Plain PL/I
(A PL/I Primer)

093-000216-00

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000216
© Data General Corporation, 1978
All Rights Reserved
Printed in the United States of America
Revision 00, March 1978

Plain PL/I
(A PL/I Primer)
093-000216

Original Release - March 1978

Preface

We wrote this manual to teach you how to program using Data General PL/I, a subset of ANSI Standard PL/I. If you are an experienced PL/I programmer, you may want to skip this manual and go right to the *PL/I Reference Manual,* (093-000204).

Data General PL/I runs on the Advanced Operating System (AOS). To acquaint yourself with the operating system, read *Learning to Use Your Advanced Operating System,* (093-000196).

Reader, Please Note:

We use the following conventions in examples in this manual:

Where	**Means**
KEYWORD	You must use the keyword (or its accepted abbreviation) as shown.
required	You must give some argument (such as a filename).
[optional]	You have the option of entering some argument. Don't enter the brackets; they only set off what's optional.
...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Additionally, we use certain symbols in special ways:

Symbol	**Means**
)	Press the NEW-LINE or RETURN key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

All numbers are decimal.

Finally, we usually show all examples of entries and system responses in THIS TYPEFACE. But, where we *must* clearly differentiate your entries from system responses in a dialog, we will use

THIS TYPEFACE TO SHOW YOUR ENTRY)
THIS TYPEFACE FOR THE SYSTEM RESPONSE

End of Preface

Contents

Chapter 1 - How to Create a PL/I Program

From Source Text to Executable Program 1-1
Writing the Source Text 1-1
Compiling the Source Text 1-1
Binding the Source Text 1-1
The Example Program EXAMPLE 1-2
Writing the Source Text of EXAMPLE 1-2
Compiling EXAMPLE.PL1 1-3
Binding EXAMPLE.OB 1-5
Executing EXAMPLE 1-5
The Source Text of Example 1-5
Identifiers 1-5
Named Constants and Variables 1-6
Literal Constants 1-6
Delimiters 1-6
Comments 1-6
PL/I Statements 1-7
Groups and Blocks 1-7
The Statements of EXAMPLE 1-7
The PROCEDURE Statement 1-7
The DECLARE Statement 1-7
Input and Output in EXAMPLE 1-8
The OPEN Statement 1-8
PUT LIST 1-9
GET LIST 1-10
Variations 1-10

Chapter 2 - Data and Its Uses

Data Types: Why They Are Important 2-1
Data Types in PL/I 2-1
Arithmetic Data Types 2-4
Precision and Scale with Arithmetic Data 2-4
Fixed Binary Data 2-5
Fixed Decimal Data 2-6
Float Binary Data 2-7
Conversions with Arithmetic Data Types 2-8
Arithmetic Operations 2-8
Conversions with Operations 2-10
Subexpressions and Precedence 2-10
Parentheses 2-11
Errors Resulting from Arithmetic Operations 2-11
Built-in Functions 2-12
Putting it All Together: A Program to Calculate Simple Interest 2-14
Character-string Data 2-15
The Character-string Operator !! 2-15
Conversions Between Arithmetic and Character-string Data Types 2-16

Chapter 3 - Conditional Branching in PL/I

IF THEN 3-2
IF THEN ELSE 3-3
Nested IF THEN Statements 3-4
Non-Iterative DO Groups 3-5
Using IF THEN in the Simple Interest Program 3-6
The Relational Operators 3-7
    Bit-string Data 3-8
    Bit Constants 3-8
    Bit Variables 3-8
Bit-string Data and The Results of Relational Operators 3-8
Bit-string Data and Logical Operations 3-9
The Bit-string Operators and the Relational Operators 3-10
Conversions and Operations with Bit-string Data 3-11
The Order of Evaluation of Operations 3-11
Multiple Branching: DO CASE 3-12

Chapter 4 - Loops and STREAM Files

Introduction to PL/I's Loops 4-1
DO WHILE 4-3
    The Bit-string Expression 4-4
Nested DO Loops 4-5
DO Increment 4-6
    Using DO Increment to Calculate Compound Interest 4-8
DO List 4-10
Stream Files and List-Directed I/O 4-12
The CLOSE Statement 4-13
GET LIST and PUT LIST with Stream Files 4-14

Chapter 5 - Arrays and Structures

Operators with Array and Structure Operands 5-2
Arrays 5-2
    Declaring Arrays 5-2
    Subscripted References 5-3
    GET LIST and PUT LIST with Arrays 5-3
        Implied DO Increment with GET and PUT LIST 5-4
    Assignment with Arrays 5-4
    Using Arrays: a Simple Sort Program 5-5
Structures 5-6
    Declaring Structures 5-6
    Structure-qualified References 5-7
    GET LIST and PUT LIST with Structures 5-8
    Assignment with Structures 5-8
    Arrays of Structures 5-10
    Connected and Unconnected Arrays 5-10
    Using Structures 5-11

Chapter 6 - Blocks

Introduction 6-1
    The Block Structure of BLOCKS 6-2
    Scope in BLOCKS 6-3
    PROCEDURE Blocks 6-3
The Syntax of Blocks 6-7
    Terminology for Block Structure 6-7
    Scope 6-8
Block Activation 6-9
    Variables and Expressions in the Declarations of Automatic Variables 6-9
    Block Structure and Program Efficiency 6-10
Procedures 6-11
    Parameters and Arguments 6-12
        Declaring Parameters 6-12
        Asterisk Extent Expressions in Parameter Declarations 6-12
        The Array and String Built-in Functions DIM, HBOUND, LBOUND, and LENGTH 6-12
        Parameter Passing in PL/I 6-13
        Rules for Passing by Value and Passing by Reference 6-14
Type of Procedures 6-15
    Procedures Invoked with Call 6-15
    Returning from a Called Procedure 6-16
    Procedures Invoked with Function References 6-17
    Returning from a Block with a GOTO 6-18
    Side Effects with Procedures 6-18
    The Scope of Procedure Names 6-18
    The Scope of Internal Procedure Names 6-19
External Procedures 6-19
    The Scope of External Procedure Names 6-19
    The DECLARE Statement for External Procedure Names 6-21
    Writing, Compiling, and Binding Programs with Several External Procedures 6-21
    Binding PL1ECIS into your Program 6-22
    COMP_INT with Internal Procedures 6-22

Chapter 7 - Input/Output

File Data 7-2
    File Constants 7-2
    File Variables 7-2
    Assignment and Comparison with FILE Data 7-3
AOS PL/I File Types 7-3
    File Attributes 7-4
    PL/I File Types and AOS Record Types 7-4
Opening PL/I Files 7-4
    Opening a File with the OPEN Statement 7-5
        Specifying the Data Set: The TITLE Keyword 7-5
        Keywords for STREAM Files 7-6
        Keywords for Record Sequential Files 7-6
        Keywords for Record Direct Files 7-7
        Default and Implied File Attributes 7-7
        Default Attributes 7-7
        Default Data Sets 7-7
        Default Line Size with Stream Output Files 7-7
        Default Page Size with Stream Output Print Files 7-7

Opening Files with the GET, PUT, READ, and WRITE statements . . . 7-8
The Default PL/I Files SYSIN and SYSPRINT . . . 7-8
File Openings and the File Control Block . . . 7-9
The ENDFILE Condition . . . 7-9
The ENDPAGE Condition . . . 7-9
I/O with Stream Files . . . 7-10
Types of I/O with Stream Files . . . 7-10
List-Directed I/O . . . 7-10
The GET LIST Statement . . . 7-10
The PUT LIST Statement . . . 7-11
The Implied DO Increment in Input Lists and Output Lists . . . 7-13
Edit-directed I/O . . . 7-13
The GET EDIT Statement . . . 7-14
The PUT EDIT Statement . . . 7-15
The FORMAT Statement . . . 7-15
The Format List . . . 7-16
The Format Items . . . 7-16
The F Format . . . 7-17
The E Format . . . 7-18
The A Format . . . 7-19
The B Format . . . 7-20
The P Format . . . 7-20
The Control Format Items . . . 7-22
The COLUMN Format Item . . . 7-22
The X Format Specifier . . . 7-23
The SKIP Format Specifier . . . 7-23
The LINE Control Format . . . 7-23
The TAB Format Item . . . 7-24
The PAGE Format Item . . . 7-24
The R Format Item . . . 7-24
GET and PUT as Format Control Statements . . . 7-25
Line-Directed I/O . . . 7-25
READ with Stream Input Files . . . 7-25
WRITE with Stream Output Files . . . 7-26
Interactive Terminal I/O under AOS . . . 7-26
Input and Output with Record Files . . . 7-27
Illegal Variables with Record I/O . . . 7-28
Record Sequential I/O . . . 7-29
The READ Statement with Record Sequential Files . . . 7-29
The WRITE Statement with Record Sequential Files . . . 7-29
Record Direct I/O . . . 7-30
The READ Statement with Record Direct Files . . . 7-30
The WRITE Statement with Record Direct Files . . . 7-31
The REWRITE Statement . . . 7-31
The DELETE Statement . . . 7-32
The Program: STATEMENT . . . 7-32

Chapter 8 - Working with Character and Bit Strings

String Data in PL/I 8-1
Operators and Built-in Functions for String Handling 8-1
Operators and Functions for String Manipulation 8-1
Functions which Return Information About Strings 8-2
Functions Using the ASCII Character Sequence 8-2
An Example: the Program PIGLAT 8-2
Concatenate 8-5
SUBSTR 8-5
LENGTH 8-6
Using SUBSTR and LENGTH to Read a Line One Character at a Time 8-7
VERIFY 8-7
INDEX 8-8
TRANSLATE 8-8
COLLATE 8-9
RANK 8-10
ASCII 8-10
STRING 8-10
An Example: A Simple Editor 8-11
Bit-string Functions 8-14
UNSPEC 8-14
BOOL 8-15

Chapter 9 - Pictured Data

Using Pictured Variables in your Programs 9-2
The Picture Specifier 9-3
How to Use the Specifiers 9-4
9, Z, and * 9-4
$, S, +, - 9-4
/, B, , , ... 9-5
CR and DB 9-5
V 9-5

Chapter 10 - Storage Classes, Based and Defined Variables

Introduction 10-1
Data Types and Storage 10-1
Aligned Data 10-2
PL/I's Storage Classes 10-3
The Meaning of the Storage Classes 10-3
Based Variables and Pointers 10-4
Pointers 10-5
Based Variables 10-5
Extent Expressions with Based Variables 10-5
Giving Storage to a Based Variable 10-5
The ALLOCATE statement 10-6
Multiple Allocations Of Storage 10-6
Freeing Storage Allocated to a Based Variable 10-7
Using Based Variables and Pointers to Reference Storage Belonging to Other Variables 10-7
The ADDR Built-in Function 10-8

Rules for ADDR 10-8
The NULL Built-in Function 10-9
Using Several Pointers to Refer to the Same Storage 10-9
Pointer-qualified Reference 10-10
Explicit Pointer-qualified Reference 10-10
Implicit Pointer-qualified Reference 10-11
Rules for Pointer-qualified Reference 10-11
Based Variable Types and Other Types 10-11
Extent Expressions and Pointer-qualified Reference 10-12
Fixing the Size of a Based Structure: the REFER Option 10-12
Cautions on the Use of Based Variables 10-13
An Example: Using Based Variables to Make a Linked List 10-13
Defined Variables 10-16
Rules for Defined Variables 10-17
Storage Sharing in PL/I 10-17
Storage Sharing in Standard PL/I 10-17
Examples for Standard PL/I Storage Sharing 10-18
Rules for Storage Sharing in AOS PL/I 10-18

Chapter 11 - The GOTO Statement and Label Data

Introduction 11-1
Label Data 11-2
Label Constants 11-2
Arrays of Label Constants 11-3
Label Variables 11-4
The GOTO Statement 11-5
Rules for the GOTO 11-6
Compiler Detection of Errors with GOTO 11-8
Proper Use of the GOTO in PL/I 11-8

Chapter 12 - Storage and Initialization of Static Variables

The Scope of Static Variables 12-1
Declaring Static Variables 12-2
Extent Expressions with Static Variables 12-3
Initialization with Static Variables 12-3
Initializing Arrays 12-3
Initializing Structures 12-3

Chapter 13 - Getting Into and Out of Blocks: Entry Data, the ENTRY Statement, Recursive Block Activation, and ON Units

Entry Data 13-1
Entry Constants 13-1
External Entry Constants 13-2
Entry Variables 13-3
Examples of Parameter Attributes for Array, Structure and Entry Parameters 13-5
The ENTRY Statement 13-6
Syntax of the ENTRY statement 13-6
An Example: the Procedure F_READ 13-7

Recursive Procedures 13-8
    Blocks in Recursive Invocation 13-8
ON Units and Conditions 13-10
    The ON Statement and the ON Unit 13-10
    Syntax of the ON Statement and the ON Unit 13-11
    Multiple ON-Units 13-12
    ON Units and Efficiency 13-12
    The SIGNAL and REVERT Statements 13-12
Conditions 13-14
    The ERROR Condition 13-14
    The ONCODE Built-in Function 13-16
    The ENDFILE Condition 13-16
    The ONFILE Built-in Function 13-17
    The ENDPAGE Condition 13-18

Chapter 14 - Calling AOS and Assembly-Language Routines from AOS PL/I

Making AOS System Calls 14-1
    A Simple Example: Calling ?DELETE to Delete a File 14-1
    Using %INCLUDE with PL1SYSID.PL 14-3
    A Complete Subroutine to Delete a File 14-3
    A More Complicated Example: Using ?GTMES to Read a CLI Command 14-4
Calling Assembly-Language Routines from PL/I 14-11

Chapter 1
How to Create a PL/I Program

From Source Text to Executable Program

In this chapter we show you how to create a simple program in AOS PL/I.

There are three steps to creating a program:

- writing the program's source text
- compiling the source text to produce object code
- binding the object code with the PL/I runtime library to produce executable code.

Writing the Source Text

Like BASIC, COBOL, or FORTRAN, PL/I is a high-level language. You write a PL/I program using the "words" and "symbols" of the PL/I language. The program as it is written in the PL/I language is called the source text. In AOS PL/I, your source text will be an AOS file containing the text of your PL/I program. In AOS, you write text files with one of the text editors. For details, see *Learning to Use Your Advanced Operating System.*

Compiling the Source Text

Next you must compile and bind the source text. The compiler is a program that recognizes the PL/I words and symbols of your source text and generates code which another program called the binder can use to produce the executable code. The code generated by the compiler is called object code. As the compiler compiles your program, it checks the source text for violations of the rules of the PL/I language. If it finds any, it gives you an error message. You correct errors by editing your source text and compiling it again.

Binding the Source Text

To get executable code, you must combine the object code with the PL/I runtime library. You do this with the AOS binder utility. This step is called binding the program, and once you have finished it, you have executable code and can execute your program.

The Example Program EXAMPLE

To see how all this works in practice, let's take a look at the PL/I program EXAMPLE. EXAMPLE asks you to input a word at the terminal and then outputs the word along with a message. The source text for EXAMPLE looks like this:

```
EXAMPLE:
   PROCEDURE;

   DECLARE(INPUT,OUTPUT)FILE;
   DECLARE CHARSTRING CHAR(20) VARYING;

/*****************************************************************/
/* THE TITLE KEYWORD IN THE OPEN STATEMENT ASSOCIATES THE       */
/* PL/I FILES INPUT AND OUTPUT WITH THE AOS GENERIC FILES       */
/* FOR TERMINAL INPUT AND TERMINAL OUTPUT.                      */
/*****************************************************************/

   OPEN FILE (INPUT) STREAM INPUT TITLE ("@INPUT);
   OPEN FILE (OUTPUT) STREAM OUTPUT TITLE ("@OUTPUT");

   PUT FILE(OUTPUT)  LIST("INPUT A WORD");

   GET FILE(INPUT)   LIST(CHARSTRING);

   PUT FILE(OUTPUT)  SKIP LIST("HERE IS YOUR WORD", CHARSTRING);

   END; /* EXAMPLE */
```

Figure 1-1. The Example Program EXAMPLE

The statements which actually output a prompt, get the word, and then output the word are on lines 16, 18, and 20. The PUT LIST statement on line 16 outputs the prompt INPUT A WORD; the GET LIST statement on line 18 gets the word you input from the keyboard; the PUT LIST statement on line 20 outputs the word along with a message. The rest of the program sets up storage for constants and variables, associates PL/I files with AOS files, and defines the program as a unit of the PL/I syntax.

The EXAMPLE:PROCEDURE statement on lines 1 and 2, together with the END statement on line 22, defines the program as a PROCEDURE block. A PROCEDURE block is the smallest syntactical unit in PL/I which may be a program in its own right.

