take many forms, and many levels of complexity. It is a very important statement in PL/I. You will learn of further forms of the assignment statement in later topics. Before continuing, you should complete the exercises at the end of this topic.

Exercises

1) By the use of brackets, show the order in which the expressions in the following statements will be evaluated.

```
/* A */    X = A * B + C;
/* B */    X = C + A * B;
/* C */    X = (A * B) * C;
/* D */    X = A ** (B ** C);
/* E */    X = (A / B) * C;
/* F */    X = (-A ** B) - (A ** B);
/* G */    X = (A + (B * C)) + ((D / E) * (F ** 2));
```

2) Write the following algebraic expressions as PL/I expressions.

a) $(A+B)^2$ `X = A + B**2`

b) $\frac{4}{A+B+C+D}$ `4 / (A+B+C+D)`

c) $\frac{4}{\frac{1}{A}+\frac{1}{B}+\frac{1}{C}+\frac{1}{D}}$ `4 / ( 1/A + 1/B + 1/C + 1/D )`

d) $\left(\frac{A-ABAR^2}{N}\right)^{1/2}$ `((A - ABAR**2)/N)**2`

e) $\frac{A+B}{C+D}$ `(A+B) / (C+D)`

3)

```
DCL FIX_B1 FIXED BIN(15,0);
DCL FIX_B2 FIXED BIN(18,3);
DCL FIX_D1 FIXED DEC(7,2);
DCL FIX_D2 FIXED DEC(7,4);
DCL FIX_D3 FIXED DEC(6,3);
DCL FL_D1  FLOAT DEC(6);
DCL FL_D2  FLOAT DEC(16);
```

Given the above declarations, what will be the precisions of results R1 to R9 in the following expressions?

```
/* A */ R1 = FIX_B1 + FIX_B2;
/* B */ R2 = FIX_B1 * FIX_B2;
/* C */ R3 = FIX_B1 / FIX_B2;
/* D */ R4 = FIX_D1 - FIX_D2;
/* E */ R5 = FIX_D1 * FIX_D2;
/* F */ R6 = FIX_D1 / FIX_D2;
/* G */ R7 = FL_D1 / FL_D2;
/* H */ R8 = FIX_D1 / FIX_D2 * FIX_D3;
/* I */ R9 = FIX_D1 * FIX_D2 * FIX_D3;
```

H (15,6) * (6,3) (15,9) (15,9)

4) Using the variables declared in question 3), what will the attributes of the results of the following expressions be? What problems might arise in each case?

Changed to Binary

```
/* A */   R1 = FIX_B1 + FIX_D2;
/* B */   R2 = FIX_D1 * FL_D1;   to float pt but
/* C */   R3 = FIX_B1 + FL_D1;
/* D */   R4 = FIX_D1 + FIX_D2 * FIX_B1;
/* E */   R5 = FL_D1 + FIX_D2 * FIX_B1;
```

5) In the following assignments, what will be the base, scale and precision of the converted source value before the precision is adjusted to match that of the target?

```
/* A */   FIX_B1 = FIX_D1;
/* B */   FIX_D1 = FIX_B1;
/* C */   FL_D1  = FIX_D1;
/* D */   FIX_D1 = FL_D1;
/* E */   FL_D1  = FIX_B2;
```

6)

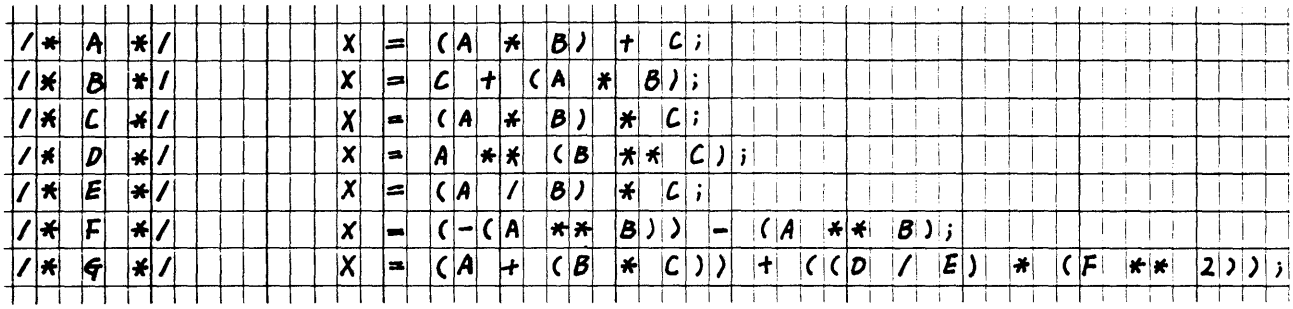
```
DCL CV1  CHAR(5);
DCL CV2  CHAR(7);
DCL CV3  CHAR(14);
DCL NUM  FIXED DEC(5,2);
```

Given the declarations above, what will be the contents of the target variable after each of the following assignments?

```
/* A */    CV2 = 'WALTER';
/* B */    CV1 = CV2;
/* C */    CV3 = CV2 || 'GABRIEL';
/* D */    CV3 = 'WALTER';
/* E */    CV3 = CV3 || 'GABRIEL';
/* F */    CV1 = '111.11';
/* G */    CV2 = '222.22';
/* H */    CV3 = CV1 || CV2;
/* I */    NUM = CV1;
/* J */    NUM = CV1 + CV2;
```

Answers

1)



2) These solutions use the minimum parentheses. Extra parentheses could be used in c) and d).

a) (A+B)**2

b) 4/(A+B+C+D)

c) 4/(1/A + 1/B + 1/C + 1/D)

d) ((A - ABAR) **2/N) ** (1/2)

e) (A+B) / (C+D)

3)

a) FIXED BIN(19,3)

b) FIXED BIN(31,3)

c) FIXED BIN(31,13)

d) FIXED DEC(10,4)

e) FIXED DEC(15,6)

f) FIXED DEC(15,6)

g) FLOAT DEC(16)

h) FIXED DEC(15,9)

i) FIXED DEC(15,9)

4)

a) FIX__D2 will be converted to FIXED BIN(24,14).

The result will be in FIXED BIN(30,14).

b) FIX__D1 will be converted to FLOAT DEC(7).

The result will be in FLOAT DEC(7).

c) Both operands will have to be converted; FIX__B1 to FLOAT BIN(15), and FL__D1 to FLOAT BIN(20).

The result will be in FLOAT BIN(20).

d) The multiplication will be done first.

FIX__D2 will be converted to FIXED BIN(24,14), and the intermediate result will be in FIXED BIN(31,14). For the addition, FIX__D1 will be converted to FIXED BIN(24,7), and the final result will be in FIXED BIN(31,14).

e) The multiplication will be done first in the same way as in d).

The result will be in FIXED BIN(31,14). For the addition, FL__D1 will be converted to FLOAT BIN(20), the intermediate result will be converted to FLOAT BIN(31), and the final result will be in FLOAT BIN(31).

5)

a) FIX__D1 will be converted to FIXED BIN(24,7).

On assignment, any fractional part of FIX__D1 will be truncated. If the value in FIX__D1 cannot be held in FIXED BIN(15), the result of the assignment will be undefined.

b) FIX__B1 will be converted to FIXED DEC(5,0).

Any value which can be held in FIX__B1 can be held in FIX__D1.

c) FIX__D1 will be converted to FLOAT DEC(7).

Approximately 1 decimal digit of precision will be lost in the assignment.

d) FL__D1 will be converted to FIXED DEC(6,q).

The value of q is not defined by the precision of FL__D1, but by the magnitude of the value held in it. The value could be too large to be held in FIX__D1, so causing an undefined result, or could be so small that no significant digits are stored in FIX__D1; only leading zeroes.

e) FIX__B2 will be converted to FLOAT DEC(6). No problems will arise, as this is the same precision as FL__D1.

6)

a) 'WALTERb'

b) 'WALTE'

c) 'WALTERbGABRIEL'

d) 'WALTERbbbbbbbb'

e) 'WALTERbbbbbbbb'

An intermediate field of 'WALTERbbbbbbbbGABRIEL' will be formed, but this will be truncated on assignment.

f) '111.1'

g) '222.22b'

h) '111.1222.22bbb'

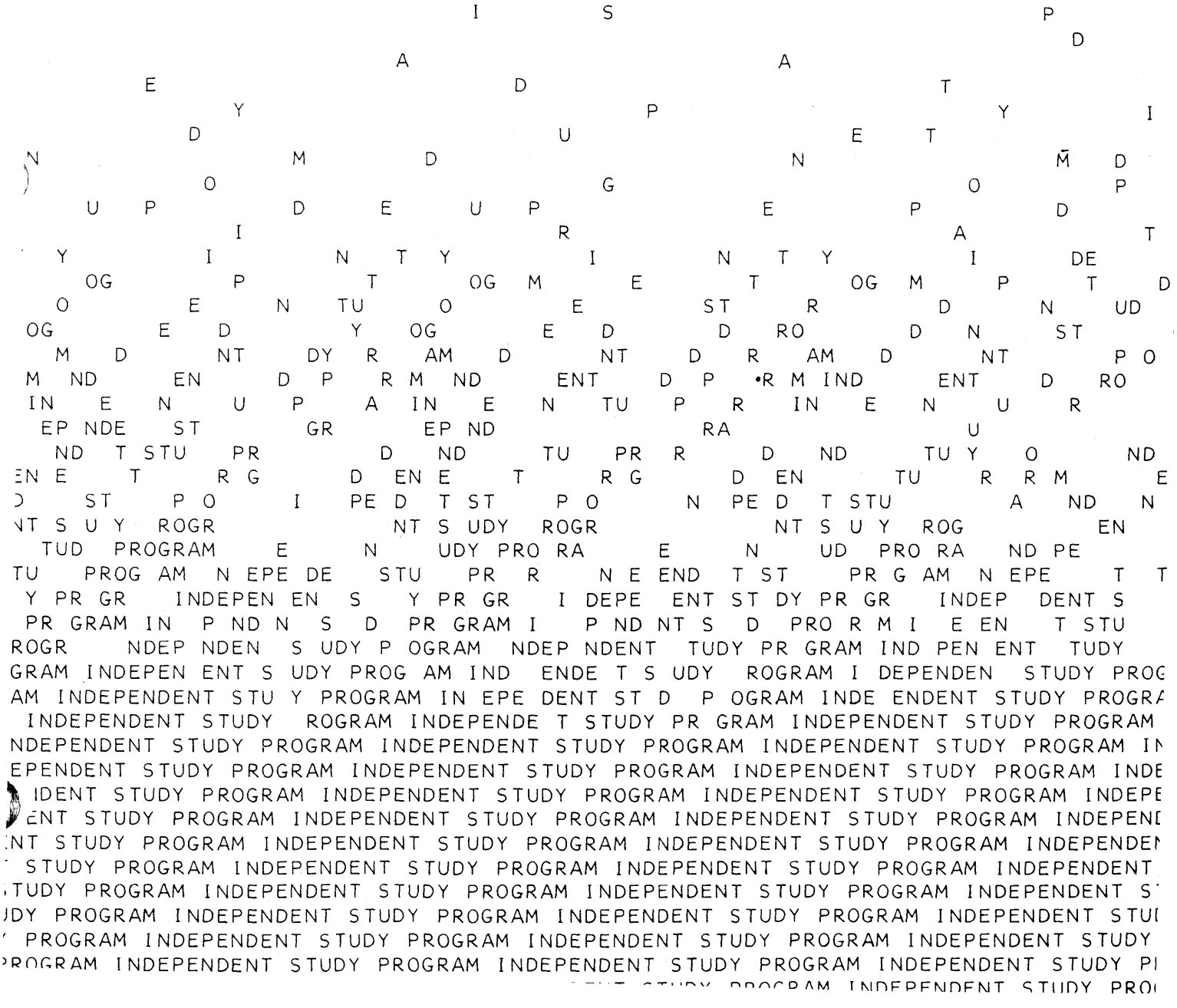
i) 111.10

j) 333.00

When the character string variables are converted to FIXED DEC(15), the fractional part will be lost.

Topic

5



Topic 5

Record Input/Output Part 1 - Move Mode

This topic deals with MOVE mode input/output. It covers the statements used, their effects, and the declaration of the files.

Objectives

On completion of the topic you should be able to:

- read, write and update records in consecutively organized files using MOVE mode input/output
- understand the process of the physical movement of data between the external device and main storage
- write file DECLARE statements
- write OPEN and CLOSE statement.

Introduction

PL/I has two distance approaches to input/output - 'record' and 'stream'. Record is more frequently used within installations because it is a much quicker form of input/output. Stream is reserved normally for particular uses, for instance the output of debugging information or when greater control is required over the format of printed output. Stream input/output will be discussed in Topic 15.

Within record input/output there are two modes - move mode and locate mode, the main difference being a question of where the data will be processed. This topic discusses move mode record input/output. Locate mode will be discussed in Topic 8.

General

Data held externally to a program is held in data sets. Data sets may be organized so that the records in them may be accessed sequentially, directly via an index or directly by the record address. These data set organizations are called SEQUENTIAL, INDEXED SEQUENTIAL, DIRECT and VIRTUAL STORAGE ACCESS METHOD. We are concerned here only with processing SEQUENTIAL data sets. The others will be discussed in later topics.

External data is represented in a PL/I program by a FILE. The actual data set to be processed is defined by Job Control Language, thus a program can be used to process different data sets without modification, so long as the data sets match the FILE attributes in the program. Job Control Language is described in more detail in the appropriate OS/VS and DOS/VS courses. A file must be given attributes by declaration. Some of these attributes relate to the organization of the external data set with which it will be associated, others relate to the way in which it is to be processed. File declarations will be dealt with later in this topic.

The READ Statement

The READ statement causes the transfer of a logical record from an external storage device to a variable in main storage. The external device may hold several logical records in one physical record or block. This is done for efficient usage of the external device. The logical records will be extracted from the physical records automatically by the READ statement. This process is called de-blocking.

The program may be processing several files, perhaps a master stock file holding information on part numbers, stock levels etc., a transaction file holding information on stock issues and receipts, and a report file to print notices of items to be re-ordered. The program will almost certainly use many variables. The READ statement then, must identify the operation to be performed, the particular file to be read from, and the variable in which the record is to be stored.

Hence:

READ FILE(file name) INTO(variable name);

The keyword READ must be the first part of the statement. The options FILE (file name) and INTO (variable name) may be in either order.

The file name is an identifier, which may be referred to outside the procedure in which it is declared. Like a procedure name, its length may not be greater than 7 characters, and it may not contain the break character.

For DOS/VS users, some attributes **must** be given in a DECLARE statement. For both DOS/VS and OS/VS users all attributes may be given in a DECLARE statement, but some may be acquired by implication from other attributes, or by default. This will be covered later in this topic.

The variable in the INTO option may have any attributes of type, base, scale, precision or length, but must match the attributes of the data in the record read. The length of the variable must be the same as the length of the logical record, and its base, scale and precision, or type and length must be the same as those of the data in the record. If the data originates from keyboard driven devices, such as card punches, diskettes, visual display units and such, it will be in a character format, even if the characters are numeric. If it comes from a magnetic disk (DASD), or tape, it may be in any format.

Wherever the record comes from, it is likely that it will contain several fields. For instance, a record on a personnel file might contain a person's name, their personnel number, department, age, sex, position etc. This information will normally be held in one record, and the different parts or fields of the record must be separated to be used in the program. Topic 6 describes techniques for doing this by using aggregates. An aggregate is a group of variables which may be referred to by a group name, which is used in the READ statement. The fields of the record are then referred to by the individual variable names.

The WRITE Statement

The WRITE statement performs the reverse function to the READ statement, and takes the form:

WRITE FILE(file name) FROM(variable name);

The statement type is identified by the first keyword, WRITE, and the file and variable are identified by the options FILE and FROM, which must follow WRITE, but may be in either order.

The contents of the variable will be copied out as the next record on the file identified.

The name used for the file must conform to the same rules that apply to the name used for the file in READ statement. It must be associated with a data set on a suitable device - not, for example, a card reader.

The variable may be of any type. If the information is to go to a printer, it must be in character format. If it is to go to a magnetic device, it may be in any format, but the variable or aggregate which is used to read it back must have the same attributes as the variable or aggregate used to write it out. The length of the logical records on the data set and the length of the variable used in the WRITE statement must be the same.

Input and Output MOVE Mode

The Input Operation

Let us consider first, data being read directly from punched cards, and look at the operation of a card reader.

A card reader may work at a rate of 1000 cards per minute, but this is very slow relative to the rate at which the CPU operates on data in main storage. Further, the operation of reading a card and storing the information in main storage is controlled by the channel. The CPU merely initiates the process and is free to do other work while the channel obtains the record required.

A similar situation occurs when data is read from a magnetic disk or tape. The rate at which data can be transferred from a disk or tape is much higher than the rate at which cards are read, but it still takes a considerable amount of time in terms of the amount of processing that can be done by the CPU.

The process of reading a physical record from the data set may be overlapped with other processing.

To achieve this, PL/I files normally use two intermediate work areas called buffers. A buffer is an area in main storage the same size as a physical record of the data set being processed. It is automatically provided and is filled by the channel. A READ statement extracts a logical record from the buffers. As the physical records are put into the buffer by the channel without using the CPU, using two buffers allows the channel to put the next physical record into one buffer while the user's program is reading the logical records from the other buffer by READ statements, and processing them.

The use of two buffers speeds the input process by allowing the reading of physical records from the data set to be overlapped with processing of data in main storage. If the physical records hold more than one logical record each, buffers are necessary as somewhere to store the physical records in main storage while logical records are extracted from them.

When a READ statement is executed, a logical record is copied from the buffer into a variable. The data read may be in any form, and the variable may be of any type. The data is not checked as it is copied. The programmer must ensure that the attributes of the variable used are the same as the attributes of the data. If a record had been written to a disk data set from a variable with attributes CHARACTER(4), and was read back into a variable with attributes FIXED DECIMAL(7), the two variables are of the same length so no program failure would occur, but, when the receiving FIXED DECIMAL(7) variable is used in a later statement, the result would be unpredictable.

The Output Operation

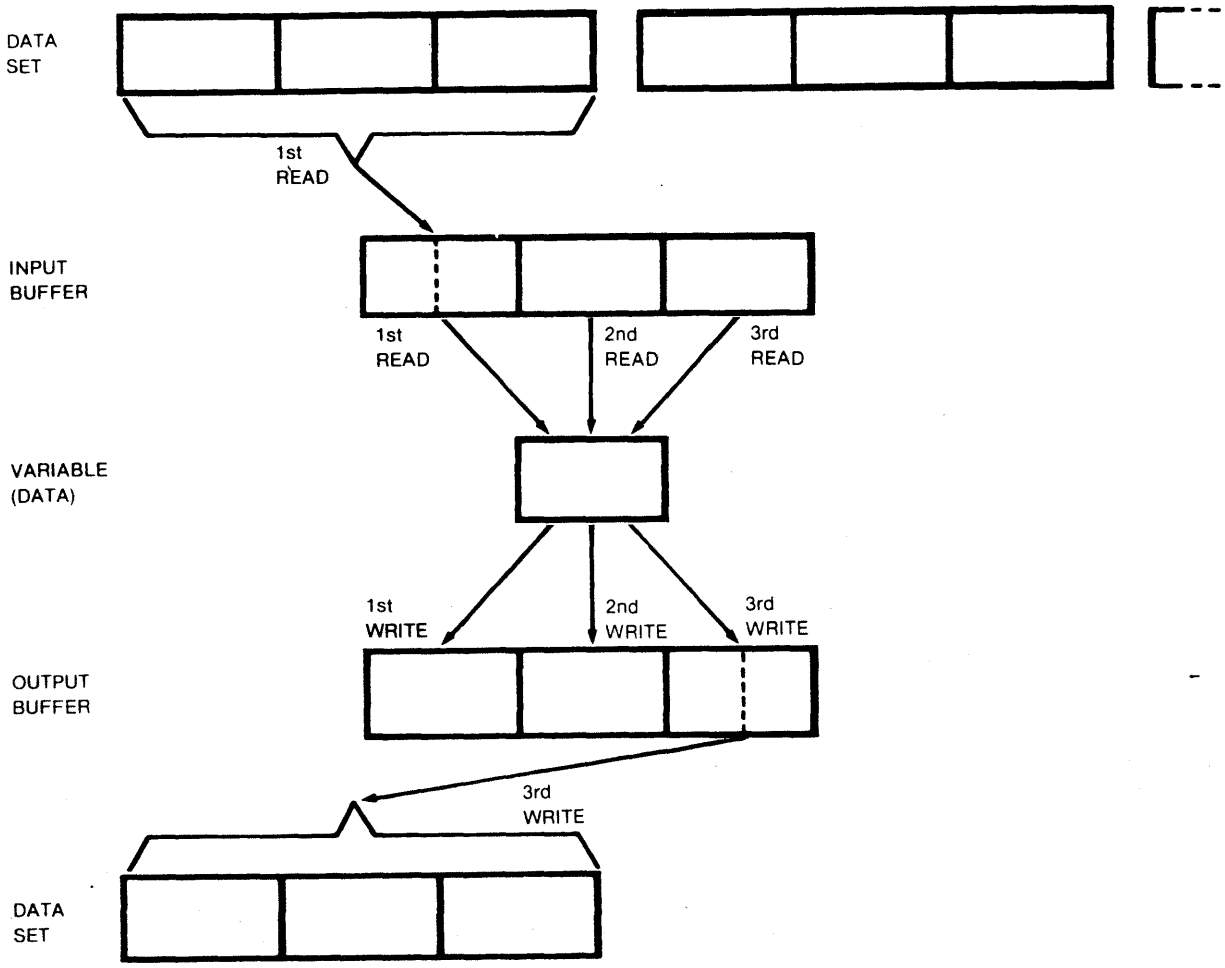
The output operation is governed by similar considerations to the input operation. The physical movement of data between main storage and the external device is governed by the channel, which may overlap its work with the work of the CPU. To facilitate the overlapping, and to allow logical records to be blocked, two buffers are normally used. While logical records are being copied to one buffer by WRITE statements, the contents of the other buffer will be copied to the data set. The data stored on the data set will be in the same format as the data in the variables written out.

'Move mode' in the topic title refers to the movement of data between data variables and buffers.

As an example of move mode processing, let us consider the program from Topic 2 which copied records containing names and addresses from one file to another. To simplify the situation, we will assume only one buffer is used for each file, and that there are three logical records in each physical record. The program, and a diagram of the move mode process, is shown on the following pages.

A Program to Copy Records from one File to Another

```
NCARD: PROCEDURE OPTIONS(MAIN);
       DCL NAME_AND_ADDRESS CHARACTER(80);
       READ FILE(CARDSIN) INTO(NAME_AND_ADDRESS);
       DO WHILE(NAME_AND_ADDRESS ¬= ' ');
          WRITE FILE(PRINTOUT) FROM(NAME_AND_ADDRESS);
          READ FILE(CARDSIN) INTO(NAME_AND_ADDRESS);
       END;
       END;
```



The first READ statement will cause the first physical record to be read from the disk into the buffer, and the first logical record to be copied from the buffer to NAME__AND__ADDRESS. The first WRITE statement will cause the contents of NAME__AND__ADDRESS to be copied as the first logical record in the output buffer. No record will be written to the output data set at this stage.

The second READ statement will not cause any records to be read from the input data set, but the second logical record will be copied from the input buffer to NAME__AND__ADDRESS, over-writing what had been put there by the first READ statement. Similarly, the second WRITE statement will not cause any output, and the third READ and WRITE statements will not cause any input or output. It is only when the fourth READ statement is executed, and there are no more records in the buffer, that the next physical record is read from the data set into the buffer, so allowing the fourth logical record to be read to NAME__AND__ADDRESS. It is only when the fourth WRITE statement is executed, and there is no room in the output buffer for more logical records, that the first record is written to the output data set, so allowing the fourth logical record to be written into the buffer.

This pattern will continue through the program. Only one READ statement in three will require a physical record to be read, and only one WRITE statement in three will cause a physical record to be written. The reading and writing of physical records will only be done when a READ or WRITE statement is executed which cannot be satisfied, so there will be no overlap of transferring physical records and other processing. However, there will still be a gain in performance as reading and writing a record of 240 bytes takes less than three times as long as reading or writing a record of 80 bytes.

If two buffers were used, as the fourth READ statement was executed, causing the first logical record to be read from the second buffer, the re-filling of the first buffer would be initiated, and would possibly be completed before all three logical records in the second buffer had been processed. Similarly, as the fourth WRITE statement was executed, causing the first logical record to be written to the second buffer, the writing of the first physical record to the output data set would be initiated, and would continue while the three logical records were written to the second buffer. Thus, there would be a double gain. The input and output operations would take less time, and the time which they took would be overlapped with other processing.

The blocking and de-blocking of records and the switching between buffers is transparent to the programmer. He simply codes READ or WRITE statements in his program whenever he wants to read or write the next logical record. Any operations required to enable those statements to be executed will be carried out automatically. The process of blocking and de-blocking and the use of buffers becomes more important to the programmer when doing locate mode processing, which will be covered in Topic 8.

You should now attempt questions 1 to 3 in the exercises at the end of this topic.

Updating Files

Records in sequential data sets which are held on a DASD may be updated in place. This is done in PL/I by the REWRITE statement:

REWRITE FILE(file name) FROM(variable name);

The REWRITE statement causes the logical record last read from a file to be over-written by the contents of the variable named. The last statement executed for a file before a REWRITE statement must have been a READ statement.

On the next page is an example of the use of the REWRITE statement. The RECEIPT file contains stock receipt records in the same order as the stock level records on the STOCK file. There is one receipt record for each stock level record. The end of the receipt records is indicated by a receipt record of -1. If any stock has been received for an item, the receipt record will hold the amount, which will be added to STOCK__LEVEL, and the new value in STOCK__LEVEL used to over-write the old value on the file.

The Use of the REWRITE Statement

```
READ FILE(RECEIPT) INTO(ADDITION)                          /* 10 */
DO WHILE(ADDITION ¬= -1);                                  /* 20 */
   READ FILE(STOCK)INTO(STOCK_LEVEL);                      /* 30 */
   DO WHILE(ADDITION ¬= 0);                                /* 40 */
      STOCK_LEVEL = STOCK_LEVEL + ADDITION;                /* 50 */
      REWRITE FILE(STOCK)FROM(STOCK_LEVEL);                /* 60 */
      ADDITION = 0;                                        /* 70 */
   END;                                                    /* 80 */
   READ FILE(RECEIPT)INTO(ADDITION);                       /* 90 */
END;                                                       /* 100 */
```

The following should be noted about this program:

STOCK__LEVEL is used as the variable into which the data is originally read, and the variable from which the new value is written. The variable used in the REWRITE statement may be any variable with suitable attributes.

File Attributes

File names must have a series of attributes which specify the attributes of the data set with which it is to be associated, and how this data set is to be processed. There are many of these attributes, which may be given by the DECLARE statement or other means. The coding may be reduced by the use of defaults and implications as shown in the following list.

Attributes	Abbreviation	Implies
FILE		
STREAM		FILE
RECORD		FILE
INPUT		FILE
OUTPUT		FILE
UPDATE		FILE RECORD
SEQUENTIAL	SEQL	FILE RECORD
DIRECT		FILE RECORD
BUFFERED	BUF	FILE RECORD SEQUENTIAL
UNBUFFERED	UNBUF	FILE RECORD SEQUENTIAL
ENVIRONMENT	ENV	FILE

The meanings of the attributes are:

FILE

Specifies that the identifier is a file name.

STREAM

Specifies that the file will be processed by stream input/output statements. These will be discussed in Topic 15.

RECORD

Specifies that the file will be processed by record input/output statements.

INPUT

Specifies that the file will be used for input only.

OUTPUT

Specifies that the file will be used for output only.

UPDATE

Specifies that the file may be used for input and output. If the file is to be processed sequentially (see below), it may be processed by the READ and REWRITE statements, not the WRITE statement.

SEQUENTIAL

Specifies that the file is to be processed in a sequential manner. It does not specify the organization of the data set. Thus, an indexed sequential data set may be processed either sequentially or directly. (Abbreviation SEQL).

DIRECT

Specifies that the file is to be processed directly, accessing specified records on the data set. Like SEQUENTIAL, it does not specify the organization of the data set being accessed. Both indexed sequential and direct data sets may be accessed directly.

BUFFERED

Specifies that the file is to be processed using buffers. All the discussion in this topic has been on the basis of buffered processing. It is not usual to process sequentially without using buffers. (Abbreviation BUF).

UNBUFFERED

Specifies that buffers are not to be used if not needed. In many situations, for instance the processing of blocked records, buffers will still be used, even though the UNBUFFERED attribute is specified. (Abbreviation UNBUF).

ENVIRONMENT

Allows the attributes of the data set which is to be associated with the file to be described by a series of options. The options of the ENVIRONMENT attribute are specified in a list, separated by spaces and enclosed in parentheses, following ENVIRONMENT. They may be in any order. The options vary slightly between OS/VS and DOS/VS, and will be discussed separately for the two operating systems. (Abbreviation ENV).

ENVIRONMENT Options For DOS/VS

OS/VS users turn to the section headed "*ENVIRONMENT OPTIONS FOR OS/VS*".

The ENVIRONMENT options under DOS/VS must fully describe the data set to be associated with the file. The options appropriate to sequential data sets with fixed length records are:

MEDIUM
F
FB
CONSECUTIVE
RECSIZE(n)
BLKSIZE(n)
BUFFERS(n)

where n is an integer constant.

MEDIUM

MEDIUM(symbolic device name, physical device type)

The MEDIUM option describes the symbolic device name for the data set and the physical device type to be used.

Symbolic Device Name

The symbolic device name is used to link, through Job Control Language (described in a later segment) a file to a physical device - a particular disk drive, magnetic tape unit, card reader etc. It takes the form SYSxxx, where xxx may be:

IPT	The system input device, normally the card reader. If SYSIPT is used as the symbolic device name, no Job Control ASSGN statement will be needed for the card reader.
LST	The system output device used for listing. If SYSLST is used, no Job Control ASSGN statement will be needed to associate the file with the line printer, but the records of the file must contain information to control the vertical spacing. How this is done will be discussed in Topic 9.
PCH	The system output device used for card punching. This will be associated automatically with the card punch.
000 to 221	Symbolic device names SYS000 to SYS221 may be used for files which are to be associated with data sets on any type of device, including the standard system input and output devices.

The choice of the symbolic device name is open to the programmer, but, in a similar way to the file name, the choice made must be reflected in Job Control.

Physical Device Type

The physical device type must be the type number of the device on which the data set is stored or is to be stored.

The following table shows devices which may be attached to a System/370 and the physical device type specifications to be used for each of them. Probably not all of these devices will be available on the computer which you will use.

Device Type	Number	Device-Type Specification
Card Readers and Punches	IBM 2540 IBM 2560 IBM 1442N1 IBM 1442N2 IBM 2520B1 IBM 2520B2 IBM 2520B3 IBM 2501 IBM 3504 IBM 3505 IBM 3525 (multi-line print) (2-line print) IBM 3881 IBM 5425	2540 2560 1442 1442 2520 2520 2520 2501 3504 3505 3525 3525T 3881 5425
Printers	IBM 1403 IBM 1404 IBM 1443 IBM 1445 IBM 3211 IBM 5203 IBM 3203	1403 1404 1443 1445 3211 5203 3203
Magnetic Tape Drives	IBM 2400 (9-track) IBM 2400 (7-track) IBM 3410/3411 IBM 3420	2400 2400 3410 3420
DASD	IBM 2311 IBM 2314 IBM 2321 IBM 3330 IBM 3340	2311 2314 2321 3330 3340
Diskette Unit	IBM 3540	3540

A file which is to be associated with a data set on a 7-track 2400 tape drive might be declared with MEDIUM(SYS008, 2400). The symbolic device name could be any number not used elsewhere in the program and which is reflected in the number used in Job Control. A file which does not use SYSLST may still be associated with a line printer, for example, MEDIUM(SYS121,3211).

Files which use the standard system input and output devices - SYSIPT, SYSLST and SYSPCH need not have a physical device type specified. They will use whatever the standard system input and output devices are. For example:

MEDIUM(SYSIPT)

MEDIUM(SYSLST)

Whether the physical device type is needed or not, the MEDIUM option must be given in file declarations.

F

Specifies that the file will contain fixed length, unblocked records.

FB

Specifies that the file will contain fixed length, blocked records. One of F and FB must be specified.

CONSECUTIVE

Specifies that the data set will have CONSECUTIVE organization. It is the default for data set organization, but does not specify how the data set is to be processed. This is done through the SEQUENTIAL attribute.

RECSIZE(n)

n is the length of logical records on the data set, in bytes. It must be the same as the length of the variables used in input/output statements for this file.

BLKSIZE(n)

n is the length of physical records on the data set, in bytes. If the record format is F, n must be the same as RECSIZE. If the record format is FB, n must be a multiple of RECSIZE.

If the record format is F, it is not necessary to specify both - the one which is not specified will default to the same value as the one which is.

BUFFERS(n)

n is the number of buffers to be used in processing the file. In DOS/VS n can be 1 or 2. The default, in DOS/VS, is 2 for CONSECUTIVE and sequentially accessed INDEXED file, 1 otherwise.

The ENVIRONMENT attribute must be supplied to describe the attributes of the data set with which the file is to be associated.

You should now attempt questions 4 and 5 in the exercises at the end of the topic, before continuing the topic at 'Alternative Attributes'.

ENVIRONMENT Options For OS/VS

The ENVIRONMENT options under OS/VS may be used to describe the attributes of the data set with which the file will be associated through Job Control Language statements, which is covered in another course. The options which are appropriate to sequential data sets with data sets with fixed length records are:

F
FB
CONSECUTIVE
RECSIZE(n)
BLKSIZE(n)
BUFFERS(n)

F

Specifies that the file will contain fixed length, unblocked records.

FB

Specifies that the data set contains fixed length, blocked records.

CONSECUTIVE

Specifies that the data set will have CONSECUTIVE organization. It is the default for data set organization, but does not specify how the data set is to be processed. This is done through the SEQUENTIAL attribute.

RECSIZE(n)

n is the length of logical records on the data set, in bytes. It must be the same as the length of the variables used in input/output statements for this file.

BLKSIZE(n)

n is the length of physical records on the data set, in bytes. If the record format is F, n must be the same as RECSIZE. If the record format is FB, n must be a multiple of RECSIZE.

If the record format is F, it is not necessary to specify both - the one which is not specified will default to the same value as the one which is.

BUFFERS(n)

n is the number of buffers to be used in processing the file. If the BUFFERS option is not specified, a default value of 2 will be taken. If it is specified it may be any value from 1 to 255. The more buffers that are allocated, the faster the program will be executed, in general, but the more storage it will use.

The ENVIRONMENT attribute may be used to describe the attributes of the data set with which the file will be associated. All of the options shown here, with the exception of the CONSECUTIVE option, which is a default, may be replaced by sub-parameters of the DCB parameter of Job Control, or, for an existing data set, by information from the data set label. For sequential data sets, the ENVIRONMENT attribute may be omitted from the file declaration, so allowing the user to specify such information as block size when the program is executed, rather than when the program is compiled. This allows one program to process data sets with different record formats and blocking factors using the same file, with no re-compilation.

If information is supplied in the ENVIRONMENT attribute, it will over-ride similar information supplied through Job Control.

You should now attempt questions 6 and 7 in the exercises at the end of this topic.

Alternative Attributes

Not all attributes may be specified for the same file. For example, a file may not be declared with both INPUT and UPDATE attributes. A group of attributes of which only one may be specified is called a group of alternative attributes. For each group of alternative attributes there is a default, as shown below.

Group Type	Alternative Attributes	Default Attribute
Usage	STREAM RECORD	STREAM
Function	INPUT OUTPUT UPDATE	INPUT
Buffering	BUFFERED UNBUFFERED	BUFFERED
Access	SEQUENTIAL DIRECT	SEQUENTIAL

These defaults are not always taken. The list below shows that all file attributes, except FILE, imply some other attributes. Where an implied attribute is not the default attribute for a group, it will over-ride the default. In this situation, no attribute for that group, except the implied attribute, may be specified. If such a clash does occur, the error will be detected when the file is opened.

Attributes	Abbreviation	Implies
FILE		
STREAM		FILE
RECORD		FILE
INPUT		FILE
OUTPUT		FILE
UPDATE		FILE RECORD
SEQUENTIAL	SEQL	FILE RECORD
DIRECT		FILE RECORD
BUFFERED	BUF	FILE RECORD SEQUENTIAL
UNBUFFERED	UNBUF	FILE RECORD SEQUENTIAL
ENVIRONMENT	ENV	FILE

File Opening and Closing

Before input or output operations may be carried out on a file, it must be opened. When a file is opened, it is associated with a data set, its attributes are checked, and buffers are allocated to it. If the file is a sequential input or update file, the buffers will be filled with the first records from the data set.

A file may be opened in one of two ways; by executing an OPEN statement for that file, or by executing an input or output statement for it. An OPEN statement causes an explicit open. Its format is:

OPEN FILE(file name)[attributes];

If attributes are supplied in OS/VS they may be any file attributes except the ENVIRONMENT attribute. For example:

```
OPEN FILE(IN) INPUT,
OPEN FILE(OUT) RECORD OUTPUT;
```

However in DOS/VS only the INPUT or OUTPUT attributes can be supplied and even then, there are restrictions (see Topic 9).

An input or output statement, without a previous explicit open, causes an implicit open.

When a file is opened, the attributes are said to be merged. At this point there must be no clash in the attributes. If the file is opened explicitly, the attributes on the OPEN statement, if any, are treated in the same way as the attributes on the DECLARE statement. They will all be processed by the following rules:

There may be only one attribute from any group of alternative attributes.