The DECLARE statements on lines 4 and 5 instruct the compiler to generate code to set up storage for the CHARACTER VARYING variable CHARSTRING and the file constants INPUT and OUTPUT. When the program refers to CHARSTRING, INPUT, and OUTPUT, it refers to the storage set up by the DECLARE statements.

The OPEN statements on lines 13 and 14 define the attributes of the PL/I files INPUT and OUTPUT, associating them with the AOS generic files for the terminal keyboard and CRT screen, @INPUT and @OUTPUT. Consequently, the GET LIST statement will get input from the keyboard, and the PUT LIST statement will output to the screen. Lines 7 through 11, finally, are a PL/I comment. When the compiler compiles a PL/I program, it ignores all characters between the comment delimiters /* and */.

Writing the Source Text of EXAMPLE

If you wanted to write the source text of EXAMPLE, you would use one of the text editor programs to create an AOS file, giving it the .PL1 extension. The file is given the .PL1 extension so that the compiler can recognize the file as a PL/I source text file.

In the following, we will name the source text file for EXAMPLE EXAMPLE.PL1.

Compiling EXAMPLE.PL1

To compile the source text file of your program, wait until you have the AOS prompt, then you type:

```
X PL1 source-text-filename
```

or in the case of EXAMPLE,

```
X PL1 EXAMPLE.PL1
```

When you type the command, the compiler looks for a file with the name you have given, and if it finds the file, it compiles it.

Since the compiler knows that PL/I source text file names have the .PL1 extension, you can also simply type

```
X PL1 EXAMPLE
```

If you want a listing, use the /L switch in your command:

```
)X PL1/L EXAMPLE
```

The compiler will output the listing to the AOS generic @LIST file. If you do not use the /L switch, the compiler will output the compile-time statistics and any error messages to your terminal.

For information on the other compiler switches, see the *PL/I Reference Manual,* (093-000204).

The compiler listing for EXAMPLE.PL looks like this:

```
SOURCE FILE:  EXAMPLE.PL
COMPILED ON 6/28/77   AT 13:35:17   BY PL/I REV  1.00

1      EXAMPLE:
2         PROCEDURE;
3
4         DECLARE(INPUT,OUTPUT) FILE;
5         DECLARE CHARSTRING CHAR(20) VARYING;
6
7         /***************************************************************/
8         /* THE TITLE KEYWORD IN THE OPEN STATEMENT ASSOCIATES THE     */
9         /* PL/I FILES INPUT AND OUTPUT WITH THE AOS GENERIC FILES     */
10        /* FOR TERMINAL INPUT AND TERMINAL OUTPUT.                    */
11        /***************************************************************/
12
13         OPEN FILE (INPUT) STREAM INPUT TITLE("@INPUT");
14         OPEN FILE (OUTPUT) STREAM INPUT TITLE("@OUTPUT");
15
16         PUT FILE(OUTPUT) LIST("INPUT A WORD");
17
18         GET FILE(INPUT) LIST(CHARSTRING);
19
20         PUT FILE(OUTPUT) SKIP LIST("HERE IS YOUR WORD", CHARSTRING);
21
22         END; /* EXAMPLE */

EXTERNAL ENTRY POINTS

NAME         CLASS       SIZE     LOC    ATTRIBUTES

EXAMPLE      CONSTANT                    ENTRY EXTERNAL
```

Figure 1-2. Compiler Listing for EXAMPLE

```
PROCEDURE EXAMPLE ON LINE 1

NAME              CLASS          SIZE   LOC      ATTRIBUTES

INPUT             CONSTANT                       FILE EXTERNAL
OUTPUT            CONSTANT                       FILE EXTERNAL
CHARSTRING        AUTOMATIC      11W     40      VARYING CHAR(20)

COMPILE-TIME STATISTICS

PHASE           TIME             SYMBOL TABLE I/O
SETUP           0:00:05
PASS1           0:00:01           0
DECLARE         0:00:00           0
PASS2           0:00:02           0
ALLOCATOR       0:00:01           0
CHAIN           0:00:06           2
PASS3           0:00:03           2
FINAL           0:00:01           0
TOTAL           0:00:19           4

LINES COMPILED 22
LINES PER MIN 66
SYMBOL TABLE PAGES USED 2, MAX 128
CONSTANTS AND INT STATIC 35
PROCEDURE CODE 172
```

Figure 1-2. Compiler Listing for EXAMPLE (continued)

As you can see, the first item on the listing is the source text of the program with line numbers. Then comes a list of the declared names in EXAMPLE, with their class size, location, and attributes. The last section of the listing is a set of compile-time statistics.

If the compiler finds an error in your program, it outputs the error messages just ahead of the compile-time statistics. For example, if we had forgotton a quotation mark in EXAMPLE, the listing would look like this:

```
SOURCE FILE:  EXAMPLE.PL
COMPILED ON 6/28/77   AT 13:48:37   BY PL/I REV 1.00

1       EXAMPLE:
2           PROCEDURE;
3
4           DECLARE(INPUT,OUTPUT) FILE;
5           DECLARE CHARSTRING CHAR(20) VARYING;
6
7           /****************************************************************/
8           /* THE TITLE KEYWORD IN THE OPEN STATEMENT ASSOCIATES THE      */
9           /* PL/I FILES INPUT AND OUTPUT WITH THE AOS GENERIC FILES      */
10          /* FOR TERMINAL INPUT AND TERMINAL OUTPUT.                     */
11          /****************************************************************/
12
13          OPEN FILE (INPUT) STREAM INPUT TITLE("@INPUT");
14          OPEN FILE (OUTPUT) STREAM INPUT TITLE("@OUTPUT");
15
16          PUT FILE(OUTPUT) LIST("INPUT A WORD );
17
18          GET FILE(INPUT) LIST(CHARSTRING);
19
20          PUT FILE(OUTPUT) SKIP LIST("HERE IS YOUR WORD", CHARSTRING);
21
22          END; /* EXAMPLE */

ERROR PL/1061 SEVERITY 4
THE SOURCE TEXT ENDS WITH AN UNRECOGNIZABLE STATEMENT.

COMPILATION ABORTED.
```

Figure 1-3. Compile-time Error in EXAMPLE

The error is in line 16. The last set of quotation marks enclosing the character-string constant ''INPUT A WORD'' is missing, and the compiler consequently interprets everything between the first set of quotation marks and the next set of quotation marks in line 20 as a character-string constant. It then cannot interpret the rest of the program as valid PL/I statements, so it aborts the compilation and prints an error message.

Errors which abort the compilation are severity 4 errors; severity 3 errors do not abort the compilation, but the compiler cannot produce working object code from the source text. Severity 2 errors are illegal in PL/I, but the code produced by the compiler will probably work. Severity 1 errors, finally, are warnings; they call your attention to features of AOS PL/I which differ from standard PL/I.

When the compiler compiles your program, it produces a file with the same name as the source text file, but with the .OB extension instead of the .PL1 extension. Thus, the file the compiler produces from EXAMPLE is called EXAMPLE.OB. This file contains the object code for the program. In order to get the executable code for the program, we have to bind the .OB file with the PL/I runtime library.

Binding EXAMPLE.OB

To bind a program you use the AOS binder utility. The AOS binder command is quite complicated, so we have included an AOS macro called PL1BIND in the PL/I package. To bind a program with the macro, you simply type

```
PL1BIND object-file-name
```

Note that you do not precede the macro with X or XEQ. To bind EXAMPLE, you would type:

```
PL1BIND EXAMPLE
```

Binding programs that contain more than one PL/I procedure or that use the Eclipse-line Commercial Instruction Set is more involved; for details, see Chapter 6.

When you execute the PL1BIND macro, the binder produces an executable program file from the .OB file and the PL/I runtime library. It gives this file the .PR extension. The executable program file for EXAMPLE is thus called EXAMPLE.PR. As the binder produces the executable program, it writes a list of the runtime routines it is binding into the program to your terminal. When the AOS prompt returns, you can execute EXAMPLE by typing

```
X EXAMPLE
```

Executing EXAMPLE

when you type X EXAMPLE, you will receive the prompt "INPUT A WORD" at your terminal. If you then input a word, say WOW, at the terminal and then hit the new line key, "HERE IS YOUR WORD" "WOW" will appear on the screen. Then the AOS prompt will appear, indicating that the system has finished executing the program.

The Source Text of Example

In PL/I terms, the source text of EXAMPLE is made up of identifiers, delimiters, and literal constants.

Identifiers are the ''words'' of a PL/I program and delimiters are its ''punctuation.'' Literal constants are values that are written directly into the source text. The identifiers, delimiters, and constants make up statements which are the ''sentences'' of a PL/I program. The statements, finally, may be combined into larger syntactic units called groups and blocks.

Identifiers

You may use the upper- or lowercase letters, the digits 1 through 9, and the underscore character (_) to make PL/I identifiers. The first character of an identifier must be a letter, but the other characters may be any combination of uppercase and lowercase letters and digits. PL/I treats upper- and lowercase letters as different characters. For example, STRING and string are two different variables. You may make identifiers up to 32 characters long.

Depending on their function, identifiers may be keywords or declared names. Keywords are identifiers that are defined as part of the PL/I language; declared names are identifiers that you define when you write your program.

In our sample program EXAMPLE, PROCEDURE, DECLARE, FILE, and OPEN are all examples of keywords. INPUT, OUTPUT, CHARSTRING, and the procedure name EXAMPLE are all declared names. You define a declared name when you declare it in your program. INPUT, OUTPUT, and CHARSTRING are defined by the DECLARE statements in which they appear; EXAMPLE is defined by its use as a label prefix on a PROCEDURE statement.

In the OPEN FILE(INPUT) STREAM INPUT TITLE("@INPUT"); STATEMENT, the first INPUT is the name of a file and hence a declared name. The second INPUT is a file attribute keyword. Though it is not good programming practice to use keywords as declared names, PL/I allows you to do so.

Named Constants and Variables

Declared names are either named constants or variables. Variables are declared names to which a program may assign values. In AOS PL/I you must define all variables with a DECLARE statement. The only variable in EXAMPLE is CHARSTRING. The GET LIST statement in line 18 assigns the string you input at the terminal to CHARSTRING.

Named constants are declared names which cannot have values assigned to them. EXAMPLE has 3 named constants: the procedure name EXAMPLE and the file names INPUT and OUTPUT. In PL/I terms, EXAMPLE is an entry constant and INPUT and OUTPUT are file constants; PL/I also has label constants. You must define all file constants with DECLARE statements, as INPUT and OUTPUT are defined in EXAMPLE. (There are some default exceptions, such as SYSPRINT and SYSIN. See the *PL/I Reference Manual*, (093-000204). You define statement labels and entry labels when you write a label prefix ahead of a statement. Label prefixes are PL/I identifiers attached to statements. A colon (:) delimiter separates the prefix from the statement. In EXAMPLE the EXAMPLE:PROCEDURE statement defines EXAMPLE as an entry constant. For more information on entry constants, see Chapter 6; for more information on label constants, see Chapter 11.

Literal Constants

Literal constants are values that you write directly in your program, such as 2.5, 1.2E+20, or "INPUT A WORD". 2.5 and 1.2+10 are arithmetic literal constants; "INPUT A WORD" is a character-string literal constant. EXAMPLE has four character-string literal constants: the AOS file names "@INPUT" and "@OUTPUT" in lines 15 and 16, and "INPUT A WORD" and "HERE IS YOUR WORD" in lines 18 and 22. As the example shows, you write character-string literal constants as character strings set off by quotation marks. PL/I also has arithmetic and bit literal constants. Arithmetic constants are introduced in Chapter 2 and bit-string constants in Chapter 3.

For details on literal constants, see the reference manual.

Delimiters

When you write identifiers and arithmetic constants in your program, you must separate them with delimiters. Delimiters may be spaces, comments, or PL/I symbols such as ( ) ; , * / + -. For a complete list, see the reference manual.

Spaces may be the ASCII blank, horizontal tab, or new line characters. You may separate identifiers and arithmetic constants with any number of spaces and you may insert spaces ahead of or behind any other delimiter. For example, A*B and A * B mean exactly the same thing.

Comments

A PL/I comment is a text set off by the comment delimiters /* and */. The text may contain any characters but /* and */. A comment may extend over several lines. You can insert a comment anywhere you can insert a space. A*B and A/*YOU CAN PUT A COMMENT HERE*/*B mean exactly the same thing.

PL/I Statements

PL/I statements end with the semicolon (;) delimiter. Because the semicolon marks the end of the statement, and because you can insert spaces anywhere you use a delimiter, you may write several PL/I statements on one line or spread one PL/I statement over several lines. Notice how we declare this structure:

```
DCL        1        MAS_STR,
                    2        P_ONE CHAR(20),
                    2        P_TWO CHAR(20),
                    2        PACKAGE CHAR(20);
```

Groups and Blocks

A group is a sequence of PL/I statements that begins with DO and ends with an END statement. There are no groups in EXAMPLE. Chapter 3 discusses non-iterative groups and Chapter 4 discusses iterative groups.

Blocks are segments of a PL/I program that may have their own storage. The block structure of a PL/I program also determines the area of the program in which a declared name has a single meaning. The entire program EXAMPLE is a single procedure block. Procedure blocks begin with PROCEDURE statements and end with END statements. The procedure block is the smallest syntactic unit that can be a program in its own right; all PL/I programs must have at least one procedure block. PL/I also has BEGIN blocks, which begin with a BEGIN statement and end with an END statement. Both PROCEDURE and BEGIN blocks may contain other PROCEDURE or BEGIN blocks. For a full explanation of blocks in PL/I programs, see Chapter 6.

The Statements of EXAMPLE

The PROCEDURE Statement

The PROCEDURE statement, and the END statement on line 22 define the single procedure block which makes up the program EXAMPLE. Since DO groups and BEGIN blocks also end with the END statement, we have added the comment /* EXAMPLE */ after the END statement in EXAMPLE to make it clear which statement the END matches.

When a procedure neither accepts data from other procedures nor returns data to them, the PROCEDURE statement has the form:

```
identifier:PROCEDURE;
```

The identifier preceding the PROCEDURE statement is a label prefix. The label prefix gives the name of the procedure. As you will see later on, when you invoke a procedure from another procedure, you use the label prefix. When you write a label prefix ahead of a PROCEDURE statement, you are also declaring the prefix as a named constant with the ENTRY data type. For details on this and on the syntax of PROCEDURE statements that accept and return data, see Chapter 6.

The DECLARE Statement

The DECLARE statement defines the names of variables and file constants. In AOS PL/I, you must declare all variables and file constants with a DECLARE statement. DECLARE statements tell the compiler how to set up the storage for the program. When your program is executed, the DECLARE statements are ignored. Consequently, you can put the DECLARE statements for a block anywhere in the block. Your programs will be easier to understand if you either put them all at the beginning, as we did in EXAMPLE, or all at the end.

The DECLARE statement for file constants and elementary variables has the form:

```
DECLARE identifier data-type-attribute;
```

The identifier may be any legal PL/I identifier; the data type attribute is a combination of keywords which gives the identifier one of the PL/I data types. The identifiers declared with DECLARE statements in EXAMPLE are the file constants INPUT and OUTPUT and the character varying variable CHARSTRING.

The DECLARE statement for CHARSTRING shows the simplest form of the DECLARE statement. It consists of the identifier followed by its attributes.

```
DECLARE CHARSTRING CHARACTER(20) VARYING;
```

You use two keywords, CHARACTER and VARYING to specify the data type, and CHARACTER is followed by parentheses which contain a value for the maximum length of the string which the variable may contain. In case you forget to give a length for the variable, the compiler will give it a default length of 1. CHARSTRING is declared with a length of 20; if you assign a string which has more than 20 characters to the variable, the string will be truncated from the right.

It is obvious from the way EXAMPLE works that character varying data has character-string values. The difference between character varying data and other types of character-string data is that a character varying variable contains both the value and its current length. Consequently, if the value in the variable is shorter than the maximum length of the variable, the program will use the actual length of the value for assignments, operations, and output. That is why the

```
PUT FILE(OUTPUT) LIST(CHARSTRING);
```

statement outputs a string exactly 3 characters long when we assigned "WOW" to CHARSTRING.

If several identifiers have the same attributes, you can factor the declarations. The DECLARE statement for the file constants INPUT and OUTPUT shows how you do this:

```
DECLARE (INPUT,OUTPUT) FILE;
```

As you can see, you enclose the list of identifiers in parentheses and separate the items in the list with commas.

Input and Output in EXAMPLE

The rest of EXAMPLE consists of statements which set up and perform input and output. PL/I accesses collections of data that are stored outside the computer's memory data sets. A data set may be a device, such as the terminal or a line printer, or an AOS disk file. You represent a data set in a PL/I program with a file constant. As we have seen, you must declare these file constants in a DECLARE statement.