If any attributes are implied, they must not contradict any attributes which are explicitly stated.

If any group of alternative attributes has no member either explicitly stated or implied, then the default for that group will be taken.

If a file is declared:

```
DCL WORK RECORD ENV(MEDIUM(SYS100,3330) F RECSIZE(80));
OPEN FILE(WORK) OUTPUT;
```

(the MEDIUM option would be omitted for an OS/VS system) the file will gain the following attributes when it is opened:

a) All attributes given will imply FILE.

b) The usage is explicitly specified as RECORD.

c) The function is explicitly specified as OUTPUT.

d) Defaults of BUFFERED and SEQUENTIAL will apply for the buffering and access.

e) The record format and record size are specified.

f) The block size will default to the record size, for DOS/VS, but Job Control and, if available, the data set label will be checked for OS/VS. If it is specified in either of these, it must match the record length. If it is not, a default of the record length will apply. Defaults of CONSECUTIVE and BUFFERS (2) will also be taken.

If a file is opened implicitly by an input or output statement, it will acquire further attributes which will be deduced from the statement type - similarly to contextual declaration.

Statement	Attributes Deduced
READ	RECORD INPUT
WRITE	RECORD OUTPUT

(If the file is declared with the UPDATE attribute, INPUT or OUTPUT will not be deduced). These attributes will be merged with the attributes on the DECLARE statement in exactly the same way as if the file had been explicitly opened with them.

When a CONSECUTIVE file is being processed for input or output under OS/VS, it is possible to avoid all declaration, if the defaults are suitable. If the file is to be used for input only, the first READ statement will cause the file to be opened. The effect will be as if the file had been declared:

```
DCL NAME FILE RECORD INPUT BUFFERED SEQL ENV(CONSECUTIVE BUFFERS(2));
```

In each case the data set attributes would have to be supplied through Job Control or from the data set label.

For DOS/VS, the minimum requirement is that the files should be declared with the ENVIRONMENT options of record format, record length, block size and MEDIUM.

Before a program is terminated, all files must be closed. When a file is closed, the reverse of opening takes place. The file is disassociated from the data set, the buffers are released and, for an output file, the last record is written out. A file may be closed explicitly by a CLOSE statement, or implicitly by executing the last statement of the program, whatever that may be. The format of the CLOSE statement is:

CLOSE FILE(file name);

For example:

```
CLOSE FILE(IN);
CLOSE FILE(WORK);
```

The major use of explicit closing is in connection with explicit opening. If a file is to be used as a work file within a program, that is, data is to be written to it at one point in the program and read back at a later point, it cannot be opened with any set of attributes which will allow this to be done. It may be done as follows:

```
DCL WRKFIL FILE RECORD ENV(MEDIUM(SYS020,3330) FB RECSIZE(120)
                                                 BLKSIZE(480));
OPEN FILE(WRKFIL) OUTPUT;
   /* WRITE DATA TO FILE */
CLOSE FILE(WRKFIL);
OPEN FILE(WRKFIL);
   /* READ THE DATA BACK */
```

The medium option should be omitted for OS/VS.

When the file is first opened, it is opened for output, and records may be written to it. When it is closed, any attributes given on the OPEN statement will be lost, so that when the file is again opened INPUT may be specified on the OPEN statement, or may be left to default, as here. When the file is re-opened, it will be positioned automatically at the beginning of the file, so that data will be read back in the order in which it is written out.

Record input/output is a very heavily used feature of PL/I. It allows efficient input and output of data in the form of records in any data format. This topic has covered one mode of record input/output for sequential data sets. Topic 8 will cover the use of locate mode for sequential data sets.

You should now complete the exercises at the end of this topic.

Exercises

1) What errors are there in the following statements?

```
/* A */ READ FILE(IN) INTO(VAR);                    valid
/* B */ WRITE FILE(OUT) INTO(VAR);                  NOT INTO
/* C */ WRITE FILE(OUT) FROM('CARD LISTING');       VAR name?
/* D */ READFILE(IN) INTO(VAR);                     SPACE R/F
/* E */ READ FILE(IN) INTO(VAR)                     NO SC.
```

Look At Example

2) The file IN1 contains records which were written from a variable with attributes FIXED DEC(5,2).

   The file IN2 contains records which were written from a variable with attributes FIXED DEC(7,1).

   Write a program to produce a third file, OUT. Each record on OUT should contain the product of the corresponding records on IN1 and IN2. The end of input data will be indicated by a record on IN1 containing 0.

   Records on OUT should be written from a variable with attributes FIXED DEC(7,2).

   Declare all necessary variables, and call the program PRODUCT.

   Hint - the processing logic could be:

   Read a record from IN1.

   If that is not 0, read a record from IN2, multiply them together, write a record to OUT and read another record from IN1. If that is not 0, ...

3) What errors could occur in the program PRODUCT?

4) *This question should be attempted by DOS/VS users only.*

   What errors are there in the following declaration?

```
FIL FILE OUTPUT BUFF REC SEQL ENV(MEDIUM(3330,SYS0008)
    F RECSIZE(256) BLKSIZE(512) CONSECUTIVE BUFFER(1));
```

6) *This question and the next should be attempted by OS/VS users only.*

   What errors are there in the following declaration?

```
FILE(FIL) UPDATE BUFFERS(4) REC CONSECUTIVE
ENV(F RECSIZE(256) SEQUENTIAL BUF BLKSIZE(256);
```

7) What advantage is to be gained by putting information about a data set in Job Control rather than the ENVIRONMENT attribute?

8) What is the minimum declaration that could be given for the following files under your operating system?

Assume that the files will be opened by a READ or WRITE statement. OS/VS programmers should ignore the MEDIUM option.

```
A */ DCL FIL1 FILE RECORD INPUT BUFFERED SEQUENTIAL
     ENV(MEDIUM(SYSØ2Ø,1442) F RECSIZE(8Ø) BLKSIZE(8Ø)
         CONSECUTIVE BUFFERS(1));

B */ DCL FIL2 FILE RECORD OUTPUT UNBUFFERED SEQUENTIAL
     ENV(MEDIUM(SYSØ3Ø,1443) F RECSIZE(132) BLKSIZE(132)
         CONSECUTIVE);
```

Answers

1) a) Valid.

   b) Invalid. The WRITE statement may not have the INTO option, it must have the FROM option.

   c) Invalid. The FROM option must have a variable name in parentheses.

   d) Invalid. There must be at least one space between READ and FILE.

   e) Invalid. The statement contains legal minimum spacing, but the semi-colon is missing.

2) One solution is:

```
PRODUCT: PROC OPTIONS(MAIN);
         DCL VAR1 FIXED DEC(5,2);
         DCL VAR2 FIXED DEC(7,1);
         DCL VAR3 FIXED DEC(7,2);
         READ FILE(IN1) INTO(VAR1);
         DO WHILE(VAR1 ¬= 0);
            READ FILE(IN2) INTO(VAR2);
            VAR3 = VAR1 * VAR2;
            WRITE FILE(OUT) FROM(VAR3);
            READ FILE(IN1) INTO(VAR1);
         END;
         END;
```

3) There might be a mistake in the records on IN1. If no record on IN1 contained 0, records would be read until there were no more. Similarly, if there were fewer records on IN2 than on IN1, the end of the file IN2 would be encountered while records were being read. PL/I has facilities for detecting and dealing with such situations. They will be covered in later topics.

   The variable VAR3 has attributes FIXED DEC(7,2). When multiplying two fixed decimal variables, the precision of the result will be (p,q), where:

$$q = q_1 + q_2$$
$$p = p_1 + p_2 + 1$$

   In this case, the operands have precisions (5,2) and (7,1). The precision of the result will be:

$$q = 2 + 1 = 3$$
$$p = 5 + 7 + 1 = 13$$

   A result of precision (13,3) will be assigned to a target of precision (7,2), with possible loss of high order digits. If this occurs the value assigned is undefined.

   These errors are examples of GIGO - Garbage In, Garbage Out. This is an old principle of computing. Garbage is defined as any data which the program cannot successfully

handle. If the programmer is not absolutely certain that the files IN1 and IN2 contain correct data, he should take steps in his program to check it.

4) Something of a tragedy.

   The abbreviation for BUFFERED is BUF.

   RECORD has no abbreviation.

   The symbolic device name and physical device type in the MEDIUM option should be in the other order.

   The block size is twice the record size when the record format has been specified as F. The keyword for the number of buffers is BUFFERS.

5) SEQUENTIAL describes how the file will be processed - in the order in which the records are stored. It is the opposite of DIRECT. CONSECUTIVE describes how the data set is organized. A CONSECUTIVE file may only be processed sequentially.

6) Another tragedy.

   FILE(FIL) is the file option as used in a READ or WRITE statement. This is a declaration and so should have an identifier followed by a series of attributes, of which FILE is one.

   The BUFFERS option should be part of the ENVIRONMENT attribute.

   There is no abbreviation for RECORD.

   CONSECUTIVE is an option of the ENVIRONMENT attribute.

   SEQUENTIAL and BUF should both be outside the ENVIRONMENT attribute.

   The right parenthesis at the end of the list of ENVIRONMENT options is missing.

7) Job Control may be varied from run to run of a program. If the block size of the data set is changed, the Job Control can be changed without changing the program and so having to re-compile it.

8) For DOS/VS:

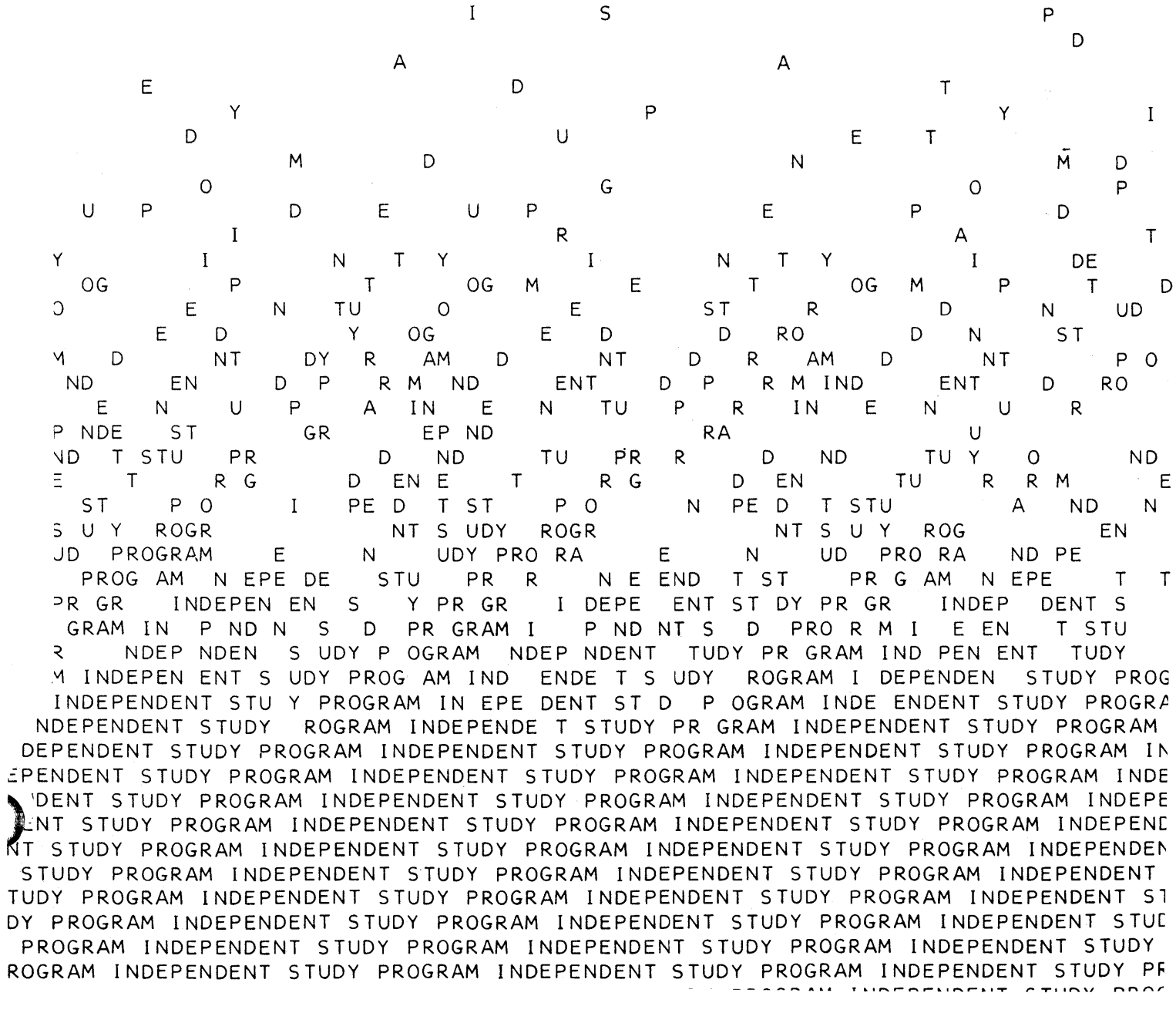
```
/* A */ DCL FIL1 ENV(MEDIUM(SYS020,1442) F RECSIZE(80) BUFFERS(1));
/* B */ DCL FIL2 UNBUF ENV(MEDIUM(SYS030,1443) F RECSIZE(132));
/* BLKSIZE COULD HAVE BEEN SUPPLIED INSTEAD OF RECSIZE */
```

For OS/VS:

```
/* A */ DCL FIL1 ENV(BUFFERS(1));
/* B */ DCL FIL2 UNBUF;
```

Topic

6



Topic 6

Data Structures and Picture Variables

The declarations of the various data elements have already been discussed. This topic will deal with the declarations of more complex data variables.

Objectives

At the end of the topic you should be able to:

- declare and use data structures
- declare and use picture variables.

Introduction

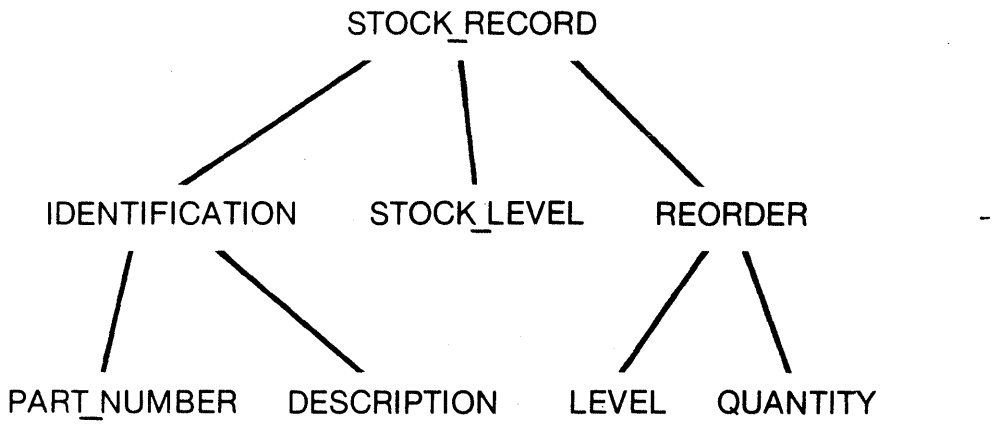
The READ and WRITE statements transfer information form/to a file; this information is held in records. It is normal for a single record to hold several items of information, some of which may be numeric. This topic covers efficient methods of isolating the items in a record, and of processing numeric information held in a character format.

Structures

Records on files commonly consist of several elements or fields. The stock records referred to in Topic 5 might each contain a part number or other item identification, an item description, the current stock level, the stock level below which they should be re-ordered and the re-order quantity. In record input/output statements, all of this information must be referred to as if it were a single variable. When the record is processed, the various parts of the record must be available. Using ordinary variables, these two requirements contradict each other. Structures overcome this contradiction.

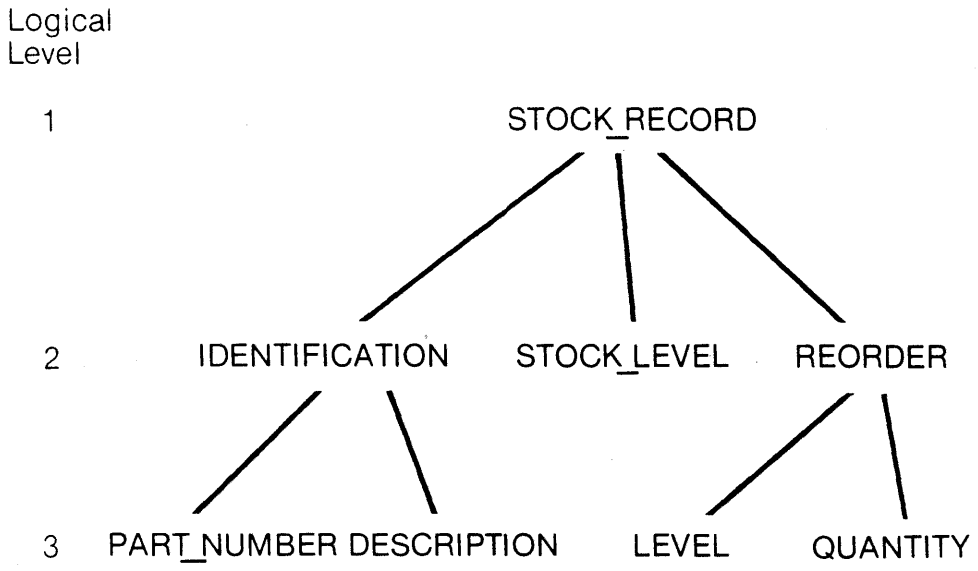
A structure is an aggregate of data items which have a logical relationship, but do not necessarily have similar attributes. It is a hierarchical organization, where the levels of the hierarchy are identified by numbers and names. The lowest level of the hierarchy contains element variables, with any attributes which element variables may have. The highest level is called the major structure, and intermediate levels are called minor structures. Major and minor structures may not have attributes of base, scale, precision or length.

The information in the stock record could be organized as follows:



The whole of the information concerning an item is known by the name STOCK__RECORD - the major structure name. The individual elements of information in the record are identified by the names PART__NUMBER, DESCRIPTION, STOCK__LEVEL, LEVEL and QUANTITY. These are the elements, and the only identifiers which will have attributes of base, scale, precision or length. Some of these elements can be grouped together into minor structures. PART__NUMBER and DESCRIPTION together make up IDENTIFICATION, a minor structure. LEVEL and QUANTITY together make up REORDER, another minor structure.

The structure could be drawn up with logical level numbers:



STOCK__LEVEL is on logical level 2, with the minor structures, although it is an element. This is because the only grouping above it is at level 1. The items on level 3 can be grouped together by minor structures at level 2, or by the major structure at level 1.

This hierarchy or structure may be declared by PL/I as:

```
DCL 1 STOCK_RECORD,
    2 IDENTIFICATION,
      3 PART_NUMBER FIXED DEC (7),
      3 DESCRIPTION CHAR (40),
    2 STOCK_LEVEL FIXED DEC (5),
    2 REORDER,
      3 LEVEL FIXED DEC (5),
      3 QUANTITY FIXED DEC (5);
```

The name of each major structure, minor structure or element must conform to the rules of names in PL/I. Each name must be preceded by a level number, which must be separated from it by at least one blank. Major and minor structures must not have any attributes of base, scale, precision or length. **The attributes follow element names in the normal way and for any level of name the list of attributes, if any, is terminated by a comma.** If the last element of the structure is the last name in the DECLARE statement, the comma is replaced by a semi-colon.

Accessing a Structure

Using the major and minor structure names, and the element names, it is possible to refer to the whole of the structure, to sub-sets of the structure and to individual elements. Thus, STOCK_RECORD will refer to all the five elements in the structure. IDENTIFICATION will refer to the two elements PART_NUMBER and DESCRIPTION. REORDER will refer to the elements LEVEL and QUANTITY. Individual elements may be referred to in the normal way by their element names. If there is no need to refer to sub-sets of a structure, then it may be declared with no minor structures:

```
DCL 1 STOCK_RECORD,
      2 PART_NUMBER FIXED DEC (7),
      2 DESCRIPTION CHAR (40),
      2 STOCK_LEVEL FIXED DEC (5),
      2 REORDER_LEVEL FIXED DEC (5),
      2 REORDER_QUANTITY FIXED DEC (5);
```

This structure contains the same information as the previous one. The minor structure names have been removed and the last two element names have been changed as a documentation aid, since the minor structure name previously indicated their use.

Level Numbers in Structures

In the two preceding examples the level numbers used corresponded to the logical levels within the structure. This is not obligatory, although they must reflect the relative levels of adjacent items in the structure:

```
DCL 1 STOCK_RECORD,
      3 IDENTIFICATION,
        7 PART_NUMBER FIXED DEC (7),
        7 DESCRIPTION CHAR (40),
      3 STOCK_LEVEL FIXED DEC (5),
      3 REORDER,
        7 LEVEL FIXED DEC (5),
        7 QUANTITY FIXED DEC (5);
```

This declaration has exactly the same meaning as the previous declaration.

Structures may have a maximum of 15 logical levels and the highest level number permitted is 255.

All the structures shown have had a similar layout - each name on a new line, each logical level indented two spaces from the next level above. Structures follow the normal rules of PL/I for spacing, with the level numbers being treated as identifiers, they must be separated from other identifiers by at least one blank but may be immediately adjacent to any separator. In this situation, the only relevant separator will be a comma. The layout used here conforms to these rules. It is usual to adopt a convention like this to aid readability and to facilitate alterations.

You should now attempt question 1 in the exercises at the end of this topic.

Using Structures

The effect of referring to a major or minor structure name will vary depending on the context in which it is used.

Input/Output

When a major or minor structure name is referred to in the INTO or FROM option of an I/O statement, it is treated as a single item. The length of the structure is the sum of the lengths of the elements. (There can be exceptions to this rule - see Topic 23, but not now).

Structure Element	Length (bytes)
PART_NUMBER	4
DESCRIPTION	40
STOCK_LEVEL	3
LEVEL	3
QUANTITY	3
Total	53

If STOCK__RECORD is used in input/output statements, the records on the file must have a length of 53. Different elements of STOCK__RECORD have different attributes. The corresponding fields of the records must have the same format as the elements of the structure. If they do not, unpredictable processing errors will occur. If a record from a data set is read into STOCK__RECORD, the first 4 bytes must have a fixed decimal format, the next 40 must have character format, and the last 9 bytes must have 3 fields of 3 bytes each in fixed decimal format. This can be most easily ensured by writing and reading the records using structures with identical declarations.

Assignment and Expressions

Major and minor structure names may be used in assignment statements in a similar manner to element variables. When so used, a structure is treated as a group of elements. The general form of a structure assignment is:

structure = element expression;
or
structure = structure expression;

where 'structure' may be a major structure or minor structure. The coding below shows examples of these forms. In this coding, all the variables are declared in one statement. In this situation, the end of a structure is indicated by a name with level 1, or a name with no level number. Thus, MAJS1 is terminated by the level 1 name STOCK__RECORD, and STOCK__RECORD is terminated by the name A, which has no level number.

```
DCL 1 MAJS1,                                                    /*  10 */
      3 MINS1,                                                  /*  20 */
        5 EL1 FIXED DEC (7),                                    /*  30 */
        5 EL2 CHAR (40),                                        /*  40 */
      3 EL3 FIXED DEC (5),                                      /*  50 */
      3 MINS2,                                                  /*  60 */
        5 EL4 FIXED DEC (7),                                    /*  70 */
        5 EL5 FLOAT DEC (6),                                    /*  80 */
    1 STOCK_RECORD,                                             /*  90 */
      2 IDENTIFICATION,                                         /* 100 */
        3 PART_NUMBER FIXED DEC (7),                            /* 110 */
        3 DESCRIPTION CHAR (40),                                /* 120 */
      2 STOCK_LEVEL FIXED DEC (5),                              /* 130 */
      2 REORDER,                                                /* 140 */
        3 LEVEL FIXED DEC (5),                                  /* 150 */
        3 QUANTITY FIXED DEC (5),                               /* 160 */
  A FLOAT DEC (6),                                              /* 170 */
  1 STK_RECORD,                                                 /* 180 */
    2 PART_NO FIXED DEC (7),                                    /* 190 */
    2 DESC CHAR (40),                                           /* 200 */
    2 STK_LEVEL FIXED DEC (5),                                  /* 210 */
    2 REORD_LEVEL FIXED DEC (5),                                /* 220 */
    2 REORD_QUANTITY FIXED DEC (5);                             /* 230 */

MAJS1 = A;                                                      /* 240 */
EL1 = A;                                                        /* 250 */
EL2 = A;                                                        /* 260 */
EL3 = A;                                                        /* 270 */
EL4 = A;                                                        /* 280 */
EL5 = A;                                                        /* 290 */

STOCK_RECORD = '';                                              /* 300 */
STOCK_RECORD = MAJS1;                                           /* 310 */
PART_NUMBER = EL1;                                              /* 320 */
DESCRIPTION = EL2;                                              /* 330 */
STOCK_LEVEL = EL3;                                              /* 340 */
LEVEL = EL4;                                                    /* 350 */
QUANTITY = EL5;                                                 /* 360 */
MINS2 = MINS2 + REORDER*2;                                      /* 370 */
EL4 = EL4 + LEVEL*2;                                            /* 380 */
EL5 = EL5 + QUANTITY*2;                                         /* 390 */
```

First look at the form:

structure = element expression;

The element expression will be evaluated according to the rules described in Topic 4. The result will then be assigned to each individual element of the structure, with conversion at each assignment, if necessary. Thus, statement 240 is exactly equivalent to statements 250-290. A will have default attribute of DECIMAL FLOAT(6), and will be converted to the attribute of each target.

Statement 300 shows a simple method of setting every element in a structure to the null value of that type of data. It is done by assigning the null string to the structure name. The null string consists of two single quote characters in adjacent positions. It has a length of zero and, on assignment, will be padded up to the length of the elements with the appropriate characters. For character strings it will be padded with blanks. For numeric variables it will be padded with zeroes. For all other data types which will meet in this segment and later, it represents the appropriate null value. Thus, statement 300 causes all the numeric elements of STOCK__RECORD to be set to zero, and the character element to be set to blanks.

It is not possible to do assignments of the form:

```
element variable = structure expression;
                   /* NOT ALLOWED */
```

The other form of the statement is:

structure = structure expression;

At its simplest, a structure expression is merely a structure name, as in statement 310. When one structure is assigned to another, both must have identical structuring. That is, not only must they have the same number of elements, but at each logical level they must have the same number of elements and minor structures, in the same order. One or both of the structures may be a minor structure, as long as the structuring within it is the same as in the major structure.

The names used for minor structures and elements do not matter, nor do the attributes of the elements, or the level numbers used. Assignment is done on an element by element basis, with conversion if necessary. Thus, Statement 310 is equivalent to statements 320 to 360, but an assignment from STOCK__RECORD to STK__RECORD, or the other way round, is not allowed, even though either could be used as the structure into which the stock records are read.

A more complicated structure expression may be an expression which involves at least one structure name. It may also involve element variables and constants. When an expression involves a major or minor structure and an element, it is expanded to operate on each element of the structure. Thus, in statement 370:

```
REORDER*2
```

will expand to:

```
LEVEL*2
```

and QUANTITY*2

When an expression involves two structures, equivalent elements of the two structures are combined. Thus, statement 370 has identical effect to statements 380 and 390.

All structures in a structure assignment statement must have identical structuring. As in simple assignment, they may be major structures, minor structures or a mixture of the two.

Structures used in input/output statements are treated as a single variable whose length is the sum of the lengths of the elements. Structures used in assignment statements are treated as a collection of element variables. The individual elements may be accessed as shown in the previous diagram. However, this is not always adequate, as the name of each element or minor structure within a structure does not have to be unique within a PL/I program.

```
DCL 1 PAYROLL,                                                    /*  10 */
      2 NAME CHAR (40),                                           /*  20 */
      2 HOURS,                                                    /*  30 */
        3 REGULAR,                                                /*  40 */
        3 OVERTIME,                                               /*  50 */
      2 RATE,                                                     /*  60 */
        3 REGULAR,                                                /*  70 */
        3 OVERTIME;                                               /*  80 */
DCL 1 PAY,                                                        /*  90 */
      2 NAME CHAR (40),                                           /* 100 */
      2 AMOUNT,                                                   /* 110 */
        3 REGULAR,                                                /* 120 */
        3 OVERTIME,                                               /* 130 */
      2 TOTAL;                                                    /* 140 */
AMOUNT = HOURS * RATE;                                            /* 150 */
TOTAL = PAY.AMOUNT.REGULAR + PAY.AMOUNT.OVERTIME;                 /* 160 */
TOTAL = AMOUNT.REGULAR + PAY.OVERTIME;                            /* 170 */
```

The declarations above are valid, but pose a problem in accessing the elements of the structures. Every element or minor structure name in a structure may be made unique by qualifying it by the names above it in the hierarchy. The fully qualified names of the various elements named REGULAR are PAYROLL.HOURS.REGULAR, PAYROLL.RATE.REGULAR and PAY.AMOUNT.REGULAR. The qualifying names must appear in order, with the highest level name first and the lowest level name last.

When any element or minor structure is accessed, it is only necessary to use sufficient levels of qualification to remove ambiguity. Thus, PAYROLL.HOURS.REGULAR may be referred to as HOURS.REGULAR, and PAY.AMOUNT.REGULAR may be referred to as PAY.REGULAR or AMOUNT.REGULAR. The statements starting at lines 160 and 170 have identical effect.

You should now attempt question 2 in the exercises at the end of this topic.

PICTURE Variables

Numeric Input/Output

Keyboard originated data - punched cards, data from terminals and data from diskettes, is in character format, but the content may be numeric. It may be read into a character string variable, but this has disadvantages. If character string variables are used in calculations, a slow conversion is done every time they are used, and the contents are then held in FIXED DECIMAL (15) format, causing the truncation of the fractional part. If they are assigned to numeric work fields, then truncation will not occur, but the conversion will be slow.

Output to the printer or to terminals is also in character format. Numeric values could be assigned to character variables for printing, but this would make the achievement of any sophisticated editing and layout of output very difficult. To overcome these difficulties, PL/I provides numeric PICTURE variables.

Numeric PICTURE Variables

Numeric PICTURE variables hold numeric data in zoned decimal format. The format of zoned decimal and character data is identical, but zoned decimal fields may only contain the characters of decimal numbers (0 to 9) and some special characters, which will be listed later.

A numeric PICTURE variable is declared with the PICTURE attribute, followed by a numeric picture specification. PICTURE may be abbreviated to PIC. The specification defines what sort of data may be held in the variable, and is enclosed in single quotes. There is no default for the specification of picture variables, it must always be given.

Although any PICTURE specification may be used for input or output, some are much more likely to be used as input specifications than others. Let's look first at specifications which are more suitable for input.

Input PICTURE Specifications

The simplest specification is for positive decimal integer values. For example:

```
DCL GRAMS PICTURE '999';
```

The picture character 9 specifies that this position may contain any of the characters 0 through 9, and so this specification defines a variable which may hold any number, in character form, from 000 to 999. It may contain neither a sign nor a decimal point, and it must be right aligned in the filed. Leading zeroes need not be included, but the whole field may not be blank.

The following are valid strings of characters to be read into GRAMS:

```
␣␣1
999
␣12
012
␣␣0
```

The following are not valid

```
1.2     contains a decimal point
␣0␣ }
␣␣␣ }   the rightmost character is blank
␣-6     contains a minus sign
```

The result of reading improper data into a picture variable is unpredictable and it may cause errors. It is the programmer's responsibility to ensure that the data put into a picture variable is valid for the specification of the variables.

Values which contain a fractional part may also be processed. It may be specified that, although the data contains no decimal point, certain digits are to be considered as fractional digits.

```
DCL AMOUNT PIC '999V99';
```

The V specifies that, although there is no decimal point in the field read, the last two digits of the field are to be assumed to be fractional digits. The V does not represent a position for a character, it represents the position of an assumed decimal point. The total width of the variable is 5 digits, of which two are fractional digits. The 9s have the same significance as above - AMOUNT may hold values in the range 0.00 to 999.99.

The following are valid strings of characters to be read into AMOUNT with the values which they represent.

Characters	Value
12345	123.45
ƀ1234	12.34
00012	0.12
ƀƀƀƀ1	0.01

The following are not valid strings of characters to be read into AMOUNT:

```
12.34 }
123.4 }        No decimal point is allowed.
-1234          No sign is allowed.
ƀ12ƀƀ          The last character is a blank.
```

If a decimal point is to be included in the input data, it must be in a fixed position, and its position must be indicated by the picture specification. However, if it is indicated, it is taken as being the indication of a period character, with no numeric significance. **The position at which the decimal point is to be assumed must also be specified.**

```
DCL KILOS PIC '99V.999';
```

As before, the V indicates the assumed position of the decimal point. It does not indicate a digit position. The period indicates the position at which a period should appear in the input. It does indicate a digit position, but does not indicate the position of a decimal point. The length of KILOS is 6 characters, and it may hold values in the range 0.000 to 99.999. The following are valid strings of characters to be read into KILOS, with the values which they represent.

Characters	Value
12.345	12.345
ƀ1.234	1.234
ƀ0.001	0.001
00.001	0.001

The following are not valid strings of characters to be read into KILOS:

```
123.45      The period is in the wrong position.
1.2345      The period is in the wrong position.
-1.234      Negative values are not allowed.
ƀ1.20ƀ      The last character is a blank.
```

In KILOS, the V and . are adjacent. They do not have to be. If they are not, the V will indicate the position of the assumed decimal point, and the . will indicate where a period character should appear in the data.

```
DCL CENTS PIC '999.99V';
```

CENTS has the V and . in non-adjacent positions. The 6 digits of input data should have a period in the fourth position, but will be considered as an integer. CENTS would be suitable for reading input which had been punched as dollars and cents, with three digits of dollars, a period, and two digits of cents, and treating it as cents.

The PICTURE specification of AMOUNT, '999', was said to be suitable for decimal integer values, as in the picture specification for CENTS. As this implies, if a V is not included in a PICTURE specification, then it is assumed that it should be after the last digit.

```
DCL CENTS PIC '999.99V';
DCL PENNIES PIC '999.99';
```

The above declarations give CENTS and PENNIES identical attributes.

The following are valid strings of characters to be read into CENTS, with the values which they represent.

Characters	Value
123.45	12345.
001.23	123.
ƀƀ1.23	123.
ƀƀƀ01	1.

The following are not valid strings of characters to be read into CENTS:

```
123456      No period.
12345.      The period is in the wrong place.
b-1.23      Negative values are not allowed.
bb1.0b      The last character is a blank.
$12.34      $ is not a permitted character.
```

You should now attempt question 3 in the exercises at the end of this topic.

If data containing a negative sign is to be read, then this information must be put into the PICTURE specification. The programmer may select whether the sign is to be the first digit in the field, or is to appear immediately before the first significant digit of the numeric value.

```
DCL BALANCE PIC '-999V.99';
```

The picture specification for BALANCE says that data read into BALANCE will be 7 digits wide. If the value is negative, the first position should contain a -, otherwise it should be blank. The rest of the specification has the same meaning as for KILOS. The last six positions should contain a decimal number with a period in the fourth position, which is also the position of the assumed decimal point. BALANCE may hold values in the range -999.99 to 999.99..

The following are valid strings of characters to be read into BALANCE, with the values which they represent.

Characters	Value
−123.45	−123.45
−b12.34	−12.34
bb12.34	12.34
−001.23	−1.23

The following are not valid strings of characters to be read into BALANCE:

```
1234.46     The first digit must be a blank or -.
bb-1.23     The - is only allowed in the first position.
-1.0bbb     The period is in the wrong position, and the
            last character is a blank.
```

If a - sign is to be included immediately before the first significant digit, then it is said to be a floating sign. It is indicated, as follows, in a PICTURE specification:

```
DCL CHANGE PIC '--9V.99';
```

The three - characters indicate that the sign may appear in any of the first three character positions, immediately before the first digit. The declaration for CHANGE is otherwise the same as the declaration of BALANCE.

The following are valid strings of characters to be read into CHANGE, with the values which they represent.

Characters	Value
-123.45	-123.45
␣␣-1.23	-1.23
␣␣10.36	10.36

The following are not valid strings of characters to be read into CHANGE:

```
-1␣␣␣␣␣        There is no period, and the last
                character is a blank.
␣␣+1.23         A + is not allowed.
```

Using these PICTURE specifications, most forms of input data may be read. Variables with these specifications may be included in structures to describe parts of input records. The variables could then be used directly in calculations. The contents will be converted first to fixed decimal format. The precision will be the same as the number of positions which can hold numeric digits.

The conversion is much quicker than the conversion of character data to fixed decimal, but will still take time. Because of this, if the value is to be used more than once in calculations, it is more efficient to assign it to a fixed decimal variable one, and then to use that variable in calculations.

Before continuing with the PICTURE specifications used in output, please do questions 4 and 5 at the end of this topic.

Output PICTURE Specifications

The main advantage of PICTURE variables in the output situation is that they provide a simple method of performing elegant editing and formatting of numeric output for reports. The normal mode of use is that a PICTURE variable is an element of a structure which describes a line of print. The result which is to go at that position is calculated and is then assigned to the PICTURE variable. The structure is then printed.

Suitable choice of PICTURE specifications might cause a monetary amount which had been calculated as -123456 cents to be printed as $**1,234.56 CR, simply by assigning the value to a picture variable with a suitable specification and then writing it out.

Simple Specifications

The simplest specifications are made up of characters 9 and V. Each 9 in a specification causes one of the characters 0 to 9 to be printed. The V causes no output. When a value is assigned to a picture variable, the decimal point in the value is aligned with the V in the specification, and the integer and fraction parts are moved into the PICTURE variable. If there are more digits in the fraction positions of the picture specification than there are in the source value, then it is padded on the right with zeroes. If there are less, then the fraction will be truncated on the right. If there are more digits in the integer part of the PICTURE specification than there are in the source value, then the source value will be padded on the left with zeroes. If there are less, the result of the assignment is undefined. The error will not necessarily be detected by the system. It is the programmer's responsibility to ensure that the PICTURE specification used is adequate.

Below are two specifications and the effect of various assignments.

Value Assigned	Characters Printed
1	001
123	123
1.23	001
0.1	000

Value Assigned	Characters Printed
1	0100
1.23	0123
12.3456	1234
0.001	0000

Zero Suppression Characters

One of the undesirable features of the specifications of PIC1 and PIC2 above is that they cause all leading zeroes to be printed. This can be overcome in several ways.

Z

If the Z picture character is included in a picture specification, it must not be preceded by any 9 picture characters, or any other zero suppression characters. A Z in any position in a picture specification causes a leading zero in that position to be replaced by a blank.

A leading zero is a zero which precedes all non-zero characters in a number. If the digit is a non-leading zero, or a digit 1 to 9, then it appears as if the picture character had been a 9.

Below are some examples and the effect of assigning values to them. The first five are covered by the rules above. The last four illustrate Z picture characters following a V picture character.

Picture Specification	Value Assigned to Picture Variable	Characters Printed
ZZZZ9	103	␣␣103
ZZZ99	1	␣␣␣01
ZZZ99	1.02	␣␣␣01
ZZZV99	1.02	␣␣102
ZZ9V99	100.12	10012
ZZZVZZ	0.01	␣␣␣01
ZZZVZZ	1.00	␣␣100
ZZZVZZ	0.00	␣␣␣␣␣
ZZZVZZ	0.001	␣␣␣␣␣

If a picture specification contains a Z picture character to the right of a V picture character, all fractional digit positions and all integer digit positions must be shown by Z picture characters. If the value assigned to the picture variable causes all positions in the picture variable to contain leading zeroes, then the picture variable will contain all blanks. If it does not, then the picture variable will contain a result as if the Z picture characters after the V picture character had been 9 picture characters.

In some situations, it may be required to replace leading zeroes by some character other than blanks. A typical application might be a program which produces checks. If leading zeroes were replaced by blanks, fraudulent alterations would be relatively simple. The * picture character obeys the same rules and acts in the same manner as the Z picture character, but suppressed leading zeroes are replaced by * characters. A picture specification may not include both Z and * picture characters.

Below are some examples of picture specifications containing the * picture character, and the effect of assigning values to them.

Picture Specification	Value Assigned to Picture Variable	Characters Printed
****9	103	**103
***9	1	***1
***V**	0.01	***01
***V**	0.00	*****

Y

The Z and * picture characters both recognize leading zeroes as a special situation, but do not recognize embedded zeroes as special. The Y picture character causes a zero at that position to be replaced by a blank, whether it is a leading zero or embedded zero. Y picture characters must not appear to the left of a Z or * picture character in a specification, but they may appear to the right of them and in any position where a 9 picture character could otherwise occur.

Below are some examples of picture specifications which contain Y picture characters, and the effect of assigning values to them.

Picture Specification	Value Assigned to Picture Variable	Characters Printed
ZZYYY	103	ƀƀ1ƀ3
ZZYYY	10000	10ƀƀƀ
Z9Y9Y9	010175	ƀ1ƀ175
YYYVY9	0.04	ƀƀƀƀ4

You should now do question 6 in the exercises at the end of the topic.

Insertion Characters

The need to insert decimal points and other characters in printed fields is met by the insertion characters. Unlike the zero suppression characters, insertion characters do not indicate positions which could hold numeric digits. Insertion characters may appear at any point in the specifications discussed so far, and will normally cause the character specified to be printed at that position. If leading zeroes are suppressed, then insertion characters may also be suppressed.

period insertion character(.)

The period insertion character causes a period to appear at the associated position. There is no relationship between the appearance of the period character and the arithmetic value. If a period insertion character does not appear adjacent to a V picture character, the value printed will appear to be different from the value assigned into the picture variable.

Here are some examples:

	Picture Specification	Value Assigned to Picture Variable	Characters Printed
1)	ZZZ9V.99	12.34	ƀƀ12.34
2)	ZV99.99	0.1234	ƀ12.34
3)	ZZ.Z9V99	123.45	ƀ1.2345
4)	ZZ.Z9V99	1.23	ƀƀƀƀ123
5)	ZZ.Z9V99	12.34	ƀƀƀ1234
6)	ZZZZ.VZZ	0.01	ƀƀƀƀƀ01
7)	ZZZZV.ZZ	0.01	ƀƀƀƀ.01
8)	ZZZZV.ZZ	0.001	ƀƀƀƀƀƀƀ

The first example above shows a typical picture specification for printing out decimal results in a 'conventional' manner. The second example has the V picture character separated from the period insertion character by two positions. When a value is assigned to the picture variable, it will be aligned with the V picture character. The period insertion character will appear where specified. This specification would be suitable if a ratio had been calculated, and it was to be printed as if it were a percentage, or if an amount had been calculated in cents and it was to be printed as if it were dollars. It should be noted that, if the variable which had had 0.1234 assigned to it were used in an arithmetic expression, it would be taken to contain the value 0.1234.

Due to the suppression of leading zeroes by the Z or * picture characters, period insertion characters may be replaced by blanks or *'s, as appropriate. If all digits to the left of the period insertion character are suppressed as leading zeroes, then the period insertion character will be suppressed as well. If the zero suppression characters are Z's the period insertion character will be replaced by a blank. If the zero suppression characters are *'s, the period insertion character will be replaced by an *. This situation is illustrated by the third to sixth examples above.

However, if the period insertion character appears to the right of a V picture character, it will not be replaced by a blank or asterisk unless:

all digit positions to the right of the V are indicated by Z or *
AND
there are no significant digits in the field.

This situation is illustrated by examples 7 and 8 above.

For most applications, periods will be wanted in output unless the whole field is suppressed, so specifications containing the sequence V., as in example 7, will be required, rather than the sequence .V, as in example 6. Suppression of insertion characters is normally more useful when using the comma insertion character.

comma insertion character(,)

The comma insertion character follows the same rules as the period insertion character, except that, when not suppressed, it causes a comma character to appear in the equivalent position.

Below are some typical uses of the comma insertion character. Example 6 shows printing of large numbers using the European format.

	Picture Specification	Value Assigned to Picture Variable	Character Printed
1)	Z,ZZ9V.99	1234.56	1,234.56
2)	Z,ZZ9V.99	123.45	ƀƀ123.45
3)	Z,ZZ9V.99	0.00	ƀƀƀƀ0.00
4)	Z,ZZZ,ZZ9V.99	1.23	ƀƀƀƀƀƀƀƀ1.23
5)	Z,ZZZ,ZZ9V.99	1234567.89	1,234,567.89
6)	Z.ZZZ.ZZ9V,99	1234567.89	1.234.567,89

The slash insertion character follows the same rules as the period and comma insertion characters. It causes a slash character to be inserted in the equivalent position. A common use for the slash insertion character is for formatting the date, held as a six figure integer DDMMYY or MMDDYY. The following would be a suitable specification for the purpose.

Z9/Y9/Y9

The Y specification characters will cause zeroes in these positions to be suppressed. It will cause the following results:

Date	Printed
101280	10/12/80
010181	ƀ1/ƀ1/81

B

The B insertion character causes a blank character to be inserted at the equivalent position in the string of characters. If it occurs in a situation where the period, comma, or slash insertion characters would be replaced by an asterisk, the B insertion character will still cause a blank character to be inserted.

Below are some uses of the B insertion character.

Picture Specification	Value Assigned to Picture Variable	Characters Printed
99B999BV.99	1234.56	01ƀ234ƀ.56
BB**	123.0	**ƀ*1ƀ23

Currency Symbol

The currency symbol is shown in IBM manuals as $. The currency symbol may be used in a static or drifting manner.

Static Currency Symbol

A static currency symbol appears as either the first or the last character in a picture specification. It acts as an insertion character in the same way as a B; it may not be suppressed. A picture specification containing a static currency symbol will cause a currency symbol to be printed as the first or the last character, as appropriate, in the printed string.

Drifting Currency Symbol

The drifting currency symbol causes leading zeroes to be suppressed, and a currency symbol to be inserted before the first digit. It is coded in a similar, but not identical, way to a Z or * zero suppression character, as two or more currency symbols in the first digit positions of a picture specification. The string of currency symbols may contain insertion characters. The period, comma and slash insertion characters will be suppressed or replaced by a currency symbol if they are not preceded by a digit. If all digit positions in the specifications are currency symbols, and there are no significant digits, the whole field will be treated as blanks.

The difference between the use of the zero suppression characters and the drifting currency symbol is that the first currency symbol does not specify a possible digit position, but is reserved for use as a currency symbol if all other digit positions contain digits. A picture specification which contains a drifting currency symbol may not also include Z or * zero suppression characters. Following are some uses of the currency symbol.

Picture Specification	Value Assigned to Picture Variable	Characters Printed
Static Currency Symbol		
$Z,ZZ9V.99	12.36	␣␣␣12.36
999V.99$	12.36	012.36$
Drifting Currency Symbol		
$$,$$9V.99	12.36	␣␣␣$12.36
$$,$$9V.99	112.36	␣␣$112.36
$$,$$9V.99	0.01	␣␣␣␣$0.01
$$,$$$V.$$	0.01	␣␣␣␣␣$.01
$$,$$$V.$$	0.00	␣␣␣␣␣␣␣␣␣

You should now attempt question 7 in the exercises at the end of this topic.

Sign Indication

The sign of a number may be indicated in several ways. We will consider first the use of + and -.

Signs may be requested by using the following specification characters: -

a) S print + if positive, - if negative

b) - print a blank if positive, - if negative

c) + print + if positive, blank if negative

S, - and + may be used in an identical way to the currency symbol, as static or drifting characters. A drifting sign indication may not appear in the same specification as a drifting currency symbol, or a Z or * zero suppression character, but may appear with a static currency symbol.

It should be noted that a static sign symbol, or the first of a string of drifting sign symbols, may not be used as a digit position, even if it is not needed for a sign with any particular number. **Thus, a picture variable with the specification '999' or '---9' could not accommodate the number 1234. The maximum number it can accommodate is 999, and the first position will print as a blank.**

Following are some uses of the S, - and + sign characters.

Picture Specification	Value Assigned to Picture Variable	Characters Printed
SZZ9V.99	1.23	+ƀƀ1.23
SZZ9V.99	−0.23	−ƀƀ0.23
$−,−−9V.99	−1.23	$ƀƀƀ−1.23
$−,−−9V.99	1.23	$ƀƀƀƀ1.23
++9V.99	1.23	ƀ+1.23
++9V.99	−1.23	ƀƀ1.23

Other Sign Indications

It is a common business practice to indicate negative values by writing either CR (credit) or DB (debit) to the right of a number. Picture specifications provide this facility by the CR picture character pair and the DB picture character pair.

CR

The characters CR may be coded in a picture specification to the right of all digit positions. If the value assigned into the picture variable is negative, the characters CR will be printed in the corresponding position. If it is positive, two blanks will be printed.

DB

The characters DB may be coded in a picture specification to the right of all digit positions. If the value assigned to the picture variable is negative, the characters DB will be printed in the corresponding position. If it is positive, two blanks will be printed.

Below are some examples of the use of the CR and DB picture specification characters.

Picture Specification	Value Assigned to Picture Variable	Characters Printed
$BZ,ZZ9V.99BCR	−12.61	$ƀƀƀƀ12.61ƀCR
$BZ,ZZ9V.99BCR	12.61	$ƀƀƀƀ12.61ƀƀƀ
$$$,$$9V.99BDB	−12.61	ƀƀƀƀ$12.61ƀDB
$$$,$$9V.99BDB	12.61	ƀƀƀƀ$12.61ƀƀƀ

You should now answer question 8 in the exercises at the end of this topic.

Picture Variables in Assignment and Expressions

Picture variables form a bridge between the character and numeric data types. This is reflected in their use in assignments and expressions, where they may sometimes be treated as character variables, and sometimes as numeric variables.

Picture Variables as a Target

When a picture variable is the target of an assignment statement, it is always treated as a numeric variable. The source must be either

a) an arithmetic constant or variable

b) an arithmetic expression

c) a character string constant or variable, the contents of which could be used as a numeric constant

The source will be edited as it is assigned.

Note that the statement:

```
DCL PICVAR PIC '$99V.99';
PICVAR = '$12.34';
```

would cause an error.

A $ is not allowed in a character string constant which is to be converted to numeric.

Picture Variables Used in a Character Context

If a picture variable is assigned to a character variable, or forms an operand in a character expression, it is treated as a character variable. The contents are the characters which would be printed if the picture variable were printed.

```
DCL PICVAR PIC '$--9V.99';                    /* 10 */
DCL PRINLINE CHAR(132);                       /* 20 */
DCL CHARVAR CHAR(7);                          /* 30 */
PICVAR = -1.23;                               /* 40 */
CHARVAR = PICVAR;                             /* 50 */
PRINLINE = 'PROFIT = '||PICVAR;               /* 60 */
```

In the above, statement 40 would assign '$b -1.23' into PICVAR. Statement 50 would assign '$b -1.23' to CHARVAR, and statement 60 would assign 'PROFITb =b$b -1.23' to PRINLINE.

Note that, although statement 50 caused the contents of PICVAR to be quite legally assigned to CHARVAR, it would cause an error to assign CHARVAR to PICVAR after statement 50.

Picture Variables used in a Numeric Context

If a picture variable is assigned to a numeric variable, or appears in an arithmetic expression, it is treated as a numeric variable, with the value which was last assigned to it. The position of the decimal point will be indicated by the V picture character, or taken as after the last digit position if there is no V in the specification. Any insertion characters and currency symbols will be ignored, but any indication of sign will be heeded.

If the current value was put into the picture variable by a READ statement, then the numeric value will be that value which would have been assigned to the variable to cause the same characters to be held.

Picture variables used in arithmetic expressions are converted to FIXED DECIMAL with precision (p,q), where p is the total number of digit position in the specification and q is the number of digit positions after the V picture character. In this context a digit position is a

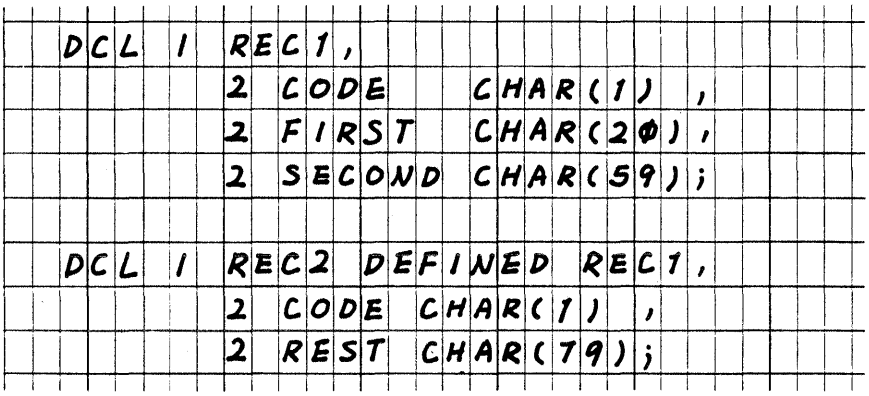
9,Z,*,Y or a drifting $, S, + or - which is not the first such character. Thus, the precision of a variable with a specification '$---9V.99' would be (5,2). Only the underscored characters count towards the precision.

The maximum number of digit positions allowed in a picture specification is 15.

You should now complete questions 9 and 10 in the exercises at the end of this topic.

DEFINED attribute

A variable may be declared, using the DEFINED attribute, so that it occupies the same area of storage as some other variable. A structure can also be declared, using the DEFINED attribute, so that it occupies the same area of storage as some other structure (or array or variable). This is useful when you want to interpret one area of storage (e.g. a record workarea) in different ways depending on the record type. For example:



```
DCL 1 REC1,
    2 CODE    CHAR(1) ,
    2 FIRST   CHAR(20) ,
    2 SECOND  CHAR(59);

DCL 1 REC2 DEFINED REC1,
    2 CODE  CHAR(1) ,
    2 REST  CHAR(79);
```

Here is a coding technique that makes use of the DEFINED attribute.

Suppose part of an output record has the specification:

```
2 OUTVALUE PIC  'Z9V.9',
```

If the data involved are valid the statement

```
OUTVALUE = VALUE;
```

will result in a suitable printed format for OUTVALUE. Let us suppose that we want four asterisks to replace OUTVALUE if we have tested and found some invalid data. We will get an error if we use the statement:

```
OUTVALUE =  '****';
```

as the character string does not conform to the picture for OUTVALUE, which requires numeric data. To get around this we can use:

```
DCL OUTSTARS CHAR(4) DEFINED OUTVALUE;
OUTSTARS =  '****';
```

The DCL statement causes the PL/I compiler to assign the same storage locations for OUTSTARS as for OUTVALUE. Since OUTSTARS has the character attribute we can readily assign the desired character string value to it.

The DEFINED attribute (abbreviation: DEF) will be described in greater detail in topic 21. The above information should be sufficient for you at this stage.

Exercises

1. Write a declaration for a structure to contain the following items. Call it CUST_STRUC. It is required to be able to refer to the last 3 items as LAST_TRAN.

```
CUST_NAME CHAR(30)
CUST_ADDR CHAR(100)
ACC_NO FIXED DEC(7)
CRED_LIM FIXED DEC(5)
CRED_LEVEL FIXED DEC(7,2)
DISCOUNT FIXED DEC(3,2)
DAY CHAR(2)
MONTH CHAR(2)
YEAR CHAR(2)
```

2.

```
DCL 1 DATE,
    2 DAY CHAR (2),
    2 MONTH CHAR (2),
    2 YEAR CHAR (2);
```

Given DATE declared as above, and CUST STRUC as declared in question 1., write statements to read a record from a file CUSTFIL into CUST_STRUC, replace the data in the last three elements of CUST_STRUC by the data currently held in DATE, and update the record on the file. Do not code any declarations.

3. Declare picture variables PIC1 to PIC4 capable of being used to read the following characters

```
ƀ12345
ƀ12345 to be interpreted as 12.345
ƀ12.34 to be interpreted as 12.34
ƀ12.34 to be interpreted as 123.4
```

4. Declare variables PIC1 to PIC3 capable of being used to read the following characters.

```
-ƀ12.34 to be interpreted as -12.34
ƀƀ-12.3 to be interpreted as -12.3
ƀ-12.34 to be interpreted as -1234
```

5. What are the largest positive and negative values which could be held in PIC1 to PIC3, declared in question 4?

6. Declare picture variables which could generate the following output.

Value Assigned	Characters Printed
1001	001001
1001	ƀƀ1001
1001	**1001
10.01	**1001
10.01	ƀƀ1ƀƀ1

7. Declare one picture variable which would cause the following outputs.

Value Assigned	Characters Printed
1	$bbbbb0.01
100	$bbbbb1.00
12345	$bbb123.45
1234567	$12,345.67

8. Declare a structure containing a character string variable and a picture variable such that, if -123456 were assigned to the picture variable, a line could be printed as follows:

   BALANCE IS $**1,234.56 CR

   Pad the structure to a length of 132 with a character variable containing blanks.

9. What values will be held if the following strings of characters are read into a variable declared as shown below:

```
DCL PICIN PIC '---9V99';
```

```
-12345
bbb123
bb-123
```

10. What will be printed if the values put into PICIN in question 9 are assigned to a variable PICOUT, and printed.

```
DCL PICOUT PIC '$SS,SS9V.99';
```

Answers

1.

```
DCL 1 CUST_STRUC,
      2 CUST_NAME CHAR (30),
      2 CUST_ADDR CHAR (100),
      2 ACC_NO FIXED DEC (7),
      2 CRED_LIM FIXED DEC (5),
      2 CRED_LEVEL FIXED DEC (7,2),
      2 DISCOUNT FIXED DEC (3,2),
      2 LAST_TRAN,
        3 DAY CHAR (2),
        3 MONTH CHAR (2),
        3 YEAR CHAR (2);
```

The level numbers 2 and 3 could be replaced by higher numbers.

2.

```
READ FILE (CUST_FIL) INTO (CUST_STRUC); -
LAST_TRAN = DATE;
REWRITE FILE (CUST_FIL) FROM (CUST_STRUC);
```

If a structure assignment were not used, the names used would have to be qualified.

```
CUST_STRUC.DAY = DATE.DAY;
CUST_STRUC.MONTH = DATE.MONTH;
CUST_STRUC.YEAR = DATE.YEAR;
```

3.

```
DCL PIC1 PIC '999999V';
         /* OR */
DCL PIC1 PIC '999999';
DCL PIC2 PIC '999V999';
DCL PIC3 PIC '999V.99';
DCL PIC4 PIC '999.9V9';
```

4.

```
DCL PIC1 PIC '-999V.99';
DCL PIC2 PIC '---99V.9';
```

There could be more - characters in PIC2's specification

```
DCL PIC3 PIC '--99.99V';
```

There could be more - characters in PIC3's specification, and the V could be omitted.

5. PIC1: 999.99 and -999.99

   PIC2: 9999.9 and -9999.9

   PIC3: 99999 and -99999

   The first - picture character does not indicate a possible decimal digit position.

6.

```
DCL PIC1 PIC '999999V';
DCL PIC2 PIC 'ZZ999V';
DCL PIC3 PIC '**999V';
DCL PIC4 PIC '**99V99';
DCL PIC5 PIC 'ZZ9YVY9';
```

In the declarations of PIC2 to PIC4, the zero suppression characters may be continued in all positions.

In the declaration of PIC5, all digit positions could be Y picture characters.

7.

```
DCL PICVAR PIC '$ZZ,ZZ9.99V';
```

8.

```
DCL 1 LINE,
      2 TXT CHAR (11) INIT ('BALANCE IS'),
      2 PIC PIC '$***,**9.99VBCR',
      2 PAD CHAR (107) INIT (' ');
```

Every character in the picture specification will occupy one position on the print line, except the V. The number of *'s in the picture specification is flexible. There must be a minimum of 2. All decimal digit positions could be shown by *.

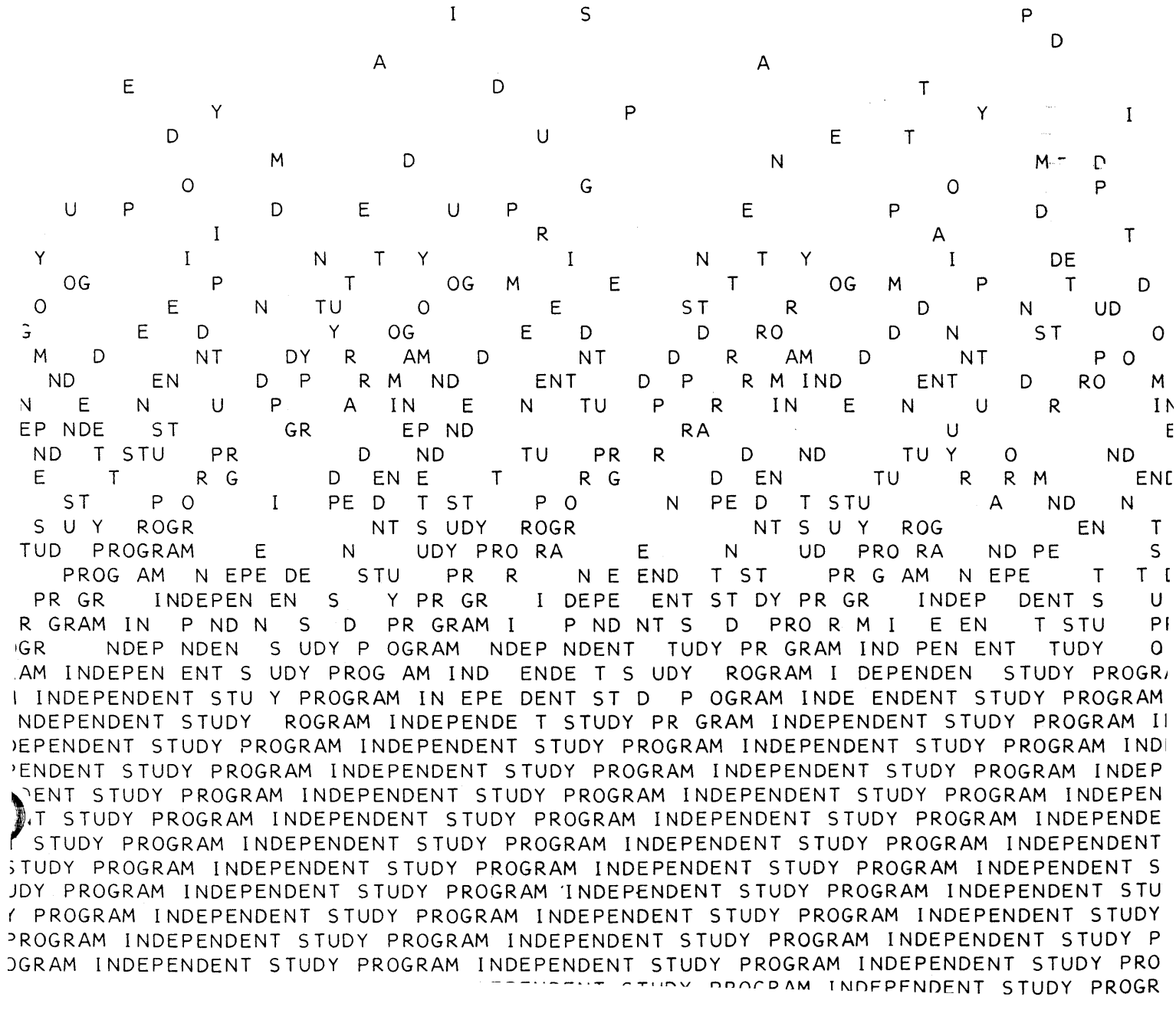
9.

Characters Read	Value Held
−12345	−123.45
ƀƀƀ123	1.23
ƀƀ−123	−1.23

10.

Original Characters Read	Characters Printed
−12345	$ƀƀ−123.45
ƀƀƀ123	$ƀƀƀƀ+1.23
ƀƀ−123	$ƀƀƀƀ−1.23

Topic **7**



Topic 7

Control of Program Flow

This topic describes the facilities available in PL/I to control the circumstances under which parts of a program will be executed. A data type, BIT strings, will be described. These are useful for holding the results of tests and acting as program flow switches. Arrays are discussed towards the end of the topic - these are useful PL/I aggregates of data which can be manipulated very easily within a DO group.

Objectives

On completion of this topic you should be able to:

- use the IF statement to control program flow
- use the various forms of DO statements to enable groups of statements to be executed repeatedly
- use SELECT groups to provide multiway conditional branches
- use the GOTO statement to control program branching
- use the LEAVE statement to transfer control from within a DO group
- use BIT string variables as program indicators
- be able to declare and use arrays.

Introduction

In all programs shown so far, all statements in the program have been executed once, in the order in which they were coded, unless they were included in a DO group. The group of statements in a DO group will be executed repeatedly until the condition on the DO statement is met, when control will pass to the first statement after the DO group. This is a very useful facility. It enables a program to produce any pay slips until there are no more employees' records to be processed. It enables a program to produce invoices until there are no more orders to process. However, there is a need for a finer level of control over which statements will be executed and when. Overtime pay should not be calculated unless the employee has worked overtime. A customer should not be invoiced for an item if there is no stock of that item to send him.

The IF Statement

The IF statement causes different groups of statements in a program to be executed, depending on some condition. It has a similar affect to the use of 'if' in speech.

'If it is sunny, I will go out, otherwise I will stay in'.

The form of the statement is:

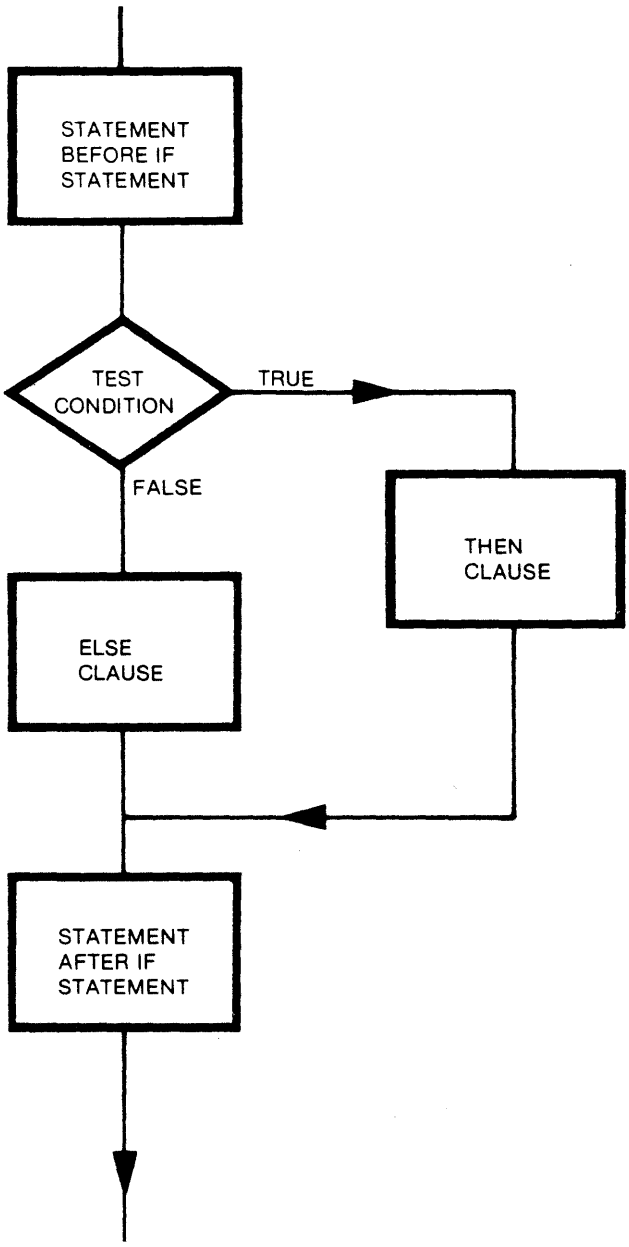
IF condition THEN then clause;
ELSE else clause;

Example

```
IF HOURS > 38 THEN OTIME = (HOURS - 38) * ORATE;
   ELSE OTIME = 0;
```

IF, THEN and ELSE are keywords. They must be separated from other identifiers by at least one blank.

The condition is typically a comparison of two element variables, or a variable and a constant, using relational operators. If the comparison is true, the THEN clause is executed, and the ELSE clause is not executed. If it is false, the THEN clause is not executed and the ELSE clause is. This flow of control is shown below.



In speech, the 'otherwise' option is not always needed. Instructions on how to prepare to go out might be:

Put on your jacket,
If it is cold, put on your overcoat.
Go out.

There is no statement of what to do if it is not cold.

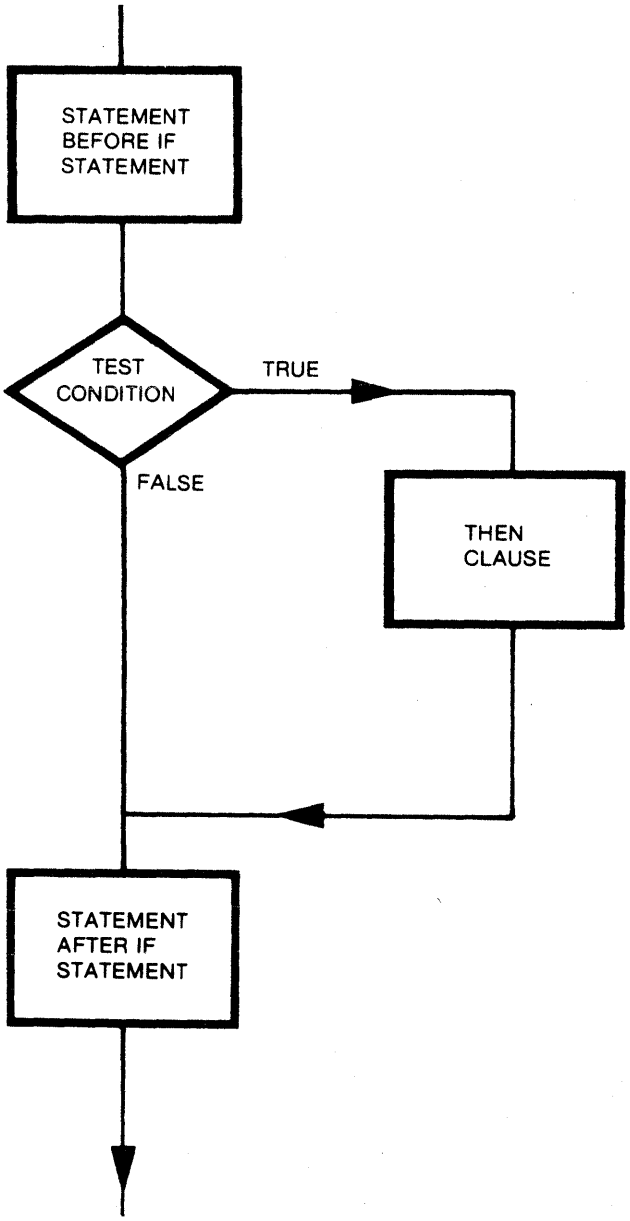
PL/I allows a similar form with the IF statement, by omitting the ELSE clause.

IF condition THEN then clause;

Example

```
PAY = HOURS * RATE;
IF HOURS > 38 THEN PAY = PAY + (HOURS - 38) * RATE * 0.5;
TAX = PAY * .3;
```

If the comparison is true, the THEN clause is executed. If it is not, the THEN clause is by-passed and control passes straight to the following statement. The flow of control is shown below.



The absence of an ELSE clause is shown by the statement after the THEN clause not starting with ELSE.

Relational Operators

The relational operator in the examples is '>', standing for 'is greater than'. The condition in these statements is 'If the value held in HOURS is greater than 38, then. . . . '

Relational Operator	Meaning	48-character Set Equivalent
=	is equal to	=
<	is less than	LT
>	is greater than	GT
< =	is less than or equal to	LE
> =	is greater than or equal to	GE
¬ =	is not equal to	NE
¬ <	is not less than	NL
¬ >	is not greater than	NG

The relational operators are shown above. They allow all the comparisons which are normally made. The composite operators, those made of two characters, like <=, must be coded in adjacent columns. The 'not' symbol is sometimes shown as ¬. If this form is used, it must be written carefully so that it is not confused with seven, 7. Of the relational operators, only = appears in the 48-character set. All others have equivalents in the 48-character set which are made up of pairs of alphabetic characters. If the 48-character set is being used, LT, GT, LE, GE, NE, NL and NG are reserved words; that is, they may not be used on their own as variable names or file names. Thus, LT would not be a valid variable name, but LTA would.

The operands of relational operators may be element expressions of any data type. They may not be structure expressions. The expressions are evaluated according to the rule laid out in Topic 4, and each reduced to a single value. When compared, both operands must have the same attributes. If they are coded with different attributes, one or both will be converted, in a similar way to that used in arithmetic and character expression evaluation.

Character Operands

If both operands are character strings, the shorter string is padded to the length of the longer string with blanks on the right hand end. Character operands may be compared by the = and ¬= operators. If the two strings, after padding, have identical contents, character by character, they will be counted as equal. If they do not, they will not be. Thus:

'THIS '	**will be equal to 'THISbb'.**
'THAT '	**will not be equal to 'bbTHAT'.**

Numeric Operands

If either or both of the operands is numeric, including PICTURE, then both operands will be converted to numeric.

If one operand is CHARACTER, it will be converted to DECIMAL FIXED(15,0), as in an arithmetic expression. If an operand is PICTURE, it will be converted to DECIMAL FIXED, with a precision as implied by the picture specification.

Once both operands are in coded arithmetic form, they will be converted to the same base and scale, as in an arithmetic expression. If the bases differ, the DECIMAL base will be converted to BINARY. If the scales differ, the FIXED scale will be converted to FLOAT.

Consider the following examples:

```
DCL CHAR1     CHAR(4)              INIT('1234');                    /* 10 */
DCL CHAR2     CHAR(6)              INIT('1234Ø');                   /* 20 */
DCL PIC       PIC'99V9'            INIT(12.1);                      /* 30 */
DCL FIX_DEC   FIXED DEC(5,2)       INIT(6.8);                       /* 40 */
DCL FLO_DEC   FLOAT DEC(6)         INIT(7.3);                       /* 50 */

   IF CHAR1 = CHAR2 THEN CHAR1 = PIC;                               /* 60 */
   IF CHAR1 > PIC THEN CHAR2 = 'BIG';                               /* 70 */
                  ELSE CHAR2 = 'SMALL';                             /* 80 */
   IF FIX_DEC >= FLO_DEC THEN FIX_DEC = FLO_DEC;                    /* 90 */
                         ELSE FLO_DEC = FIX_DEC;                    /* 100 */
```

In line 60, CHAR1 and CHAR2 are both character strings, and will be compared as character strings. As CHAR1 is shorter than CHAR2, it will be padded to the length of CHAR2 with blanks on the right hand end, giving a value '1234bb'. This will be compared, character by character, with CHAR2.

CHAR1	**1234bb**
CHAR2	**12340b**

The fifth character will not compare as equal. The THEN clause will be by-passed, and control will pass to line 70.