The OPEN Statement

Before you can use a PL/I file constant to represent a data set, your program must associate the file constant with the data set and give the file constant attributes which describe the way in which the program will interpret and use the data in the data set. PL/I terms this *opening* the file. You can open a file in your program with an OPEN statement or with the READ, WRITE, PUT, and GET statements, which transmit data between the data set and the program. When you open a file with the OPEN statement, you open it explicitly; when you open it with one of the input/output statements, you open it implicitly. To attach a file to an AOS generic data set, you must open the file explicitly; hence, we have used the OPEN statement in EXAMPLE.

The OPEN statement looks like this:

```
OPEN FILE(file-exp) file-attributes;
```

The file expression has a file constant as a value. In EXAMPLE, the expressions are simply file constants, but they might also be file variables or functions which return a file value. For details on file variables, see Chapter 7.

The file attributes are keywords which describe how the data is stored in the file, how the file will be accessed, how it will be used in the program, and what data set it is to be associated with. There are three basic file types in AOS PL/I: stream files, record sequential files, and record direct files. The data in the stream files is stored in the form of a stream ASCII characters and can only be accesed sequentially. When a program reads a stream file, it converts the data from ASCII characters; when it writes a stream file, it converts the data to ASCII characters. The data in record files is not converted, but is stored in its internal representation. The data in record sequential files may only be accessed sequentially, while the data in record direct files may be accessed randomly. In the first part of this primer, we will only use stream files; for full information on the other PL/I file types, see Chapter 7.

You use the STREAM keyword in the OPEN statement to indicate that a file is a stream file. You may use STREAM files either as input or output files; when you use a stream file as an input file, the OPEN statement will have the INPUT keyword; when you use it as an output file, the OPEN statement will have the OUTPUT keyword.

When you want to associate the PL/I file with an AOS data set which has a different name from the name of the PL/I file, you have to use the TITLE keyword. The TITLE keyword looks like this:

```
TITLE(char-string-exp)
```

The character-string expression must yield a character string which is the name of an AOS file. When the program opens the file, it will associate the PL/I file with the AOS file. The AOS file must already exist if the program associates it with a PL/I INPUT file; if it does not exist and the program associates it with a PL/I OUTPUT file, the program will create the AOS file.

We use the TITLE keyword in EXAMPLE to associate our PL/I files with the AOS generic dataset for terminal input, @INPUT, and the AOS generic dataset for terminal output, @OUTPUT. These files have special qualities which make them particularly useful for interactive terminal I/O. You must use the TITLE keyword with AOS generic data sets because the " @ " character that distinguishes them is not legal in PL/I identifiers. For details, see the reference manual.

In EXAMPLE, the file INPUT is a stream file associated with the data set "@INPUT" and the file OUTPUT is a stream output file associated with "@OUTPUT". When we put all of the attributes for INPUT together, we get the OPEN statement:

```
OPEN FILE(INPUT) STREAM INPUT TITLE("@INPUT");
```

When we do the same for OUTPUT, we get the OPEN statement:

```
OPEN FILE(OUTPUT) STREAM OUTPUT TITLE("@OUTPUT");
```

PUT LIST

The first statement which actually performs I/O is the PUT LIST statement on line 16. PUT LIST writes data to a STREAM OUTPUT file. Its syntax looks like this:

```
PUT FILE(file-exp) LIST(output-list);
```

The file expression must have as its value a file constant for a file which has been opened as a STREAM OUTPUT file. The output list is a list of variable names or expressions separated by commas. When PL/I executes a PUT LIST statement, it converts the value of each variable or expression to a character string, adds a space, and writes it to the data set associated with the file expression. If the variable or expression has a character-string value, the PUT LIST statement surrounds it with quotation marks before it adds the space (unless the OUTPUT file has the PRINT attribute). Hence, the output of the PUT LIST statement on line 16, whose output list consists of a single character-string constant, looks like this:

```
"INPUT A WORD"
```

The output list of the PUT LIST statement on line 20, on the other hand, has two items, each of which is a character-string value, so the output looks like this:

```
"HERE IS YOUR WORD" "WOW"
```

The PUT LIST statement on line 20 also contains the SKIP keyword. The SKIP keyword makes the PUT LIST statement insert a new line character in the data stream before it begins outputting the data in the output list. As you can see from EXAMPLE, when your output file is a terminal or lineprinter, the new line character makes the output begin on a new line. If you want to skip more than one line, you can use integer values in parentheses following SKIP. For details, see Chapter 7.

GET LIST

The GET LIST statement gets data from a STREAM INPUT file. The statement's syntax looks like this:

```
GET FILE(file-exp) LIST(input-list);
```

The file expression must have as its value a file constant which has been opened as a STREAM INPUT file. The input list is a list of variables. Commas separate the items on the list.

When you use GET LIST to read a file, each item in the file must be followed by a comma or a space. The GET LIST statement reads characters until it finds a comma or a space. When it finds a comma or a space, it converts the characters it has read to the data type of the first variable on the input list and assigns them to the variable. It then reads the next set of characters until it finds a comma or a space, converts the value to the data type of the second variable, and assigns the value to the variable. It continues in this fashion until it has assigned values to all the variables in the list.

For example, if you respond to EXAMPLE's prompt "INPUT A WORD" with GEORGE WASHINGTON, the program will only read GEORGE, since there is only one variable in the input list. If you want it to read GEORGE WASHINGTON, you have to enclose the string in quotation marks.

When you use GET LIST with the AOS generic file @INPUT, you don't have to worry about adding a space or a comma after your last input; the system adds the terminator for you.

Variations

1. To make sure you have understood this chapter, write your own version of EXAMPLE and get it working. Use the same statements, but make up your own declared names, comments, and prompts. You may also want to vary the program's format and the length of the character varying variable. When you get the program finished, play with it. Try different kinds of inputs and see what happens.

2. If you want to get rid of the quotation marks around your program's output, you can add the PRINT keyword to the OPEN statement for your output file. When you open a file as a STREAM OUTPUT PRINT file, the output is automatically formatted for the line printer. Character values are left without quotes and the output of each value begins at the tab stop following the end of the preceding value's output. When you open the file OUTPUT with the PRINT attribute, the OPEN statement looks like this:

   ```
   OPEN FILE(OUTPUT) STREAM OUTPUT PRINT TITLE("@OUTPUT");
   ```

   If you input HEY, the output looks like this:

   ```
   HERE IS YOUR WORD HEY
   ```

End of Chapter

Chapter 2
Data and Its Uses

Data Types: Why They Are Important

Each piece of data that a program processes has a data type. The data type of a piece of data defines what kind of values, the range of values, and what you can do with the values it represents. In the program EXAMPLE, for instance, there were four kinds of data: file constants, which represented data sets; character-string constants with the character data type; a character-varying variable; and the entry constant EXAMPLE. The different kinds of data were not interchangeable; if we had written the GET LIST statement like this:

```
GET FILE(CHARSTRING) LIST(INPUT);
```

it would have been meaningless, since a character varying variable cannot represent a data set and you cannot assign character-string values to file data.

Data Types in PL/I

A piece of data in a PL/I program may have one of 13 different data types. We introduced four of them in EXAMPLE: character varying data, character data, file data, and entry data. In this chapter, we will give more details on the character and character varying data types, and will introduce AOS PL/I's three arithmetic data types: fixed binary, fixed decimal, and float binary. We will discuss operations and built-in functions for arithmetic data and character-string data. We will also introduce you to the conversions that PL/I performs when you use different data types in an operation, or assign a data item of one data type to a variable of another data type.

The following example program TYPES shows some of the data types we will be working with.

```
TYPES:
   PROCEDURE;

   DECLARE OUT FILE;

   /****************************************************************/
   /* INT, FLO, AND DEC ARE VARIABLES OF THE THREE ARITHMETIC     */
   /* DATA TYPES                                                  */
   /****************************************************************/

   DECLARE INT FIXED BINARY;
   DECLARE FLO FLOAT BINARY;
   DECLARE DEC FIXED DECIMAL(4,2);

   /***************************************************************/
   /*   CHAR  IS  A  CHARACTER  VARIABLE                         */
   /***************************************************************/

   DECLARE CHAR CHARACTER(10);

   OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");
```

Figure 2-1. The Example Program TYPES

```
/****************************************************************/
/* ASSIGN THE SAME VALUE TO ALL FOUR VARIABLES.  THE            */
/* VALUE IS A LITERAL ARITHMETIC CONSTANT WITH THE FIXED        */
/* DECIMAL DATA TYPE.                                           */
/****************************************************************/

INT = 3;
FLO = 3;
DEC = 3;
CHAR= 3;

PUT FILE(OUT) LIST(INT,FLO,DEC,CHAR);

/****************************************************************/
/*  USE THE ARITHMETIC VARIABLES AS OPERANDS IN ARITHMETIC      */
/*  OPERATIONS. OUTPUT THE RESULT.                              */
/****************************************************************/

PUT FILE(OUT) SKIP LIST(INT + FLO,INT * DEC,INT / FLO,
                        DEC - FLO);

END;  /* TYPES */
```

Figure 2-1. The Example Program TYPES (continued)

The output for this program looks like this:

```
                3 3.000000E+00 3.00 " 3 "
6.000000E+00 9.00 1.000000E+00 0.000000E+00
```

What happens in TYPES seems simple enough: the program assigns the same arithmetic value, 3, to variables of the three arithmetic data types and to a variable with the character data type. It then outputs the values of the variables. Next, it uses the arithmetic variables as operands in the arithmetic operations + (addition), * (multiplication), / (division), and - (subtraction), and outputs the results of the operations.

If what happens is so simple, why is the output so complicated? The reason is that the single value 3 is converted to three arithmetic and one character data types in the program, and the data type of a piece of data determines how PUT LIST outputs it.

These conversions are a consequence of the data types of the arithmetic literal constant 3 and the variables INT, FLO, DEC, and CHAR. As we already know from EXAMPLE, the DECLARE statement for a variable determines the variable's data type.

INT's DECLARE statement declares it as fixed binary data. Fixed binary data items can only have integer values. The integers are stored as 15-digit binary numbers, so the values cannot be less than -32768 or greater than 32767.

FLO is float binary data. Float binary data is stored as a binary mantissa with a binary exponent, so the values need not be integers. In AOS PL/I, float binary values may range from approximately 5.4E-79 to 7.2E+75 (E denotes the power of 10).

DEC is fixed decimal data. Fixed decimal data is stored as a string of decimal digits with a decimal-point specifier. The numbers in parentheses following the DECIMAL keyword give the precision and scale of the values the variable may contain. Precision is the number of digits in a value. Scale is the number of digits to the right of the decimal point. With a precision and scale of 4,2, DEC may have values ranging from -99.99 to +99.99. The maximum precision for a fixed decimal data item is 16. The scale may range from 0 to the precision.

CHAR is character data. The values of character data are fixed-length strings of ASCII characters. As with character varying data, the number in parentheses following the CHARACTER keyword gives the length of the string. If the string assigned to a CHARACTER variable is longer than the variable's length, the string is truncated from the right; if it is shorter, it is padded to the right with blanks until it has the necessary length.

TYPES assigns all these variables the literal arithmetic constant 3. Like other data items, arithmetic constants have data types. 3 is an integer and is not in a context requiring fixed binary data. It therefore has the fixed decimal data type. Like fixed decimal variables, fixed decimal constants have precision and scale; 3 has a single digit to the left of the decimal place, so its precision is 1 and its scale is 0.

The program uses the assignment statement to assign the constant 3 to the variables. You may use the assignment statement to assign values to elementary variables, arrays to arrays, or structures to structures. When used to assign values to elementary variables, it has the form:

```
variable = expression;
```

For its use with arrays and structures, see Chapter 5.

An expression is a program element that has a value. Expressions may be constants, single variables, or the result of an operation on other expressions. The 3 in the assignment statement, the variables INT, FLO, DEC, and CHAR in the first PUT LIST statement, and the operations INT + FLO and so forth in the second PUT LIST statement are all expressions.

When an assignment statement or a statement such as the GET LIST statement assigns an expression to a variable, it converts the value of the expression to the data type of the variable. Thus, the first assignment statement converts the fixed decimal constant 3 to the fixed binary value 3. The second assignment statement converts the fixed decimal constant 3 to the float binary value 3. The third does not convert 3 to a different data type, but it does change its precision and scale: the constant has a precision of 1 and a scale of 0, but the variable has a precision of 4 and a scale of 2, so the 3 is converted to a value with this precision and scale. The fourth statement assigns the arithmetic constant to a character variable. The constant is therefore converted from a number into a string of ASCII characters.

We can see the effects of these conversions in the first line of the output. The output from the first variable in the output list, INT, is the integer 3. This is what we would expect, but the reality is more complicated than the appearance: the PUT LIST statement converts all the data it outputs into ASCII characters; according to the rules for the conversion of fixed binary data into ASCII characters, fixed binary data is output as decimal integers.

The second output makes the conversion which took place on assignment clear: the 3 has become 3.000000E+00. This has happened because it was converted to a float binary value on assignment, and when float binary values are converted to ASCII characters, they are written in scientific notation. 3.000000E+00 is 3 multiplied by 10 to the 0.

The third output shows how the value 3 was given a new precision and scale when it was assigned to DEC. It remains 3, but since the variable has the precision and scale 4,2, it is stored as the decimal number 03.00. When PUT LIST converts fixed decimal data items to ASCII characters, it suppresses leading 0's, but retains the 0's after the decimal point, so 03.00 becomes 3.00.

The fourth output shows the conversion to a character string. The CHAR = 3; statement converts the arithmetic value 3 to a string of ASCII characters. The rule for conversion of integers to character strings states that the character string will consist of the digits of the integer plus three blanks on the left. When this four-character string is asigned to CHAR, it is padded with six more blanks on the right to give it the length 10 required by CHAR's declaration. The quotation marks in the output, "□□□3□□□□□□", show that the 3 is a character- string value and not an arithmetic value, and the position of the 3 in relation to the quotation marks show how the padding has been added.

The second line of output comes from the second PUT LIST statement. This statement outputs the values resulting from the operations INT + FLO, INT * DEC, INT / FLO, and DEC - FLO. Now that we know what the different output forms are, we can see that conversions also take place when you perform operations with data items of different data types.

The output for the first operation, INT + FLO, indicates that the result is a float binary value. Apparently, the fixed binary value 3 contained in INT was converted to a float binary value before it was added to FLO. The output for the second operation, INT * DEC, indicates that the fixed binary value was converted to a fixed decimal value before it was multiplied, while the outputs of the third and fourth operations indicate that the fixed binary value in INT and the fixed decimal value in DEC were both converted to float binary values before the operation was performed.

Although the conversions PL/I performs are complicated, they are also completely automatic. You need concern yourself with them only if there is a chance that the conversion may not be possible or that it will change the values involved. Later, we will deal with situations where either might occur.

Arithmetic Data Types

TYPES has already introduced you to the three arithmetic data types in AOS PL/I. This section explains how you write constants and declare variables of these data types, and gives more details concerning precision, scale, and the range of values for each data type.

Precision and Scale with Arithmetic Data

We have already seen from TYPES what precision and scale mean with fixed decimal data: the precision of a fixed decimal value is the total number of digits it contains; its scale is the number of these digits that are to the right of the decimal place.

Fixed binary and float binary data also have precision, but here the situation is more complicated. The precision of a fixed binary data item is the number of binary digits the value requires. Thus, 8, which has a decimal precision of 1, has a binary precision of 4. (1000 is the binary equivalent of 8.) The precision of a float binary data item is the number of binary digits in the mantissa.

The precision of a fixed binary or float binary data item also depends on what is being done with it: the same value will have a computational precision, which determines how many digits are involved in computation, and a conversion precision, which determines how many digits are involved when the value is converted to a character string.

To see what this can mean, take a look at the following:

```
PRECS:
          PROCEDURE;

          DECLARE LARGE FIXED BINARY(15);

          DECLARE SMALL FIXED BINARY(2);
          DECLARE OUT FILE;

          OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

          SMALL = 255;
          LARGE = SMALL;
          PUT LIST(SMALL,LARGE);

END; /* PRECS */
```

The output for this program is 3 255. The output is a consequence of the difference between computational and conversion precision. Like all fixed binary data items, SMALL and LARGE both have the same computational precision of 15 binary digits; they therefore can take exactly the same range of values. Consequently, you can assign 255 to SMALL and then assign SMALL to LARGE without losing any of the value. However, it is the conversion precision, and not the computational precision, which determines how many digits of the value will be converted to a character string. SMALL has a conversion precision of 2, so only the 2 rightmost are converted. These digits are 11, and the decimal value of the binary number 11 is 3.

As you can see from the example, the precision you give when you declare a fixed binary or float binary variable is the variable's conversion precision; if you always declare your variables with conversion precisions that are the same as their computational precisions, you will avoid the kinds of problems illustrated by the example.