In line 70, as PIC is PICTURE, the comparison will be done in numeric form. CHAR1 contains a valid number, which will be converted to FIXED DECIMAL(15,0) and held as 1234. The 12.1 in PIC will be converted to FIXED DECIMAL(3,1), and the two numbers will be compared. As the value in CHAR1 is greater, the test will be true, and the THEN

clause will be executed. The ELSE clause on line 80 will be by-passed, and control will pass to line 90.

Lines 90 and 100 cause FIX__DEC and FLO__DEC to be set to the value of whichever is the smaller. If the value held in FIX__DEC is greater than or equal to the value held in FLO__DEC, the THEN clause will be executed. If not, the ELSE clause will be executed.

In all the examples, ELSE has been positioned directly under THEN, and both clauses have been assignment statements. If desired, ELSE could come immediately after the semi-colon of the THEN clause. Positioning it under THEN helps make the program more readable. The THEN and ELSE clauses of an IF statement may contain any executable statement, or a group of statements. The full options will be covered later in this topic.

You should now complete question 1 in the exercises at the end of this topic.

Logical Operators

Decisions are often made on more than one condition - 'If it is sunny and I have nothing better to do, I will go out' or 'If it is cold or it is raining, I will stay in'. PL/I allows such tests to be made by means of the logical operators shown as follows:

Logical Operator	Meaning	48-character Set Equivalent
&	and	AND
|	or (inclusive)	OR
¬	not	NOT

As before, the 48-character set equivalents of the symbols are reserved words.

The logical operators may be used to link together expressions involving relational operators in the following ways.

&

If two expressions are linked by the & operator, and if the expression to the left of the operator gives a true result and the expression to the right of the operator gives a true result, then the whole expression has a true result, otherwise it has a false result.

Examples:

```
IF (WEATHER = 'FINE') & (INTERESTS = 'NONE') THEN ACTION = 'GO OUT';
                                              ELSE ACTION = 'STAY IN';

IF (A = B) & (C = D) THEN X = Y;
                     ELSE X = Z;

IF (A > 0) & (A <= 10) THEN C = A * B;
                       ELSE C = 0;
```

The first example shows the first 'prose' example as it might appear in a PL/I program. The second shows two arithmetic comparisons.

The third example tests whether A is in the range greater than 0 but not greater than 10. In PL/I, each operand of & or the other logical operators must be a complete relational expression.

You may not code:

```
IF (A > 0) & ( <= 10 ) THEN C = A * B;
                       ELSE C = 0;
```

The | operator is coded as the same character that is used to make the concatenation operator, | |. With the & operator, both operands must give a true result for the whole expression to have a true result. With the | operator, if either of the operands is true, or if both are true, the final result is true.

Examples:

```
IF (TEMP < 40) | (WEATHER = 'RAIN') THEN ACTION = 'STAY IN';
IF (STOCK < ORDER) | (CREDIT > LIMIT) THEN MSG = 'DON''T DELIVER';
                                      ELSE MSG = 'DELIVER';

IF (A <= 0) | (A > 10) THEN C = 0;
                       ELSE C = A * B;
```

The first example is the PL/I equivalent of the second 'prose' example. If the temperature is low, stay in. If it is raining, stay in. If it is both cold and wet, stay in.

The second example shows an order processing situation. Delivery will be stopped by insufficient stock or by the customer exceeding his credit limit.

The third example will cause C to be set to 0 if A is 0, negative or greater than 10. If it is greater than 0 but not greater than 10, C will be set to A * B.

The ¬ operator is used as a prefix operator. It reverses the result of the expression to which it is attached. If (A>B) gives a true result, ¬(A>B) will give a false result. When attached to a simple expression, as here, it is often clearer to rewrite the expression than it is to use the ¬ operator. Thus, ¬(A>B) is the same expression as (A<=B), which probably has a more obvious meaning to anyone reading the program. However, the ¬ prefix may be attached to a more complicated expression, which itself forms only a part of a larger expression.

Example

```
IF (A > B) & ¬((C > D) | (E > F)| (X = Y)) THEN Y = Z;
                                          ELSE Y = 2 * Z;
```

In such a situation the ¬ operator may make the expression easier to understand.

Operator Hierarchy

In all the IF statements shown, brackets have been used to show the order in which expressions will be evaluated. The relational operators and the logical operators may be added to the hierarchy of operators to show the order in which expressions without brackets will be evaluated. The full hierarchy of operators is shown as follows:

Operators	Priority
** prefix + prefix – ¬	highest
* /	
infix + infix –	
| |	
< ¬< <= = >= > ¬>	
&	
|	lowest

Note:

In an expression with several operators at the same level of priority, if they are at the highest level, they will be processed from right to left; if they are at any other level, they will be processed from left to right.

Following this hierarchy, the expression:

```
A+B**C**-D>E*F/C>D&¬(F**G>X-Y)
```

is a valid expression, but it would probably be easier to understand its meaning, and be sure it would be interpreted as intended, if it were written as:

```
((A+(B*C**(-D)))>(E*F)) | (C>D) & ¬((F**G) > (X-Y))
```

The use of parentheses in expressions does not incur any penalty. If they are not used in expressions, a misunderstanding of the priority of operators will normally not cause a failure, but may cause the program to produce incorrect results. If the programmer coding the expression above, with parentheses, had been mistaken about the priorities of the operators ÷ and &, the significance of the expression would be completely changed.

You should now attempt question 2 in the exercises at the end of this topic.

THEN and ELSE Clauses

In all the examples in this topic, THEN and ELSE clauses have been assignment statements. They may be any executable statement, but may not be DECLARE, PROCEDURE or END statements.

Examples

```
IF STOCK > ORDER THEN STOCK = STOCK - ORDER;      /* 10 EXAMPLE 1 */
                 ELSE WRITE FILE(REPORT)          /* 20 */
                      FROM(STOCK_OUT_MSG);        /* 30 */

BIGGEST = A;                                      /* 10 EXAMPLE 2 */
IF B > BIGGEST THEN BIGGEST = B;                  /* 20 */
IF C > BIGGEST THEN BIGGEST = C;                  /* 30 */

IF A > B THEN IF A > C THEN BIGGEST = A;          /* 10 EXAMPLE 3 */
                       ELSE BIGGEST = C;          /* 20 */
              ELSE IF B > C THEN BIGGEST = B;     /* 30 */
                            ELSE BIGGEST = C;     /* 40 */

BIGGEST = A;                                      /* 10 EXAMPLE 4 */
IF B > C THEN IF B > BIGGEST THEN BIGGEST = B;     /* 20 */
                            ELSE;                 /* 30 */
         ELSE IF C > BIGGEST THEN BIGGEST = C;    /* 40 */
```

In the first example, the ELSE clause is a WRITE statement.

The second example uses only THEN clauses. After it has been executed, BIGGEST will hold the largest value in A, B and C.

The third and fourth examples show different ways of solving the same problem as the second example. The IF statement is an executable statement, and in these examples it has been used as the THEN and ELSE clauses. These are nested IF statements. IF statements may be nested up to 49 deep.

When IF statements are nested, each ELSE is associated with the innermost, unmatched THEN.

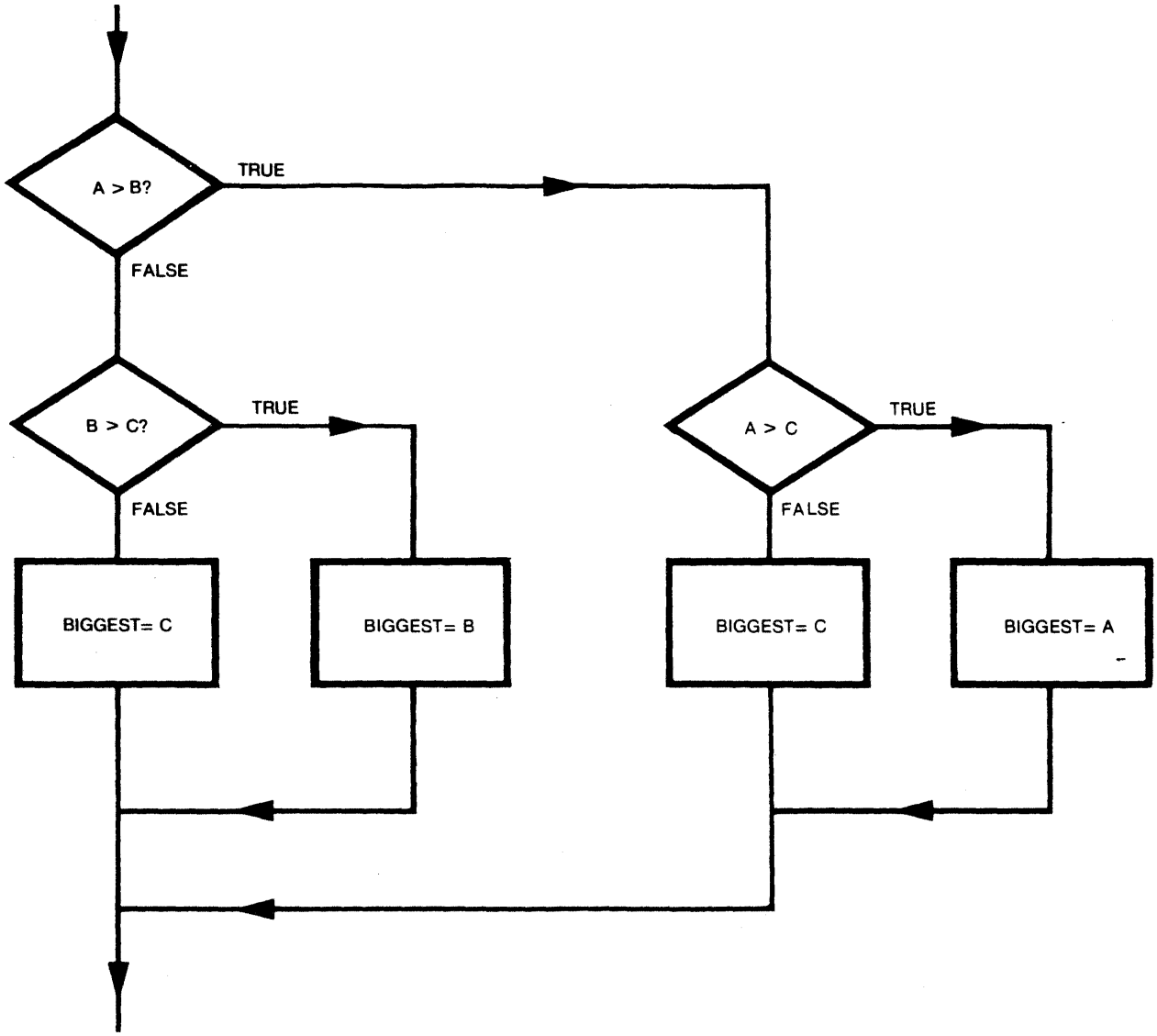
In example 3:

The ELSE on line 20 will be associated with the second THEN on line 10

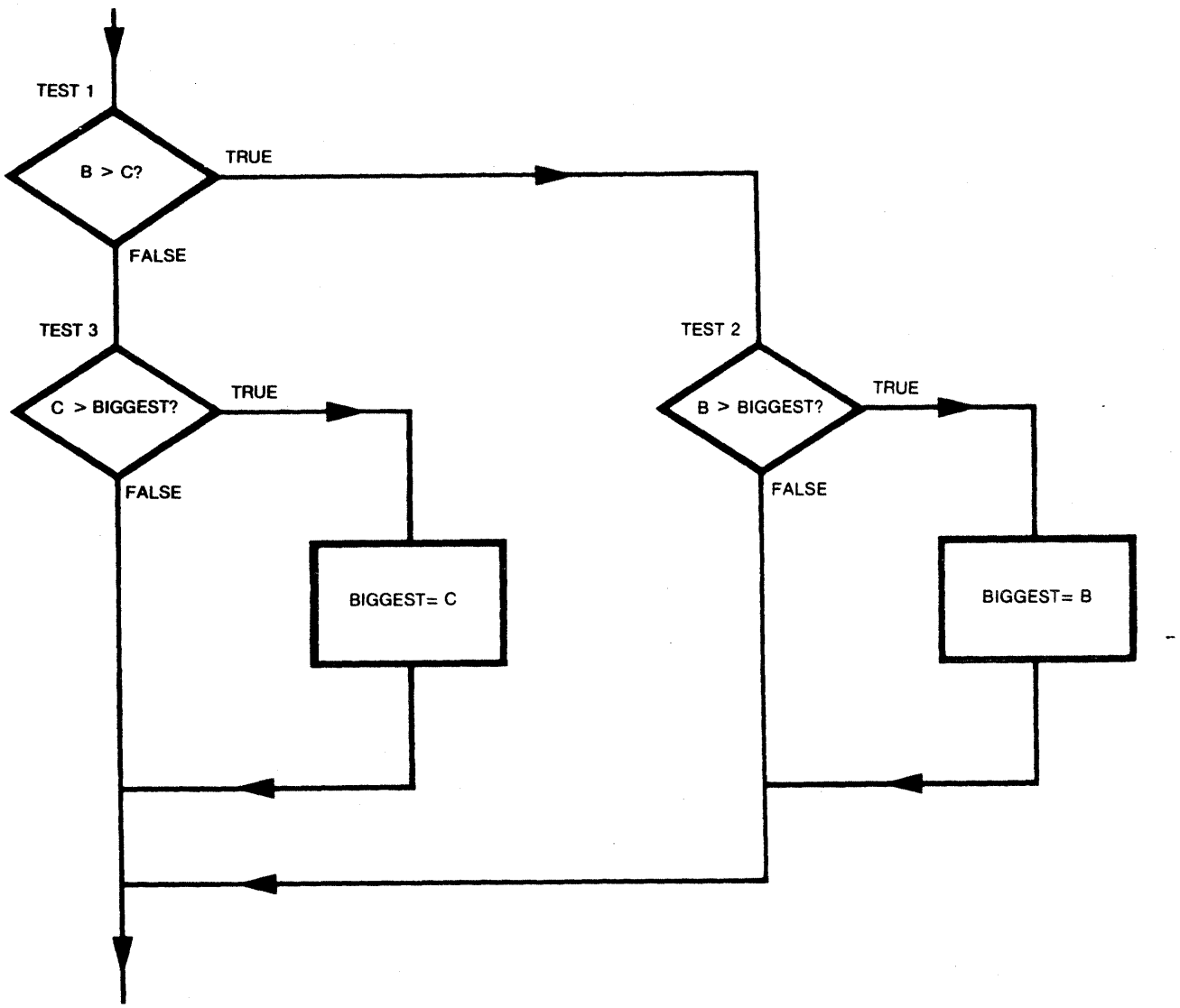
The ELSE on line 30 will be associated with the first THEN on line 10

The ELSE on line 40 will be associated with the THEN on line 30.

Below is a flowchart for example 3.



The flowchart for example 4 is shown below.



In the program, all IF statements have ELSE clauses, but in the flowchart only the first test has both THEN and ELSE clauses. The others have THEN clauses only.

If the ELSE clause of the second test, B>BIGGEST, were left off, the ELSE clause of the first test would be taken as the ELSE clause of the second test, so changing the logic of the program. This problem is overcome by using the null statement, consisting of only a semi-colon, as a dummy ELSE clause for the second test. The null ELSE clause is needed only in nested IF statements to cause the correct logical construction.

You should now attempt questions 3 and 4 in the exercises at the end of this topic.

In many programs, action to be taken as a THEN or ELSE clause will be more than one statement. They cannot be coded simply as a series of statements after the THEN or ELSE, because the second and subsequent statements would not be considered as part of the THEN or ELSE clause; they would be considered as statements following the IF statement. If a THEN or ELSE clause is to be more than one statement, it must be grouped together in a DO group.

DO Groups

An example of a DO group was shown in Topic 2 in the form:

```
DO WHILE (NAME_AND_ADDRESS ¬= ' ');
  .
  .
  .
END;
```

The group of statements between the DO statement and the END statement will be executed repeatedly until the condition in the WHILE option is false. This is one form of DO group which allows repetitive execution of a group of statements. In its simplest form, the DO group merely groups together a set of statements.

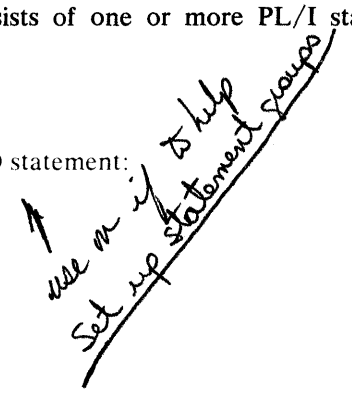
Simple DO Group

A simple DO group consists of one or more PL/I statements preceded by a simple DO statement:

DO;

and terminated by an END statement:

END;



Below are some examples of simple DO groups and how they might be used.

```
DO;
    SHORT_ORDER = ORDER_STOCK;
    ORDER = STOCK;
    REORDER = REORDER_QTY;
    WRITE FILE(REPORT) FROM(REORDER_LINE);
END;

DO;
    SHORT_ORDER = 0;
    STOCK = STOCK_ORDER;
    REORDER = 0;
END;

IF (STOCK < ORDER) THEN DO;
                            SHORT_ORDER = ORDER_STOCK;
                            ORDER = STOCK;
                            REORDER = REORDER_QTY;
                            WRITE FILE(REPORT) FROM(REORDER_LINE);
                        END;
                   ELSE DO;
                            SHORT_ORDER = 0;
                            STOCK = STOCK_ORDER;
                            REORDER = 0;
                        END;
```

Both DO groups contain executable statements, but they may contain any statements. If a simple DO group appears in the normal flow of execution between other executable statements, the DO and END have no effect. Its most common use is as the THEN and ELSE clauses of an IF statement, as in the above example.

The whole of the first DO group will be the THEN clause, and all the statements in it will be executed if the test is true. If it is false, the whole of the first DO group will be by-passed, and all of the statements in the second DO group will be executed. The DO and END statements are not executed. They delimit the group.

A DO group may contain any PL/I statements, so it may contain nested DO groups. If DO groups are nested, each END statement will be associated with the nearest preceding unattached DO statement, as below.

```
DO;                         /* DO NO1 */
.
.
    DO;                     /* DO NO2 */
    .
    .
    END;                    /* MATCHES DO NO2 */
    .
    DO;                     /* DO NO3 */
    .
    .
    END;                    /* MATCHES DO NO3 */
.
.
END;                        /* MATCHES DO NO1 */
```

The maximum permitted depth of nesting is 49.

Iterative DO Groups

Iterative DO groups cause the statements within the DO group to be executed a number of times depending on options on the DO statement.

WHILE Option

The option may be the WHILE option:

DO WHILE (expression);

The expression may be any expression which can be used in an IF statement. **Before** the statements in the group are executed, the expression is evaluated. If it gives a true result, the statements in the group are executed, and control passes back to the beginning of the group to re-evaluate the expression. This continues until the expression gives a false result, when control passes to the statement following the END statement for the group.

Below is an example to add together the first N integers (1+2+3+.....+N). If N is zero or negative, the statements in the DO group will not be executed at all, and the ISUM will be 0 at the statement after END;

```
I = 1;
ISUM = 0;
   DO WHILE (I <= N);
      ISUM = ISUM + I;
      I = I + 1;
   END;
/* NEXT STATEMENT */
```

UNTIL Option

The general format of the UNTIL option is

DO UNTIL (expression);

The expression may be any expression which may be used in an IF statement. **After** the statements in the group are executed, the expression is evaluated. If it gives a true result, control passes to the statement following the END statement for the group. If it gives a false result, control passes back to the beginning of the group and the statements in the group are re-executed.

Below is another example which adds together the first N integers. In this case, if N is zero or negative, the UNTIL expression will never be true and the program will be in a continuous loop.

```
I = 1;
ISUM = 0;
   DO UNTIL(I = N + 1);
      ISUM = ISUM + I;
      I = I + 1;
   END;
/* NEXT STATEMENT */
```

Note that the major difference between the WHILE option and the UNTIL option is that in the former case the expression is evaluated at the beginning of the DO group and in the latter case it is evaluated at the end of the DO group. This implies that with the UNTIL option, the DO group will always be executed at least once. This is not true with the WHILE option.

Index Variables

DO groups may also be executed repetitively by assigning a series of values to an index variable, and executing the statements of the DO group once with the index variable set to each value.

This type of DO group may be specified in several ways.

Its simplest form is:

DO variable = expression, expression, ...;

Examples:

```
DO I = 10, 20, 3*N, X**2, 1;

DO CHARS = 'A', 'BCD', CHAR1||'IJK';
```

The variable may be any type of element variable and is called the index variable. The expressions may be any sort of element expression whose result can be converted to the attributes of the index variable. All the expressions are evaluated before the group is entered,

and the statements of the group are executed once with the index variable set to the result of each of the expressions in turn.

The group which would follow the first DO statement would be executed with I set to 10, then 20, 3*N, X**2 and 1. The group which would follow the second DO statement would be executed with CHARS set to 'A', 'BCD' and the character value of CHAR1 extended with the characters 'IJK'. Any reference to the index variable within a DO group will refer to the value to which it was set before that execution of the group.

Execution of the DO group with the index variable set to the result of any particular expression may be made conditional by use of the WHILE option.

```
DO I = 1,10,39 WHILE(A>B),2*X;
```

The group will be executed once with I set to 1 and once with I set to 10. It will be executed with I set to 39 only if A is greater than B. Whether or not it is executed with I set to 39, it will then be executed with I set to 2*X. 2*X will be evaluated before the group is entered for the first time, so that, if X is changed within the group, this will not influence the values to which I will be set.

However, the expression A>B is evaluated after I has been set to 39, using the values currently in A and B. The expression in the WHILE option is the only expression used in the control of DO groups which is evaluated during the execution of the DO group; all others are evaluated before the group is first executed.

You should now attempt question 5 in the exercise at the end of this topic.

TO Option

The index variable may be set to a series of values by one or more iterative specifications:

DO variable = expression 1 TO expression 2;

Examples:

```
DO VAR = 10 TO 30;
DO IND = A*B TO R-20;
```

TO is a keyword, and must be separated from preceding and following identifiers by at least one space.

When the DO group is executed, both expressions are evaluated. The index variable is set equal to the value of the first expression. If it does not exceed the value of the second expression, the DO group is executed. 1 is then added to the value of the index variable. Control passes to the top of the group and the index variable is tested again. This continues until the value in the index variable exceeds the value of the second expression, when control passes to the statement after the END statement. An iterative DO group in PL/I will not be executed at all if the value of the first expression is greater than the value of the second expression.

```
DO I = 1 TO N;
```

would cause the DO group to be by-passed if N were 0 or negative. If N were 1 it would be executed once only.

Above you have been shown statements to calculate the sum of the first N integers, using a DO statement with the WHILE and UNTIL options. This may be coded more simply using iterative DO statements.

```
ISUM = Ø;
DO I = 1 TO N;
   ISUM = ISUM + I;
END;
/* NEXT STATEMENT */
```

You should now attempt question 6 in the exercises at the end of this topic.

BY Option

An increment other than 1 may be specified for an iterative DO group by using the BY option:

```
DO VAR = 1Ø TO 3Ø BY 2;
DO IND = A*B BY -3 TO R-2Ø;
```

The BY option may precede or follow the TO option. The value of the expression in the BY option is used as the increment for the index variable. The expression is evaluated once only before the DO group is entered. If the expression used in the BY option has a negative value, the group will be executed until the index variable is less than the value of the expression in the TO option.

```
DO I = 1Ø TO 3Ø BY 2;
```

will cause the group following to be executed 11 times with I having value of 10, 12, 14, 16, 18, ~~10, 11, 14, 16, 18~~ and 30.

```
DO I = 3Ø TO 1Ø BY -2;
```

will cause the group following to be executed 11 times with I having values 30, 28, 26, 24, 22, 20, 18, 16, 14, 12 and 10.

```
DO I = 3Ø TO 1Ø;
```

will cause the group following to be by-passed, since the value of I, 30, will exceed the limit value, 10, the first time.

These types of specification may also be qualified by the WHILE or UNTIL options:

DO variable = expression 1 BY expression 2 TO expression 3 { WHILE / UNTIL } (expression 4);

Examples

```
DO I = 1 TO 1Ø UNTIL(ERR>LIMIT);
DO J = 1Ø TO 3*X BY 2 WHILE(N*J ¬= M);
```

In the UNTIL case, the index variable is incremented and tested against the limiting value. If it is not out of the range the group is executed and the UNTIL expression is then evaluated on reaching the end of the group. If the expression is true then control passes to the statement after the END. Otherwise control passes back to the start of the group and the index variable

is incremented again. In the WHILE case, after the index variable has been incremented and tested against the limiting value, the expression in the WHILE option is evaluated. If the control variable is used in the expression in the WHILE option, the incremented value is used. If the expression is true the group is executed, otherwise control passes to the statement after the END.

Where there is no definite maximum, but a value in a control variable is still needed, an iterative specification may be used with a BY option but no TO option.

DO variable = expression 1 BY expression 2 { WHILE } (expression 3);
{ UNTIL }

Example.

```
DO I = 7 BY 7 WHILE(ERR<LIMIT);
DO J = K BY L UNTIL (J > X);
```

The DO groups will be executed repeatedly, with the value of the index variable being incremented, until they are stopped by the WHILE or UNTIL option. These options are not obligatory but will normally be present. If they are not, some other means will have to be used to stop the execution of the loop, or it will continue until the index variable is unable to hold the value being assigned to it. Other techniques for stopping the execution of the DO group will be discussed later in this topic.

Using this form, the BY option must be specified, even if it is 1. If it is not, the index variable will not be incremented and the group will be executed once at the most.

REPEAT option

The TO and BY options allow the index variable to be incremented by fixed negative or positive amounts. In contrast, the REPEAT option, an alternative to the TO and BY options, allows the index variable to be incremented non-linearly. It is used in the following way.

```
DO I = 1 REPEAT 2*I;
.
.
END;
```

In this example the initial value of the index variable (I) is 1. On subsequent executions of the loop the evaluation of the REPEAT expression is assigned to the index variable. Thus, the index variable will have the following values when the DO group is executed

1, 2, 4, 8, 16, ...

Note that with the REPEAT option no terminal value is specified and thus to halt execution of the loop, an UNTIL or WHILE option will need to be specified.

The simplest form of an iterative DO group was defined as:

DO variable = expression, expression,....;

The more complicated forms may also have more than one specification:

DO variable = specification, specification,....;

where each specification may be:

expression 1 BY expression 2 TO expression 3 { UNTIL / WHILE } (expression 4);

Example

```
DO I = 1 TO 10 WHILE(ERR > LIMIT),20 TO 30 BY 2;
```

When execution terminates under control of the first specification, it starts under control of the second. If the condition ERR>LIMIT stopped execution of the group under control of the first specification as it was about to be executed with I set to 6, the group would be executed a total of 11 times, with I set to 1, 2, 3, 4, 5, 20, 22, 24, 26, 28 and 30.

The specifications which may be used to control the execution of a DO group may take varied forms. Very often, the specification will take a simple form, either:

DO WHILE (expression);
or
DO variable = constant TO constant;

If more complicated forms are used, it is important to remember that:

1. Expressions in the TO and BY options are evaluated once only, before the group is entered.
2. Expressions in the WHILE option are evaluated every time they are referred to, after the new value has been assigned to the index variable and before the group is executed.
3. Expressions in the UNTIL option are evaluated every time they are referred to, after the group has been executed.
4. The current evaluation of the expression on the REPEAT option is assigned to the index variable. It is not a fixed increment.

You should now attempt questions 7 and 8 in the exercises at the end of this topic.

SELECT Statement

Earlier in the topic you learned about the IF statement and how it could be nested within THEN and ELSE clauses as follows:

```
IF A>B THEN IF A>C THEN X=1;
                  ELSE IF A=C THEN X=2;
                                   ELSE X=3;
       ELSE IF A=B THEN X=4;
                  ELSE X=5;
```

As the level of nesting increases so does the difficulty of understanding the logic of the program coding.

The SELECT statement, which heads a select group, provides a multiway conditional branch and also is an alternative to nested IF statements. The general format of the SELECT statement and group is as follows:

```
SELECT(E);
  WHEN (E1, E2, E3) action 1;
  WHEN (E4, E5) action 2;
  .
  .
  OTHERWISE action n;
END;
```

Here E, E1 etc. are expressions. When control reaches the SELECT statement, the expression E is evaluated and its value saved. The other expressions are evaluated in the order specified in the coding and compared with the saved value of E. As soon as an expression is found to have a value equal to that of E the corresponding action of the WHEN clause is executed. No further expressions of WHEN clauses are evaluated. If none of the expressions is equal to the expression in the SELECT statement then the action of the OTHERWISE statement is executed.

The action after a WHEN or OTHERWISE clause can be a single or compound statement, a DO group, a SELECT group or a BEGIN block. After the action has been carried out, control returns to the first statement after the END of the SELECT group unless the flow of control has been altered by the specified action.

Example

```
SELECT(FIGURE);
   WHEN ('SQUARE') AREA = R * R;
   WHEN ('CIRCLE') AREA = 3.14 * R * R;
   WHEN ('SPHERE') AREA = 4 * 3.14 * R * R;
   OTHERWISE AREA = Ø;
END;
```

The expression in the SELECT statement can be omitted, in which case each expression in the WHEN clauses is evaluated and converted to a bit string. The corresponding action will then be executed if any of the bits have value '1'B.

Example

```
SELECT;
   WHEN(A > B) STRING = 'BIGGER';
   WHEN(A = B) STRING = 'EQUAL';
   WHEN(A < B) STRING = 'SMALLER';
END;
```

In the above example there was no need for an OTHERWISE clause because one of the WHEN clause actions must be executed. However, if the OTHERWISE clause is omitted and the execution of a SELECT group does not result in a WHEN clause action being executed, then the program will terminate abnormally.

```
SELECT;
   WHEN (A > 0) VALUE = '+VE';
   WHEN (A < 0) VALUE = '-VE';
END;
```

In the above example, if the value of A is 0 then neither of the WHEN actions will be executed. There is no OTHERWISE clause specified in the SELECT group and so in this case the program will terminate abnormally.

At the beginning of this topic there is an example of nested IF statements. Using nested SELECT groups the same logic can be presented in a more comprehensible fashion as follows:

```
SELECT;
   WHEN(A>B) SELECT;
                WHEN(A>C) X=1;
                WHEN(A=C) X=2;
                WHEN(A<C) X=3;
             END;
   WHEN(A=B) X=4;
   WHEN(A<B) X=5;
END;
```

You should now attempt question 9 in the exercises at the end of this topic.

Transfer of Control

GOTO Statement

DO, IF and SELECT statements may cause groups of statements to be executed or not. Another method of causing statements to be executed is to branch to any statement from any other, by means of a GOTO statement. Below is the main body of the card listing program from topic 2, re-written using GO TO statements.

```
NEXT:  READ FILE(CARDSIN) INTO(NAME_AND_ADDRESS);
       IF(NAME_AND_ADDRESS = ' ') THEN GOTO OUT;
       WRITE FILE(PRNTOUT) FROM(NAME_AND_ADDRESS);
       GO TO NEXT;
OUT:   ;
```

NEXT and OUT are called label constants. A label constant may be attached to the beginning of any PL/I statement. It consists of an identifier of up to 31 characters, and is separated from the statement by a colon. The colon character does not appear in the 48-character set, and is replaced by two periods in adjacent position (..). There may be one or more spaces between the label constant and the colon, and between the colon and the statement. The main use of a label is to identify a statement which is to be branched to by a GOTO statement.

The format of the GOTO statement is:

GOTO label;

There must be at least one space between GOTO and the label. There may be one space between GO and TO, but not more. The effect of the statement is to cause the next statement executed to be the one identified by the label, as if it had been the statement following the GOTO statement.

The label must be on an executable statement. A GOTO statement may not branch to a label on:

A PROCEDURE statement
A DECLARE statement
A statement inside a repetitive DO group, unless the GOTO statement is also inside it.

A GOTO statement may branch out of a DO group. If it does, the group is terminated and, if it has a repetitive specification, it cannot be branched back into.

A GOTO may form all or part of a THEN or ELSE clause on an IF statement, as shown above. If it does, it is executed and the statements following the IF statement will not be executed. When a GOTO statement appears as the THEN clause of an IF statement, the IF statement is sometimes called a conditional GOTO statement. Statements following a conditional GOTO statement will be executed only if the condition is not met. The statement following an ordinary GOTO statement can only be executed if it bears a label, and this label is referred to elsewhere in a GOTO statement. It is an error if the statement following an unconditional GOTO statement does not bear a label.

When studying programs with GOTO statements it becomes difficult to decide how control was passed to any particular labelled statement in a program. It may possibly have been branched to from one of several GOTO statements. The coding of GOTO statements hence makes a program more difficult to maintain or modify and they should be avoided whenever possible. In fact, in PL/I, by the careful use of switches and DO groups they can be avoided altogether, with one exception (see topic 19, Handling Exceptional Conditions).

LEAVE Statement

The LEAVE statement is used to transfer control from within a DO group to the first executable statement after the END statement of the DO group.

Example:

```
DO;
.
.
   LEAVE;
.
END;
A: /* NEXT STATEMENT */;
```

In this example, if the LEAVE statement is executed, then the next statement to be executed will be the one labelled A.

The LEAVE statement can specify a label, in which case control will be passed to the first statement that follows on from the DO group which has that specified label on the DO statement.

Example:

```
A: DO;
   .
      B: DO I = 1 TO 10;
         .
         IF J > 1 THEN LEAVE A;
         .
         END;
   .
   END;
C: /* STATEMENT */ ;
```

In this case if the LEAVE statement is executed than control will be passed to the statement labelled C.

Bit String Data

Bit string variables and constants hold data of a binary nature -1 or 0, Yes or No, True or False. The information is held as a series of binary 1s and 0s at a rate of one per bit, eight per byte.

A bit string variable is declared with the attribute BIT, and a length, in a similar way to a character string variable. For example:

```
DCL BITS BIT(8);
DCL ATTEND BIT(6);
DCL TRUTH BIT(1);
```

The length is the number of binary digits it may hold, and may not exceed 32767. The default is 1.

A bit string constant consists of a string of 0 and 1 characters, preceded by a single quote, and followed by a single quote and a B. For example:

```
'0010'B
'1'B
'10010110'B
(2)'11001'B
```

The last example demonstrates that a bit string constant may have a repetition factor attached to it, like a character string constant. The length of a bit string constant is the number of 0 and 1 characters in the string multiplied by the repetition factor, if applicable. The lengths of the examples are 4, 1, 8 and 10 respectively. The maximum permitted length depends on the space available to the compiler, but will never be less than 4096.

Bit string variables may be initialized with bit string constants:

```
DCL BITS BIT(8) INIT('11111111'B);
DCL ATTEND BIT(6) INIT('000000'B);
DCL TRUTH BIT(1) INIT('0'B);
```

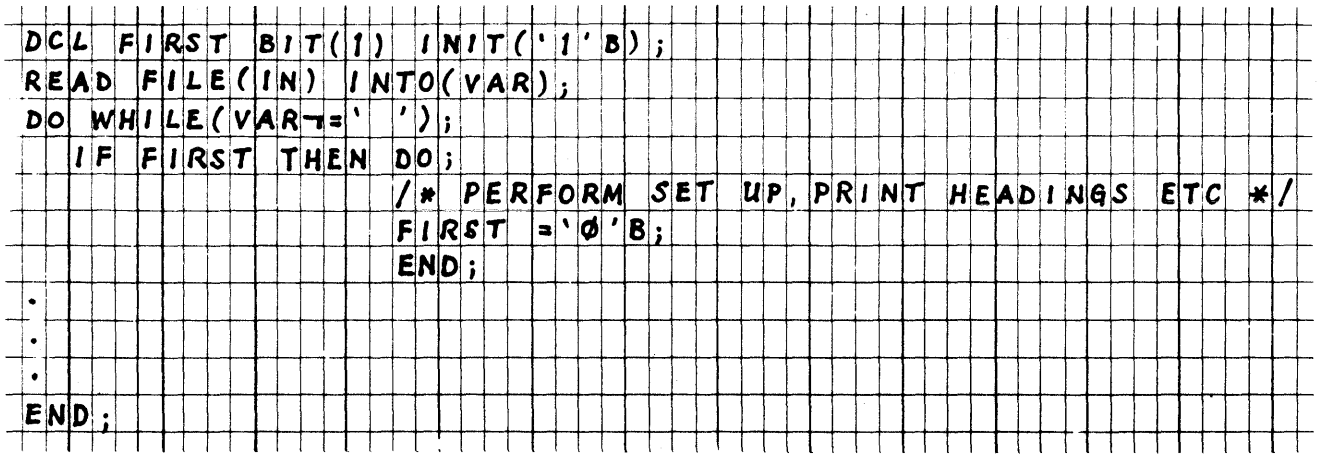
The Use of Bit Strings of Length 1

In an IF statement, or the WHILE or UNTIL option of a DO statement, the result of the test is a bit string of length 1. This will be '1'B if the test is true and '0'B if the test is false. Bit string variables may be used to hold results of comparisons and logical expressions and may be used in IF statements and WHILE options, either as the whole of the expression, or as a part of the expression. For example:

```
TRUTH = A>B & C<D;
IF A>B & C<D THEN TRUTH = '1'B;
                ELSE TRUTH = '0'B;
IF TRUTH THEN X=Y;
         ELSE X=Z;
IF ¬TRUTH & TEXT='LAST' THEN X=0;
```

The assignment statement and the first IF statement have identical effects. Having assigned a value to TRUTH, it may now be used as in the second IF statement, as the expression in the test. If TRUTH contains a 1 bit, the THEN clause will be executed, otherwise the ELSE clause will be executed. In the last statement, TRUTH is an operand of the ¬ operator. If TRUTH contains a 1 bit, it will be switched to a 0 bit. If it contains a 0 bit, it will be switched to a 1 bit. This will then be ANDed with the result of comparing the contents of TEXT with 'LAST'. That is, if TRUTH contained a 0 bit and TEXT contained 'LAST' the THEN clause will be executed, otherwise it will not.