Fixed Binary Data

You declare fixed binary variables with the FIXED, BINARY, or FIXED BINARY keywords. All fixed binary variables have a computational precision of 15 binary digits, and the default precision is 15. Hence, you should declare fixed binary variables without a precision attribute.

Examples of declarations:

```
DECLARE INT FIXED BINARY;
DECLARE WHOLE_NUMBER FIXED(3);
DECLARE INT_VAL BINARY(10);
```

You write fixed binary constants as decimal integers. The compiler interprets the integers as fixed binary data if they appear in contexts which require fixed binary values or in operations with other fixed binary data items.

For example, in the expression INT + 5, 5 is interpreted as a fixed binary value because it is added to a fixed binary variable. In the expression 1/3, on the other hand, 1 and 3 are interpreted as fixed decimal values.

The fact that fixed binary values are stored as 15-digit binary integers has consequences for assignments to fixed binary variables, conversions of items of other data types to fixed binary data, and expressions with fixed binary operands.

1. If your program assigns a value that has places to the right of the decimal point to fixed binary variable, the value will be truncated from the right. For instance,

   ```
   INT = 0.99
   ```

   will assign the value 0 to INT.

2. If your program assigns values larger than 32767 or smaller that -32768 to fixed binary variables, or if it converts values outside these limits to fixed binary data, the program is in error. Note that AOS PL/I cannot detect such errors. For instance:

   ```
   DECLARE INT FIXED BINARY;
             .
             .
             INT = 10;
             .
             .
             INT = INT + 32767;
   ```

   The expression INT + 32767 produces a value larger than the fixed binary data type can take. The program will perform the operation, but INT will have the value -32759.

Fixed Decimal Data

You use the DECIMAL or FIXED DECIMAL keywords to declare fixed decimal variables. You specify the precision and scale in parentheses following the keywords. If p is the precision and q the scale, then the precision specifier must look like this: (p,q). p and q must be integers. The precision can range from 1 to 16 and the scale can range from 0 to the precision. If you do not specify precision and scale, the compiler assigns a default precision and scale of 7,0.

Examples of declarations and the range of values the variable may contain:

Declaration	**Range of Values**
DECLARE DEC FIXED DECIMAL(4,2);	-99.99 +99.99
DECLARE SMALL DECIMAL(6,6);	-.999999 +.999999
DECLARE DEFAULT FIXED DECIMAL;	-9999999 +9999999

You write fixed decimal literal constants as decimal numbers. The compiler interprets decimal integers as fixed decimal data unless the context requires a fixed binary value; decimal numbers with the decimal point are always fixed decimal data. As we have seen, the form of the fixed decimal constant determines its precision and scale; 3 had a precision and scale of 1,0; 3.14159 has precision and scale of 6,5; 1000.01 hass a precision and scale of 6,2. As with variables, the maximum precision is 16; a fixed decimal constant therefore cannot have more than 16 digits.

Because the precision and scale of fixed decimal variables determine the range of values the variable can represent, you have to make sure that a variable's precision and scale will allow it to represent all the values that may be assigned to it in your program. If you assign a value to a fixed decimal variable that has more places to the left or right of the decimal place than the precision and scale of the variable allow, the value will be truncated. Extra places to the right of the decimal place are truncated from the right; extra places to the left are truncated from the left. Consequently, if the precision and scale are wrong, you may lose the most significant digits of your value.

Example:

```
PRECISIONS:
          PROCEDURE;

          DECLARE OUT FILE;
          DECLARE SMALL FIXED DECIMAL(,2);
          DECLARE LARGE FIXED DECIMAL(7,3);

          OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

          LARGE = 99.99;
          SMALL = LARGE;
          PUT FILE(OUT) LIST(SMALL);
          LARGE = 999.999;
          SMALL = LARGE;
          PUT FILE(OUT) LIST(SMALL);
END; /* PRECISIONS */
```

The output for this program is 99.99 99.99. As the example shows, PL/I does not check whether the variable on the left-hand side of the assignment statement has a precision and scale sufficient to contain the range of values of the expression on the right-hand side. When the assignment takes place, the excess digits are simply truncated. When LARGE is assigned to SMALL the first time, SMALL gets the value 0099.990. Since the truncated digits are 0's, the value does not change. When LARGE is assigned to SMALL the second time, however, the two most significant digits are lost.

If you are running AOS PL/I on an ECLIPSE-line machine with the Commercial Instruction Set and include the PL1ECIS file when you bind your program (see Chapter 6 for how to do this), the ERROR condition will arise if an expression produces a fixed decimal result with a precision larger than 16; if you have not included the PL1ECIS file, the system will not detect the error. AOS PL/I detects errors in all cases involving conversions from other data types which produce fixed decimal values with a scale larger than 16.

Float Binary Data

You declare float binary variables with the FLOAT and FLOAT BINARY keywords. Parentheses containing an integer precision may follow the keywords. The value of the integer determines both the variable's computational precision and its conversion precision.

Float binary variables may have one of two computational precisions: in single-precision variables, the mantissa has a precision of 21 binary digits; in double-precision variables, it has a precision of 53 binary digits.

If you declare a float binary variable with a precision less than or equal to 21, it is a single-precision variable. If the precision is greater than 21 but less than or equal to 53, it is a double-precision variable. Precisions less than 21 or 53 only determine how many digits of the mantissa are converted to a character string. As with fixed binary variables, the best way to avoid difficulties with the difference between computational and conversion precisions is to declare float binary variables with no precision attribute at all, which gives you the default single precision of 21, or declare them with 53, which gives double precision.

The following program gives examples for the declaration of float binary data. It also shows how computational precision and conversion precision work:

```
FLOAT_PREC:
            PROCEDURE;

            DECLARE OUT FILE;

            DECLARE S_PREC_1 FLOAT BINARY(4);
            DECLARE S_PREC_2 FLOAT;
            DECLARE D_PREC FLOAT(53);

            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

            S_PREC_1 = 1/3;

            S_PREC_2 = S_PREC_1;
            D_PREC = S_PREC_2;

            PUT LIST(S_PREC_1, S_PREC_2, D_PREC);

END; /* FLOAT_PREC */
```

The output is 3.3E-01 3.333333E-01 and 3.333333134651184E-01. As the assignment of S_PREC_1 to S_PREC_2 shows, both variables have the same computational precision; however, since S_PREC_1 has a conversion precision of 4, the PUT LIST statement only converts the first four digits of the mantissa to a character string. The assignment of S_PREC_2 to D_PREC shows the difference precision can make in a value: since the mantissa of S_PREC_2 is shorter than that of D_PREC, PL/I adds the extra binary digits when it converts the value to double precision.

You write float binary constants with scientific notation. The mantissa is a decimal fraction. It is followed by E and then a positive or negative number for the exponent:

Float Binary Constant	Decimal Value
3.55E+2	355
3.33E-2	.0333

All float binary constants have double precision.

Conversions with Arithmetic Data Types

As we saw in TYPES, PL/I automatically converts a value of one arithmetic data type to a value of another arithmetic data type when your program requires it. The conversions take place when you assign a value of one arithmetic data type to a variable of another arithmetic data type, or when you use operands of different data types in arithmetic operations.

While the conversions can be quite complicated, you need worry about them only when values might be lost in the process. This can happen:

1. If you assign a value with digits to the right of the decimal point to a fixed binary variable, the digits to the right will be truncated.
2. If you assign to a fixed binary variable a value smaller than -32768 or larger than 32767, you will lose the value.
3. If you assign to a fixed decimal variable a value with a scale larger than that of the variable, the rightmost digits will be truncated.
4. If you assign to a fixed decimal variable a value that is larger than the precision scale of the variable allowed, the leftmost digit will be truncated.
5. If you assign a double-precision float binary value to a single-precision float binary value, the rightmost binary digits of the mantissa will be truncated.
6. If you asign a value to a float binary variable and the value is larger or smaller than the computation precision allowed by the variable, you will raise the error condition. For details on the error condition, see Chapter 13.

Arithmetic Operations

PL/I has the following arithmetic operators:

Symbol	Operation
+ with one operand	unary +
- with one operand	unary -
+ with 2 operands	addition
- with 2 operands	subtraction
*	multiplication
/	division
**	exponentiation

The data items you use with operators are called operands. In A * B, the variables A and B are the operands. With arithmetic operations, the operands must be arithmetic data items. PL/I does not convert operands of other data types when they appear in arithmetic operations.

In general, the arithmetic operations work the way you would expect them to. The unary + and the unary - have one operand; the result of the unary + operator has the same data type, precision, scale, and sign as its operand; the result of the unary - operator has the same data type, precision, and scale, but the value has the opposite sign:

```
DECLARE VAL FIXED DECIMAL;
.
.
.
VAL = -20;
.
.
.
```

+VAL will have the value -20; -VAL will have the value +20.

Addition, subtraction, and multiplication have as their results the sum of the operands, the difference of the operands, and the product of the operands. With all these operations, PL/I follows certain rules of precedence. See the section *Subexpressions and Precedence* in this chapter.

Since division of integers may produce non-integer results, you cannot use / when both operands are fixed binary data; instead, you must use the DIVIDE built-in function, which we explain below. As you might expect, neither / nor DIVIDE will work if you divide by 0.

The PL/I expression to raise a value X to a power Y is X ** Y. Note that the operands with exponentiation are evaluated from right to left. PL/I first evaluates Y, then it evaluates X, and then it evaluates X** Y. In general, exponentiation produces a float binary result; for exceptions, see the reference manual. PL/I defines the results for cases of exponentiation involving negative values, zero, or negative exponents as follows:

1. If the value being raised to a power is 0 and the exponent is greater than 0, the result is 0.

2. If the value being raised to a power is not equal to 0 and the exponent is 0, the result is 1.

3. If the value being raised to a power is 0 and the exponent is less than or equal to 0, the program raises the error condition.

4. If the value being raised to a power is negative and the exponent is not a fixed binary or fixed decimal integer, the program raises the error condition.

Examples:

```
DECLARE VAL FIXED DECIMAL(8,2);
DECLARE EXP FIXED DECIMAL(8,2);
DECLARE RES FLOAT BINARY(53);

VAL = 0;
EXP = 22;
RES = VAL ** EXP;
```

Result = 1

```
VAL = 108;
EXT = 0;
RES = VAL ** EXP;
```

Result = 0

```
VAL = 0;
EXP = -10;
RES = VAL ** EXP;

VAL = -5;
EXP = 2.5;
RES = VAL ** EXP;
```

Error Conditions

Conversions with Operations

If the operands with addition, subtraction, multiplication, and division have different data types, AOS PL/I converts them to a common data type. The following table shows how these conversions work. With exponentiation, the operands are converted to the float binary types, with some exceptions. For details, see the *PL/I Reference Manual,* (093-000204).

Operand Type	**Operand Type**	**Common Type**
fixed binary	fixed decimal	fixed decimal
fixed binary	float binary	float binary
fixed decimal	float binary	float binary

Note that AOS PL/I differs in this respect from standard PL/I, where the common type for fixed decimal and fixed binary is fixed binary.

The compiler returns the following error message when it encounters an expression requiring a non-standard conversion:

```
ERROR PL/I_0162 SEVERITY 2 BEGINNING ON LINE --
MIXED DECIMAL AND BINARY OPERANDS ARE CONVERTED TO DECIMAL.
STANDARD PL/I WOULD CONVERT THEM TO BINARY. WRITE THE DECIMAL
BUILT-IN FUNCTION AROUND THE BINARY OPERAND TO CONFORM TO
STANDARD PL/I.
```

Unless you are converting a standard PL/I program which will not work unless fixed binary is the common type, you can ignore this message.

The result of the operation will have the common data type. PL/I automatically gives the result data type a precision and scale large enough to take all values that might result from the operation. The only limitation is that the precision and scale of the result cannot exceed the implementation limits for the data type.

Subexpressions and Precedence

As you might expect, operands of arithmetic operations may be other arithmetic expressions:

```
DECLARE (INT_1, INT_2) FIXED BINARY;
DECLARE DEC FIXED DECIMAL(6,2);
DECLARE FLO FLOAT BINARY;

FLO = INT_1 * INT_2 + FLO/DEC;
```

In the above example, INT_1 * INT_2 + FLO/DEC is a single expression which contains the subexpressions INT_1 * INT_2 and FLO / DEC. These subexpressions in turn each contain two subexpressions, namely the variables INT_1, INT_2, FLO, and DEC. These subexpressions have no further subexpressions, so they are termed primitive expressions. The primitive expressions in PL/I are unsigned literal constants, variables, and function references.

When PL/I evaluates an expression containing subexpressions, it follows certain rules of order. In our example, it performs the multiplication and division before it performs the addition. Since addition, multiplication, and division are all performed from left to right, the exact order of evaluation of the subexpressions is:

```
INT_1, INT_2, INT_1 * INT_2, FLO, DEC,
FLO / DEC, INT_1 * INT_2 + FLO / DEC.
```

The order of precedence is as follows:

• Operations peformed first:	unary +, unary -, exponentiation
• Operations performed second:	multiplication, division
• Operations performed last:	addition, subtraction

If two operations have different levels of precedence, the one with the highest is performed first.

If two operations other than unary +, unary -, and exponentiation have the same level of precedence, the leftmost operation is performed first. In our example, the * operation is therefore performed before the / operation.

If there is more than one exponentation operator or a unary operator and one or more exponentiation operators, the operations are performed right to left.

Subexpression operands are converted the same way primitive expression operands are converted. In our example, INT_1 and INT_2 both have the fixed binary data type,, so INT_1 * INT_2 has a fixed binary result. DEC is a fixed decimal and FLO is a float binary, so DEC / FLO has a float binary result. Since the two operands of the + operation have the fixed binary and float binary types, the whole expression has the float binary type.

Parentheses

You can use parentheses to specify the order in which you want subexpressions to be evaluated. When PL/I evaluates expressions in parentheses, it starts with the innermost set of parentheses, evaluates the expression they contain according to the rules just given, evaluates the expression which contains the parenthesized expression and so forth until it has evaluated the entire expression. If there are not enough parentheses to completely determine how the expression is to be evaluated, PL/I evaluates the part of the expression for which parentheses are lacking according to the general rules of precedence.

For example, if we used parentheses to obtain the same results we obtained through precedence above, our expression would look like this:

```
FLO = ((INT_1 * INT_2) + (FLO /DEC));
```

If we wanted to add INT_2 to FLO, multiply the result by INT_1, and divide the product by DEC, we could specify this order with parentheses as follows:

```
FLO = ((INT_1 * (INT_2 + FLO)) / DEC);
```

Errors Resulting from Arithmetic Operations

If the result of an operation does exceed the implementation limits for the data type, your program is in error. What happens depends on the data type of the result and the kind of computer you have.

1. If an operation has fixed binary operands and the result exceeds the implementation limits, the error is not detected.

2. If the operation has fixed decimal operands and the result has a precision larger than 16 digits, the error will not be detected unless you have bound your program with the PL1ECIS file. If you have done so, the program will raise the error condition.

3. If the operation has float binary operands and the result exceeds the implementation limits, the program will raise the error condition.

When your program raises the error condition, PL/I will stop program execution, close all files, issue a run_time error message, and return to the CLI. You can also write ON units which tell the computer what action to take when the condition arises. ON units are discussed in Chapter 13.

Usually, you will have no trouble figuring out which operations in your program might produce results too large or small for fixed binary or float binary data to handle. However, the conversion rules for fixed decimal data sometimes produce unexpected results. For example:

```
DEC_OVERFLOW:
            PROCEDURE;

            DECLARE FULL FIXED DECIMAL(4,2);
            DECLARE OUT FILE;

            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

            FULL = 10.5;
            PUT FILE(OUT) LIST(FULL + 1/2);
END; /* DEC_OVERFLOW */
```

This program seems straightforward enough: if FULL is 10.5, then FULL + 1/2 will be 11. The output is, however, 1.000000000000000. What happened? Arithmetic operations in PL/I are performed according to the rules of algebra; consequently, the program divided 1 by 2 before it added the result to FULL. The divide operator always gives the maximum scale possible. In this case, the result is a fraction, so it has the precision and scale 16,16. When a value with a 16,16 precision and scale is added to a value with a precision and scale of 4,2, the result has a precision and scale of 19,16, which exceeds the implementation limit. You can avoid the problem simply by replacing 1/2 with .5.

Built-in Functions

Built-in functions are routines that are supplied to you with your AOS PL/I software. The routines perform tasks such as calculating various mathematical functions, converting from one data type to another, and returning information about the data in your program.

To use a built-in function, you write the function name along with the values you want the function to work with.

When the computer executes the built-in function, it returns the calculation result to the function name. The function name (or function reference) is thus an expression. You can use function references just like any other expression.

To see how built-in functions work, let's take a look at the DIVIDE and ROUND mathematical functions and the FIXED conversion function.

The DIVIDE function lets you get around the problems posed by the / operator. You can use DIVIDE with any kind of arithmetic data. The result will have the common type of the operands. DIVIDE also lets you specify the precision and scale of the result. For example, we could use it with fixed binary operands like this:

```
DECLARE (DIVIDEND,DIVISOR,QUOTIENT) FIXED BINARY;

DIVIDEND = 4;
DIVISOR = 2;

QUOTIENT = DIVIDE(DIVIDEND,DIVISOR,15);
```

In this example, the function would, of course, return a fixed binary value 2, with a precision of 15 binary digits.

You can also use DIVIDE to solve the precision and scale problems we encountered with the expression FULL + 1/2. FULL had the precision and scale 4,2, and if we wanted the result of 1/2 to have the same precision and scale, we could write our expression like this:

```
FULL + DIVIDE(1,2,4,2);
```

The first two numbers are of course the dividend and divisor; the second two give the precision and scale of the result.

The items in parentheses following the name of a built-in function are the function's arguments. You write the arguments as a list of expressions. Commas separate the members of the list from each other. In general, you can use any expression that has the proper data type as an argument, but there are some limitations. For example, with DIVIDE, the dividend and divisor may be any expression that has an arithmetic value, but you must use integer constants for the precision and scale. For details on the data types of the arguments for a given built-in function, see the *PL/I Reference Manual,* (093-000204).

The ROUND built-in function rounds an arithmetic value to a given precision and scale. The function takes two arguments. The first is the arithmetic expression to be rounded. If the expression is a fixed binary or fixed decimal, the second argument is a signed integer constant indicating the position within the first argument to be rounded. The position immediately left of the decimal is position 0. Two positions left of the decimal is position -1. The position immediately right of the decimal is position 1. If the expression in the first argument is float binary, the exponent is first subtracted from the second argument and the result indicates the position to be rounded in the mantissa. If we wanted to use ROUND to round the result of a fixed decimal calculation, we could do it like this:

```
DECLARE PRINCIPAL FIXED DECIMAL(10,2);
DECLARE RATE FIXED DECIMAL(4,4);
DECLARE TIME FIXED BINARY;
DECLARE INTEREST FIXED DECIMAL(10,2);

INTEREST = ROUND(PRINCIPAL * RATE * TIME,2);
```

This rounds the result of the calculation to a precision and scale of 16,2. The example also shows how you can use an expression containing subexpressions as an argument.

The DECIMAL conversion function converts fixed binary or string values to fixed decimal values. You cannot use the function with float binary values. The function's first argument is the value to be converted; its second and third arguments are integer constants that represent the precision and scale of the fixed decimal value it is to be converted to.

You use this function anytime you want to force a conversion to the fixed decimal data type. For example, if a division with fixed binary operands has a fractional result, you can use neither DIVIDE nor the / operator. However, if you use DECIMAL to convert one of the operands to a fixed decimal value, you can use either operator. For instance:

```
DECLARE(INT_1,INT_2) FIXED BINARY;
DECLARE QUOTIENT FIXED DECIMAL(4,2);

INT_1 = 2;
INT_2 = 4;
QUOTIENT = DIVIDE(DECIMAL(INT_1,4), INT_2, 4,2);
```

The result here will be 00.50, since it is fixed decimal and has a precision and scale of 4,2. This example also shows how you can use one built-in function as an argument in another built-in function.

Putting it All Together: A Program to Calculate Simple Interest

To see how the things we have discussed thus far fit together, let's make a PL/I program which asks the user for the principal, rate, and number of years for a loan, calculates the simple interest, and returns the amount of interest to the user. The calculation itself is certainly no problem:

```
interest = principal * rate * time
```

Getting the data and outputting the result pose no problems either. To get the data, you use GET LIST with the variables for principal, rate, and time, and to output the result, you use PUT LIST.

In fact, the only thing that requires any thought at all is declaring our variables. As for their names, let's call them what they are: INTEREST, PRINCIPAL, RATE, and TIME. Since we are dealing with money and bankers don't throw away the change, INTEREST, PRINCIPAL, and RATE should all have the fixed decimal data type. What kind of precision and scale should they have? RATE will of course always have fractional values, so its scale will be as large as its precision. Bankers usually figure loans in quarters of a percent, or .0025, so a precision and scale of 4,4 should serve our purposes. The scale of INTEREST and PRINCIPAL is obviously 2; the precision will depend on the size of the loans the program is to handle. Let's assume that the maximum loan will be $99,999.99 and give them a precision and scale of 7,2. TIME is the number of years and will therefore always have integer values less than 32,767. Consequently, the fixed binary data type will serve perfectly well.

We will also have to use ROUND in the program to keep PL/I from truncating values when it assigns them to fixed decimal variables with smaller scales.

The program looks like this:

```
SIMP_INT:
            PROCEDURE;

            DECLARE (PRINCIPAL,INTEREST) FIXED DECIMAL(7,2);
            DECLARE RATE FIXED DECIMAL(4,4);
            DECLARE TIME FIXED BINARY;

            DECLARE (IN,OUT) FILE;

            OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

            PUT FILE(OUT) LIST("INPUT PRINCIPAL,RATE,AND TIME");
            GET FILE(IN) LIST(PRINCIPAL,RATE,TIME);

            INTEREST = ROUND(PRINCIPAL * RATE * TIME,2);

            PUT FILE(OUT) LIST("THE INTEREST IS",INTEREST);
END; /* SIMP_INT */
```

You may want to get your own version of this program working. Try some variations, like separate prompts and GET statements for the principal, rate, and interest, or adding a variable for the number of payments and then calculating the amount of each payment as well as the total interest.

Character-string Data

We have already encountered two kinds of character-string data: character-string constants and character varying variables. PL/I also has two other kinds of character-string variables: character variables and aligned character variables. We will talk about character variables here; aligned character variables are discussed in Chapter 10.

You declare character variables with the CHARACTER keyword. An expression in parentheses following the keyword gives the length of the variable:

```
DECLARE STRING CHARACTER(10);
```

STRING has a length of 10 ASCII characters. If the declaration does not give a length, the default length is 1.

The difference between character variables and varying character variables is that the length of character variables is fixed. If you assign a value to a character variable which is longer than the length of the variable, the value will be truncated from the right; if you assign a value to the variable which is shorter than the length of the variable, it will be padded with blanks from the right.

To see how this works, replace the character varying variable in your version of EXAMPLE with a character variable. If we had declared CHARSTRING like this in our example:

```
DECLARE CHARSTRING CHARACTER(20);
```

and input "WOW", our output would have looked like this:

```
"HERE IS YOUR WORD" "WOW                                   "
```

Since the value we assigned to the variable was only 3 characters long, while the variable has a length of 20, PL/I added 17 blanks.

The Character-string Operator !!

PL/I has one character-string operator. The !! operator converts its operands to character-strings and then attaches the second character string to the first. This operation is called concatenation. Take the following as an example:

```
CONCATENATE:
            PROCEDURE;

            DECLARE VARSTRING CHARACTER(10) VARYING;
            DECLARE FIXSTRING CHARACTER(10);
            DECLARE DECNUM FIXED DECIMAL(3,2);

            DECLARE OUT FILE;

            OPEN FILE(OUT) STREAM OUTPUT TITLE ("@OUTPUT");

            VARSTRING = "NUMBER IS;"
            FIXSTRING = "THIS;"
            DECNUM = 3.14;

            PUT FILE(OUT) LIST(FIXSTRING !! VARSTRING !! DECNUM);

END; /* CONCATENATE */
```

The output of this program is "THIS NUMBER IS 3.14". As the space THIS and NUMBER shows, when you use a character ariable with !!, the variable's padding is included in the result string; when you use a character varying variable, you only get the actual number of characters assigned to the variable. The 3.14, finally, shows how !! converts arithmetic operands to character strings.

One of the common uses of !! is writing long character-string constants in your source text. Unless you use !!, you cannot write a character- string constant which extends past the end of a line; with !!, you can write constants of any length, although you can only output character- string values of up to 132 characters long.

For instance, if you wanted to give a complete set of directions to the user of the simple interest program, you could write your prompt like this:

```
PUT FILE(OUT) LIST("THIS PROGRAM CALCULATES SIMPLE"!!
                   "INTEREST. INPUT YOUR5 PRINCIPAL, THE"!!
                   "RATE OF INTEREST AS A DECIMAL FRACTION,"!!
                   "AND THE NUMBER OF YEARS AS AN INTEGER.");
```

The output will look like this:

```
"THIS PROGRAM CALCULATES SIMPLE INTEREST. INPUT YOUR PRINCIPAL, THE RATE
OF INTEREST AS A DECIMAL FRACTION, AND THE NUMBER OF YEARS AS AN INTEGER."
```

Try using character-string variables and the concatenate operator to personalize your interest program. One way to do it would be to ask for the borrower's name at the beginning of the program and then output the name and a message along with the amount of the payments.

PL/I also has a number of built-in functions wnich take character-string arguments. For details about these, and about string-handling in general, see Chapter 8.

Conversions Between Arithmetic and Character-string Data Types

As you have seen, PL/I can convert arithmetic data to character-string data and vice-versa. These conversions are important because GET LIST converts ASCII characters to other data types when it reads an input file, and PUT LIST converts data of other data types to ASCII characters when it writes data to an output file.

The rules for converting arithmetic data to character-string data are simple: with fixed decimal and fixed binary values, the arithmetic value is converted to a character string that is the value's representation as a fixed decimal number.

If the value is negative, it is preceded by a - sign; if it is fixed decimal and has a scale greater than 0, it will have a decimal point. With float binary values, the arithmetic value is converted to a character string which is the value's representation in scientific notation. Remember that it is the conversion precision and not the computational precision of a fixed binary or float binary variable which determines how many digits of the variable are converted to a character string. If the variable's conversion precision is less than the precision of the value it contains, the conversion to the character string will produce inaccurate results.

Conversion from character-string data to arithmetic data is only possible if the character string contains nothing but characters that can be interpreted as arithmetic values. Hence, PL/I will convert the strings "-3", "+212", "48.62", and "-3.50E+3", but not "$22.95". Note that the string can contain blanks, as long as the blanks do not fall between characters in the value. For example, "   45   " will work.

As with arithmetic constants, the data type of the converted value depends upon its form and context. Integers are either fixed binary or fixed decimal values, depending on the variable they are assigned to; decimal values with the decimal point are fixed decimal; values written in scientific notation are float binary. If the values are assigned to an arithmetic variable of a different data type, they are converted.

If PL/I cannot perform a character-string to arithmetic conversion when it executes your program,the error condition arises.

End of Chapter

Chapter 3
Conditional Branching in PL/I

A program branches when it executes its statements non-sequentially. If a program always executes a branch, the branch is unconditional; if the program may execute a branch or not, depending on the condition of its data, the branch is conditional. PL/I has statements for both unconditional and conditional branches.

The statements for unconditional branches are the GOTO, which branches to a labeled statement, and the CALL, which branches to a procedure block. We discuss the CALL statement with procedure blocks in Chapter 6.

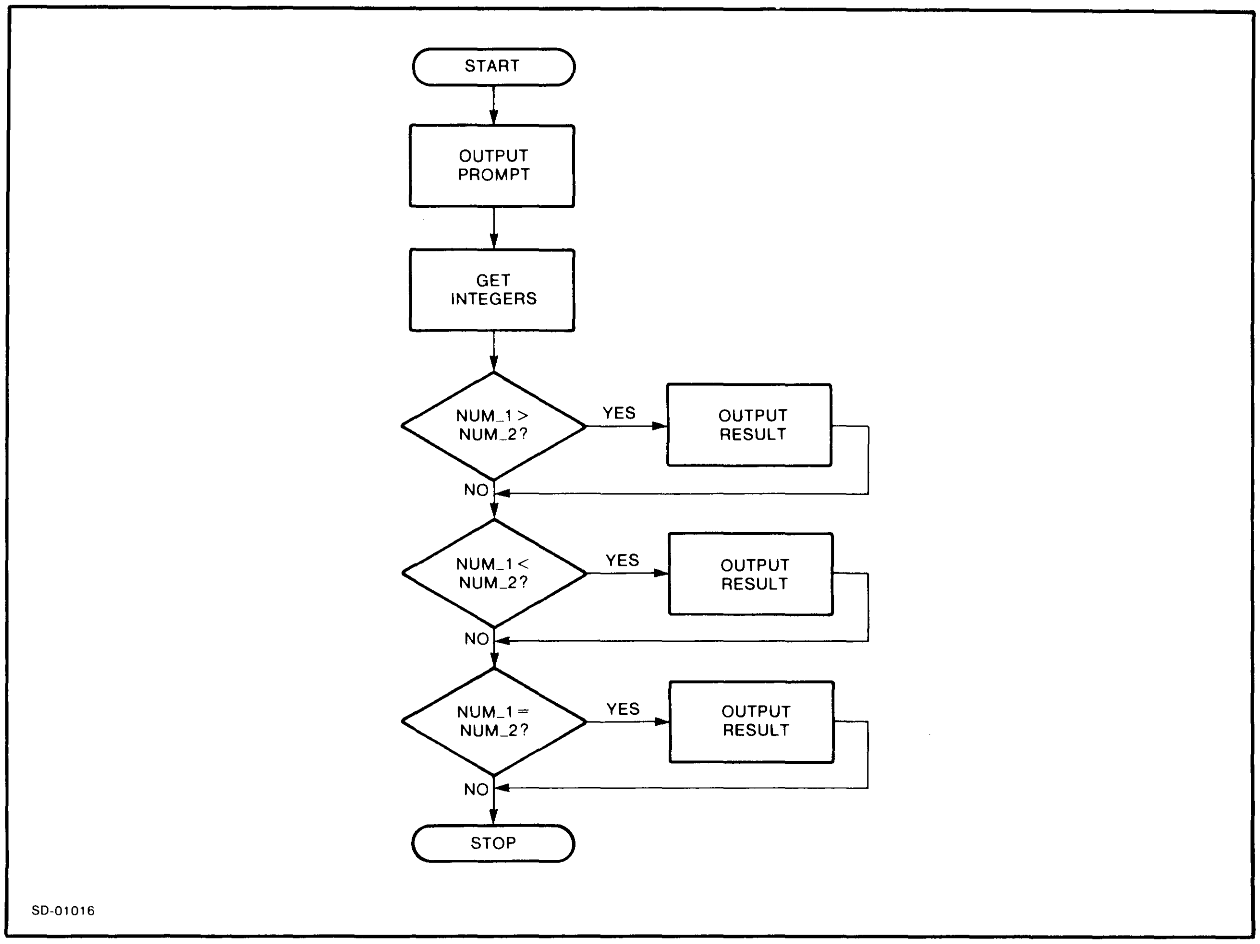
PL/I's branching and looping constructs minimize the need for the GOTO; consequently, we discuss it in Chapter 11 of the primer.

In this chapter, we will take a look at AOS PL/I's two conditional branching statements, the IF THEN [ELSE] statement and the DO CASE [OTHERWISE] statement. Since the IF THEN statement uses relational operators, bit-string expressions, and bit-string operators, we will also introduce you to bit-string data and the relational and bit- string operators.

To see how a conditional branch works, let's look at a program that accepts two integers and tells us whether they are equal and if not, which of them is greater.

```
COMPARE:
            PROCEDURE;

            DECLARE (NUM_1,NUM_2) FIXED BINARY;
            DECLARE (IN,OUT) FILE;

            OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

            PUT FILE(OUT) LIST("INPUT TWO INTEGERS");
            GET FILE(IN) LIST(NUM_1,NUM_2);

            IF NUM_1 >NUM_2
                      THEN PUT FILE(OUT) LIST("NUM_1 IS BIGGER THAN NUM_2");

            IF NUM_1 <NUM_2
                      THEN PUT FILE(OUT) LIST("NUM_2 IS BIGGER THAN NUM_1");

            IF NUM_1 = NUM_2
                      THEN PUT FILE(OUT) LIST("NUM_1 = NUM_2");

END; /* COMPARE */
```

The three IF THEN statements do the work in this program. Each of the statements tests one of the three possible relationships between the integers; if the relationship is true, the clause following the THEN is executed; otherwise, the clause is not executed. In either case, the program then executes the statement following the IF THEN statement. Since you need to know how the flow of control works with IF THEN to understand the rest of this discussion, we've included a flow chart for COMPARE.



IF THEN

The formal syntax of IF THEN looks like this:

```
IF exp THEN statement;
```

As you can see, the IF THEN statement has two parts: an expression following the IF keyword and a statement following the THEN keyword. The expression following the IF has to have a value that the computer can evaluate as true or false; the statement following the THEN is called a clause. The clause must contain a single PL/I statement. If the expression is true, the statement is executed; if it is false, the program executes the statement following the IF THEN statement. We will explain later in this chapter what kinds of expressions the computer can evaluate as true or false; for now, the only expressions we will use will be simple relational operations like the ones in COMPARE.