Bit string variables of length 1 are often used as simple indicators in a program. For example, in the following program extract, certain functions must be performed for the first record processed. FIRST will give a true result in the IF test the first time it is executed, but will give a false result afterwards, as it is assigned '0'B within the THEN clause of the IF statement.



A bit string variable of length 1 is often used to process a sequential input file until the end of data is reached. In the following program extract, when the first attempt to read beyond the end of data occurs the ON ENDFILE statement will be invoked and a zero bit will be assigned to MORE_DATA. The program then continues from the location just beyond the point where the end of data condition was recognized. This will be at the end of the DO group whereupon the DO WHILE (MORE_DATA); statement is executed. This is equivalent to DO WHILE (MORE_DATA = '1'B);. Thus, when MORE_DATA contains a zero bit the DO GROUP will no longer be executed. ON statements will be discussed in greater detail later, but this will serve for now.

```
DCL MORE_DATA BIT(1) INIT ('1'B);
ON ENDFILE(TRANS) MORE_DATA = '0'B;
READ FILE(TRANS) INTO (INREC);
DO WHILE(MORE_DATA);
        /* PROCESS THE INREC */
READ FILE(TRANS) INTO (INREC);
END;
.
.
.
```

The Use of Bit Strings of Length Greater Than 1

Bit strings of any length may be used in similar ways to bit strings of length 1. If a larger bit string is used as the expression in an IF statement, then if any bit in the string is a 1 bit, the string gives a true result. The following strings would all give a true result:

```
'11111111'B
'10000000'B
'00000001'B
```

The following strings would all give a false result if used in a test:

```
'00'B
'0000'B
'00000000'B
```

The string ATTEND could be used as the attendance register for one student for six days. In the test:

```
DCL ATTEND BIT(6);
IF ATTEND THEN . . .
```

if the student attended on any day or days, the THEN clause will be executed. If the NOT operator (¬) is applied to a bit string, every bit has its setting reversed. Thus, every bit which is 1 is set to 0, every bit which is 0 is set to 1. In the test:

```
IF ¬ATTEND THEN . . .
```

the setting of every bit in ATTEND will be reversed. If any bit in the string which is the result of this operation is 1, the THEN clause will be executed. The effect of the test is that if the student failed to attend on *any day,* the THEN clause will be executed.

When the AND and OR operators (& and |) are applied to bit strings, they operate on each bit position separately to produce a result string of the same length as the longer operand. When the AND operator is used, each position in the result string will be 1 if the equivalent position in both operands is 1, otherwise it will be 0. For example:

```
    001101001
&   101100110
    ---------
    001100000
```

This operation could be used with the string ATTEND to establish whether a student attended on a particular day. The requirement is to obtain a 1 bit if the student attended on that day, and a 0 bit for all others. Thus, to establish whether the student attended on day 2:

```
IF ATTEND & '01000'B THEN DO . . .
```

If the student attended on days 2, 3, 5 and 6, the effect will be:

```
    011011
&   010000
    ------
    010000
```

The test will give a true result, and the THEN clause will be executed.

If the OR Operator is used, the result string will contain a 1 in each position if either or both of the operands had a 1 in that position:

```
    0100110
|   1010101
    -------
    1110111
```

This could be used to change the setting of some of the bits in a string. If there were a query regarding the marking of the attendance of a student on day 3, it could be turned on by:

```
ATTEND = ATTEND | '00100'B;
```

Whatever the setting of the third bit before the statement is executed, it will be 1 afterwards. The other bits will not be changed.

Bit Strings and Concatenation

Bit strings may be manipulated by the concatenation operator in the same way as character strings. When two bit strings are concatenated, the result string consists of the two operand strings joined together. The length of the result string will be the sum of the length of the operands. Thus:

```
'00'B||'111'B gives '000111'B
'0101'B||'1010'B gives '01011010'B
'0'B||'1011010'B gives '01011010'B
```

Padding and Truncation of Bit Strings

Bit strings are padded or truncated in similar circumstances, and in a similar way to character strings.

If a bit string is assigned to another bit string which is shorter, it will be truncated on the right. Thus:

```
ATTEND = '000000111'B;
```

will assign '000000'B to ATTEND.

When two bit strings of unequal lengths are operated on, the shorter will be padded to the length of the longer with binary zeroes on the right. Thus:

'1111111'B & '1111'B

will have the same effect as:

'1111111'B & '1111000'B

and will give a result of '1111000'B.

Bit strings provide a very efficient and compact method of processing binary data. If your background is non-mathematical, the & and | operations may, at first sight, appear a little strange. Their operation on single bit strings is similar to their common English usage. When applied to larger strings, they continue to work in the same way, but work on each position in the strings.

You should now attempt exercise 10 at the end of this topic.

Arrays

In Topic 6, structures were discussed. A structure is an aggregate of elements, not necessarily with similar attributes. The elements are identified by unique names, which may need levels of qualification to make them unique.

An array is an aggregate of elements with identical attributes. The elements are identified by the array name and a subscript - a subscript is a number which uniquely identifies a particular element in an array. Arrays have many uses, such as holding tables of values and sets of similar measurements.

Arrays must be declared. Their declarations give the name of the array, the attributes of each element and how many elements there are. The number of elements is put in brackets immediately after the array name and is called the **dimension attribute.** There may be spaces between the array name and the brackets, but not attributes.

```
DCL AVE_WEIGHT (6) FIXED DEC(3);
```

The above declaration will give an array of six elements, each with attributes FIXED DEC(3). The individual elements of the array may be used as element variables, and are identified by the array name and a subscript. In its simplest form, the subscript is an integer enclosed in brackets. The integer must not be greater than the dimension attribute of the array.

```
AVE_WEIGHT(3)=10;
```

will assign 10 to the third element of AVE__WEIGHT.

A subscript may be any expression whose result can be assigned to a FIXED BINARY (15,0) field. If it is an expression, it will be evaluated and its result truncated to an integer.

```
AVE_WEIGHT(AGE/10-1) = 10;
```

If AGE holds a value of 26 the element of AVE__WEIGHT accessed will be:

```
  26/10-1
= 2.6-1
= 1.6
= 1
```

So far we have been talking about one-dimensional arrays, i.e. arrays which need only one subscript to identify a particular element within them. The need for two or more dimensional tables is met by two or more dimensional arrays, up to a limit of 15 dimensions. In declaring a multi-dimensional array, the various dimension attributes are separated by commas. A two dimensional array, which might hold a table of average weights, classified by age group and sex, would be declared:

```
DCL TWO_DIM(2,6) FIXED DEC(3);
```

2 and 6 are the two **dimension attributes** of the array. TWO__DIM will have 12 elements and might be interpreted as follows:

	15→20	21→30	31→40	41→50	51→60	61→70
MALE	1,1	1,2	1,3	1,4	1,5	1,6
FEMALE	2,1	2,2	2,3	2,4	2,5	2,6

TWO__DIM

In the above diagram element (2,3) of array TWO__DIM would contain the average weight for females between 31 and 40 years of age. For a two-dimensional array the first subscript could be interpreted as a 'row number' and the second subscript as a 'column number'. When accessing the elements of a multi-dimensional array, there must be as many subscripts as there are dimensions; the subscripts are separated by commas. The subscripts must be in the same order as the dimensions, and no subscript may exceed the value of the equivalent dimension attribute.

```
TWO_DIM(1,5)=10;
WEIGHT=TWO_DIM(1,(AGE-1)/10);
```

The constant given for the dimension attribute is called the **upper bound** for that dimension. The **lower bound** is 1. The number of integers between the lower and upper bound is called the **extent**. In some circumstances, it may be more convenient to specify a lower bound other than 1. This might be useful if the array were to hold, say, average weights for all ages between 70 and 80, or densities at all temperatures from -10°C to +10°C. If the lower bound is not 1, it must be specified as an optionally signed constant, preceding the upper bound in the declaration, and separated from it by a colon:

```
DCL WEIGHT(2,70:80) FIXED DEC(3);
DCL DENSITY(-10:10) FLOAT(6);
```

The extent for a dimension which has both bounds specified is:

upper bound - lower bound + 1

The total number of elements for WEIGHT will be 2*(80-70+1) = 22. The number of elements for DENSITY will be 10-(-10)+1 = 21.

The range of subscripts allowed for any dimension which has both lower and upper bounds specified is from the lower bound to the upper bound. The lowest value which may be specified as the lower bound is -32767. The highest value which may be specified as the upper bound is 32767. The upper bound must be greater than the lower bound.

Referring to Arrays

A reference to an element of an array may appear anywhere that a reference to an ordinary variable may appear. In most situations, it does not matter whether an array is declared:

```
     DCL   AR(2,6);
or   DCL   AR(6,2);
```

as long as references to the elements of the array are consistent with the declaration. Iterative DO groups are often used where all the elements of an array are to be processed in a similar way. If the array has more than one dimension, it may be necessary to nest DO groups, as shown below.

```
DCL AR(6,2);

DO I=1 TO 6;
   DO J=1 TO 2;
      AR(I,J)=J+(I-1)*2;
   END;
END;
```

This code will give identical results to the following:

```
DCL AR(6,2);

AR(1,1) = 1;
AR(1,2) = 2;
AR(2,1) = 3;
AR(2,2) = 4;
AR(3,1) = 5;
AR(3,2) = 6;
AR(4,1) = 7;
AR(4,2) = 8;
AR(5,1) = 9;
AR(5,2) = 10;
AR(6,1) = 11;
AR(6,2) = 12;
```

You should now attempt question 11 in the exercises at the end of this topic.

Array Assignment

If exactly the same operation is to be carried out on all elements of an array, it may be done by an array assignment statement.

Array assignments, like structure assignments, may take two forms:

array name = element expression;
array name = array expression;

An array expression is an expression, one of whose operands is an un-subscripted array name. In the first form, the expression on the right-hand side is evaluated and the result is assigned to each element of the array, e.g.

```
AR=0;
```

will assign 0 to every element of the array AR.

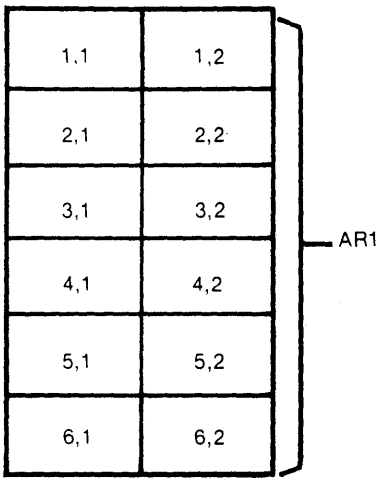
In the second form, all arrays in the statement must have the same number of dimensions, and each dimension must have the same bounds; having the same extents is not adequate.

Given the following arrays:

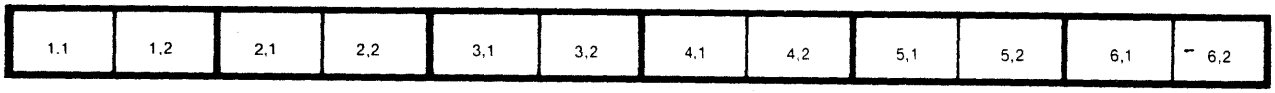
```
DCL AR1(6,2);
DCL AR2(0:5,2);
DCL AR3(6,2);
```

AR1 and AR3 could appear in the same assignment statement, but AR2 could not appear with either of the others.

Array elements are stored in contiguous storage with the right-hand subscript of each element varying fastest. For a two-dimensional array this means that the array elements are stored 'row by row'. For example, AR1 above is a '6 by 2' array which can be represented as follows:



The elements of AR1 will be stored in the following order:



When an array expression is evaluated and assigned to another array, this is done in the same order as that in which the array is stored. Thus,

```
AR1=2*AR3;
```

will be executed as if it were coded:

```
DO I=1 TO 6;
   DO J=1 TO 2;
      AR1(I,J)=2*AR3(I,J);
   END;
END;
```

For this assignment, it does not matter in what order the expression is evaluated. In other situations it will

```
AR1=AR1(3,2)*AR3;
```

will expand to:

```
DO I=1 TO 6;
   DO J=1 TO 2;
      AR1(I,J)=AR1(3,2)*AR3(I,J);
   END;
END;
```

The order here is vital, as AR1(3,2) will be changed part way through the calculations, and the new value will be used in the remaining calculations (see question 12 at the end of this topic).

Arrays in Input and Output

An array name may be used in READ and WRITE statements in the same way as a structure name. If so used, it refers to the whole space occupied by the array. If an input record contained the average weights for six age groups and two sexes, with all the age groups for one sex followed by all the age groups for the other sex, the record could be read as follows:

```
DCL TWO_DIM(2,6) FIXED DEC(3);
READ FILE(IN) INTO (TWO_DIM);
```

For input and output, it is important to consider the order in which the elements of a multi-dimensional array are stored in main storage, i.e. with the right hand subscript varying fastest. The order of the array elements must be reflected in the order of the fields in the input records.

You should now attempt questions 12 and 13 in the exercises at the end of this topic.

Arrays and Structures

If necessary, an array may be an element of a structure. If a record contained the name of a customer followed by the codes of 6 items which the customer has bought, it could be read into STRUC1, declared:

```
DCL 1 STRUC1,
      2 NAME          CHAR(30),
      2 ITEMS(6)      CHAR(10);
```

When a structure contains an array, it is called a **structure of arrays.** Any or all of the elements of a structure may be arrays. The elements of ITEM may be referred to in unqualified form - ITEM(I), or qualified form - STRUC1.ITEM(I), as required.

It is also possible to have **arrays of structures.**

An array of structures is a structure which has dimensions on the major structure name or on one or more minor structure names. The elements may also be arrays. The record which could be read by STRUC1, does not reflect the information which would normally be available in such a situation. Normally there would be at least a quantity associated with each item, and the record would be held as:

CUSTOMER	ITEM	QUANTITY	ITEM	QUANTITY

This information could be read into the structure:

```
DCL 1 STRUC2,
      2 NAME          CHAR(30),
      2 ITEMS(6),
        3 CODE        CHAR(10),
        3 NUMBER      PIC '999';
```

The minor structure ITEMS will be repeated six times, like the elements of an ordinary array. The structure will be organized in main storage with 30 bytes for the name, followed by 10 bytes for the first code, 3 bytes for the first number, 10 bytes for the second code, 3 bytes for the second number, and so on.

It is important to understand that the declaration of STRUC2 will cause a different organization from STRUC3, declared:

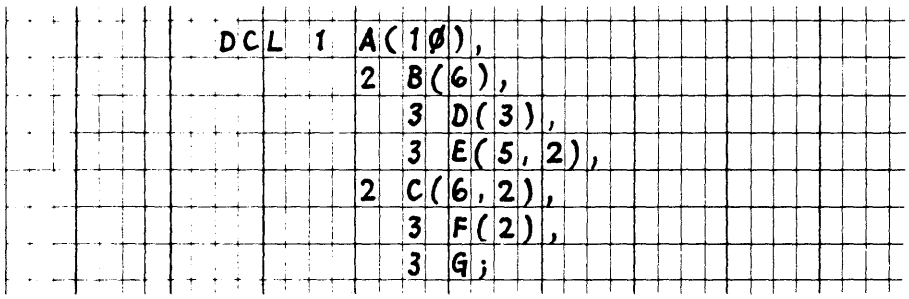
```
DCL 1 STRUC3,
    2 NAME          CHAR(30),
    2 ITEMS,
      3 CODE(6)     CHAR(10),
      3 NUMBER(6)   PIC '999';
```

STRUC3 will be organized in main storage with the 30 bytes of NAME, followed by 6 groups of 10 bytes each for the codes, followed by 7 groups of 3 bytes each for the numbers.

Considering STRUC2, the elements may be accessed by unqualified or qualified names. Although the dimension is on ITEMS, the subscript in the reference may be on any level of qualification. The following are all valid references to the same elements:

```
CODE(I)
STRUC2.ITEMS.CODE(I)
STRUC2.ITEMS(I).CODE
STRUC2(I).ITEMS.CODE
```

If a structure has subscripts at all levels:



references to each element must have the correct number of subscripts in the correct order, but, if qualified names are used, the subscripts may all be grouped together after any level of qualification, or they may be attached in groups at different levels. The correct number of subscripts in the correct order must be supplied. The following all refer to the same element of D:

```
D(8,6,2)
A.B.D(8,6,2)
A.B(8,6,2).D
A.B(8).D(6,2)
A(8).B(6).D(2)
A(8,6,2).B.D
```

They all refer to the second element of D within the eighth level of A and the sixth level of B.

The following all refer to the same element of G:

```
G(8,4,1)
A(8,4,1).G
A.C(8,4,1).G
A(8).C.G(4,1)
```

They all refer to the element G within the eighth level of A, the fourth level of the first dimension of C and the first level of the second dimension of C.

You should now attempt question 14 in the exercises at the end of this topic.

Initialization of Arrays

When an array is initialized, an initial value must be provided for each element that is to be initialized. Arrays are initialized in the order in which the elements are stored, with the right-most subscript varying fastest. If the number of initial values supplied is greater than the number of extents in the array, the excess values will be ignored; if the number of initial values supplied is less than the number of extents in the array, the remaining elements will not be initialized.

The initial values may be supplied as a list of constants, separated by commas as shown below:

```
DCL AR1(2,2) FIXED DEC INIT(1,2,3,4);
DCL AR2(2,2) FIXED DEC INIT(1,2);
DCL AR3(2,2) FIXED DEC INIT(1,2,2,3,4);
```

These declarations will cause the following initializations:

Element	Initial Value
AR1(1,1)	1
AR1(1,2)	2
AR1(2,1)	3
AR1(2,2)	4
AR2(1,1)	1
AR2(1,2)	2
AR2(2,1)	uninitialized
AR2(2,2)	uninitialized
AR3(1,1)	1
AR3(1,2)	2
AR3(2,1)	2
AR3(2,2)	3

The last initial value in the declaration of AR3 will not be used.

Where some elements of an array are not to be initialized, an * may be used at the appropriate point in the list of initial values. Thus

```
DCL AR4(2,2) FIXED DEC INIT(1,*,*,1);
```

will cause AR4(1,1) and AR4(2,2) to be initialized to 1, and AR4(1,2) and AR4(2,1) to be uninitialized.

Where many elements of an array are to be initialized to the same value, that value may be shown once with an iteration factor, as in the coding below:

```
DCL AR5(8,8) FIXED DEC INIT((8)Ø,(8)*,1,2,(46)Ø);
DCL AR6(3,3) CHAR(4) INIT((4)'A',(3)(4)'B',(5)(1)'ABCD');
```

The first 8 elements of AR5 will be initialized to 0, the next 8 will not be initialized, the next 2 will be initialized to 1 and 2 respectively, and the remaining 46 will be initialized to 0.

When a character string variable is initialized, the initial value may be expressed as a character string with a **repetition factor**, which is coded in a similar manner to an **iteration factor**. When only one factor precedes a character string constant, it is taken to be a repetition factor, so that the first element of AR6, above, will be initialized to 'AAAA'. When two factors precede the constant, the first is taken to be the iteration factor, and the second to be the repetition factor, so that the next three elements of AR6 will each be initialized to 'BBBB'. Where no repetition factor is required, a repetition factor of 1 must be supplied, so that the last five elements of AR6 will each be initialized to 'ABCD'.

If the lengths of constants which are being used to initialize arrays of character string variables do not match the lengths of the elements, padding or truncation will occur on an element by element basis, so that:

```
DCL AR7(1Ø) CHAR(4) INIT((1Ø)'ABCD');
```

will cause a character string of length 40 to be generated and assigned to the first element of AR7. It will be truncated to a length of 4 and AR7(1) will be initialized to 'ABCD'. The rest of the array will be uninitialized.

Initialization of Structures of Arrays

```
DCL 1 ST(1Ø),
      2 A FIXED DEC,
      2 B FIXED DEC;
```

If all of the elements of the above array of structures are to be initialized enough initial values must be specified. The declaration:

```
DCL 1 ST(1Ø),
      2 A FIXED DEC INIT(Ø),
      2 B FIXED DEC INIT((1Ø)Ø);
```

will cause only the first A to be initialized, but all B's will be initialized.

You should now answer questions 15 and 16 in the exercises at the end of this topic.

Exercises

1. Write statements to fulfill the following requirements:

   a) Add the absolute value of X to Y.

   (The absolute value is the value of X, ignoring the sign. If X is positive, it is X. If X is negative, it is -X).

   b) Assign 'HEADING' to TEXT if ITEM contains 1, otherwise blank out TEXT.

   c) If the setting of SWITCH is 'OFF', change it to ON.

2. Write statements to fulfill the following requirements:

   (a) If there is enough stock and the customer is within his credit limit, put 'DELIVER' into MSG, otherwise, put 'HOLD' into MSG. The current stock level is in STOCK, the order quantity is in ORDER, the customer's current level is in CREDIT, and his credit limit is in CRED__LIM.

   b) If an employee has not been late and his output is 100, add BONUS to PAY. No special action should be taken otherwise. The number of times late is held in TIMES__LATE and the output level in OUTPUT.

   c) Given information held in the same variables as in b) above, if the employee has either been late or has failed to reach 100 output, set ACTION to 'SACK', otherwise set ACTION to 'KEEP'.

3. Using nested IF statements, write statements to meet the following requirements:

   (a) Using the same variables as in question 2 (a), if the customer's credit is within the limit, set MSG to 'DELIVER' if there is adequate stock, otherwise set MSG to 'BACK ORDER'. If the credit is not within the limit, set MSG to 'NO CREDIT'.

   (b) If a person is male, they receive a pension if aged 65 or over. If a person is female, they receive a pension if aged 60 or over. A person's sex is indicated by 'M' or 'F' in SEX. Their age is held in AGE. MSG should be set to 'PENSION' or 'NO PENSION' to indicatethe position.

4. The following code should perform the following function:

   If a worker is hourly paid and has worked more than 38 hours, he should receive overtime pay.

   If a worker has worked less than 38 hours, he should have some pay deducted.

```
IF HRS > 38 THEN
      IF JOB = 'HOURLY' THEN PAY = PAY + (HRS - 38) * O_T_RATE;
      ELSE
IF HRS < 38 THEN PAY = PAY - (38 - HRS) * RATE;
```

   What alterations are needed to make this work?

5. Write code to fulfill the following requirements:

   a) When a customer orders goods, if there is sufficient stock to meet the order, and the new order will not take him over his credit limit, take the following action:

Put DELIVER into MSG.

Deduct the quantity of the new order from STOCK.

Add the value of the new order to his amount of credit.

If the conditions are not met, simply put 'HOLD' into MSG.

The current stock level is held in STOCK.

The customer's credit limit is held in CRED__LIM.

His current credit level is held in CREDIT.

The quantity ordered is in ORDER.

The value of the order is in VALUE.

b) FINISH will hold 'PAINT' if a room is to be painted. If it is, calculate the volume of paint and the cost of paint as:

```
QTY = (LENGTH + WIDTH) * HEIGHT * 2/COVER;
COST = QTY * PRICE;
```

c) Write code to sum the contents of A, X, G, I and D into SUM, using a DO group.

d) Modify your answer to (c) so as not to add in I if the sum of A, X and G is greater than 100.

6. The program NCARD reads and lists cards until a blank card is read:

```
NCARD: PROCEDURE OPTIONS(MAIN);
       DECLARE NAME_AND_ADDRESS CHARACTER(80);
       READ FILE(CARDSIN) INTO (NAME_AND_ADDRESS);
       DO WHILE(NAME_AND_ADDRESS ¬= ' ');
          WRITE FILE(PRINTOUT) FROM (NAME_AND_ADDRESS);
          READ FILE(CARDSIN) INTO(NAME_AND_ADDRESS);
       END;
END;
```

The data is to be changed. The first card will contain, in columns 1 to 3, the number of cards which follow. Write the executable statements needed to read the first card into the structure REC1, and use NUM to control the number of cards read.

```
DCL 1 REC1
      2 NUM PIC '999V'.
      2 PAD CHAR(76);
```

7.

a) Write code to sum the squares of all even integers up to 1000, but stop if the sum at any point exceeds 1,000,000.

b) Write code using the REPEAT option to sum the series:

```
1, 2, 4, 16, 256 ....
```

and stop when the sum exceeds 1,000,000.

8. What is the difference between:

```
DO I = 1 TO 10 WHILE(I¬=J);
.
.
.
END;
```

and

```
DO I = 1,2,3,4,5,6,7,8,9,10 WHILE(I¬=J);
.
.
.
END;
```

9.

a) Rewrite the solutions to exercise 3 using SELECT groups.

10.

a) Set SW, a bit string of length 1, such that:

```
IF SW THEN X = Y;
      ELSE X = Z;
```

will cause Z to be assigned to X.

b) SW1 and SW2 are two bit strings of length 1. Code an IF statement such that the THEN clause will be executed if SW1 is switched on and SW2 is switched off.

(c) The bit string HEALTH, of length 6, contains information concerning illnesses suffered by a patient. Code an IF statement which will cause the THEN clause to be executed if the patient has had any of of the illnesses.

(d) Code an IF statement to execute the THEN clause if the patient has had the third or fifth illnesses.

11.

```
DCL AR(6,2) FIXED DEC(3);
DCL MSGS(10) CHAR(40);
```

a) Write code to set all elements of AR to zero.

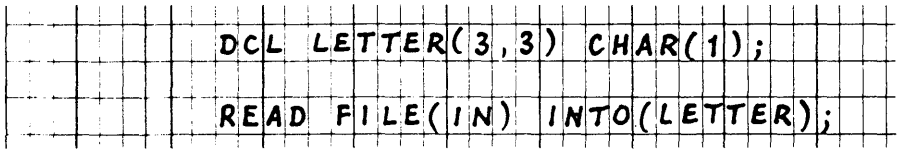
b) Write code to write the elements of MSGS to the file PRINT, stopping when an element is found which is blank.

12. What will be in AR1 after executing the following code?

```
DCL AR1(6,2);
DCL AR3(6,2);

AR1(3,2) =3;
AR3 =2;
AR1 =AR1(3,2)*AR3;
```

13.



If the record read contained:

```
ABCDEFGHI
What will be in LETTER(2,1)?
```

14. A record on a file will contain the maximum temperature, minimum temperature and rainfall for each month of a year. The information will be in the order:

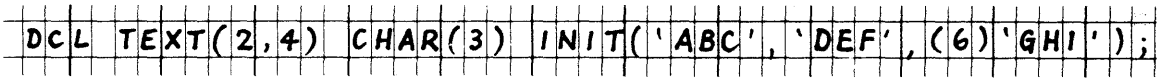
```
TEMP TEMP RAINFALL TEMP TEMP RAINFALL ...
```

Declare a suitable structure to hold this record, letting each temperature be called TEMP, and the rainfall RAINFALL. Call the whole structure WEATHER.

15. What will be the effects of the following initializations?

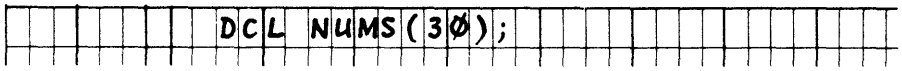
a)

b)



16. What is the easiest way to obtain the following initializations?

a)



The first 10 elements to be set to 1. The next 10 elements to have any value, and the last 10 elements to be set to 1.

b)

Fill the whole array with character zeroes.

Answers

1. a)

```
IF X < Ø THEN Y = Y - X;
         ELSE Y = Y + X;
/* OR */

IF X <= Ø THEN Y = Y - X;
          ELSE Y = Y + X;
/* OR */

IF X > Ø THEN  Y = Y + X;
       ELSE    Y = Y - X;
/* OR */

IF X >= Ø THEN Y = Y + X;
          ELSE Y = Y - X;
```

b)

```
IF ITEM = 1 THEN TEXT = 'HEADING';
            ELSE TEXT = ' ';
```

c)

```
IF SWITCH = 'OFF' THEN SWITCH = 'ON';
```

2. a)

```
IF STOCK >= ORDER & CREDIT < CRED_LIM THEN MSG = 'DELIVER';
                                      ELSE MSG = 'HOLD';
```

b)

```
IF TIMES_LATE = Ø & OUTPUT = 1ØØ THEN PAY = PAY + BONUS;
```

c)

```
IF TIMES_LATE > 0 || OUTPUT < 100 THEN ACTION = 'SACK';
                     ELSE ACTION = 'KEEP';
```

3. a)

```
IF CREDIT < CRED_LIM THEN IF STOCK >= ORDER THEN MSG = 'DELIVER';
                                              ELSE MSG = 'BACK ORDER';
                     ELSE MSG = 'NO CREDIT';
```

b)

```
IF SEX = 'M' THEN IF AGE >= 65 THEN MSG = 'PENSION';
                                ELSE MSG = 'NO PENSION';
              ELSE IF AGE >= 60 THEN MSG = 'PENSION';
                                ELSE MSG = 'NO PENSION';
```

This problem could be solved without using nested IF statements:

```
IF (SEX = 'M' && AGE >= 65) || (SEX = 'F' && AGE >= 60)
    THEN MSG = 'PENSION';
    ELSE MSG = 'NO PENSION';
```

4. The layout of the code is deceptive. The ELSE clause will be associated with the nearest preceding unmatched THEN: the one on the second IF. The logical construction may be corrected by the use of a null ELSE clause.

```
IF HRS>38 THEN IF JOB = 'HOURLY' THEN PAY=PAY+(HRS-38)*O_T_RATE;
                                 ELSE;
          ELSE IF HRS < 38 THEN PAY = PAY -(38-HRS)*RATE;
```

5. a)

```
IF STOCK >= ORDER & (CREDIT + VALUE) <= CRED_LIM THEN
                                    DO;
                                       MSG = 'DELIVER';
                                       STOCK = STOCK - ORDER;
                                       CREDIT = CREDIT + VALUE;
                                    END;
                                                        ELSE
                                    MSG = 'HOLD';
```

b)

```
IF FINISH = 'PAINT' THEN DO;
                            QTY = (LENGTH+WIDTH)*HEIGHT*2/COVER;
                            COST = QTY*PRICE;
                         END;
```

c)

```
SUM = 0.0;
DO S = A,X,G,I,D;
   SUM = SUM + S;
END;
```

The index variable, S, should be chosen to have suitable attributes, considering that the implied attributes of the items in the list vary. A safe course is for it to have the same attributes as SUM.

d)

```
SUM = 0.0;
DO S = A,X,G,I WHILE(SUM <= 100),D;
   SUM = SUM + S;
END;
```

6.

```
READ FILE(CARDSIN) INTO(REC1);
DO I = 1 TO NUM;
   READ FILE(CARDSIN) INTO(NAME_AND_ADDRESS);
   WRITE FILE(PRNTOUT) FROM(NAME_AND_ADDRESS);
END;
```

In the program in topic 2, the data was restricted because a card could not be processed which was all blank. This restriction is removed, but replaced by a limit of 999 cards, due to the specification for NUM.

7. a)

```
ISUM = 0;
DO I = 2 BY 2 TO 1000 WHILE(ISUM < 1000000);
   ISUM = ISUM + I **2;
END;

   /* OR */

ISUM = 0;
DO I = 2 BY 2 TO 1000 UNTIL(ISUM >= 1000000);
   ISUM = ISUM + I**2;
END;
```

b)

```
SUM = 0;
DO I = 1, 2 REPEAT(I*I) UNTIL(SUM >= 1000000);
   SUM = SUM + I;
END;
```

8. In the first case, every time I is set, it is tested against J.

   If it is the same, the whole group is abandoned.

   In the second case, I is only compared with J for the last setting of I, i.e., 10.

9. a)

```
SELECT;
   WHEN(CREDIT < CRED_LIM) SELECT;
                              WHEN(STOCK >= ORDER) MSG = 'DELIVER';
                              OTHERWISE MSG = 'BACK ORDER';
                           END;
   OTHERWISE MSG = 'NO CREDIT';
END;
```

b)

```
SELECT(SEX);
   WHEN('M') SELECT;
                WHEN(AGE >= 65) MSG = 'PENSION';
                OTHERWISE MSG = 'NO PENSION';
             END;
   WHEN('F') SELECT;
                WHEN(AGE >= 60) MSG = 'PENSION';
                OTHERWISE MSG = 'NO PENSION';
             END;
END;
```

Note that the OTHERWISE clauses could have been replaced by WHEN clauses.

10.

```
/* A */ SW = '0'B;

/* B */ IF SW1 & (¬SW2) THEN ...
                        ELSE ...

/* C */ IF HEALTH THEN ...
                  ELSE ...

/* D */ IF HEALTH & '00010'B THEN ...
                             ELSE ...
```

11. a)

```
DO I=1 TO 6;
   DO J=1 TO 2;
      AR(I,J) =Ø;
   END;
END;
```

The order of the DO groups could be reversed, as long as the first subscript varies from 1 to 6, and the second varies from 1 to 2.

b)

```
DO I=1 TO 1Ø WHILE(MSGS(I)¬=' ');
   WRITE FILE(PRINT) FROM (MSGS(I));
END;
```

When MSGS is tested, the value of I will be incremented before the test is made.

12.

```
AR1(1,1)=6      AR1(4,1)=12
AR1(1,2)=6      AR1(4,2)=12
AR1(2,1)=6      AR1(5,1)=12
AR1(2,2)=6      AR1(5,2)=12
AR1(3,1)=6      AR1(6,1)=12
AR1(3,2)=6      AR1(6,1)=12
```

13. D.

14.

```
DCL 1 WEATHER(12),
      2 TEMP(2),
      2 RAINFALL;
```

15. (a)

AR(1,1)=0	AR(2,1)=1
AR(1,2)=0	AR(2,2)=1
AR(1,3)=0	AR(2,3)=1
AR(1,4)uninitialized.	AR(2,4)=1

(b) TEXT(1,1)='ABC'
TEXT(1,2)='DEF'
TEXT(1,3)='GHI'

The rest of the array will be uninitialized. (6) 'GHI' will cause the 6 to be treated as a repetition factor and so create a string of length 18, which will be truncated to 3.

16. a)

```
DCL NUMS(30) INIT((30)1);
```

If it does not matter what value goes into the middle 10 elements, it is alright to put 1 into them.

```
DCL NUMS(30) INIT((10)1,(10)*,(10)1);
```

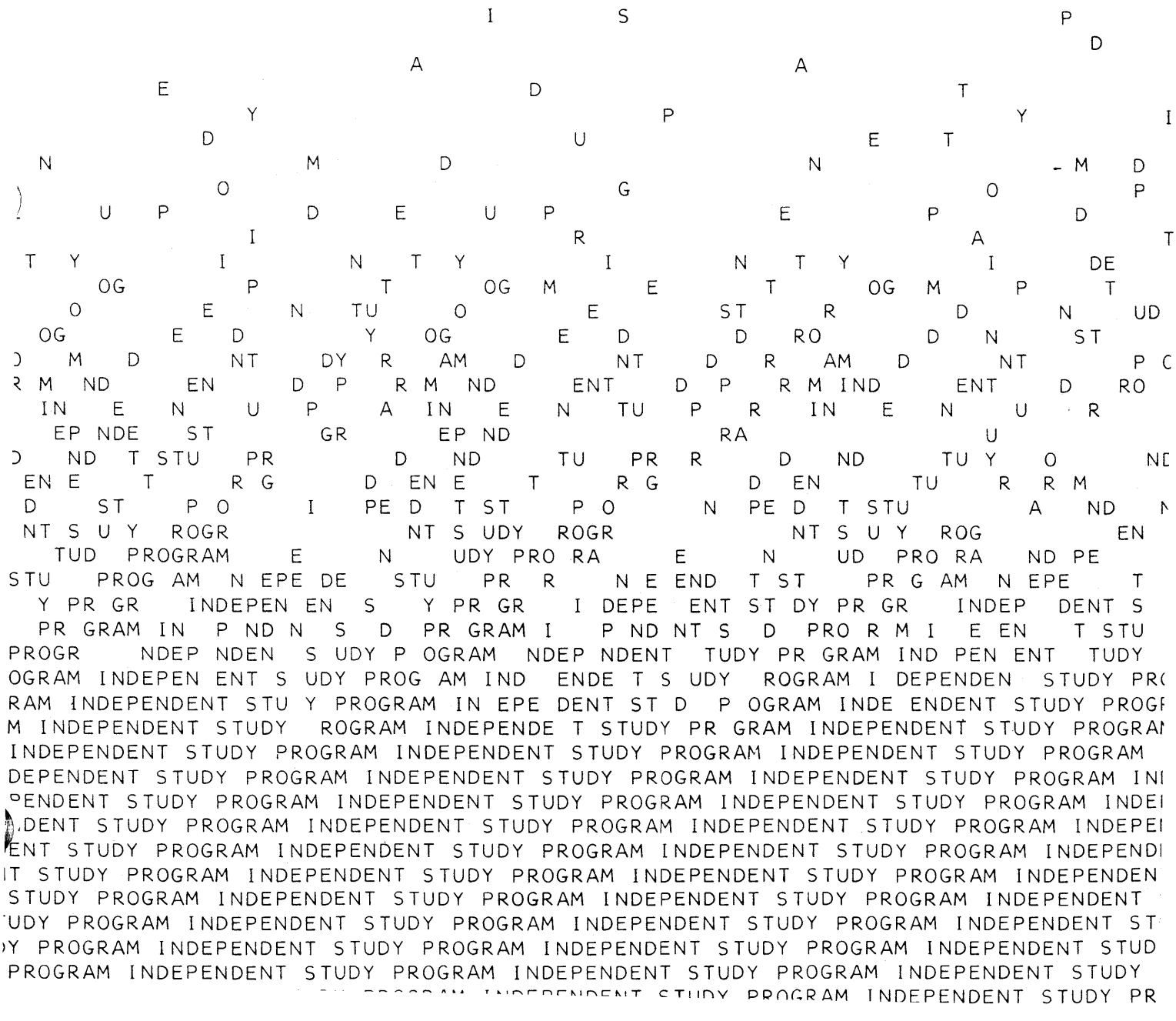
would also be correct.

b)

```
DCL CHARNUM(1000) CHAR(20) INIT((1000)(20)'0');
```

Topic

8



Topic 8

Record Input/Output Part 2 - LOCATE Mode

This topic covers locate mode record input/output. It also deals with pointer and based variables.