Relational operators compare their operands with each other. If the relation expressed by the operator is true, the operation is true; otherwise, it is false. PL/I has the following relational operators:

<	less than
^<	not less than
<=	less than or equal to
>	greater than
^>	not greater than
>=	greater than or equal to
=	equal
^=	not equal

Note: The not operator ^is either a shift-N or a shift-6, depending upon your keyboard.

For the time being, we will restrict ourselves to using these operators with arithmetic operands; we will explain their use with other operands later in this chapter.

IF THEN ELSE

If you look at the flowchart for COMPARE, you will see that the program is not particularly efficient: even when the two integers satisfy the first IF THEN statement, they are tested two more times. PL/I lets us solve this difficulty with the ELSE clause. You add the ELSE clause to an IF THEN statement to stipulate what you want done if the expression following IF is false. IF THEN with ELSE has the form:

```
IF exp THEN statement; ELSE statement;
```

Like the THEN clause, the ELSE clause may contain only a single statement.

For example, if you wanted to do nothing more than check whether one of the values in COMPARE was greater than the other, you could do it with a single IF THEN ELSE statement:

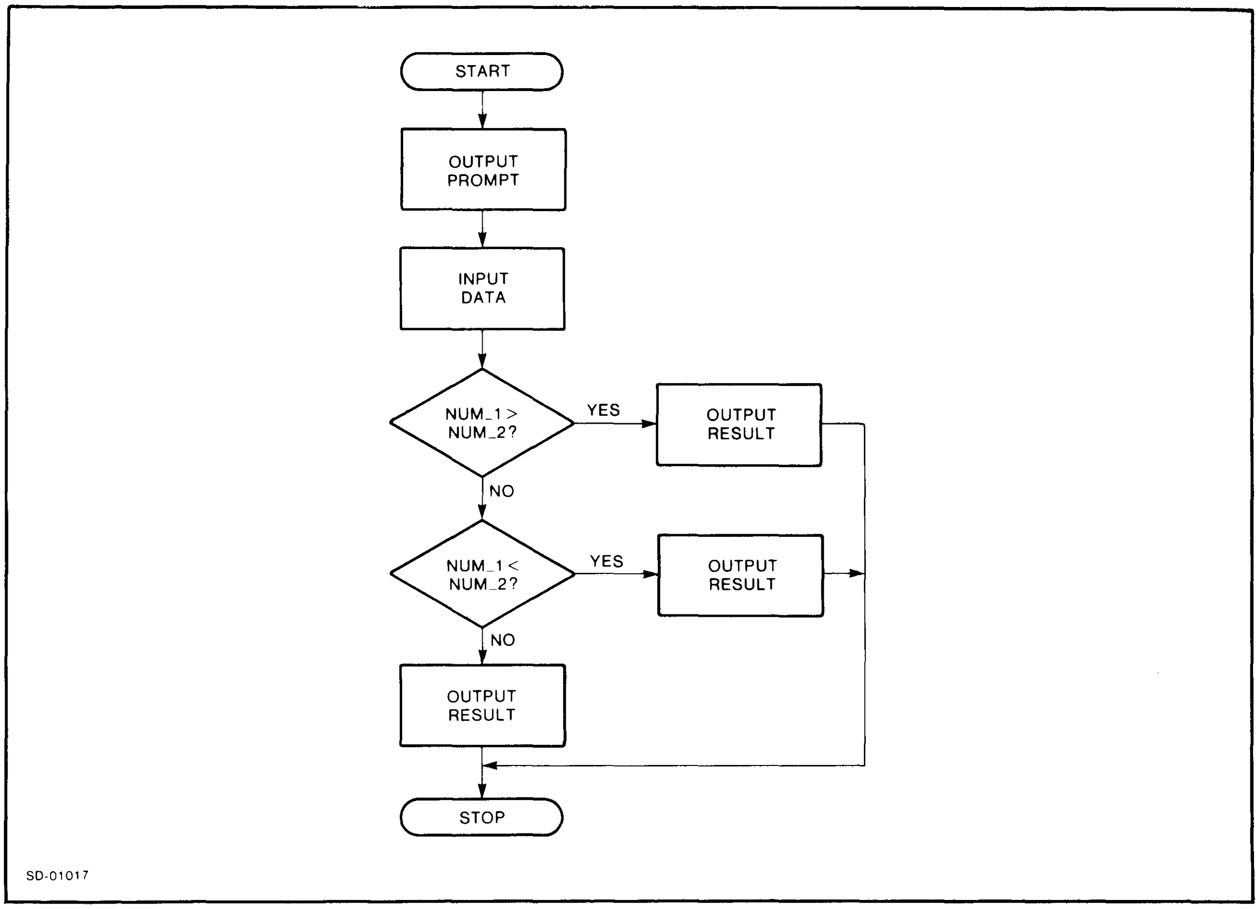
```
IF NUM_1 <NUM_2
           THEN
                      PUT FILE(OUT) LIST("NUM_1 IS LESS THAN NUM_2");

           ELSE
           PUT FILE(OUT) LIST("NUM_1 IS NOT LESS THAN NUM_2");
```

If we rewrite COMPARE with IF THEN ELSE, it will compare the integers twice at most; if the integers satisfy the relational operation of the first IF THEN ELSE statement, it will only compare them once:

```
COMPARE_2: /* VERSION WITH ELSE */
   PROCEDURE;
      .
      .
   IF NUM_1 >NUM_2
      THEN
         PUT FILE(OUT) LIST("NUM_1 IS BIGGER THAN NUM_2");
      ELSE
      IF NUM_1 <NUM_2
         THEN
            PUT FILE(OUT) LIST("NUM_2 IS BIGGER THAN NUM_1");
         ELSE
            PUT FILE(OUT) LIST("NUM_2 = NUM_1");
END; /* COMPARE_2 */
```

The program works like this: if NUM_1 > NUM_2 is true, it executes the first IF THEN statement's THEN clause. If it isn't, it executes the ELSE clause, which contains another IF THEN statement. If NUM_1 < NUM_2 is true, it executes that IF THEN statement's THEN clause. If it isn't, it executes that IF THEN statement's ELSE clause. Since any value which is neither greater than NUM_1 nor less than NUM_1 must be equal to NUM_1, there is no need to perform another test, and the second ELSE clause can contain a simple PUT LIST statement.

What happens if NUM_1 is bigger than NUM_2 and the program executes the first IF THEN statement's THEN clause? Since the second IF THEN statement is part of the first IF THEN statement's ELSE clause and the third PUT LIST statement is part of the second IF THEN statement's ELSE clause, everything following the THEN clause of the first IF THEN statement is part of its ELSE clause and is simply passed over if the THEN clause is executed. The flow chart for this program follows. Note the differences between it and the flowchart for COMPARE.



Nested IF THEN Statements

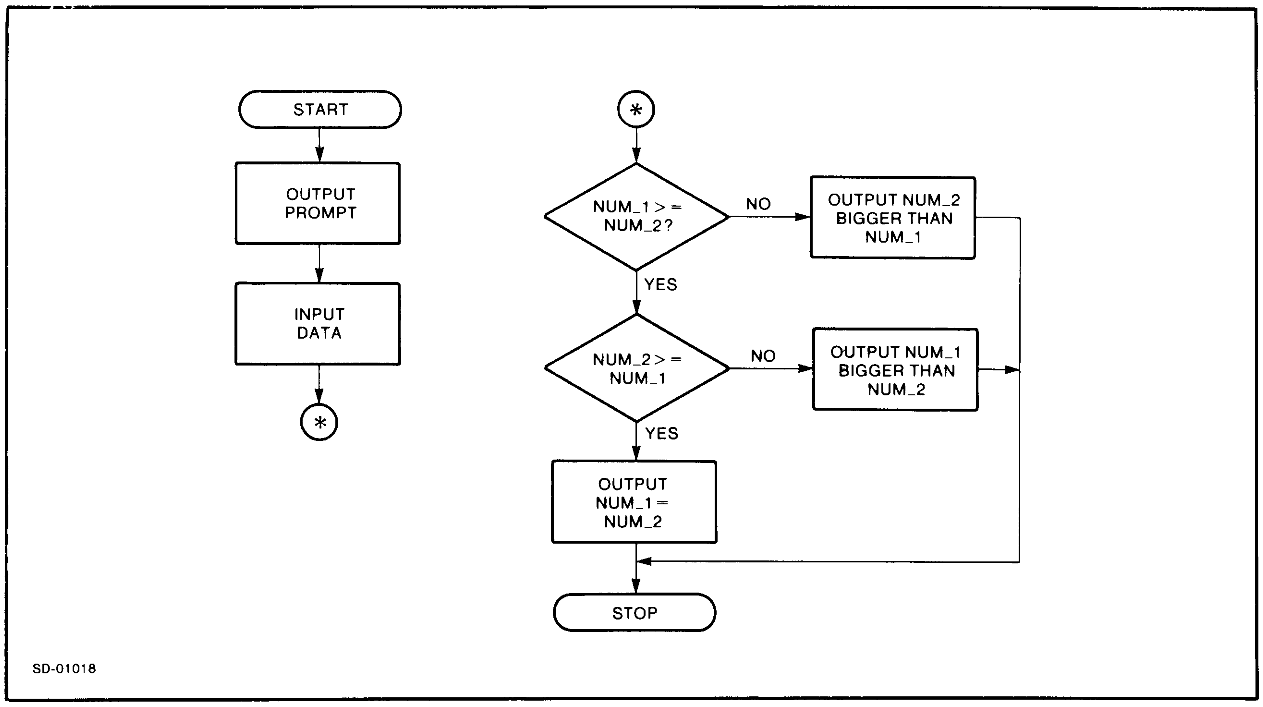
In COMPARE_2, we used an IF THEN statement in the ELSE clause of another IF THEN statement. Of course, you can use IF THEN statements in the THEN clause, too. When you do this, you have nested IF THEN statements. Nested IF THEN statements can make a program difficult to understand, but since you may run into programs that use them, we include a version of COMPARE with nested IF THEN's here:

```
COMPARE_3: /* VERSION WITH NESTED IF THEN'S */
        PROCEDURE;
        .
        .
    IF NUM_1 > =NUM_2
        THEN
            IF NUM_2> =NUM_1
            THEN
                PUT FILE(OUT) LIST("NUM_2 = NUM_1");
            ELSE
                PUT FILE(OUT) LIST("NUM_1 IS BIGGER THAN NUM_2");
            ELSE
                PUT FILE(OUT) LIST("NUM_2 IS BIGGER THAN NUM_2");
END; /* COMPARE_3 */
```

When you use ELSE clauses with nested IF THEN's, the innermost ELSE belongs to the innermost preceding IF, the next innermost ELSE to the next innermost preceding IF, and so forth. In COMPARE_3, the first ELSE belongs to the second IF and the second ELSE belongs to the first IF. As you can see, proper indentation makes it easier to understand the relationship between IFs and ELSEs.

When you run this program, it works like this: if NUM_1 is not greater than or equal to NUM_2, NUM_2 must be bigger than NUM_1; consequently, the ELSE clause for the first IF THEN statement outputs that message (remember, the first IF THEN statement's ELSE clause is the second ELSE clause). If NUM_1 > =NUM_2 is true, then the second IF THEN statement is executed; if it is true too, then NUM_1 = NUM_2. Hence, the THEN clause for the second IF statement outputs "NUM_1 = NUM_2". If the expression in the second IF THEN statement is false, then NUM_1 is bigger than NUM_2, and the ELSE clause for the second IF THEN statement outputs that message.

The flowchart looks like this:



As you can see, COMPARE_3 is no more efficient than COMPARE_2, which is much easier to understand.

Non-Iterative DO Groups

The PL/I DO group allows you to use a sequence of statements anywhere you can use a single statement. PL/I has both iterative and non-iterative DO groups. We will discuss the iterative DO groups in the next chapter; we will deal only with the non-iterative DO.

The non-iterative DO group consists of a sequence of statements which begin with DO and end with END. All of the statements in the group are executed as a single unit. For example, if you use a DO group in the THEN clause of an IF THEN statement and the expression in the IF clause is true, all the statements in the DO group will be executed before the statement following the IF THEN statement is executed.

For example, we could rewrite COMPARE with DO groups like this:

```
COMPARE_4: /* VERSION WITH DO-GROUPS */
           PROCEDURE;
           .
           .
           IF NUM_1 >NUM_2
                      THEN
                                 DO;
                                 PUT FILE(OUT) LIST("NUM_1 IS BIGGER THAN NUM_2");
                                 STOP;
                                 END;

IF NUM_1 <NUM_2
                      THEN
                                 DO;
                                 PUT FILE(OUT) LIST("NUM_2 IS BIGGER THAN NUM_1");
                                 STOP;
                                 END;
                                 PUT FILE(OUT) LIST("NUM_1 = NUM_2");

END; /* COMPARE_4 */
```

We haven't seen the STOP statement before. It has the form:

```
STOP;
```

The STOP statement does the same thing as a program's final END statement: it stops program execution, closes all open files, and returns control to the operating system. Since STOP and the final END statement for a program do the same thing, you need only use STOP when you want the program to STOP before it has reached the final END statement. Please note that a DO group that contains a STOP statement must still have an END statement for the DO.

How COMPARE_4 works is plain enough: if NUM_1 is bigger than NUM_2, the Do group in the THEN clause of the first IF THEN statement is executed. The group's PUT LIST statement outputs the message that NUM_1 is bigger than NUM_2 and the STOP statement stops the program. If NUM_2 is bigger than NUM_1, the DO group in the second IF THEN statement is executed; again, the proper message is output and the program is stopped.

If NUM_2 is neither greater than nor less than NUM_1, NUM_2 = NUM_1 and we can output a message to that effect. Since the last output is at the end of the program, we need not follow it with a STOP.

You probably noticed that COMPARE_4 does not use the ELSE. It does not need it, because the STOP statements in the branches terminate the program whenever a relationship is true.

Using IF THEN in the Simple Interest Program

Branches can help you catch bad input before it produces bad output. At present, the SIMP_INT program in Chapter 2 will work even if the user inputs a 95% rate of interest, a negative principal, or a negative number of years. What would a branch that output an error message and stopped the program when someone gave a rate of interest above 18% look like?

The expression in the IF part of the IF THEN is clear enough: RATE > .18. Since we want the branch to do two things, the THEN clause will have to be a DO group:

```
IF RATE > .18
           THEN
           DO;
           PUT FILE(OUT) LIST("DON'T BE A LOANSHARK! RATES"!!
                                          "HIGHER THAN 18% ARE USURY");
           STOP;
                      END;
```

Make some branches that catch errors in your interest or mortgage program. Have fun with the error messages. You may also want to try nested IF THENs and IF THENs with ELSE to look for several errors in one sequence of statements.

The Relational Operators

So far, we have dealt intuitively with the relational operators. If NUM_1 was 10 and NUM_2 was 100, NUM_1 < NUM_2 was obviously true, and we did not concern ourselves with how the computer made the comparison or what "true" actually meant. Now it is time to see how the relational operators work.

How the computer makes the comparison depends on the data types of the operands. If the operands have arithmetic data types, PL/I converts them to a common type the same way it does for arithmetic operations and then compares the operands algebraically: if operand 1 - operand 2 is 0, the two are equal; if it is greater than 0, operand 1 is larger than operand 2; if it is less than 0, operand 2 is larger than operand 1.

If the operands have character-string data types, PL/I pads the shorter character string on the right with blanks until both strings are the same length. Consequently, if one operand has the value "A" and the other "ABC", PL/I will pad the "A" until it is "A□□". Then it starts on the left and compares them character by character according to the character's value in the ASCII code. If it finds that the two string's first characters have different values, the string whose first character has the larger ASCII value is greater. If the first characters are identical, the computer makes the same check on the second characters. Again, the string whose second character has the larger ASCII value is greater. The comparison continues in this fashion until the computer finds two characters which are not the same or until it has compared all the characters in the two strings. Only if each character in the first string is identical to its corresponding character in the second string are the strings equal.

According to the above rules, the following relationships are all true:

```
"A" < "B"
"A" < "AAA"
"BBC" < "BCA"
```

Note that in the ASCII code, any lower-case letter has a higher value than any upper-case letter. Consequently, the following relationship is true:

```
"Z" < "a"
```

Since the non-alphabetic characters in the ASCII character-set also have values, you can compare them. For example, the following relationships are true:

```
"#$" < "$"
"1" < "A"
```

As you might expect, the operands with the comparison operators must have the same general data types. You can compare any type of arithmetic value with any other type of arithmetic value, or any type of character-string value with any other type of character-string value, but you cannot compare arithmetic values with character-string values. You can use the comparison operators with all of the PL/I data types, but the comparisons have special rules and limitations for each type. For details, see the *PL/I Reference Manual,* (093-000204).