Objectives

On completion of the topic you should be able to:

- read, write and update records in consecutively organized files using LOCATE mode input/output
- write statements to declare POINTERs and BASED variables.

Introduction

Topic 5 looked at move mode record input/output in DOS/VS and OS/VS systems. In this topic we will look at the alternative form of record input/output - locate mode. Before the input/output process can be understood, two new types of variables have to be met: pointers and based variables. These will be explained in sufficient depth in order that you can understand locate mode processing.

MOVE Mode Review

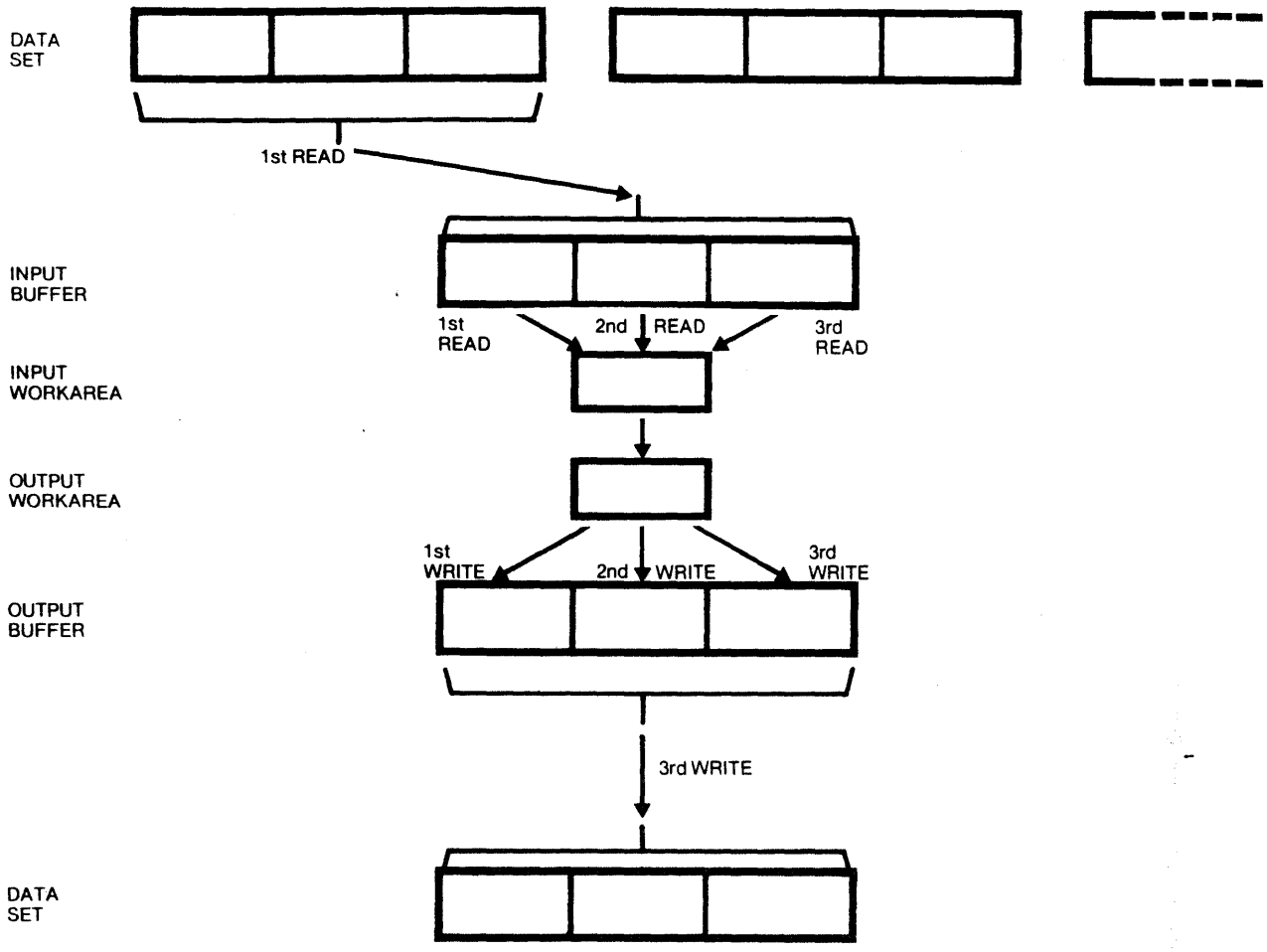
Before looking at locate mode input/output, let us review the processes of move mode file input/output using single buffers.

We will look at a program which reads records from an input file INFIL, into a structure INSTRUCT. For each logical record which it reads, it puts information into OUTSTRUCT, and writes this out as a logical record to OUTFIL.

The end of processing is indicated by the first three bytes of an input record containing 'END'.

The program and the process are illustrated below.

```
MOVE:  PROC OPTIONS (MAIN);
       DCL INFIL RECORD INPUT ENV( /* SUITABLE
                  ENVIRONMENT FOR DOS OR OS */ );
       DCL OUTFIL RECORD OUTPUT ENV( /* SUITABLE
                  ENVIRONMENT FOR DOS OR OS */ );
       DCL 1 INSTRUCT,      /* INPUT STRUCTURE */
             2 IDENT  CHAR(3),
             2 INDAT1 FIXED DEC(7),
             2 INDAT2 FIXED DEC(5);
       DCL 1 OUTSTRUCT,     /* OUTPUT STRUCTURE */
             2 OUTDAT1 FIXED DEC(3),
             2 OUTDAT2 FIXED DEC(7);
       READ FILE(INFIL) INTO(INSTRUCT);
       DO WHILE( IDENT ¬= 'END' );
          /* BUILD OUTSTRUCT FROM INSTRUCT */
          WRITE FILE(OUTFIL) FROM(OUTSTRUCT);
          READ FILE(INFIL) INTO(INSTRUCT);
       END;
       END /* MOVE */;
```



The first READ statement executed will cause the buffer associated with INFIL to be filled with the first physical record on INFIL. The first logical record will be copied from the buffer to INSTRUCT. In the processing of this record, information will be assigned to OUTDAT1 and OUTDAT2. Execution of the first WRITE statement will cause the first logical record of output to be copied to the buffer for OUTFIL. No physical records will be written to OUTFIL.

The processing of the next two logical records from INFIL will not cause any physical records to be read from INFIL, nor any physical records to be written to OUTFIL. Each READ or WRITE will cause data to be copied from the input buffer, or to the output buffer.

Reading the fourth logical record will cause the second physical record to be read from INFIL to the input buffer, and the first of the logical records in that physical record to be copied from the input buffer to INSTRUCT. Having processed the record, the fourth WRITE will cause the first physical record to be written to OUTFIL, and the contents of OUTSTRUCT will be copied to the first position in the output buffer.

At any stage in the execution of the program, as long as it is not in the middle of executing a READ or WRITE statement, one logical record from INFIL and one logical record from

OUTFIL will exist in main storage in two copies - one in a buffer and one in INSTRUCT or OUTSTRUCT.

If the input records can be accessed directly in the input buffer, and the output records can be built up directly in the output buffer, we will be able to save duplicating the records in main storage, and save the copying of data between buffers and structures where it may be worked on.

Locate mode record input/output allows this to be done.

A common problem in data processing is that a series of input records being read contains mixed record types, and as each record is read it is not known what type it will be. A typical situation is to have a record giving information about a customer - name, address, account number, discount rate etc., followed by a series of records, each detailing one purchase - item number, quantity, price etc. Before each record is read, it is not known whether it will be another transaction for the last customer, or the identification of the next customer. Only when the record has been read can some record type identification field be inspected.

Locate mode record input/output provides one of PL/I's solutions to this problem.

Declarations

Move Mode Declarations

The declaration of INSTRUCT in the first example will cause a total of 10 bytes to be reserved. In a READ statement which names INSTRUCT in the INTO option, 10 bytes will be copied from the input buffer to the 10 bytes of space reserved for INSTRUCT. A reference to IDENT will access the first three bytes of this information, interpreting them as CHARACTER data. Reference to INDAT1 will access the next four bytes of data, starting at the fourth byte of the structure, and interpret it as FIXED DECIMAL data. Reference to INDAT2 will access the next three bytes of data, starting at the eighth byte of the structure, and interpret it as FIXED DECIMAL data.

Locate Mode Declarations

The requirements of locate mode processing are that the declaration of a variable or structure should not cause space to be allocated to it, but when, in the case of INSTRUCT, the locate mode input statement is executed, reference to IDENT should cause a reference to the first three bytes of the next logical record; reference to INDAT1 should cause access to the next four bytes of that logical record, and reference to INDAT2 should cause access to the next three bytes of it.

To enable this to happen, the structure, or element variable, must be declared to be BASED on a pointer variable.

Pointer Variables and Based Variables

Pointer Variables

A pointer variable is a variable which may hold main storage addresses of variables. It may acquire the POINTER attribute explicitly, by declaration e.g.

```
DCL P POINTER;
```

or contextually, by appearing in the BASED attribute of the declaration of a variable or structure.

The keyword POINTER may be abbreviated to PTR.

A pointer variable does not hold any predictable address when the program starts executing. It may be set to an address by being referred to explicitly or implicitly in a locate mode input/output statement.

BASED Attribute

A variable is defined as being based by being declared with the BASED attribute.

```
DCL VAR FIXED DEC (5,2) BASED (PT);
```

The BASED attribute must include the name of a pointer variable in brackets. The use of PT in the declaration of VAR contextually declares it to be a pointer variable. The BASED attribute can come in any order with the other attributes, but where attributes include information in brackets, it must not come between the preceding keyword and those brackets.

The BASED attribute may be attached to an element variable name, an array name or a major structure name, but may not be attached to a minor structure name or a structure element name. A structure must occupy a continuous area of main storage. If the BASED attribute appeared on a minor structure or an element of a structure, it would imply that that part of it was to occupy a different area of main storage, which it cannot do.

The following are valid declarations of based variables:

```
DCL 1 ST BASED (PT),
      2 MS1,
        3 EL1 CHAR(3),
        3 EL2 FIXED(5),
        3 EL3 FIXED(5);
DCL AR (20,3) BASED (P) FIXED (7,2);
DCL ITEM FIXED BASED(PT) DEC(15);
```

Note that both ST and ITEM are based on PT. This is quite legal. It is the means of dealing with the situation described earlier, where a record being read could be any of several types.

LOCATE Mode Input/Output Statements

The locate mode input and output statements cause a pointer variable to be set to indicate the beginning of the next logical record in the buffer, so that variables based on that pointer may be used to access that record. The statements are:

READ FILE(filename) SET(pointer);

LOCATE based variable FILE(filename);

READ Statement

When the locate mode READ statement is executed, the specified pointer is set to point to the next logical record in the input buffer for that file. If there are no more logical records in that buffer, then the buffer is re-filled with the next physical record on the file, and the pointer is set to point to the beginning of the buffer.

Note that no variable is named in the statement. Reference to any variable which is based on the specified pointer will now cause reference to the area in the buffer following the address pointed to by the pointer. The area in the buffer can continue to be referred to until the pointer is set to some other address. No movement of data is caused by the locate mode READ statements, except the movement of physical records from the file to the buffer as necessary.

LOCATE Statement

When the LOCATE statement is executed, the pointer on which the based variable is based is set to point to the next logical record position in the output buffer for that file. As with move mode, the length of the variable and the record length for the file must be the same. If there is not room for any more logical records in the buffer, then the contents of the buffer are written out as the next physical record on the file, and the pointer is set to point to the beginning of the buffer.

Reference to any variable which is based on the same pointer as the named based variable will cause reference to the space in the buffer immediately following the address pointed to by the pointer. The output record may now be built. The physical record will be written out when a WRITE or LOCATE statement tries to add a logical record which cannot be put into that buffer.

It is most important to remember, when using locate mode output, that the results may not be assigned to the output structure until the LOCATE statement has been executed. If this is not done, then reference to the based variables before the execution of the first LOCATE statement will cause unpredictable, but almost certainly undesirable results, as the address pointed to by the pointer will be unpredictable.

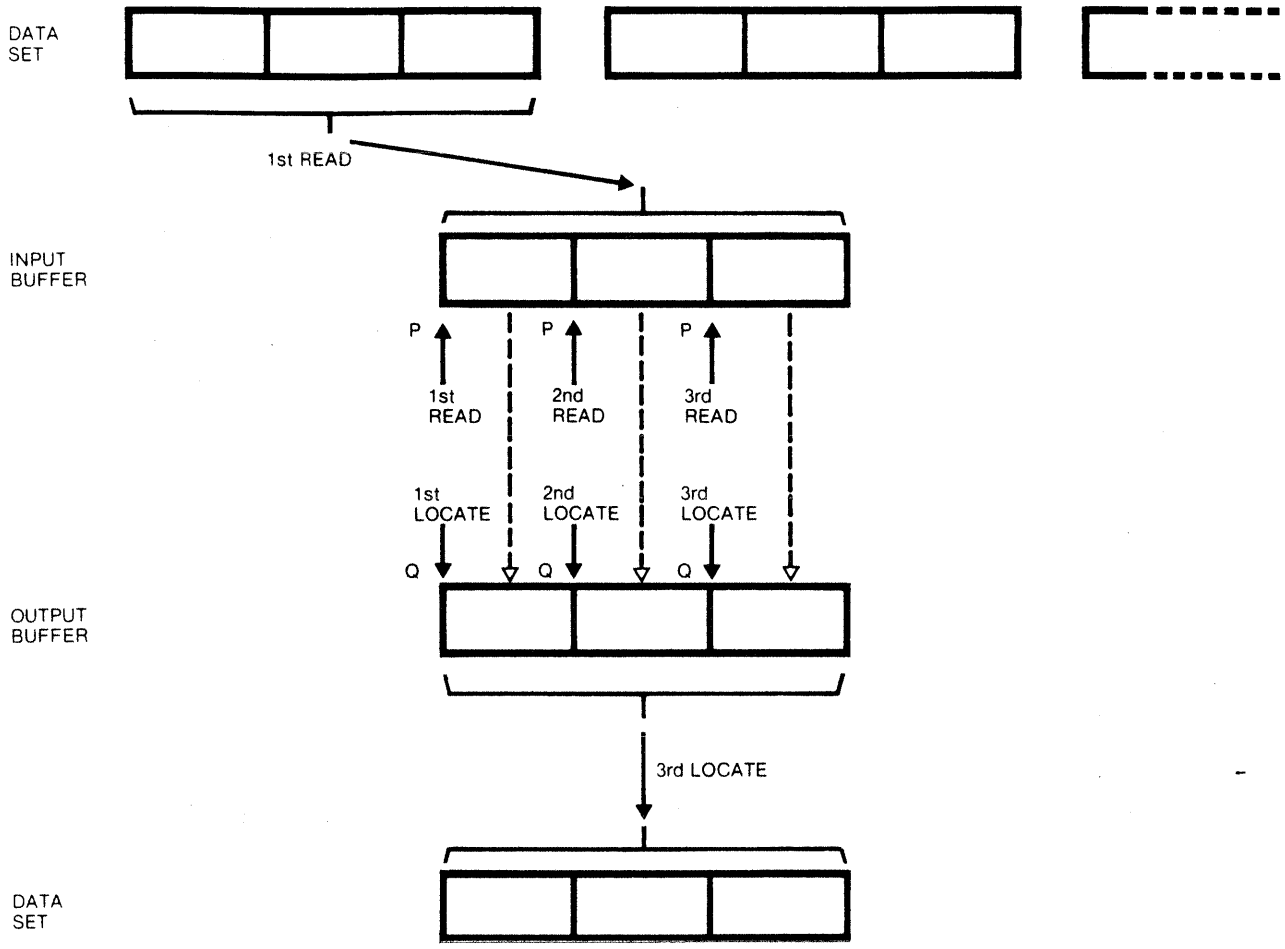
Remember:

LOCATE,
then assign.

The use of the LOCATE statement enables the output records to be built up directly in the output buffer. They do not have to be moved there after they have been built up.

The next coding example and its associated process illustrate the previous program in this topic rewritten for locate mode processing.

```
LCTE:  PROC OPTIONS (MAIN);
       DCL INFIL RECORD INPUT ENV( /* SUITABLE ENVIRONMENT
                                      FOR DOS OR OS */ );
       DCL OUTFIL RECORD OUTPUT ENV ( /* SUITABLE ENVIRONMENT
                                         FOR DOS OR OS */ );
       DCL 1 INSTRUCT BASED(P),          /* INPUT STRUCTURE */
             2 IDENT  CHAR(3),
             2 INDAT1 FIXED DEC(7),
             2 INDAT2 FIXED DEC(5);
       DCL 1 OUTSTRUCT BASED(Q),         /* OUTPUT STRUCTURE */
             2 OUTDAT1 FIXED DEC(3),
             2 OUTDAT2 FIXED DEC(7);
       READ FILE(INFIL) SET(P);
       DO WHILE (IDENT ¬= 'END' );
          LOCATE OUTSTRUCT FILE(OUTFIL);
          /* BUILD OUTSTRUCT FROM INSTRUCT */
          READ FILE(INFIL) SET(P);
       END;
       END;/* LCTE */
```


DATA SET
1st READ
INPUT BUFFER
P
1st READ
P
2nd READ
P
3rd READ
1st LOCATE
2nd LOCATE
3rd LOCATE
Q
Q
Q
OUTPUT BUFFER
3rd LOCATE
DATA SET

Pointer Usage

The use of P and Q in the declarations of INSTRUCT and OUTSTRUCT contextually declares them to be pointers. Execution of the first READ statement causes the first physical record to be read from INFIL into the buffer, and P to be set pointing to the beginning of it. Reference to IDENT in the DO statement will refer to the first three bytes of the input buffer. Assuming that they do not contain 'END', the LOCATE statement will cause the pointer Q to point to the first byte of the output buffer. Before this statement is executed, the address to which Q points is now known. It may even be an invalid address. When it has been executed, and Q points to the buffer, anything assigned to OUTDAT1 will be assigned to the first two bytes of the output buffer, and anything assigned to OUTDAT2 will be assigned to the next four bytes of the buffer.

Execution of the next READ statement will cause P to point to the second logical record in the input buffer, so that the next reference to IDENT will refer to the first three bytes of the second logical record. Similarly, the second execution of the LOCATE statement will cause Q to point to the second logical record position in the output buffer. References to OUTDAT1 and OUTDAT2 will refer to the first two and next four bytes of that record, so that the second output record can be built. The first output record cannot now be accessed.

A similar process will occur with the third READ and LOCATE statements.

When the fourth READ statement is executed, there will be no more logical records in the input buffer. The second physical record will be read in, and P will point to the first byte of the buffer, now containing the fourth logical record. When the fourth LOCATE statement is executed, there will be no more room in the output buffer, so the first physical record will be written out, and Q will point to the first byte of the buffer.

This process will continue until an input record is read with 'END' in the first three bytes. When the program terminates, the last physical record will be written out, containing as many logical records as have been LOCATEd in it, whether valid data was assigned to them or not.

In this program, the use of locate mode will have saved main storage for INSTRUCT nd OUTSTRUCT, 16 bytes, against which must be offset the 8 bytes needed for the two pointers. It will have saved little main storage. The principal saving will have been the execution time saved in not having had to copy each input logical record from the input buffer to INSTRUCT, and each output logical record from OUTSTRUCT to the output buffer.

Processing Files with Mixed Records

Advantages of Locate Mode

Processing files with a mixture of record types does not pose great problems in locate mode. The keys to processing are:

1) The locate mode READ statement does not name a variable, only a pointer.

2) Many variables may be based on the same pointer.

The following example shows some code to process records from a file containing customer identification records, identified by 'CUST' in the first four bytes, and transaction records, containing 'TRAN' in the first four bytes. The order in which they will be read is not known. The last record will be identified by 'END' in the first three bytes.

```
TWOREC:  PROC OPTIONS(MAIN);                                          /*  10 */
         DCL INFIL RECORD INPUT ENV( /* SUITABLE ENVIRONMENT
                                        FOR DOS OR OS */ );
         DCL CHK CHAR(4) BASED(P);                                    /*  40 */
         DCL 1 CUST_REC BASED(P),                                     /*  50 */
               2 IDENT          CHAR(4),                              /*  60 */
               2 CUSTOMER       CHAR(30),                             /*  70 */
               2 ACC_NUMBER     PIC '99999V',                         /*  80 */
               2 DISCOUNT       PIC '99V';                            /*  90 */
         DCL 1 TRANSACT_REC BASED(P),                                 /* 100 */
               2 IDENT  CHAR(4),                                      /* 110 */
               2 ITEM   PIC '99999V',                                 /* 120 */
               2 QTY    PIC 'ZZ9V'                                    /* 130 */
               2 PRICE  PIC '99V99';                                  /* 140 */
         READ FILE(INFIL) SET(P);                                     /* 150 */
      A: DO WHILE( CHK ¬= 'END' );                                    /* 160 */
            IF CUST_REC.IDENT = 'CUST' THEN                           /* 170 */
               DO;                                                    /* 180 */
               /* PROCESS CUSTOMER RECORD */                          /* 190 */
               END;                                                   /* 200 */
                                                ELSE                  /* 210 */
               IF TRANSACT_REC.IDENT = 'TRAN' THEN                    /* 220 */
                  DO;                                                 /* 230 */
                  /* PROCESS TRANSACTION RECORD */                    /* 240 */
                  END;                                                /* 250 */
                                                      ELSE            /* 260 */
                  DO;                                                 /* 270 */
                  /* PROCESS ERROR */                                 /* 280 */
                  END;                                                /* 290 */
            READ FILE(INFIL) SET(P);                                  /* 300 */
         END   /* OF A */;                                           /* 310 */
      END /* TWOREC */;                                              /* 320 */
```

CHK and the two structures, CUST__REC and TRANSACT__REC, are based on the same pointer, P. When a record is read, in statement 150 or 300, P will point to the beginning of it in the input buffer. The first four bytes of it may now be referred to as CHK, CUST__REC.IDENT or TRANSACT__REC.IDENT. They all refer to the same area of storage. Statement 160 refers to them as CHK, to check whether they contain 'END'. If they do, processing is ended. If not, statement 170 refers to them as CUST__REC.IDENT, to check whether they contain 'CUST'. If they do, the fields of the record are accessed and interpreted by the names of the elements of CUST__REC, for the record is a customer record. If not, statement 220 refers to them as TRANSACT__REC.IDENT, to check whether they contain 'TRAN'. If they do, the fields of the record are accessed and interpreted by the names of the elements of TRANSACT__REC, for the record is a transaction record. If the first four

bytes contain none of these things, it is processed as an error, before the next record is read at line 300.

There are some points to note about this type of program.

If the records were being read from punched cards, they would be 80 byte records. CHK has a length of 4 bytes, CUST_REC has a length of 41 bytes and TRANSACT_REC has a length of 16 bytes. As long as the structures are not longer than the records, that is quite alright. If the structures are all shorter than the record read, then the end of the record cannot be accessed, but presumably it will not contain useful information.

In this program, the customer record contains a discount rate for the customer. This information will be needed when processing the transactions for that customer. However, when the READ statement is executed which reads the first transaction for the customer, the pointer P, which was the means of locating DISCOUNT, will be moved to point to the first transaction record. Also, the customer record may be over-written, as it may have been the last record in the block. Thus when processing a file in locate mode, information in an input record is available only until the execution of the next READ for that file. Any information which will be needed after that must be assigned to a variable outside the buffer before the next READ is executed.

Similar considerations apply to locate mode output. Once a LOCATE statement has been executed, the contents of the previous record are unobtainable. If information from the previous record is required to build any record, then it must be duplicated in a variable outside the buffer.

File Declaration

It is not specified in a file declaration whether a file is to be processed in move mode or locate mode. A file may be processed in both move mode and locate mode by using both move mode and locate mode input/output statements. However, it is more efficient to use one mode only for any one file.

File Update

Sequentially processed files on DASD may have their records modified if they are declared with the UPDATE attribute.

In move mode, the READ statement causes the next record to be copied to a variable. When the REWRITE statement is executed, the contents of the variable named in the statement are copied back to the file in place of the record last read. The variable may be the same as that used in the READ statement, but need not be.

Updating in locate mode is done within the buffer. The statement used has the form:

```
REWRITE FILE(file name);
```

Example

```
REWRITE FILE (UPFIL);
```

No variables or pointers are specified. It causes the last record accessed to be re-written to the file. If any records on a file are being updated by locate mode statements, no input records for that file should be changed in the buffer, unless they are to be updated.

The unit of data transfer between files and main storage is the physical record. This applies in file updating as well as in reading and writing. When a locate mode REWRITE statement is executed, the whole block is marked as a block to be re-written. It is re-written to the file when the whole block has been processed, i.e. when the first logical record is read from the next physical record. Thus, if any other logical record in the block is changed, or if the record or records to be updated are changed after their REWRITE statement has been executed, the information written to the data set when the block is re-written will not be correct.

Locate mode input/output allows input records to be accessed directly in the buffers and output records to be built up directly in the buffers by means of based variables and pointers. This has implications in that the data can only be accessed while it remains in the buffers, and while the pointers point to the correct position for it. If this is remembered, it allows quicker program execution and reduced main storage requirements. It also facilitates the processing of files which have a mixture of record types, when the order of records is not know.

Before continuing with the next topic, you should complete the exercises at the end of this topic.

Exercises

1. What errors do the following statements contain?

```
/* A */  READ FILE(INFILE) SET(P);
/* B */  READ FILE(INFILE) INTO (VAR);
/* C */  READ FILE(INFILE) INTO (VAR) SET(P);
/* D */  LOCATE FILE(OUTFILE);
/* E */  LOCATE FILE(OUTFILE) OUTVAR;
/* F */  LOCATE OUTVAR FILE(OUTFILE);
/* G */  REWRITE FILE(UPFILE) SET(P);
```

2. Assuming INFILE and OUTFILE are suitably declared, are there any errors in the following section of code?

```
DCL INREC CHAR(80) BASED (P);
DCL OUTREC CHAR(80) BASED (Q);
READ FILE (INFILE) SET (P);
OUTREC= INREC;
LOCATE OUTREC FILE (OUTFILE);
```

3. Assuming that INFILE and OUTFILE are suitably declared, are the ny errors in the following section of code?

```
DCL INREC CHAR(80) BASED (P);
DCL OUTREC CHAR(80) BASED (P);
READ FILE (INFILE) SET(P);
LOCATE OUTREC FILE (OUTFILE);
OUTREC= INREC;
```

4. Assuming that INFILE and OUTFILE are suitably declared, what problems would the following code cause, and how could they be overcome?

```
DCL INREC1 PIC '9999'    BASED(P);
DCL INREC2 PIC '99999'   BASED(Q);
DCL OUTREC PIC '999999'  BASED(R);
READ FILE (INFILE) SET (P);
READ FILE (INFILE) SET (Q);
LOCATE OUTREC FILE (OUTFILE);
OUTREC= INREC1 + INREC2;
```

5. Using locate mode, write a program to read records from a personnel file until one is found with a personnel number of 53000. When it is read, change the MARITAL_STATUS field to 'M' and update the record on the file.

   Call the program WEDDING.

   The following declarations are appropriate for the program. Do not code them in your program, but put a comment to indicate where they would go.

```
/* FOR A DOS PROGRAM */
DCL PERSFIL RECORD UPDATE SEQL
    ENV(MEDIUM(SYS017,3330) FB
    RECSIZE(61) BLKSIZE(610));

/* FOR AN OS PROGRAM */
DCL PERSFIL RECORD UPDATE SEQL;
/* INPUT STRUCTURE FOR PERSONNEL RECORD */
DCL 1 PERS_REC BASED (P),
        2 NAME          CHAR(30),
        2 PERS_NUM      FIXED(5),
        2 DEPT          FIXED(3),
        2 JOB           CHAR(15),
        2 SALARY        FIXED(5),
        2 DATE_OF_BIRTH,
          3 (YEAR,MONTH,DAY) PIC '99',
        2 SEX           CHAR(1),
        2 MARITAL_STATUS CHAR(1);
```

Answers

1. a) No errors. This is a valid locate mode READ statement, assuming that INFILE is an input file and P is a pointer.

   b) No errors. This is a valid move mode READ statement.

   c) A READ statement may not have both the INTO and SET options.

   d) It does not identify a based variable.

   e) The name of the variable must be immediately after LOCATE.

   f) No errors.

   g) There should be no SET option. The correct form is:

2. If these are the only references to these files and pointers, Q will hold an undefined address when the assignment statement is executed.

   The LOCATE statement should be executed before the assignment statement.

3. Both records are based on the pointer P. When the READ statement is executed, P will point to the input buffer. The input record could be referred to now as INREC or OUTREC. They both refer to the same space. When the LOCATE statement is executed, P will point to the output buffer, and the record in the input buffer can no longer be accessed. The assignment statement will have no effect.

There are two simple solutions.

1) Change the name of the pointer for OUTREC.

2) Change the output to move mode.

```
DCL INREC CHAR(80) BASED(P);
READ FILE (INFILE) SET(P);
WRITE FILE (OUTFILE) FROM (INREC);
```

Using a based variable in the FROM option is permitted, as long as its pointer indicates a suitable address.

4. This problem is not well suited to locate mode processing. The assignment statement refers to the contents of two records from the same file. Although different pointers are used for INREC1 and INREC2, the first record may not still be available, once the second record has been identified by the second READ statement. There are two simple solutions:

1) Remove the BASED attributes from INREC1 and INREC2 and use move mode input.

2) Assign INREC1 to a non-based variable before the second READ statement is executed, and use this variable in the assignment statement.

5.

```
WEDDING: PROC OPTIONS(MAIN);
         /* THE DECLARATIONS COULD GO HERE, OR BEFORE
         ANY OTHER STATEMENT IN THE PROGRAM, AS LONG AS
         THEY ARE BEFORE THE FINAL END STATEMENT */

         READ FILE(PERSFIL) SET(P);
         /* IF THIS WAS NOT THE REQUIRED RECORD, KEEP
         READING UNTIL IT IS FOUND */
         DO WHILE (PERS_NUM ¬= 53000);
            READ FILE(PERSFIL) SET(P);
         END;
         /* HAVING FOUND IT, UPDATE IT */
         MARITAL_STATUS = 'M';
         REWRITE FILE(PERSFIL);
         END;
```

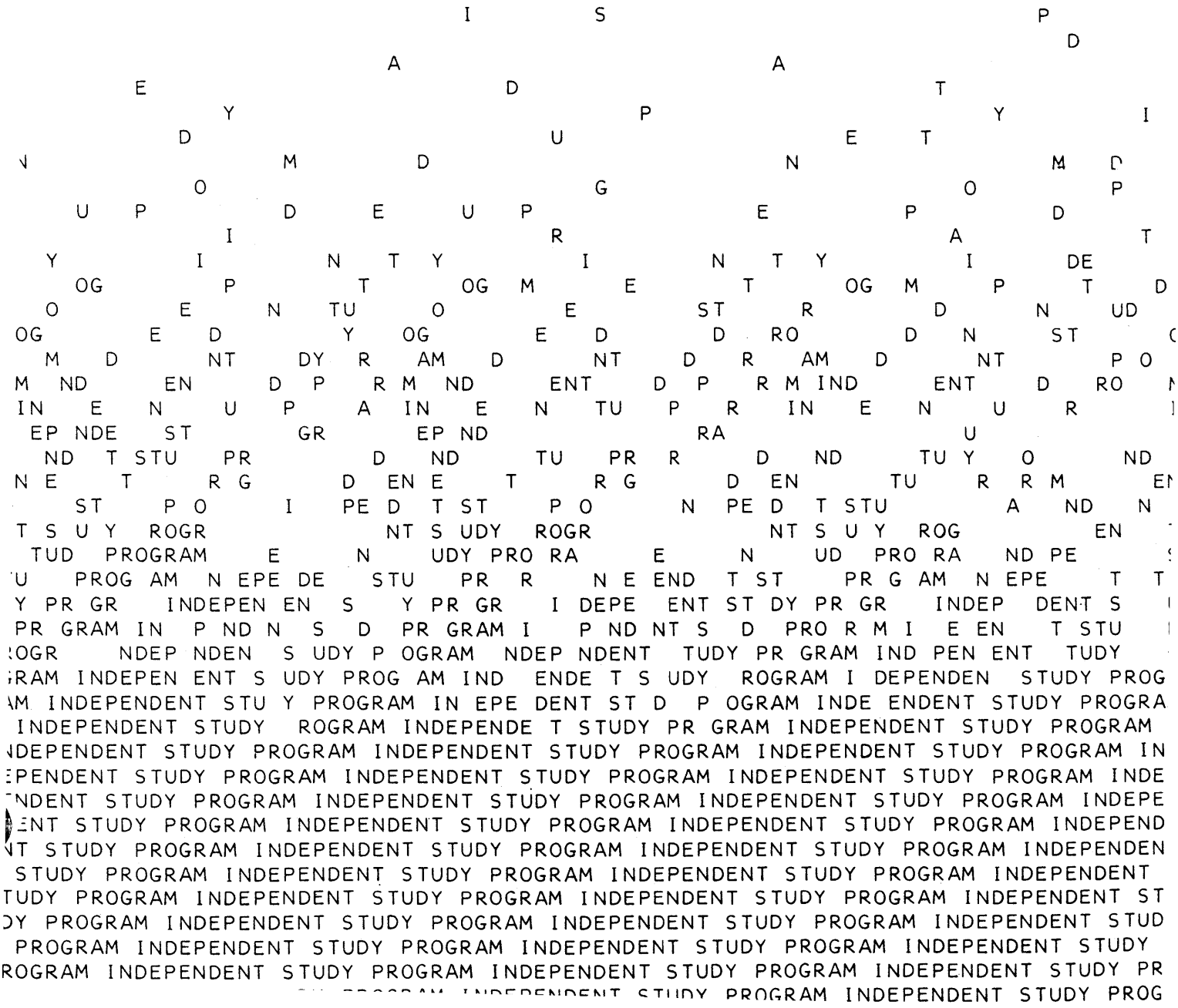
Note:

There are many ways to write this program, as there are for most programs.

This is one way.

It is generally regarded as bad practice to write a program which produces no printed output. It might be useful here to print out what the record has been changed to.

Topic **9**



Topic 9

Input and Output - Further Considerations

This topic will look generally at PL/I files and the type of input/output supported by PL/I. The need for OPEN and CLOSE statements and the action resulting from the execution of these statements will be discussed.

Objectives

On completion of this topic you should be able to:

- explain the differences between STREAM and RECORD I/O
- determine which file attributes are in force for a specified filename at any stage during the execution of a given program
- describe how PL/I files are associated with OS/VS or DOS/VS data sets
- use OPEN and CLOSE statements in appropriate situations
- code valid DISPLAY statements using REPLY and EVENT options
- state the situations in which the TRANSMIT and RECORD conditions will be raised.

Terminology

Confusion can arise as to the differences between a data set and a file. This is of more importance to the OS/VS programmer than the DOS/VS programmer. The latter tends to use the term 'file' to mean both file and data set. However, in this segment, the terms 'file' and 'data set' will be used in the context explained below.

Data Set

A data set is an actual collection of data on a physical device, such as on cards or on disk, completely independent of any program.

File

A file in PL/I is a symbolic representation of a data set which enables the programmer to deal with the logical aspects of the data, rather than with its physical organization. Except for the MEDIUM option in DOS/VS where the type of device must be specified, the programmer can write a PL/I program without specifically referring to any data set. Thus a file is nothing more than a declaration by the programmer of the way in which he intends to manipulate a data set.

Access Method

These are the routines which link the file in the program to the physical data set. Normally these are brought in from a library on execution of an I/O statement.

Data Set Organization

Data can be organized in several different ways depending upon the methods of retrieval required. Below is a table which summarizes the different organization. Alongside each one is the associated PL/I file and access methods.

OS/DOS NAME	PL/I NAME	ACCESS
SEQUENTIAL	CONSECUTIVE	Sequential
INDEXED SEQUENTIAL	INDEXED	Sequential or Direct
DIRECT	REGIONAL (1) REGIONAL (2) REGIONAL (3)	Sequential or Direct
VSAM key-sequenced VSAM entry-sequenced VSAM relative record	VSAM	Sequential or Direct

NOTE:

1. Regional(2) files only exist in OS/VS.
2. The type of PL/I file must be stated as an ENVIRONMENT option in the file declaration statement. If omitted, then CONSECUTIVE is assumed by default.

In later topics each organization will be discussed in full detail. The PL/I coding of the file declaration statements and I/O statements required will be introduced. In this topic we will not be referring to any specific organization.

Stream and Record I/O

Two types of data transmission can be used in PL/I: stream-oriented and record-oriented (hereafter referred to as Stream I/O and Record I/O, respectively).

In Stream I/O, the data set must have CONSECUTIVE organization (and therefore must be accessed sequentially), but the physical organization of the data is ignored within the program and it is treated as a continuous stream of characters. Data is converted from character form to a programmer defined internal form on input; the converse happens on output. For example on input, if the data is to be read into a fixed decimal field, it will be converted to fixed decimal format before being assigned to the field. If there were alphabetic characters in the data then an error would occur and the program would probably terminate. The details of Stream I/O will be discussed in Topic 15.

In Record I/O, the data set consists of discrete records which are transferred without any conversion. It is therefore necessary for the programmer to be fully aware of the way in which data is stored on the external medium.

Stream I/O is less efficient both in execution time and physical storage economy (e.g. disk space) - and it can only handle consecutively organized, sequentially accessed data sets. It is, however, easier to use, particularly for punched card input and printed output and can be very useful for testing purposes. It would not, however, be recommended for normal production programs.

A summary of the differences is given in the table below.

	STREAM	RECORD
Format of External Data	Character	Any
Statements	GET/PUT	READ/WRITE/LOCATE REWRITE/DELETE
Data Conversion	Yes	No
Access	Sequential	Sequential/Direct
Speed	Stream is slower than Record I/O because of the data conversion item by item	
File organization	CONSECUTIVE	CONSECUTIVE INDEXED REGIONAL VSAM

Opening PL/I Files

Since opening a file associates a file definition with a data set and prepares it for I/O, all PL/I files must be opened before they can be used. The various events which occur at OPEN time are as follows:

a) If it is an output file, a check is made to see if there is room for it.

b) If it is an input file being read from disk or tape, then label checking routines are preformed.

c) If it is an output file being written to disk or tape, then label writing is performed.

d) The PL/I file is associated with the physical data set.

e) The file attributes are merged.

The last two events will be discussed in this topic.

Associating PL/I Files With DOS/VS Data Sets

Unit Record Devices

A file in a PL/I program is associated with a physical data set on execution of the program. This link is set up at 'OPEN' time as follows:

In a file declaration one of the mandatory options of the environment is the MEDIUM option. The 'symbolic device name' SYSxxx in this option corresponds to an identical one in the JCL assignment statement.

```
DCL X FILE RECORD OUTPUT ENV(MEDIUM(SYSØØ7,1403)...);
```

In Job Control

```
// ASSGN SYSØØ7, X'ØØE'
```

Thus by means of the 'symbolic device name', the link has been set up between the file, known as X, in the program and the physical device with address X'00E'.

DASD

In this case the initial link is set up by the name of the file. This must correspond with an identical filename in a DLBL statement. The 'symbolic device name', is not taken from the MEDIUM option as above but from the EXTENT statement associated with the DLBL statement.

```
DCL X FILE ... ENV(MEDIUM(SYSØØ8,3330)...);
```

In Job Control

```
// DLBL X 'MASTER STOCK FILE'
// EXTENT SYSØ1Ø
// ASSGN SYSØ1Ø, X'15A'
```

In this example no use is made of the 'SYS008' in the MEDIUM option and hence its value is unimportant. However an entry must be present.

The 'symbolic device unit' can be omitted from the EXTENT statement. In this case, the value is taken from the MEDIUM option.

```
DCL X FILE ... ENV(MEDIUM(SYSØØ8,3330)...);
```

In Job Control

```
// DLBL X,'MASTER STOCK FILE'
// EXTENT
// ASSGN SYS008,X'15A'
```

Tape Devices

With tapes, as with DASD, the link is set by the filename and this must correspond with the filename in the TLBL statement. The 'symbolic device name' in the ASSGN statement must match that in the MEDIUM option.

```
DCL X FILE RECORD ... ENV(MEDIUM(SYS007,2400) ... );
```

In Job Control

```
// TLBL X,'MASTER STOCK FILE'
// ASSGN SYS007,X'182'
```

In all these cases, the other operand appearing in the MEDIUM option (2400 in the last example) must correspond with the physical device whose address is specified in the ASSGN statement.

TITLE Option

The TITLE option allows the programmer to associate a PL/I file with several different physical data sets.

```
DCL X FILE...ENV(MEDIUM(SYS007,3330)...);
OPEN FILE(X) TITLE('MASTER');
```

In Job Control

```
// DLBL MASTER,'INVENTORY FILE'
```

In this case the link is not made by the filename but by the name given in the TITLE option. Thus, the association of the file with different data sets is achieved by closing the file and reopening it with a different name in the TITLE option. The expression in the TITLE option can be a variable whose character string value at execution time is used.

Example

```
DCL 1 INREC,
      2 FIELD1...,
      2 FILE_ID CHAR(7);
DCL DETAIL FILE INPUT...;
DCL MASTER FILE INPUT...;

OPEN FILE(DETAIL);
READ FILE(DETAIL)INTO(INREC);
/* NAME OF FILE TO BE PROCESSED HAS
   NOW BEEN READ INTO 'FILE_ID'      */

OPEN FILE(MASTER)TITLE(FILE_ID);

/* CODE TO PROCESS MASTER FILE */

CLOSE FILE(MASTER);
```

In Job Control

```
// DLBL DETAIL,'...
// DLBL MASTER1,'MASTER1'...
// DLBL MASTER2,'MASTER2'...
// DLBL MASTER3,'MASTER3'...
```

Associating PL/I Files With OS/VS Data Sets

OS/VS programmers may omit this section and recommence with the section called ASSOCIATING PL/I FILES WITH OS/VS DATA SETS).

A file in a PL/I program is associated with a physical data set on execution of the program. This link is set at 'OPEN' time as follows:

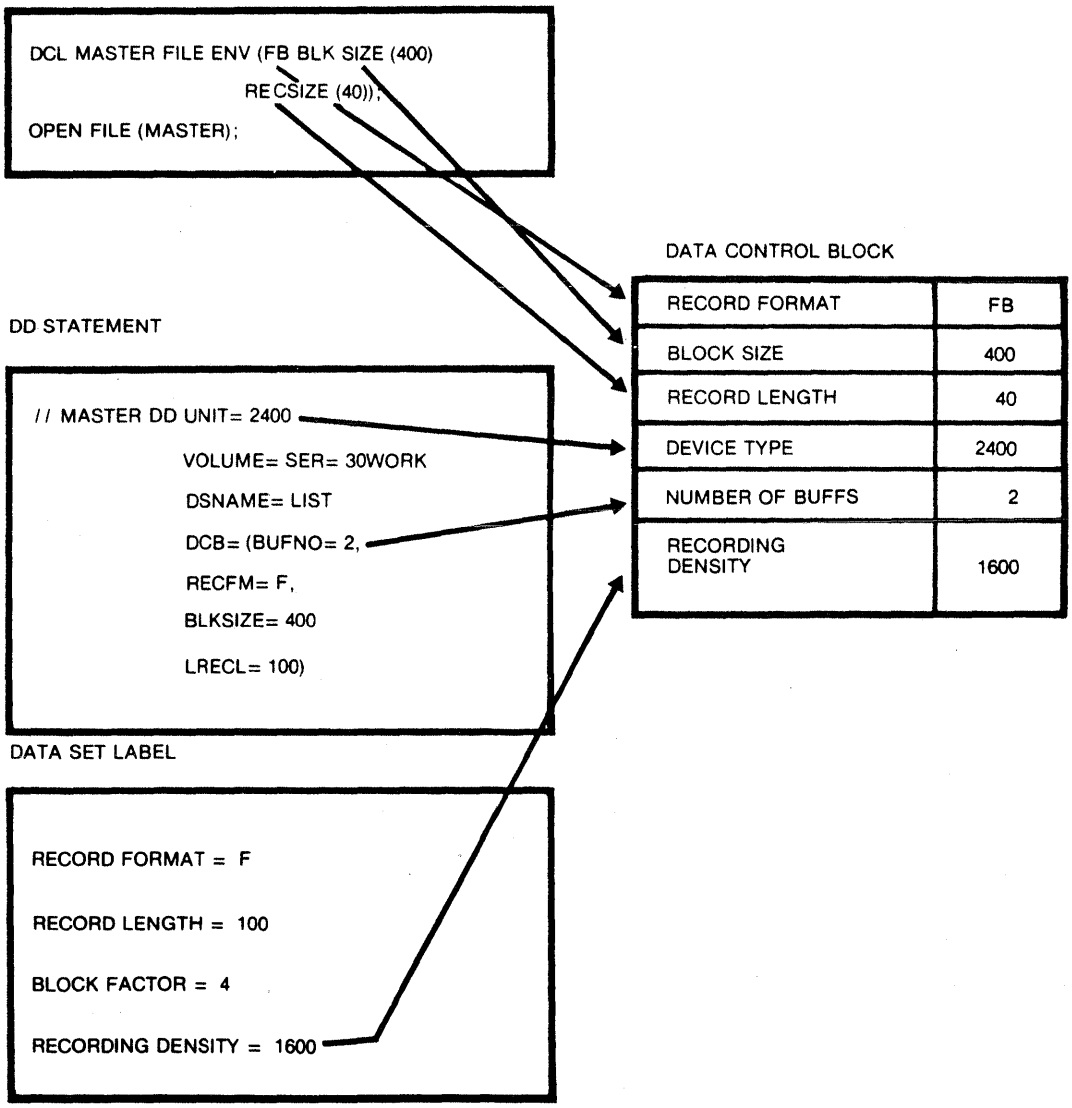
The filename in the file declaration statement is taken as the link and it must correspond with the name of a DD statement in the Job Control.

```
DCL X FILE RECORD OUTPUT;
```

In Job Control

```
//X DD...
```

At OPEN time PL/I library subroutines create a skeleton data control block and commence filling it in with attributes from the DECLARE and OPEN statements and with any that they imply. **The routines then examine the DD statement to see if there is any information there which is still required. Finally, if the data set exists, the routines will try and fill any gaps remaining with information from the Data Set Label (DSCB).**



TITLE Option

The TITLE option allows the programmer to associate a PL/I file with several different physical data sets.

```
DCL X FILE...;
OPEN FILE(X)TITLE('MASTER');
```

In Job Control

```
//MASTER DD ...
```

In this case the link is not made by the filename but by the name given in the TITLE option. Thus file X can be associated with several data sets by closing and then reopening with different names in the TITLE option. The expression can be a variable whose character string value at execution time is used. Example

```
DCL 1 INREC,
      2 FIELD1 ...
      2 FILE_ID CHAR(7);
DCL DETAIL FILE INPUT...;
DCL MASTER FILE INPUT...;

OPEN FILE(DETAIL);
READ FILE(DETAIL)INTO(INREC);
/* NAME OF FILE TO BE PROCESSED HAS
   NOW BEEN READ INTO 'FILE_ID'      */

OPEN FILE(MASTER)TITLE(FILE_ID);

/* CODE TO PROCESS MASTER FILE */

CLOSE FILE(MASTER);
```

In Job Control

```
//DETAIL DD DSN=...
//MASTER1 DD DSN=MASTER1,...
//MASTER2 DD DSN=MASTER2,...
//MASTER3 DD DSN=MASTER3,...
```

File Attributes

File attributes were introduced and the usage explained in Topic 5. We will now give a brief recap of some of the features already mentioned with the intended aim of explaining how, when a file is opened, the complete list of attributes for the file is built up. You should refer to the figure called 'file declarations' in SECTION I of the PL/I LANGUAGE REFERENCE MANUAL (for a complete list of attributes). Most of the attributes mentioned in this figure were discussed in Topic 5. Two new ones will now be introduced.

KEYED

KEYED implies that keys will be used when creating or accessing a file (see later topics on INDEXED, REGIONAL and VSAM files).

PRINT

The PRINT attribute specifies that the data of the file is ultimately to be printed and implies that the file is STREAM and OUTPUT.

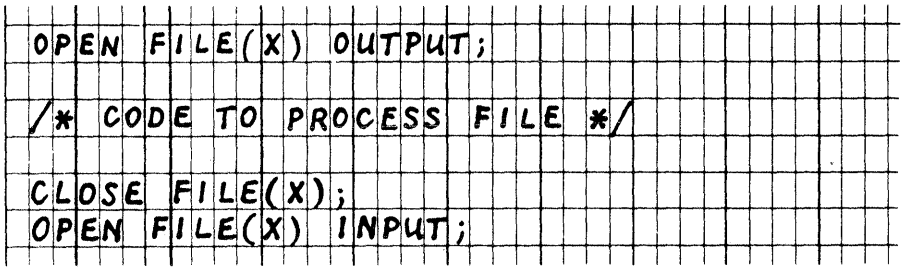
Explicit Attributes

These are the attributes which are mentioned by name in the file declaration statement.

Explicit File Open

It is possible to add further attributes for a file on the OPEN statement.

Example



In this example we are first of all opening file X as an output file and then later on in the program as an input file. In this case neither OUTPUT nor INPUT must be an explicit attribute otherwise there will be a contradiction of attributes for the file (see UNDEFINED FILE CONDITION later in the topic). In OS/VS it is possible to add any attribute except EXCLUSIVE, EXTERNAL or ENVIRONMENT. In DOS/VS, only INPUT or OUTPUT can be added and then only if the file is CONSECUTIVE and UNBUFFERED.

We have now met two situations when there is a need for an explicit OPEN statement.

1) When using the TITLE option.

2) When there is a requirement to add attributes at OPEN time.

There is only one other situation when there is this need and that is when the page size and line size of a STREAM output file needs changing (see Topic 15 on STREAM I/O).

Implicit File Open

If none of the above situations apply then the file can be OPENed implicitly i.e. when the first I/O statement referring to the file is executed. This method is as efficient as an explicit OPEN and also there is no requirement to code an OPEN statement.

In OS/VS only, an implicit OPEN deduces various attributes for the file according to the

following table:

	Statement Identifier	Attributes Deduced
	GET	STREAM, INPUT
	PUT	STREAM, OUTPUT
*	READ	RECORD, INPUT
*	WRITE	RECORD, OUTPUT
	LOCATE	RECORD, OUTPUT SEQUENTIAL, BUFFERED
	DELETE	RECORD, UPDATE
	REWRITE	RECORD, UPDATE

* READ and WRITE only imply INPUT and OUTPUT where UPDATE has not been specified.

Implicitly opening a file in OS/VS is equivalent to issuing an OPEN statement that specifies the deduced attributes.

Implied Attributes

So far we have seen two sources from which file attributes can be determined:

explicit declarations
explicit/implicit OPEN time attributes

From these attributes various others can be determined. For example, UPDATE implies RECORD, because a STREAM file cannot be updated. A full list of implied attributes is given in the table in the LANGUAGE REFERENCE MANUAL. There is no need to learn the table. Once you have understood it you will see that all the implications are perfectly logical.

Default Attributes

In Topic 5 you learned about alternative attributes i.e.

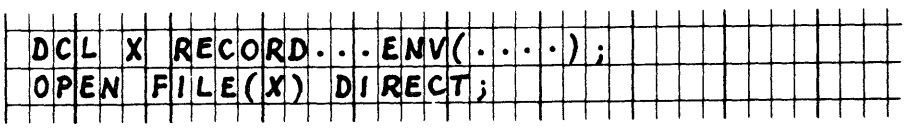
```
        STREAM/RECORD
        INPUT/OUTPUT/UPDATE
*       SEQUENTIAL/DIRECT
**      BUFFERED/UNBUFFERED
```

If in any group, no member is mentioned, then the default is taken (the underlined member). Hence a final source of attributes for a file is the default attributes if none of the alternatives have been specified.

* Since **all** STREAM files are sequential, the SEQUENTIAL attribute is only used for RECORD files.

** You only specify BUFFERED or UNBUFFERED for SEQUENTIAL RECORD or VSAM files. Other files are also buffered or unbuffered, but you have no control over their buffering.

Example 1 - OS/VS only



EXPLICIT ATTRIBUTES: RECORD
OPEN ATTRIBUTES: DIRECT
IMPLIED ATTRIBUTES: FILE KEYED (both implied by DIRECT)
DEFAULT ATTRIBUTES: INPUT

Hence the full attributes of the file are given in the statement.

```
DCL X FILE RECORD DIRECT KEYED INPUT ENV(....);
```

Example 2

```
DCL Y KEYED....ENV(....);
/* THE NEXT STATEMENT IS THE FIRST I/O STMT FOR Y */
READ FILE(Y) INTO(WORK);
```

OS/VS

EXPLICIT ATTRIBUTES: KEYED
OPEN ATTRIBUTES: RECORD, INPUT
IMPLIED ATTRIBUTES: FILE
DEFAULT ATTRIBUTES: SEQUENTIAL BUFFERED

DOS/VS

EXPLICIT ATTRIBUTES: KEYED
IMPLIED ATTRIBUTES: FILE RECORD
DEFAULT ATTRIBUTES: INPUT SEQUENTIAL BUFFERED

Hence in either case the full attributes of the file are given in the statement:

```
DCL Y FILE RECORD INPUT KEYED SEQUENTIAL BUFFERED ENV(....);
```

File Variables

One attribute which has not been mentioned yet is VARIABLE. If this attribute is used then all other attributes are excluded.

Identifiers which have been declared as files in the ordinary way (file constants) may then be assigned to the file variable at any stage during the program. This enables the files referenced

by I/O statements to be selected dynamically, according to the outcome of programmed tests. Example

```
DCL PERSON FILE VARIABLE;
DCL (ADULT,CHILD) FILE RECORD INPUT ENV(.....);

IF AGE > 18 THEN PERSON = ADULT;
            ELSE PERSON = CHILD;
READ FILE(PERSON) INTO (PROFILE);
```

UNDEFINED FILE Condition

If during the build up of the attributes of a file a contradiction is obtained, then what is known as the UNDEFINED FILE CONDITION is raised. In the topic called 'HANDLING EXCEPTIONAL CONDITIONS' you will learn how to code PL/I statements to handle such conditions. At present it is sufficient to know that the default course of action is for a message to be printed and the program to terminate.

Example 1

```
DCL X FILE OUTPUT ENV(...);
OPEN FILE(X) INPUT;
```

Example 2

```
DCL Y FILE DIRECT ENV(INDEXED ......);

/* FILE Y IS OPENED IMPLICITLY BY THE NEXT STMT */
LOCATE FIELD FILE(Y);
/* IMPLYING SEQUENTIAL */
```

Closing PL/I Files

When a file is closed it is disassociated from the data set. If the file is an output file, the remaining records in the buffer are also written out to the data set.

Implicit CLOSE

All files which are still open at the end of the job will be closed by PL/I routines before control is returned to the operating system. Hence normally there is no need for an explicit CLOSE.

Explicit CLOSE

The formal CLOSE statement is only required in three situations:

a) When the file needs re-opening using the TITLE option.

b) When the file needs re-opening with different attributes added.

c) When tape ENVIRONMENT options are required to control the action of a magnetic tape when it is closed. These options are:

LEAVE

LEAVE which prevents the tape from being rewound.

REREAD

REREAD (OS/VS only) which rewinds the tape to permit reprocessing of the data set (or volume if it is a multivolume data set).

UNLOAD

UNLOAD (DOS/VS only) which rewinds the tape and unloads it.

Example

```
CLOSE FILE(T_FILE) ENV(LEAVE);
```

Multiple File OPEN and FILE CLOSE

Opening or closing more than one file with a single statement can save execution time, even though it results in a temporary increase in the use of internal storage.

Example

```
OPEN FILE(A), FILE(B) TITLE('XYZ'), FILE(C);
CLOSE FILE(P), FILE(Q) ENV(LEAVE);
```

Thus if there is a need to explicitly open or close one file, any other files should be opened or closed at the same time, if appropriate.

Operator Communication

DISPLAY Statement

This is a STREAM output statement which displays character strings on the operator's console and hence is useful for operator communication. The general format of the statement is:

DISPLAY(element expression)[REPLY(character variable)

[EVENT(event variable)]];

REPLY Option

The REPLY option specifies a character variable into which any response from the operator will be assigned.

EVENT Option

The EVENT option permits asynchronous I/O (i.e., overlap of I/O and CPU processing). By specifying EVENT (event variable) in the DISPLAY statement, the output to the console will commence simultaneously with the execution of the subsequent instructions. An event variable can be any identifier, and is contextually declared by appearing after the keyword EVENT.

Example

```
DO WHILE(SW);
   DISPLAY('GIVE PASSWORD');
   REPLY(PASSWORD)EVENT(RESPONSE);
     /* OTHER PROCESSING */
   WAIT(RESPONSE);
   IF PASSWORD = CURRENT THEN DO;
                                 DISPLAY('PASSWORD ACCEPTED');
                                 SW = '0'B;
                                 END;
```

Notice how the DISPLAY instruction is issued well before the reply is needed, so that other processing can take place while the response is being keyed in.

WAIT Statement

Just before the response is required, the WAIT instruction is issued. The general format is:

WAIT(event);

where the event is the identifier in the corresponding EVENT option. At this point program execution will be suspended until the I/O operation is complete.

Notes:

1. The maximum length of the displayed message and the reply is 72 characters.
2. The DISPLAY statement can be coded without the REPLY option.
3. The EVENT option can only be used if the REPLY option has been used.

Further examples of the use of EVENT and WAIT will be met in the topics 'INDEXED ORGANIZATION' and 'REGIONAL ORGANIZATION'.

Exceptional Conditions

Here we will discuss two possible reasons why an INPUT/OUTPUT operation may fail. Full details of these and other reasons can be found in Topic 19, 'HANDLING EXCEPTIONAL CONDITIONS'. At this stage it is sufficient to know that the resulting action is to print a message (which includes information as to the location of the error) and to terminate the program.

RECORD Condition

We will only consider here fixed length records. Variable length records will be dealt with later. The condition can be raised in either a READ INTO, WRITE or LOCATE statement. It is caused by the record length specified for the file being different to the size of the variable in the I/O statement.

Example

```
DCL Y FILE RECORD OUTPUT ENV(F RECSIZE(80));
DCL X FILE RECORD INPUT  ENV(F RECSIZE(80));
DCL A CHAR(81);
DCL B CHAR(79) BASED(P);
READ FILE(X) INTO(A);
LOCATE B FILE(Y);
```

Both the READ and LOCATE statements would cause RECORD condition to be raised since A and B are not 80 bytes long (the record size of the files).

Most input and output is to and from structures. Hence it is important that the sizes of the structures are known and checked to ensure that they are the same as the record size of the file.

TRANSMIT Condition

This is raised when there is a permanent transmission error such as a hardware error. It signifies that any data transmitted is potentially incorrect. One of the causes could be an incorrectly specified block size.

Summary

A PL/I file is a logical description of a data set, and it expresses the way in which the data is to be accessed, together with aspects of its physical organization. This is achieved by means of attributes, which may be either explicitly specified, deduced from I/O statements, implied by other attributes or, finally, generated by default.

A file, however, is of no use until it has been associated with a data set (where the data physically resides) and although certain aspects of this data set may be specified in the ENVIRONMENT attribute, the actual link between program and data set is created by the operating system, via the JCL DD or DLBL statement.

In the exercises which follow there are some questions on LOCATE/MOVE mode processing. This has not been covered in this topic but it is essential at this stage that you understand the differences between the two modes. If you are at all unhappy with the differences, you are advised to refer back to Topics 5 and 8.

Exercises

1. Write an OPEN statement to link the following file declaration and the JCL statement appropriate to your operating system.

```
DCL X FILE ....ENV(....);
.
.
.
// DLBL Y ....
```

```
DCL X FILE;
.
.
.
//Y DD ...
```

2. What will be the complete set of attributes for the files A and B after they have been opened?

```
DCL A DIRECT...ENV(....);
OPEN FILE(A);
GET FILE(B);
```

3. When is it necessary to code an explicit OPEN statement?

4. (OS/VS only) What is wrong with the following coding and what will be the result of it?

```
DCL C FILE RECORD;
OPEN FILE(C) PRINT;
```

5. What is wrong with the following coding?

```
DCL D FILE VARIABLE INPUT;
```

6. Compare the examples in this topic under the headings TITLE OPTION and FILE VARIABLES. Suggest advantages/disadvantages of using either method for processing more than one data set.

7. What is the major difference between MOVE and LOCATE mode processing and what overheads are involved in each method.

8. When accessing a data set DIRECTly which mode must you be in?

9. Rewrite the following coding in LOCATE mode.

```
DCL (X,Y) FILE ...;
DCL (A1, A2) PIC '999';
READ FILE(X) INTO(A1);
A2 = A1 ** 2;
WRITE FILE(Y) FROM(A2);
END;
```

Answers

1.

```
OPEN FILE(X) TITLE('Y');
```

2. a) FILE DIRECT RECORD INPUT KEYED ENV(CONSECUTIVE);

   b) FILE STREAM INPUT ENV(CONSECUTIVE);

3. a) When using the TITLE option

   b) When adding OPEN time attributes

   c) When altering the page size or line size of STREAM files.

4. The merged attributes include PRINT, which implies STREAM. This is in confilct with the explicit declaration of RECORD. The UNDEFINED FILE condition will be raised when the OPEN statement is executed.

5. No other attributes can be specified if VARIABLE is specified.

6. TITLE is more flexible in that the names of files and even the number of files can be specified at execution time. However, the file must be closed and then re-opened after each data set has been processed. With FILE VARIABLES, if the names or number of files needs altering, then it will become necessary to alter the program and re-compile it. However, the advantage of FILE VARIABLES over the TITLE OPTION is that all files can be opened together and then closed together, this is faster in execution speed.

7. In MOVE mode records are moved from the input buffer to the work area before being processed and then moved to the output buffer before being written. In LOCATE mode records are processed in the input buffer and created in the output buffer.

   In MOVE mode there is the overhead of moving records from one part of storage to another. In LOCATE mode the value of a pointer has to be saved. The latter is normally more efficient.

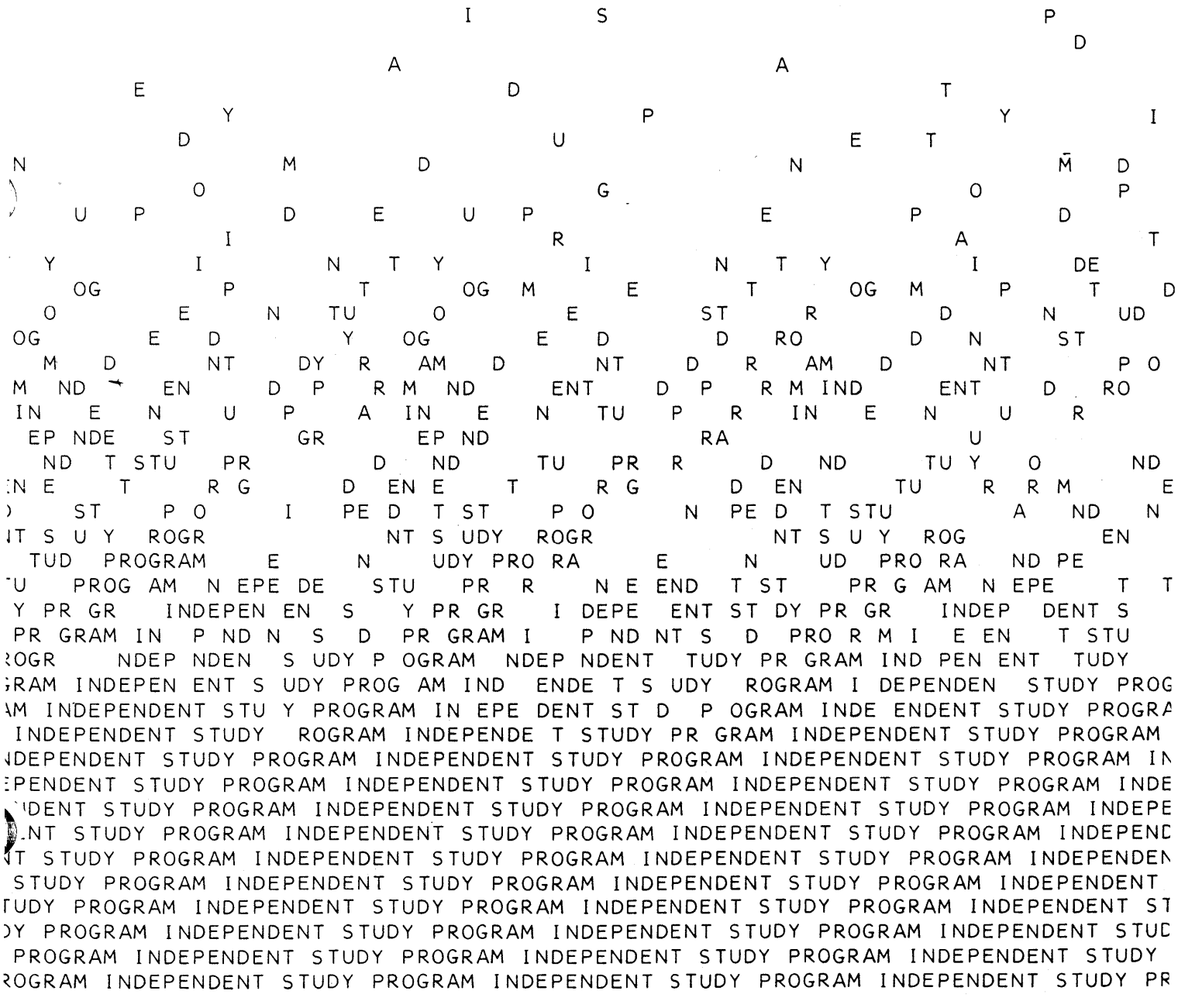
8. MOVE mode. LOCATE implies SEQUENTIAL processing (see section headed 'IMPLICIT FILE OPEN'). Note that an exception to this rule is with VSAM files. Then LOCATE mode can be used for DIRECT processing (see topic called VSAM ORGANIZATION).

9.

```
DCL (X,Y) FILE ...;
DCL A1 PIC '999' BASED(P);
DCL A2 PIC '999' BASED(Q);
READ FILE(X) SET(P);
LOCATE A2 FILE(Y);
A2 = A1 ** 2;
END;
```

Topic

10



Topic 10

Consecutively Organized Data Sets

In this topic you will learn about the processing of consecutively organized data sets.

Objectives

By the end of the topic you should be able to:

- state the record formats supported for CONSECUTIVE data sets
- code statements to create, read and update CONSECUTIVE data sets
- code statements illustrating the correct use of CTLASA control characters
- describe the use of the IGNORE option for CONSECUTIVE data sets.

Review

In data sets with the CONSECUTIVE organization, records can only be added or retrieved in strict physical sequence. When the CONSECUTIVE data set is created, each new record occupies the next vacant space on the output medium.

The sequence of records on a CONSECUTIVE data set is, therefore, identical with the order of the original output. You should appreciate that only **sequential** access may be used with such a data set, since there is no way of accessing any record **directly**. The file which is associated with a CONSECUTIVE data set must therefore have the attribute SEQUENTIAL.

If there is a requirement to insert or delete records then the data set must be re-created and the modifications made at the relevant positions in the new data set. It is possible, however, to update records which already exist on the data set.

Record Formats Supported

All PL/I formats are supported for CONSECUTIVE files:

Format	Details
F	Fixed length, unblocked
FB	Fixed length, blocked
V	Variable length, unblocked
VB	Variable length, blocked
U	Undefined

(An undefined format record is a variable length record without the standard variable control bytes i.e. the BDW and RDW, see topic 14, 'VARIABLE LENGTH RECORDS' for further details).

In other words, whether or not the data set consists of blocked or unblocked records with variable or fixed lengths, it is possible for it to be processed by a PL/I program.

Accessing Sequential Data Sets

There is a figure in the PL/I Language Reference Manual in the chapter called 'RECORD ORIENTATED TRANSMISSION' which lists all the required file declarations and input/output statements for processing SEQUENTIAL data sets. Locate this table now and study the various possibilities. Note that LOCATE or MOVE mode can be used when the BUFFERED attribute is specified, otherwise MOVE mode must be used.

Update

The UPDATE attribute specifies that a RECORD file is to be used both for input and for output. This attribute is required when a record is to be read, updated and then written back into the data set.

Records may be READ as for an input file, but the statement REWRITE may be used to rewrite the latest record read.

Example

```
READ FILE(OLD) INTO(OLDFIELD);
/* CODING TO ALTER DATA IN OLDFIELD */
REWRITE FILE(OLD) FROM(OLDFIELD);
/* OR IN LOCATE MODE */
READ FILE(OLD) SET(POINTVAR);
/* CODING TO ASSIGN NEW DATA TO THE INPUT BUFFER */
REWRITE FILE(OLD);
```

Notice the second REWRITE statement has no FROM option. This may be used after READ.. in the **input buffer.** The REWRITE then causes immediate transfer of the updated record to the output buffer.

Although READ and REWRITE statements may alternate, as shown, this need not be so. Several READ statements may follow each other, but with CONSECUTIVE files a REWRITE must always be preceded by a READ statement. It is always the last record read which is updated.

IGNORE Option

The general format is

READ FILE(file-expression) IGNORE(n);

This option is used with SEQUENTIAL, INPUT or UPDATE files to skip records. The expression n is converted to an integer, if necessary, and that number of records is skipped and will not be processed.

READ specified without an INTO or SET option may be used in place of IGNORE(1).

Example 1

```
READ FILE(F);
READ FILE(F) IGNORE(1);
```

The above are equivalent, each causing a single record to be skipped.

Example 2

Suppose that a variable block consists of two types of records, the first record in each block being of the following format:

FIELD NAME	BYTES	FORMAT
PART_NO	1 - 6	CHARACTER
NO_OF_DEPTS	7 - 8	PICTURE

and the rest of the block consisting of records, one for each department, of the following format:

FIELD NAME	BYTES	FORMAT
DEPT_NAME	1 - 10	CHARACTER
NO_IN_STOCK	11 - 15	PICTURE

Then the following coding will check the first record to see if the part is 'current', if not then the whole block will be skipped and the first record of the next block read in:

```
DCL 1 PART,
      2 PART_NO CHAR(6),
      2 NO_OF_DEPTS PIC'99';
DO WHILE(SW);
   READ FILE(STOCK) INTO(PART);
   IF  /* CODING TO CHECK IF PART IS OBSOLETE */
       THEN READ FILE(STOCK) IGNORE(NO_OF_DEPTS);
       ELSE DO;
                /* CODE TO PROCESS REST OF BLOCK */
            END;
END;
```

When the first record of the block has been read in, the 'PART_NO' is checked and if necessary the rest of the block is ignored. Note that the READ-IGNORE statement does not in fact read any records, it only skips records. The next READ-INTO statement will read the first record of the following block.

EVENT Option

The EVENT option was introduced in Topic 9, 'INPUT AND OUTPUT - FURTHER CONSIDERATIONS' in association with the DISPLAY statement. Further uses will now be introduced.

If a file is buffered then the number of buffers required for that file can be stated as an ENVIRONMENT option. If there is only one buffer then there is a delay in processing the data while a block of records is being moved from external storage to that buffer. This delay can be overcome by having two buffers. While data is being processed from one buffer, a block of records can be moved into the other. When the former buffer has been processed, another block of records can be moved into it while the later buffer is processed. This overlap of processing and movement of data from external storage applies similarly in the output process. The EVENT option can be used with unbuffered CONSECUTIVE files to simulate this overlap and so increase the speed of processing. This can be achieved as follows:

```
DO WHILE(SW);
   READ FILE(X) INTO(A);
   READ FILE(X) INTO(B) EVENT(REC_B);

   /* PROCESS A */

   WAIT(REC_B);

   /* PROCESS B */

END;
```

Within the loop, the record in A can be processed while the next record is being read into B, the latter being named as EVENT (REC__B). When this event is complete the record in B will be processed. Thus there is some overlap: the processing of record A and the reading in of record B.

Control of Printer Spacing

One particular device which allows only **sequential** output processing is the printer. A technique will be presented now which gives the programmer control over the spacing of output print lines.

CTLASA Control Characters

One of the options of the ENVIRONMENT is CTLASA. This indicates that the first byte of the output record is to act as a printer control byte only and not to be printed. To accommodate this byte, the RECSIZE option of the ENVIRONMENT should be increased by one. Note that in OS/VS it is more usual to specify the CTLASA and RECSIZE options as parameters of the DD JCL statement. In the PL/I Language Reference Manual, in the chapter called 'RECORD ORIENTATED TRANSMISSION' there is a table which lists out the possible CTLASA values. Locate this table now. As can be seen these are actions which will be taken **before** printing occurs. In the same table there are listed CTL360 code values. These are specified and used in a similar way to CTLASA codes but the action takes place either **after** or **without** printing. We will consider only CTLASA control characters in this topic.

The more commonly used values are '+', '$\overline{b}$', '0', '-', '1'.

The '+' could be used, say, if there was a requirement to return to the beginning of the current line with the intention of underlining that line. The '1' is a skip to channel 1 which is normally the start of a new page. You should ensure that the correct control character is in the first byte of an output structure before issuing WRITE or after issuing LOCATE.

Example 1

```
DCL 1 HEAD,
      2 CTLHEAD CHAR(1),
      2 H1 CHAR(40),
      2 H2 CHAR(30),
      2 H3 CHAR(62);

DCL 1 DETAIL DEFINED HEAD,
      2 CTLDETAIL CHAR(1),
      2 D1 CHAR(30),
      2 D2 CHAR(50),
      2 D3 CHAR(52);

DCL X FILE RECORD OUTPUT ENV(RECSIZE(133) CTLASA F);

CTLHEAD = '1';
H1      = '';
H2      = 'HEADING';
H3      = '';
WRITE FILE(X) FROM(HEAD);
CTLDETAIL = '_';
D1        = '';
D2        = 'FIRST';
D3        = '';
WRITE FILE(X) FROM(DETAIL);
CTLDETAIL = '';
D2        = 'SECOND';
WRITE FILE(X) FROM(DETAIL);
```

On execution of the above program, 'HEADING' will be printed at the top of a new page, two lines will be left blank (space 3) before 'FIRST' is printed and then 'SECOND' will be printed on the following line.