Now that we know how the comparisons work, what exactly does "true" mean? To explain what it means, we have to introduce a new type of data.

Bit-string Data

The computer stores all of the values in your program as sequences of bits. For example, it stores the fixed binary value 7 as the following sequence:

```
0000000000000111
```

When you use the value in your program, the computer interprets this sequence of bits as the integer value 7. In general, you need not concern yourself with the sequence of bits that actually represents a value; however, PL/I does have data whose value is simply the sequence of bits it contains. This data is called bit-string data. PL/I has bit-string constants and two kinds of bit-string variables: BIT variables and ALIGNED BIT variables. Here, we will introduce the constants and BIT variables; we deal with ALIGNED BIT variables in Chapter 10.

Bit Constants

PL/I has four formats for writing bit-string constants. For now we will only deal with the simplest: the B format. For the other formats, see the reference manual. In the B format, a bit-string constant is a sequence of 1's and 0's. You put the sequence between quotation marks and follow it with a B, as in:

```
"1001001"B
```

Bit Variables

You declare bit variables with the BIT keyword. Like character variables, they have a fixed length, which you indicate in parentheses following the BIT keyword. The default length is 1.

```
DECLARE FLAG BIT(3);
```

The length is the number of bits in the string. If you assign a value to a bit variable that is shorter than the length of the variable, the value is padded to the right with 0 bits. For example, after the assignment:

```
FLAG = "1"B;
```

FLAG will have the value "100"B. The following assignment, on the other hand, truncates the constant from the right, giving FLAG the value "101"B.

```
FLAG = "101011"B.
```

Bit-string Data and The Results of Relational Operators

What does all of this have to do with the relational operators? The result of a relational operation is a bit-string 1 bit long: if the relation in the operation is true, the operation has the value "1"B; if it is false, it has the value "0"B. You can use not only relational operations, but any bit-string expression following the IF in IF THEN. If the bit-string expression contains any "1" bits, the computer will execute the THEN clause; if it contains any "0" bits, it will not execute the clause but will execute the ELSE clause if there is one. Consequently, if we write:

```
IF "010"B THEN STOP;
```

the expression following the IF will always be true and the program will always execute the STOP statement. Conversely, if we write:

```
IF "OOO"B THEN STOP;
```

the expression following the IF will always be false and the program will never execute the STOP statement.

Since the relational operators have bit-string results, we can assign the result of a relational operation to a bit variable and then use the variable instead of the relational operation in an IF THEN statement:

```
DECLARE FLAG BIT;
DECLARE (NUM_1, NUM_2) FIXED BINARY;

.
FLAG = NUM_1 < NUM_2;
IF FLAG
         THEN
                  PUT FILE(OUT) LIST("NUM_1 < NUM_2");
         ELSE
         PUT FILE(OUT) LIST("NUM_1 < NUM_2");
.
```

The value of flag will be "1"B if NUM_1 < NUM_2 is true and "0"B if it is false. It will therefore work exactly the same way as the relational operation itself in the IF THEN statement. Since the computer can check the value of a variable faster than it can make a comparison, you can speed up programs which make the same comparisons over and over again by using bit variables instead of the comparisons.

Bit-string Data and Logical Operations

Some high-level languages have a special logical data type which has the values true or false and upon which you can peform such logical operations as AND, NOT, and OR. PL/I has no such type, but as we have seen, it will interpret bit-string expressions in IF THEN statements as having the value true or false. PL/I also has a set of three bit-string operators that work the same way as the logical operators found in other languages.

The operators are:

Symbol	Name
&	AND
!	Inclusive OR
^	NOT (logical complement)

The operands with the bit-string operators must be bit-string expressions; no conversion takes place, so you cannot use other types of expressions as operands.

& and ! each take a pair of bit-strings as operands. If the operands have different lengths, the shorter operand is padded on the right with 0 bits until both are the same length. The operators then start at the left and compare the two operands bit for bit. The result is a bit string the same length as the operands. The sequence of bits it contains will depend on the operands and the operator. If the operator is &, the result bit-string will have 1 if the operands both have 1 in the corresponding position; otherwise, it will have 0. For example, the result of "1010"B & "1000"B is "1000"B. If the operator is !, the result bit-string will have 1 if either or both of the operands have 1 in the corresponding position. The result of "1010"B ! "1000"B is "1010"B.

The ^ operator only takes a single operand. It simply makes every "0" bit of the operand into a "1" bit and vice-versa. The result of ^ "1010"B is "0101"B and that of "1000"B is "0111"B.

You may have already noticed that if you interpret "1"B as true and "0"B as false the way PL/I does, the results of operations on individual bits with these operators are the same as the truth tables for the logical operators AND, OR, and NOT. In the following, we put the truth tables and the results of the bit-string operations side- by-side.

Truth Table for AND

Operand values		Result
T	T	T
T	F	F
F	T	F
F	F	F

Results of &

Operand values		Result
"1"B	"1"B	"1"B
"1"B	"Ø"B	"Ø"B
"Ø"B	"1"B	"Ø"B
"Ø"B	"Ø"B	"Ø"B

Truth Table for OR

Operand values		Result
T	T	T
T	F	T
F	T	T
F	F	F

Results of !

Operand Values		Result
"1"B	"1"B	"1"B
"1"B	"Ø"B	"1"B
"Ø"B	"1"B	"1"B
"Ø"B	"Ø"B	"Ø"B

Truth Table for NOT

Operand	Result
T	F
F	T

Results of ↑

Operand	Result
"1"B	"Ø"B
"Ø"B	"1"B

Figure 3-1. Bit Operations and the Logical Truth Tables

The Bit-string Operators and the Relational Operators

Since the relational operators return "1"B or "0"B as their results, you can use the bit-string operators like logical operators with them. For example, you could combine several error checks in one IF THEN statement like this:

```
IF (PRINCIPAL < 0)!(RATE > .18)!(TIME < 1)

        THEN
        DO;
                PUT FILE(OUT) LIST("INPUT ERROR. CHECK YOUR INPUT.");
        STOP;
        END;
```

The computer evaluates the expression following IF as follows: first it compares PRINCIPAL with 0. Then it compares RATE with .18. Each of these operations yields a bit-string result, and the first ! uses these results as operands. If either or both of the relations is true, the ! operator will yield the value "1"B. Then the computer evaluates the TIME < 1 expression and uses its result together with the result of the first ! with the second !. If either the first ! or the comparison has the result "1"B, the whole expression will have the result "1"B and the THEN clause will be executed. As you can see, the THEN clause will be executed if any of the errors appears. If we had used & as the operator instead of !, all of the errors would have had to appear in order for the THEN clause to be executed.

Conversions and Operations with Bit-string Data

If you assign bit-string values to arithmetic and character-string variables or vice-versa, PL/I will convert the bit-string values to arithmetic and character-string values and these values to bit-string values. For details on these conversions, see the reference manual.

PL/I will also perform concatenations and comparisons with bit-string operands. If you concatenate a pair of bit- strings, the operator works the same way as it does with character strings: "100"B !! "11"B has the result "10011"B. If you concatenate a bit string with an operand of another data type, PL/I converts the bit string to a character string before it performs the concatenation. Consequently, "ABC" !! "1001"B has the result "ABC1001".

Comparison with bit-string operands works the same way as it does with character-string operands: if the bit strings have different lengths, the shorter string is padded on the right with 0's until it is as long as the longer string. Then the computer begins comparing the strings. The comparison begins at the left and continues bit-for-bit until the computer finds a position where one string has a 0 bit and the other has a 1 bit. Since a 1 bit is larger than a 0 bit, the string with the 1 bit is larger. According to this rule, the following relations are true:

```
"011"B < "100"B
"101"B < "110"B
```

The Order of Evaluation of Operations

As we saw with the arithmetic operators, PL/I has rules that determine how an operation is evaluated. Now that we have seen all of PL/I's operators, we can describe how expressions containing operations are evaluated.

1. Order of Precedence

   Since you can combine different kinds of operators in expressions, you need to know the order of precedence for all operators. If several operators have the same precedence, the operations except for unary +, unary -, and ** are performed from left to right. The latter operations are performed from right to left.

   The operations with the highest precedence are performed first and those with the lowest are performed last.

Example 3-2.

Highest:	** ^ unary + unary -
	* /
	+ (addition) - (subtraction)
	!!
	= ^= < ^< <= > ^> >=
	&
Lowest:	!

Figure 3-2. The Order of Precedence

For example, the following expression without parentheses is evaluated as if it had the parentheses of the second version:

```
IF A**B < B + C ! FLAG_1 & FLAG_2 THEN . . .

IF (((A**B) , (B + C)) ! (FLAG_1 & FLAG_2)) THEN
```

Obviously, your programs will be easier for others to understand if you use parentheses instead of relying on the order of evaluation.

2. Evaluation with parentheses

   You can use parentheses with all kinds of operations. Evaluation begins with the innermost sets of parentheses and continues until the entire expression contained in the outermost set has been evaluated.

Try adding some IF THEN statements with bit-string operators to your program that calculates simple interest.

Multiple Branching: DO CASE

The IF THEN ELSE statement is clear and easy to use when there are only two alternatives. If there are more alternatives than that, you have to repeat the tests over and over or use nested IF THEN statements.

AOS PL/I has another method of branching that allows your program to choose between as many alternatives as you want: the DO CASE statement. Note that this statement is an extension to standard PL/I and that you cannot compile programs that use it on standard PL/I compilers.

The syntax of DO CASE looks like this:

```
DO CASE (fixed-bin-exp);
statement-1;
.
.
statement-n;

END;
[OTHERWISE statement]
```

As you can see, the DO CASE statement has three parts: the DO CASE statement itself, a list of statements terminated by an END statement, and an optional OTHERWISE clause. As you could within statements in the THEN and ELSE clauses of the IF THEN statement, you can use DO groups as statements.

The DO CASE statement works like this: when the computer executes the DO CASE statement, it finds the value of the fixed binary expression in parentheses following the CASE keyword. Then it executes the statement in the list of statements whose position in the list corresponds to the value of the expression. For instance, if the expression has the value 1, it will execute the first statement; if it has the value n, it will execute the nth statement. When it is done executing the statement, it executes the statement following the DO CASE statement.

The optional OTHERWISE clause works like the ELSE clause in the IF THEN statement: the statement following OTHERWISE is executed only if the fixed binary expression has a value less than 1 or greater than the number of statements on the list. Note that if you do not include the OTHERWISE clause, an out of range value for the fixed binary expression will raise the error condition.

To see how DO CASE works, let's look at the program called TITLES. This program asks for your name, then asks what title you wish -- Miss, Mrs., Ms., or Mr. Finally, it outputs your name with the correct title.

```
TITLES:
   PROCEDURE;

   DECLARE NAME CHARACTER(20) VARYING;
   DECLARE TITLE CHARACTER(4) VARYING;
   DECLARE TITLE_NO FIXED BINARY;
   DECLARE (IN,OUT) FILE;

   OPEN FILE(IN) STREAM INPUT TITLE ("@INPUT");
   OPEN FILE(OUT) STREAM OUTPUT TITLE ("@OUTPUT"):

   PUT FILE(OUT) LIST("WRITE YOUR NAME. ENCLOSE"!!
                      "IT IN QUOTES LIKE THIS: ""JOHN K.DOE""");
   PUT FILE(OUT) SKIP LIST("NAME?");
   GET FILE(IN) LIST(NAME);
   PUT FILE(OUT) SKIP LIST("IF YOU ARE MISS, TYPE 1");
   PUT FILE(OUT) SKIP LIST("IF YOU ARE MRS., TYPE 2");
   PUT FILE(OUT) SKIP LIST("IF YOU ARE MS., TYPE 3");
   PUT FILE(OUT) SKIP LIST("IF YOU ARE MR., TYPE 4");

   PUT FILE(OUT) SKIP LIST("NUMBER?");
   GET FILE(IN) LIST(TITLE_NO);

   /***************************************************************/
   /* THE DO CASE ASSIGNS TITLE THE TITLE CORRESPONDING TO THE */
   /* NUMBER INPUT BY THE USER.                                */
   /***************************************************************/

   DO CASE (TITLE_NO);

   /* TITLENO = 1 */
       TITLE  = "MISS";

   /* TITLENO = 2 */
       TITLE  = "MRS.";

   /* TITLENO = 3 */
       TITLE  = "MS.";

   /* TITLENO = 4 */
       TITLE  = "MR.";

   END; /* DO CASE */

     OTHERWISE TITLE = "";

     PUT FILE(OUT) LIST("HELLO, "!!TITLE!!" "!!NAME);

   END; /* TITLES */
```

Figure 3-3. The Sample Program TITLES

As you can see, the program asks the user to input a number for the proper title and then assigns this number to the fixed binary variable TITLE_NO. The variable is the fixed binary expression in the DO CASE statement, and consequently the value of TITLE_NO decides which of the assignment statements on the list is executed. We have given each of the statements on the list a comment that tells you what value of TITLE_NO will cause its execution. The compiler doesn't need the comments, of course, but they make it easier for others to understand what's happening. Each assignment statement assigns the prop0er title to TITLE, and the PUT LIST statement concatenates TITLE to NAME. If the user inputs a number that is less than 1 or greater than 4, the program executes the OTHERWISE clause, which assigns a null string to TITLE.

TITLES is of course a trivial example, but it should give you some idea of the possibilities. You can use DO CASE anytime a program has to choose between several alternatives, and since the expression in the DO CASE statement has an arithmetic value, you can compute which case you want.

Try setting up your interest program so that it asks the borrower's name and title, or write a program with DO CASE which lets a user input two values and choose which arithmetic operation he wants to perform on them.

End of Chapter

Chapter 4
Loops and STREAM Files

Introduction to PL/I's Loops

A loop is a single sequence of statements your program may execute several times in succession. In PL/I, you use iterative DO groups to construct loops. AOS PL/I has three different kinds of iterative DO groups: the DO WHILE, the DO increment, and the DO list.

The DO statement in the DO WHILE loop contains a bit-string expression. Before each execution of the statements in the loop, PL/I checks whether the bit-string expression is true. If it is, it executes the loop again. Otherwise, it executes the statement following the loop. For example,

```
I = 0;

DO WHILE (I = 4);

I = I + 1;

END;
```

PL/I will continue executing this loop as long as I is not greater than 4.

The DO statement in the DO increment contains a control variable and two or three expressions. On the first execution of the loop, PL/I sets the control variable to the first expression. It then checks whether the value of the control variable has passed the value of the second expression. If it hasn't it executes the loop. When it is done, it changes the value of the control variable by the amount of the third expression and checks again to see whether the control variable's value has passed the value of the second expression. If it hasn't, PL/I executes the loop again; otherwise, it executes the statement following the loop. For instance,

```
DO I = 1 TO 4;

      PUT FILE(OUT) LIST(I);

END;
```

When PL/I enters this loop, it sets I to 1. It then executes the loop ands adds 1 to I. It continues executing the loop and adding 1 to I until I is greater than 4. In the above example, I has the value 5 after the loop is completed.

The DO statement in the DO list has a control variable and a list of expressions. On each execution of the loop, PL/I sets the control variable to the value of the next expression. The loop is executed as many times as there are expressions in the list.

DO list looks like this:

```
DO VAR = 1,1,3,-5;

      VAL = VAR + 1;

END;
```

On the first execution of the loop, PL/I assigns 1 to VAR, on the second, 1 again, and so on until it has executed the loop for each of the values in the list.

To show you how iterative DO loops work, we have rewritten the program SIMP_INT with a DO WHILE loop. The new version of the program does not stop after it has calculated the interest, but instead asks you if you want to continue. If you do, the program asks for new input and repeats the interest calculation.

The program looks like this:

```
  SIMP_INT:  /* WITH DO WHILE */
   PROCEDURE;

   DECLARE (PRINCIPAL,INTEREST) FIXED DECIMAL(7,2);
   DECLARE RATE FIXED DECIMAL(4,4);
   DECLARE TIME FIXED BINARY;

 /* COMMAND HOLDS THE VAL WHICH CONTROLS THE LOOP */

   DECLARE COMMAND CHARACTER(1);

   DECLARE (IN,OUT) FILE;

   OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
   OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

   COMMAND = "C";  /* "C" TURNS ON THE LOOP */

 /*****************************************************/
 /* THE DO WHILE LOOP COMES BACK FOR MORE INPUT UNTIL */
 /* THE USER TYPES "S" TO STOP IT                     */
 /*****************************************************/

   DO WHILE (COMMAND ↑= "S");

     PUT FILE(OUT) LIST("INPUT PRINCIPAL,RATE,AND TIME");
     GET FILE(IN) LIST(PRINCIPAL,RATE,TIME);

     INTEREST = ROUND(PRINCIPAL * RATE * TIME,2);

     PUT FILE(OUT) LIST("THE INTEREST IS", INTEREST);

     PUT FILE(OUT) SKIP LIST("TO STOP, TYPE S;"!!
                             "TO CONTINUE, TYPE C");
     GET FILE(IN) LIST(COMMAND);
   END; /* DO WHILE */

END; /* SIMP_INT */
```

Figure 4-1. The SIMP_INT Program with a DO WHILE Loop

The DO WHILE loop in the program is designed to repeat until the user stops it by typing "S". To get the loop started, we assign COMMAND the value "C" immediately before the DO WHILE statement. This makes sure that the loop will be executed at least once. Once the program is in the loop, it asks for data, performs the interest calculation, and outputs the data. Then it asks the user to type "S" if he wants to stop and "C" if he wants to continue. The program assigns the character typed by the user to COMMAND and then returns to the top of the loop, where it checks whether COMMAND is still not equal to "S". If the user typed "S", the program goes to the statement following the DO WHILE group,"END; /* SIMP_INT */". Otherwise, it repeats the loop.