Example 2

```
IF LINE_CT = 50 THEN DO;
                     CTLHEAD = '1';
                     H1      = ' ';
                     H2      = 'HEADING';
                     H3      = ' ';
                     WRITE FILE(X) FROM(HEAD);
                     LINE_CT = 0;
                   END;
                 ELSE DO;
                     CTLDETAIL = ' ';
                     D1        = ' ';
                     D2        = 'FIRST';
                     D3        = ' ';
                     WRITE FILE(X) FROM(DETAIL);
                     LINE_CT = LINE_CT + 1;
                   END;
```

This example is a section of coding which maintains a line count. When the line count reaches 50, a new page is started.

Control of Source Statement Listing

As well as controlling the spacing of the output from a program it is also possible to control the source statement listing (the listing of the coded program). Normally when coding in PL/I, you code between columns 2 to 72. Column 1 is reserved for a subset of the CTLASA characters and it is these which control the source listing. The permitted characters are:

```
b
0
-
+
1
```

and they have the same action as if they were used as full CTLASA characters. If any other character appears in column 1 then an error message is generated and the character is replaced by a blank.

Example

```
1 B: PROC;
0    DCL A CHAR(5);
       /* LIST OF DECLARATIONS */
     DCL Z PIC '99';
-      /* CALCULATIONS START HERE */
```

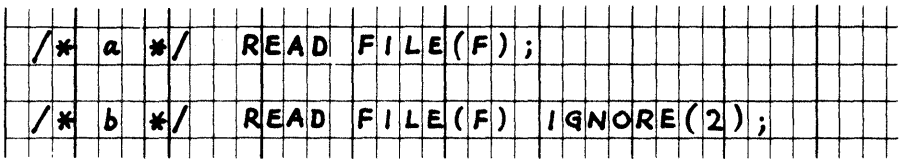
This will ensure that the procedure starts at the top of a new page, there is a blank line before the declaration statement and two blank lines before the calculations commence.

Summary

CONSECUTIVE data sets are the simplest form of data organization. You have seen here how to process them either within buffers (locate mode) or in a programmer-defined variable (move mode). Some of the other facilities such as the IGNORE, EVENT and WAIT options will be mentioned again when other data organizations are covered. You should ensure that you understand all the I/O statements listed in the table in the manual before you look at the other organizations.

Exercises

1. Briefly, what is the main difference in meaning between the keywords CONSECUTIVE and SEQUENTIAL.
2. What is the effect of the following



3. What is the function of the REWRITE statement? Under what conditions can it be used?
4. a) Write a record output statement for the unbuffered file UNBUF, so that subsequent processing may be overlapped with the actual output of data from the field OAREA.

   b) Write the statement which will ensure the output in the above example is complete before processing resumes.

5. An output structure has been declared as follows:

```
DCL 1 HLINE,
      2 CTL CHAR(1),
      2 FILL1 CHAR(50),
      2 HEADING CHAR(30),
      2 FILL2 CHAR(52);

DCL 1 LINE DEF HLINE,
      2 ASACHAR CHAR(1),
      2 FLDA CHAR(25),
      2 FLDB PIC '(8)9',
      2 FLDC CHAR(22),
      2 SPARE CHAR(77);
```

Write PL/I statements to print HEADING at the top of a new page, to skip a line, and to print the contents of LINE.

You should indicate where you would assign values to HEADING, FLDA, FLDB and FLDC but need not write the actual statement.

Answers

1. CONSECUTIVE refers to the data set organization. SEQUENTIAL refers to the way in which the data will be accessed.
2. a. The next record will be skipped

   b. The next two records will be skipped.
3. The REWRITE statement is used to replace or alter the latest record input from a given data set. It can only be used when the file has the UPDATE attribute and the most recent I/O statement for the file was a READ.
4.

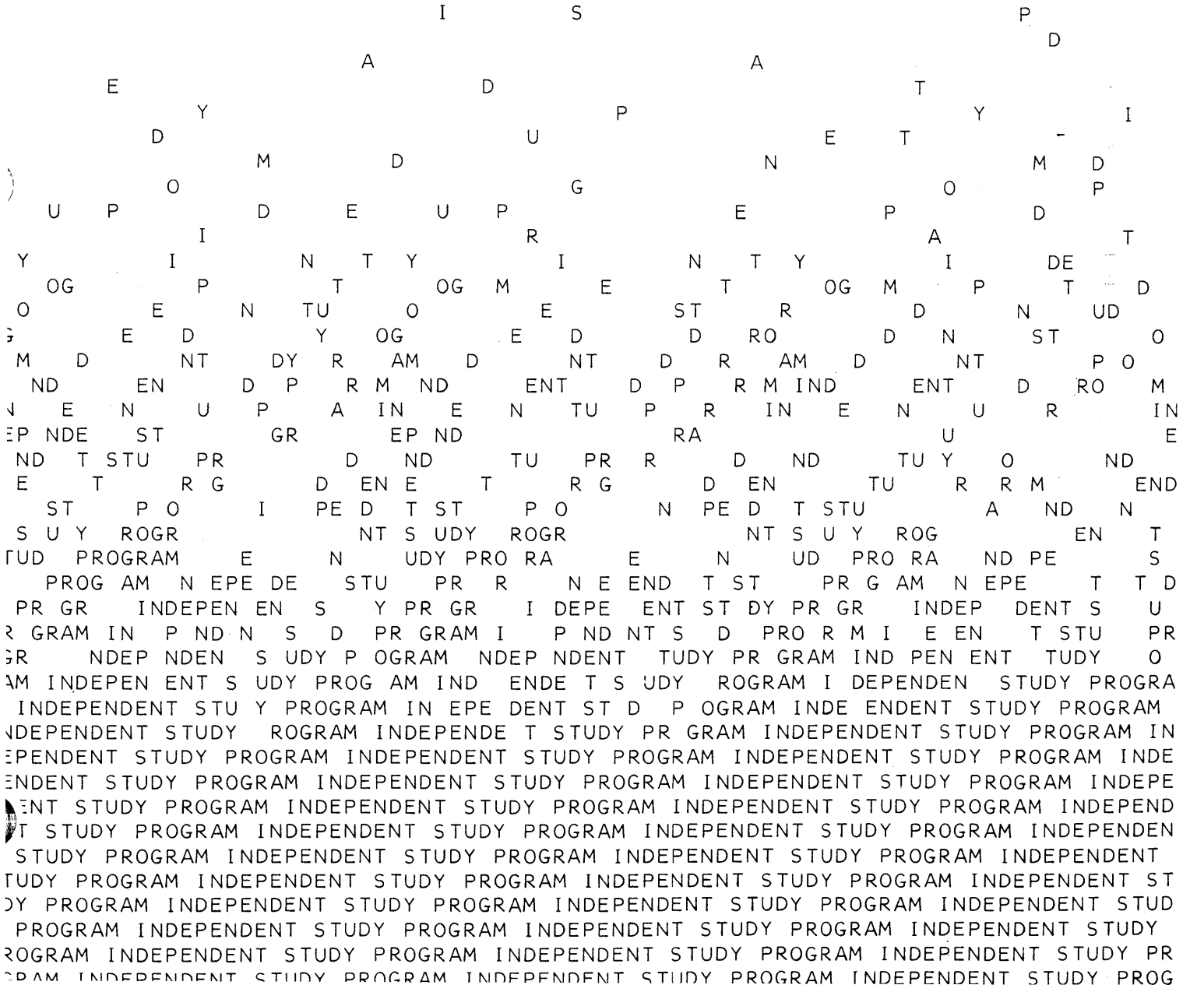
```
/* A */  WRITE FILE(UNBUF) FROM(OAREA) EVENT(ANY_IDENTIFIER);
/* B */  WAIT(ANY_IDENTIFIER);
```

5.

```
CTL = '1';
/* ASSIGN HEADING VALUES */
WRITE FILE(X) FROM(HLINE);
ASACHAR = '0';
/* ASSIGN DETAIL VALUES */
WRITE FILE(X) FROM(LINE);
```

Exercise

A



Exercise A

At this point, you have sufficient PL/I knowledge to code simple PL/I programs. The specifications for three such programs are given below as well as sample data and suggested job control statements. Having coded your solution, have it punched along with the data and job control. You are then in a position to test your solution.

We have not yet covered the testing and correcting or programs; this will come later. However by studying the output listings produced by the compiler, attempts at correcting the program (if necessary) can be made. While you are waiting for your output to be returned, you can carry on reading further topics.

Exercise 1

Description of the Program

The purpose of the program is to calculate the areas of circles and the volumes of spheres for given radii.

Record Format and Message Layout

Input

Filename is INFILE.

Record length is 80.
Blocksize is 80.
Logical unit for DOS/VS programmers is SYSIPT.

The records contain the radius in the first two characters. The remaining portion of the record is blank.

You may assume that the radii are numeric but no check has been made for zero or negative radii.

Output

Filename is OUTFILE.

Record length is 60.
Logical unit for DOS/VS programmers is SYSLST.

The following notes refer to the printer spacing chart below.

Radius

The input radius.

Area

The area of the corresponding circle, truncated to 2 decimal places. If the radius is invalid, then AREA is to be replaced by '****'.

Volume

The volume of the corresponding sphere, truncated to 2 decimal places. If the radius is invalid, then VOLUME is to be replaced by '****'.

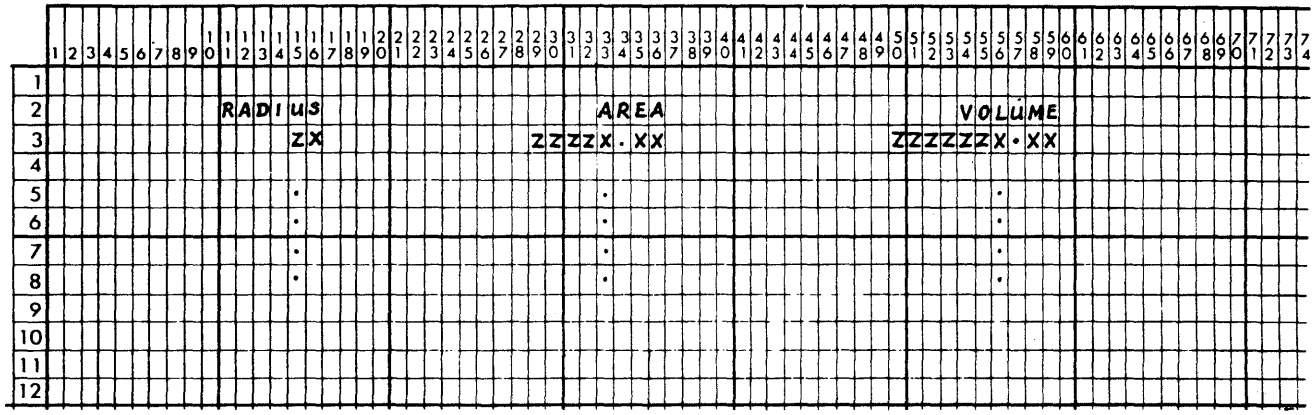


Figure XA.3: AREA/VOLUME REPORT

Processing Requirements

Main Program Logic

Read a record.

Check for validity.

Calculate the area and volume and print a line.

Continue reading and printing until the end of INFILE is reached.

Formulas

$$\text{AREA} = 3.1416 * \text{RADIUS}^2$$

$$\text{VOLUME} = \frac{4*3.1416*\text{RADIUS}^3}{3}$$

Test Data

The following should be punched and used to test your program:

```
  3
  5
 60
 90
 99
 00
 -7
 -5
  6
 ↑
COL 1
```

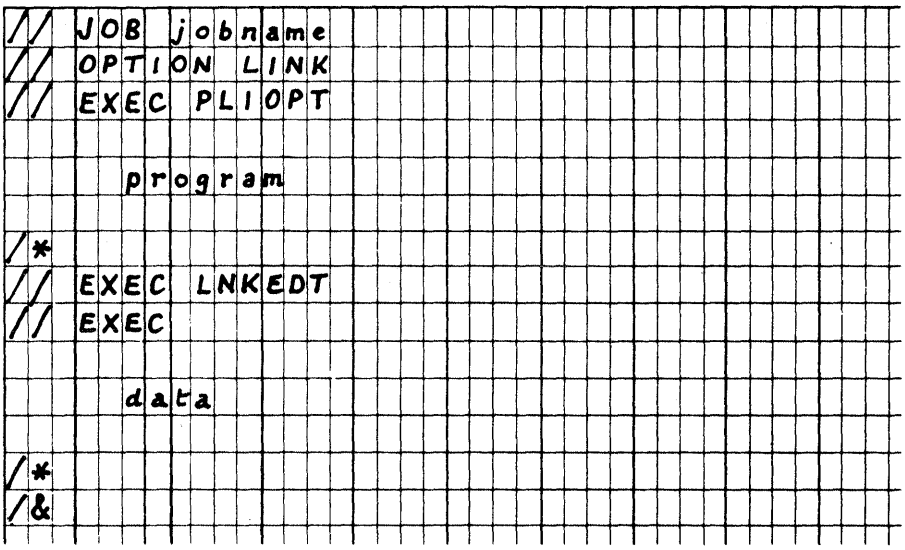
Expected Output

RADIUS	AREA	VOLUME
3	28.27	113.09
5	78.53	523.59
60	11309.75	904780.31
90	25446.95	3053634.00
99	30790.81	4064387.00
0	****	****
-7	****	****
-5	****	****
6	113.09	904.78

Job Control Statements

The following job stream will be required to run your program:

DOS/VS



OS/VS

```
//jobname JOB
//  EXEC PLIXCLG
//PLI.SYSIN DD *

      program

/*
//GO.INFILE DD *

      data

/*
//GO.OUTFILE DD SYSOUT=A,DCB=(RECFM=F,BLKSIZE=60)
```

Exercise 2A - Scientific

Description of the Problem

The purpose of this program is to calculate the mean and variance of various samples, the latter being the input to the program. (Do not worry if you do not know what variance is).

Record Formats and Message Layouts

Input

Filename is CARDIN.

Record length is 80.
Blocksize is 80.
Logical unit for DOS/VS programmers is SYSIPT.

A record can contain up to and including 40 fields, each of 2 numeric characters. Each field represents one sample.

The unused portion of all input records is blank.

You may assume that the contents of the fields have been checked and are, hence, valid. However, there may be records with no samples in them.

Output

Filename is REPORT.

Record length is 70.
Logical unit for DOS/VS programmers is SYSLST.

The following notes refer to the printer spacing chart below.

SAMPLES

The number of items in the sample.

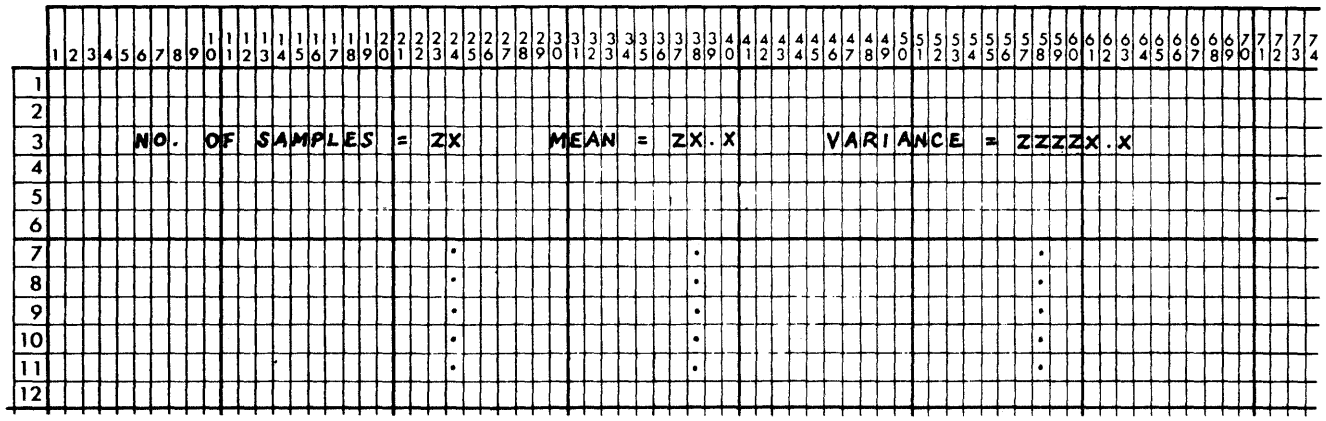
MEAN

The average value of the items in the sample, truncated to one decimal point.

VARIANCE

The variance of the items in the sample, truncated to one decimal place.

The entries in the MEAN and VARIANCE fields are to be '****' when there are no samples in the input record.



Processing Requirements

Main Program Logic

Read a record.

Check for no samples.

Calculate the necessary values and print a line for each card read (see formulas below).

Continue reading and printing until the end of CARDIN is reached.

Formulas

$$\text{MEAN} = \frac{\text{SUM OF SAMPLES}}{\text{NO. OF SAMPLES}}$$

$$\text{VARIANCE} = \frac{\text{SUM OF (SAMPLE-MEAN)}^2}{\text{NO. OF SAMPLES}}$$

Test Data

The following should be punched and used to test your program.

```
0204060810121416
242634364446
0102030405060708091011121314151617181920212223242526272829303132333435363738394080808080808080

01000001020304050698
0000
↑
COL 1
```

Note that the fifth card is blank.

Job Control Statements

The job stream required to run the program is the same as for exercise 1, except that, for OS/VS, the output filename for the 'GO' step is REPORT, the BLKSIZE is 70 and the input filename for the 'GO' step is CARDIN.

EXPECTED OUTPUT

```
NO. OF SAMPLES =  8     MEAN =  9.0     VARIANCE =  21.0
NO. OF SAMPLES =  6     MEAN = 35.0     VARIANCE =  67.6
NO. OF SAMPLES = 40     MEAN = 20.5     VARIANCE = 133.2
NO. OF SAMPLES =  7     MEAN = 80.0     VARIANCE =   0.0
NO. OF SAMPLES =  0     MEAN = ****     VARIANCE =  ****
NO. OF SAMPLES = 10     MEAN = 12.0     VARIANCE = 825.5
NO. OF SAMPLES =  2     MEAN =  0.0     VARIANCE =   0.0
```

Exercise 2B - Commercial

Description of the Problem

The purpose of this program is to prepare an inventory report.

Record Formats and Message Layouts

Input

Filename is CARDIN.

Record length is 80.
Record length is 80.
Blocksize is 80.
Logical unit for DOS/VS programmers is SYSIPT.

Col #	Field	
1- 6	Item number	(6 characters)
7-26	Description	(20 characters)
27-30	Unit cost	(PIC '99V99')
31-34	Unit sell price	(PIC '99V99')
35-38	Quantity on hand	(PIC '9999')
39-80	Unused	

You may assume that the contents of the fields have been checked and are, hence, valid.

Output

Record length is 82.

Logical unit for DOS/VS programmers is SYSLST.

Col #	Field	
1- 6	Item Number	
9-28	Description	
31-34	Quantity on hand	(suppress high order zeroes)
37-41	Unit cost	(ZZ.XX)
44-53	Total cost	($$$,$$X.XX)
56-60	Unit sell	(ZZ.XX)
63-72	Total sell	($$$,$$X.XX)
76-82	Profit Ratio	(ZXX PCT, or *******)

Processing Requirements

Total cost is the product of quantity on hand and unit cost.

Total sell is the product of quantity on hand and unit sell.

PROFIT RATIO.

Compute the profit ratio on each item, as follows:

(Unit sell - unit cost)/unit cost.

Round to the nearest whole percentage.

If less than or equal to 25%, print asterisks in the profit ratio field. If greater than 25%, print the percentage with 'PCT' following.

Accumulate a final total cost and a final total sell equal to the totals of the total cost and total sell fields, respectively, and print these totals two lines below the last detail line. (See expected output).

Test Data

The following should be punched and used to test your program.

```
014713WIDGETS              017301980319
127912GADGETS              091214980020
232417DEALIES              007800980100
299887DILLIES              003300490500
316621SNAZZLES             003400790178
421488FRAZZLES             003400790700
517329BLOOPERS             001900390000
691312WHOOPSIES            143419980003
814612WHOPPERS             253229790102
923178THING-A-MA-BOBS      017802590113
↑                          ↑
COL 1                      COL 27
```

Job Control Statements

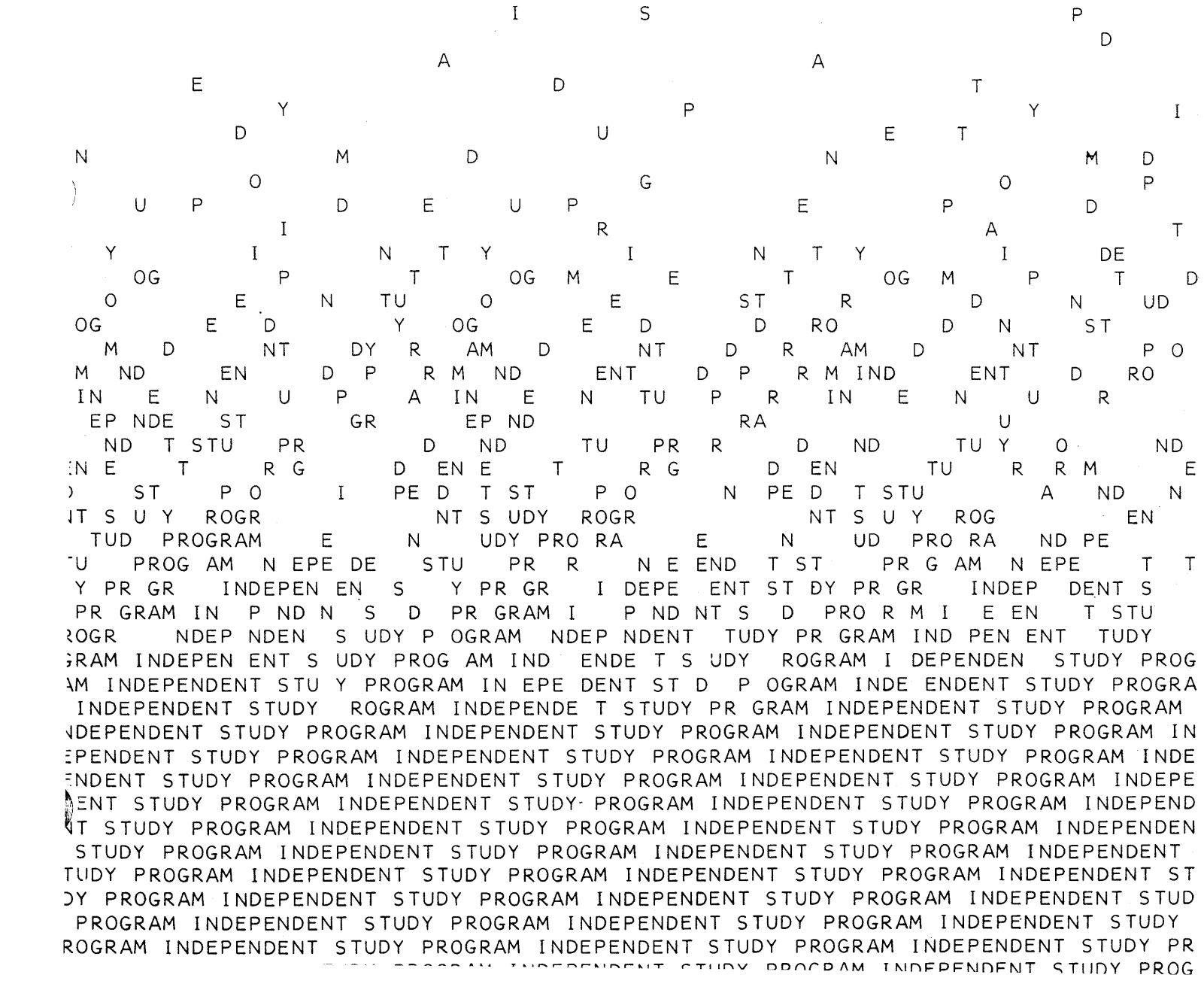
The job stream required to run the program is the same as for exercise 1, except that, for OS/VS, the output filename for the 'GO' step is OUTFILE, the BLKSIZE is 82 and the input filename for the 'GO' step is CARDIN.

Expected Output

ITEM NO	ITEM DESCRIPTION	ON HAND	UNIT COST	TOTAL ITEM COST	UNIT SELL	TOTAL ITEM SELL PRICE	PROFIT RATIO
014713	WIDGETS	319	1.73	$551.87	1.98	$631.62	*******
127912	GADGETS	20	9.12	$182.40	14.98	$299.60	64 PCT
232417	DEALIES	100	.78	$78.00	.98	$98.00	26 PCT
299887	DILLIES	500	.33	$165.00	.49	$245.00	48 PCT
316621	SNAZZLES	178	.34	$60.52	.79	$140.62	133 PCT
421488	FRAZZLES	700	.34	$238.00	.79	$553.00	133 PCT
517329	BLOOPERS	0	.19	$0.00	.39	$0.00	105 PCT
691312	WHOOPSIES	3	14.34	$43.02	19.98	$59.94	39 PCT
814612	WHOPPERS	102	25.32	$2,582.64	29.79	$3,038.58	*******
923178	THING-A-MA-BOBS	113	1.78	$201.14	2.59	$292.67	46 PCT
				$4,102.59		$5,359.03	

Exercise

B



Exercise B

At this point you should be able to code a solution to the following problem. You have now learned enough PL/I to do so. On completion, have it punched up along with the given data and the suggested job control statements. You are then in a position to run and test the program. While you are waiting for the punching to be done or the job to be run, you should carry on with the text. Two later topics which may help you debug the program are as follows:

TOPIC 16, 'CONTROLLING THE COMPILER' - this will help you remove compile time errors.

TOPIC 20, 'TESTING AND DEBUGGING AIDS' - this will help you remove execution time errors.

Description of the Problem

Program Objective

The purpose of this program is to produce a report showing the rate of depreciation in the value of second-hand cars. The input data is based on published tables of second-hand car prices.

Record Formats and Message Layouts

Input

Depreciation detail records

Filename is CARFILE.
Record length is 80.
Blocksize is 80.
Logical unit for DOS/VS programmers is SYSIPT.

There are two types of record in this file:

1. Car-name records identified by the character 'A' in column one (see Fig. B.1) and containing the name of the car.
2. Depreciation records identified by the character 'B' in column one (see Fig. B.2). Each depreciation card contains up to four sets of depreciation details relating to the preceding 'A' card.

For every 'A' record there will be one or more 'B' records.

The end of the input is indicated by a record with the character '9' in column one.

The unused portion of all input records is blank.

You may assume that the sequence of the records and the contents of the fields have already been checked.

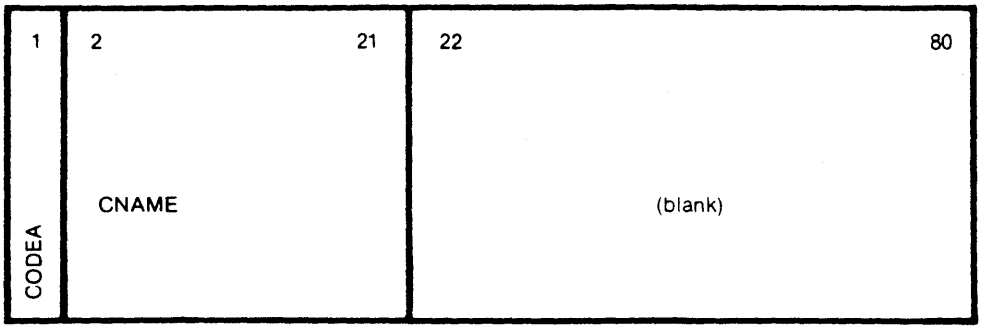


Figure B.1 CARFILE: CAR NAME RECORD FORMAT

CODEA	Record type identifier: Character 'A'
CNAME	Car name: Character, left aligned.

COLUMN	1	2	9	16	18	25	32	34	41	48	50	57	64	66
	CODEB	NPRICE	CPRICE	AGE	NPRICE	CPRICE	AGE	NPRICE	CPRICE	AGE	NPRICE	CPRICE	AGE	(BLANK)

Figure B.2 CARFILE: DEPRECIATION DETAIL RECORDS

CODEB	Record type identifier: Character 'B'
NPRICE	Price when new: numeric character, right aligned, cents up to seven digits.
CPRICE	Current price: format as for NPRICE.
AGE	Age of car: numeric character, right aligned, years up to two digits.

Output

Printed Depreciation Report

Filename is PRFILE.
Record length is 70.
Logical unit for DOS/VS programmers is SYSLST.

The following refer to the printer spacing chart (Fig. B.3 below).

CNAME	Car name, taken directly from the input record: character(20).
AVPERC	Average annual depreciation, expresses as a percentage, calculated to one decimal place, rounded up and printed with suppression of leading zeroes up to the first integer digit. If negative, a floating sign should be provided.
AVDEPN	Average annual depreciation, expressed in decimal dollars, rounded up to the nearest whole new penny. Source and format as for AVPERC.
AGE	Age of car, taken from the input record with suppression of leading zero.
FLAG	A constant, '****', which is to be printed *only* if the average annual depreciation is less than eleven percent after rounding.
HTOT	Hash total. The sum of the CPRICE fields in all the input records, edited as shown on Fig. B.3.
MEANPERC	Mean annual depreciation of all cars submitted, expressed as a percentage with suppression of leading zero.
	For further clarification of the format of the report, see Fig. B.4, SAMPLE OUTPUT.

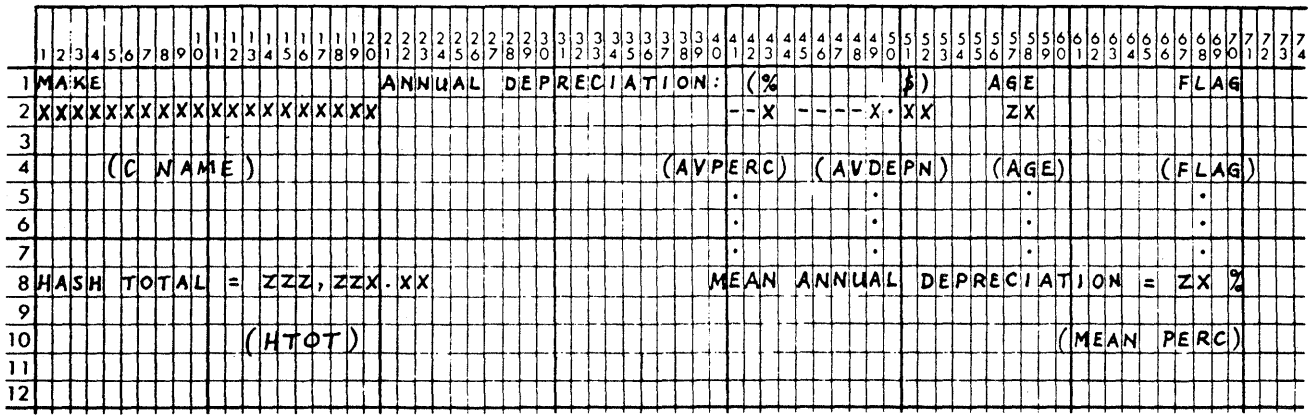


Figure B.3 PRFILE: DEPRECIATION REPORT LAYOUT.

Processing Requirements

Main Program Logic

Print the Heading.

Read a name record and a detail record.

Calculate the necessary value and print one line for each set of NPRICE, CPRICE and AGE (see Fig. B.3).

Continue reading and printing until the end of CARFILE is reached. This will be indicated by a character '9' in column one.

Calculate the mean annual depreciation and print the last line of the report.

Percentage depreciation formulas

The average percentage depreciation (d) of a particular car of age (a) can be calculated from the current selling price (c) and the price when new (n) using the following formula:

$$d = 100\ (1 - (c/n)^{1/a})$$

This gives a compound rate of depreciation.

The average depreciation in dollars (d) of a particular car of age (a) can be calculated from the current selling price (c) and the price when new (n) using the following formula:

$$d = \frac{n - c}{a}$$

MEANPERC

The overall average percentage depreciation for all cars submitted (MEANPERC) is the average of the individual values of AVPERC.

Rounding

By 'Rounding' we mean that a percentage of 14.5 should be printed as 15 and one of 14.4 as 14.

Test Data

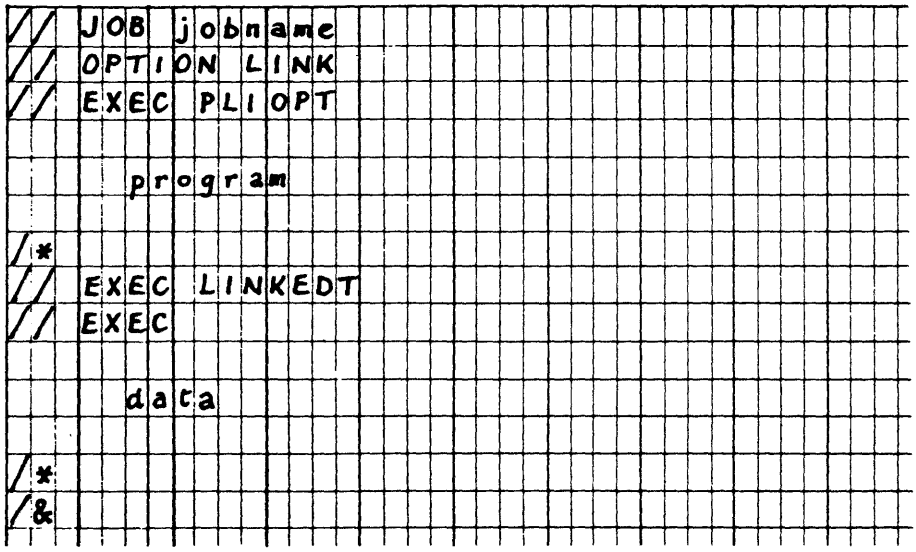
The following information should be punched and used to test your program:

```
AFLASHMAN DE LUXE
B033919900624500903391990071040080349075009400007036793301435400 6
ACONQUESTIDOR BANGER
B006185000110001200618500012975110053899001850009005389900204500 8
B005400000239750800540000027540060054999003204905006807500530200 2
AFLIEDERMAUS 1000
B009209900589990200942250069499 01
AMAZAWATEE 7.5 LITRE
B053400003489990406017990425500030700700053302502
AKAMIKAZE BOREALIS 50
B017735000425000601773500050990050122700006052504013269000720000 3
AGNOMOBILE RUSHTON
B7432751491342103224009907342100 2
AGLADSTONE STEAM CAR
B0001050026500072
9999999999
↑
Col 1
```

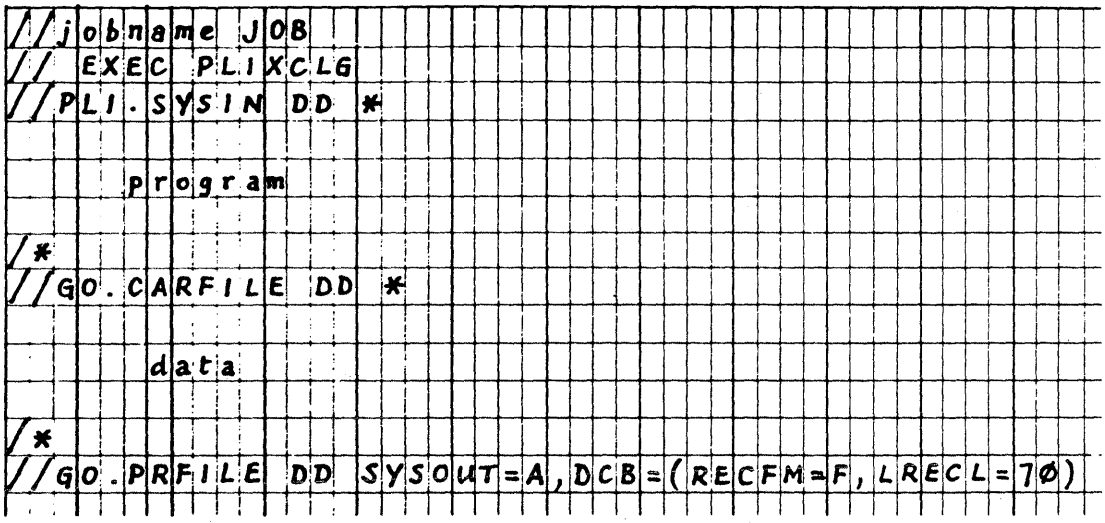
Job Control Statements

The following job stream will be required to run your program.

DOS/VS



OS/VS


//jobname JOB
// EXEC PLIXCLG
//PLI.SYSIN DD *
program
/*
//GO.CARFILE DD *
data
/*
//GO.PRFILE DD SYSOUT=A,DCB=(RECFM=F,LRECL=70)

EXPECTED OUTPUT

MAKE	ANNUAL DEPRECIATION: (%	$)	AGE	FLAG
FLASHMAN DE LUXE	17	307.50	9	
	18	335.20	8	
	17	364.39	7	
	15	373.99	6	
CONQUESTIDOR BANGER	13	42.38	12	
	13	44.43	11	
	11	39.33	9	
	11	41.81	8	
	11	42.89	7	
	11	44.10	6	
	10	45.90	5	****
	12	75.28	2	
FLIEDERMAUS 1000	20	165.50	2	
	26	247.26	1	
MAZAWATEE 7.5 LITRE	10	462.50	4	****
	11	587.66	3	
	13	838.38	2	
KAMIKAZE BOREALIS 50	21	224.75	6	
	22	252.72	5	
	16	155.44	4	
	18	202.30	3	
GNOMOBILE RUSHTON	13	8397.77	3	
	43	7529.45	2	
GLADSTONE STEAM CAR	-7	-36.65	72	****
HASH TOTAL = 81,452.07	MEAN ANNUAL DEPRECIATION 15%			

Figure B.4: SAMPLE OUTPUT