As you can see, the syntax of iterative DO groups is like the syntax of non-iterative DO groups: the group begins with a DO statement and ends with an END statement; the entire group is the syntactical equivalent of a single statement; and all the statements in the group are executed as a unit. Your programs will be easier to understand if you use indentation to make it clear which statements belong to a DO group and which END statement corresponds to a given DO statement.

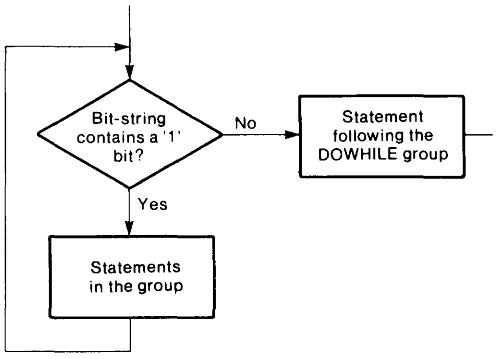
DO WHILE

The formal syntax of DO WHILE looks like this:

```
DO WHILE (bit-string-result);

statements

END;
```

Note that you must put the bit-string result in parentheses. The statements in the loop will be executed as long as the bit- string result contains at least 1 "1" bit.

A flowchart for the loop looks like this:


Bit-string contains a '1' bit?
No
Statement following the DOWHILE group
Yes
Statements in the group

SD-01019

Some consequences of the way the DO WHILE works:

1. If the bit-string expression contains all "0" bits the first time the DO WHILE statement is executed, the loop will never be executed.

2. If the group is executed, all the statements in it are executed regardless of whether the execution of one of the statements changes the value of the bit-string result. Take the following:

```
VAL = 1;
DO WHILE(VAL > 0);
          VAL = VAL - 1;
          PUT LIST(VAL);
END;
```

Because PL/I executes all the statements in the group, the PUT LIST statement will output 0. If you want to keep this from happening, you have to set up your group so that the statement which changes the value of the bit-string expression is the last statement in the group or you have to branch out of the group. One way to do this would be an IF THEN statement with STOP:

```
VAL = 1;
DO WHILE(VAL > 0);
          VAL = VAL - 1;
          IF VAL = 0 THEN STOP;
          PUT LIST(VAL);
END;
```

The Bit-string Expression

The bit-string expression in the DO WHILE statement works exactly like the bit-string expression in the IF THEN statement. Generally, the expression will be a relational operation, but you can use bit-string constants, bit-string variables, and expressions with bit-string operators as well. For example, if you wanted to have a loop run forever, you could write your DO WHILE statement with the bit-string constant "1"B:

```
DO WHILE ("1"B);
```

Since the value of the constant will never change, the loop cannot turn itself off.

You can also use bit-string variables as flags (see Chapter 2) to turn DO WHILE loops on and off. If we added a bit-string variable to SIMP_INT4 and used it as a flag, the DO WHILE loop might look like this:

```
FLAG = "1"B;
        DO WHILE(FLAG);
                  PUT FILE(OUT) LIST("INPUT PRINCIPAL,RATE,AND TIME");
                  GET FILE(IN) LIST(PRINCIPAL,RATE,TIME);

                  INTEREST = ROUND(PRINCIPAL * RATE * TIME,2);

                  PUT FILE(OUT) LIST("THE INTEREST IS," INTEREST);

                  PUT FILE(OUT) SKIP LIST("TO STOP, TYPE S;"!!
                                          "TO CONTINUE, TYPE C");
                  GET FILE(IN) LIST(COMMAND);
                  FLAG = COMMAND ^= "S";

END; /* DO WHILE */
```

Here, we get the loop going by assigning "1"B to the bit variable FLAG. Then, at the end of the loop, we assign the character input by the user to COMMAND and assign the result of the relational operation COMMAND ^= "S" to FLAG. If COMMAND does not have the value "S", FLAG will have the value "1"B; otherwise, it will have the value "0"B, which will stop the loop.

There is no particular reason why we should use a flag in SIMP_INT, but if a DO WHILE loop has many repetitions, a flag can speed up execution.

Of course, you can also use the bit-string operators in the bit-string expression. There is an example of this in the next section.

Nested DO Loops

Since you can use a DO group anywhere you can use a single statement, you can put DO groups inside of other DO groups. Such DO groups are called nested DO groups. Each nested group counts as a single statement of the group in which it is nested. Hence, the nested group cannot contain any statements of the group it is nested in. To show you how this works, let's take a DO group with two statements and replace the statements with other DO groups:

```
DO;
    statement
    statement
END;
```

can become

```
DO;
    DO;
         statements
    END;
         statements
END;
```

The nested group can itself contain DO groups, so the above may become

```
DO;
    DO;
       statements
       DO;
              statements
       END;
              statements
       END;
       statement
END;
```

Since a DO group may have any number of statements, and you can replace any of these with a DO group, you can also have parallel DO groups nested within a group:

```
DO;
    DO;
         statements
    END;
    DO;
         statements
    END;
END;
```

As a practical example, let's make a DO WHILE loop for SIMP_INT4 which looks for input errors and asks the user for new input if it finds errors. Since the input statements in SIMP_INT4 are part of a DO WHILE loop, the loop for error detection will be nested in the larger DO WHILE loop.

Let's assume that the errors we are looking for are negative or 0 values of PRINCIPAL, rates of interest over .18, and negative values of TIME. We can check for all of these with a single expression:

```
DO WHILE ((PRINCIPAL <= 0)!(RATE > .18)!(TIME <= 0));
```

This DO WHILE statement will only be executed if at least one of the relationships in the bit-string expression is true, and it will be executed as long as at least one of the relationships is true.

The other things we need in the loop are an error message and a GET LIST statement which gets new input. The finished loop will look something like this:

```
DO WHILE ((PRINCIPAL <= 0)!(RATE > .18)!(TIME <= 0));
          PUT FILE(OUT) SKIP LIST("BAD INPUT. THE"!!
                                              "PRINCIPAL MUST BE GREATER THAN 0, THE"!!
                                              "INTEREST MUST BE LESS THAN OR EQUAL TO .18, AND THE"!!
                                              "TIME MUST BE GREATER THAN 0. PLEASE"!!
                                              "INPUT NEW VALUES.");
          PUT FILE(OUT) SKIP LIST("NEW INPUT?");
          GET FILE(IN) LIST(PRINCIPAL,RATE,TIME);
END; /* ERROR-CATCHER */
```

We will want to put this loop right after the GET LIST statement in the outer loop. The nested loops will look like this:

```
DO WHILE (COMMAND ↑= "S");
          PUT FILE(OUT) LIST("INPUT PRINCIPAL,RATE,AND TIME");
          GET FILE(IN) LIST(PRINCIPAL,RATE,TIME);

/* THIS LOOP CATCHES INPUT ERRORS */

          DO WHILE ((PRINCIPAL <= 0)!(RATE > .18)!(TIME <= 0));

                    PUT FILE(OUT) SKIP LIST("BAD INPUT. THE"!!
                                              "PRINCIPAL MUST BE GREATER THAN 0, THE"!
                                              "INTEREST MUST BE LESS THAN OR EQUAL TO .18, AND
                                              THE"!!
                                              "TIME MUST BE GREATER THAN 0. PLEASE"!!
                                              "INPUT NEW VALUES.");
                    PUT FILE(OUT) SKIP LIST("NEW INPUT?");
                    GET FILE(IN) LIST(PRINCIPAL,RATE,TIME);
          END; /* ERROR-CATCHER */

          INTEREST = ROUND(PRINCIPAL * RATE * TIME,2);

          PUT FILE(OUT) LIST("THE INTEREST IS," INTEREST);

          PUT FILE(OUT) SKIP LIST("TO STOP, TYPE S;"!!
                                                  "TO CONTINUE, TYPE C");
          GET FILE(IN) LIST(COMMAND);
END; /* DO WHILE */
```

Try adding some error-catching loops to your own interest program.

DO Increment

The DO increment gives you more control than the DO WHILE. You can use it when you want a loop that runs a fixed number of times, or when you want to increase or decrease the value of a variable by a set amount on each execution. For example, a loop that counts from 0 to 10 by 2 would look like this:

```
DECLARE I FIXED BINARY;

DO I = 0 TO 10 BY 2;
      PUT FILE(OUT) LIST(I);
END;
```

This loop would produce the output 0 2 4 6 8 10.

When the program executes the DO statement the first time, it assigns the 0 to I and checks whether I is larger than 10. Since it is not, the program executes the loop. When it is finished, it adds 2 to I and checks whether I is less than 10 again. It keeps doing this until I = 10. Since 10 is still not greater than 10, it executes the loop again. This time, the addition of the 2 makes I larger than 10, and the program goes on to the statement following the DO group.

The DO increment will also work backwards:

```
DO I = 10 TO 0 BY -2;
      PUT FILE(OUT) LIST(I);
END;
```

This loop first assigns 10 to I and then decreases the value of the variable by 2 each time it executes the loop. It stops executing the loop when the value of the control variable is less than the value of the second expression.

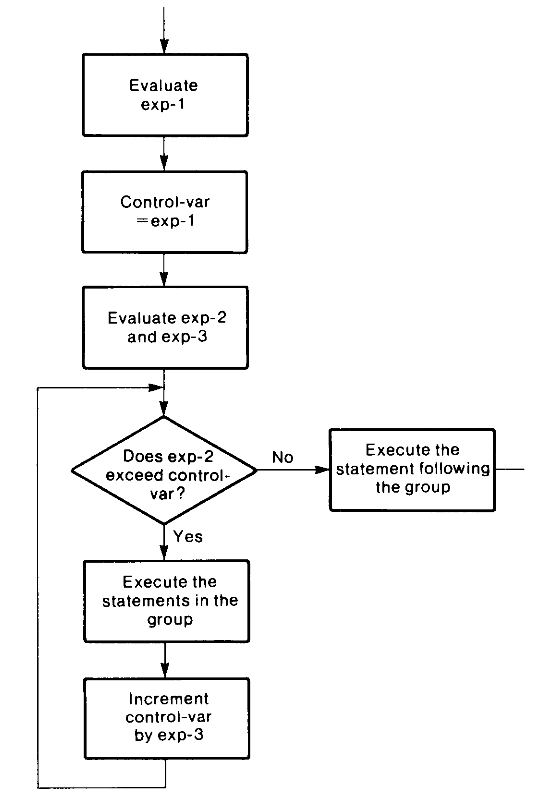
The syntax of the DO increment is as follows:

```
DO fixed-bin-var = fixed-bin-exp TO fixed-bin-exp
      [BY fixed-bin-exp];

      statements

END;
```

The variable following DO is called the control variable. It must have the fixed binary data type. The fixed-bin-expressions may be any expressions that yield a fixed binary result. Note that arithmetic operations that yield fixed binary results in standard PL/I may not do so in AOS PL/I. In AOS PL/I mixed fixed binary and fixed decimal operands have fixed decimal results. Consequently, you cannot use expressions with mixed operands in the DO increment statement.

The first expression in the statement gives the control variable its initial value. The second expression sets the value that the control variable must exceed before execution will stop. The BY keyword and the expression following it are optional. They give the amount by which the control variable will be increased or decreased on each execution. If BY and the expression are not given, the control variable will be incremented by 1.

The flow chart for DO increment looks like this:



Evaluate exp-1
Control-var =exp-1
Evaluate exp-2 and exp-3
Does exp-2 exceed control-var?
No
Execute the statement following the group
Yes
Execute the statements in the group
Increment control-var by exp-3

SD-01020

You should note several things about the way PL/I executes the DO increment:

1. The expressions in the DO statement are evaluated before the first execution of the loop. They are not re-evaluated, so changes in their values during execution of the loop will not affect the number of times the loop is executed.

   For instance:

   ```
   DECLARE(I,N) FIXED BINARY;
             .
             N = 3;
             DO I = 1 TO N;
                N = N + 1;
             END;
   ```

   This loop will give N the value 4 in the first execution, 6 in the second execution, and 9 in the third execution, but the loop will nevertheless only be executed 3 times, because N had the value 3 when the expressions in the DO statement were evaluated.

2. When PL/I evaluates the expressions, it evaluates the first expression first and then expressions 2 and 3. The order in which it evaluates the latter two expressions is not defined, so the evaluation of either expression must not affect the value of the other.

3. PL/I tests whether the value of the control variable has exceeded the value of the second expression before every execution of the statements in the group. Consequently, if the control variable exceeds the value of the second expression on the first execution of the DO statement, the group will never be executed. This would happen with the DO loop in the last example if N had a value less than 1 when the DO statement was executed.

4. As with the DO WHILE loop, all the statements in the group are executed once execution starts. Even if a statement in the group increases the value of the control variable beyond the second expression, execution will continue until the control variable is evaluated again. As with the DO WHILE, you can overcome this by branching out of the loop or by making sure that the statement that changes the value of the control variable is the last statement in the loop.

Using DO Increment to Calculate Compound Interest

Now that we know how PL/I handles loops, we can convert our simple interest program into a compound interest program. Compound interest is calculated by computing the simple interest for a period, adding the simple interest to the principal, and then using the new principal to calculate the interest for the next period.

All we need to convert our simple interest calculation to a compound interest calculation are two new variables and the DO increment loop that does the calculation. The first new variable is the control variable for the loop; let's call it I. The second new variable is the new principal; let's call it NEWPRINC. Of course, it will be a fixed decimal variable. For safety's sake, let's give it one more significant digit than we gave PRINC. The declarations will look like this:

```
DECLARE I FIXED BINARY;
DECLARE NEWPRINC FIXED DECIMAL(8,2);
```

The loop will look like this:

```
NEWPRINC = PRINCIPAL;
DO I = 1 TO TIME;
          INTEREST = ROUND(NEWPRINC * RATE,2);
          NEWPRINC = NEWPRINC + INTEREST;
END;
INTEREST = NEWPRINC - PRINCIPAL;
```

Since we need to keep the original principal in order to find the total interest, we assign the value of PRINCIPAL to NEWPRINC before the loop starts and use NEWPRINC in the actual interest calculation. When we are done, we can find the interest simply by subtracting PRINCIPAL from NEWPRINC.

We can make SIMP_INT into COMP_INT simply by inserting the compound interest loop into the program in place of the simple interest calculation.

```
DO WHILE(COMMAND) ^= "S");
          PUT FILE(OUT) LIST("INPUT PRINCIPAL,RATE,AND TIME");
          GET FILE(IN) LIST(PRINCIPAL,RATE,TIME);

/************************************************/
/* COMPOUND INTEREST CALCULATION */
/**************************************************/

          NEWPRINC = PRINCIPAL;
          DO I = 1 TO TIME;
                    INTEREST = ROUND(NEWPRINC * RATE,2);
                    NEWPRINC = NEWPRINC + INTEREST;
          END; /* COMPOUND INTEREST LOOP */

          INTEREST = NEWPRINC - PRINCIPAL;

          PUT FILE(OUT) LIST("THE INTEREST IS," INTEREST);

          PUT FILE(OUT) SKIP LIST("TO STOP, TYPE S;"!
                                             "TO CONTINUE, TYPE C");
          GET FILE(IN) LIST(COMMAND);
END; /* DO WHILE */
```

Of course, lots of variations are possible. One that might be worth doing would be to calculate compound interest by the day, the month, or the year. To do this, you have to divide the yearly rate by 365 when you compound it by day, and by 12 when you compound it by the month. You also have to multiply the number of years by 365 when you calculate by the day and 12 when you calculate by the month. One way to do this would be to ask the user how he wants to calculate the interest and then use a DO CASE statement to set up the alternatives he wants:

```
DO CASE (INT_TYPE);

/* 1: YEARLY COMPOUNDING */
;
/* 2: MONTHLY COMPOUNDING */
DO;
RATE = RATE / 12;
TIME = TIME * 12;
END;

/* 3: DAILY COMPOUNDING */
DO;
RATE = RATE / 365;
TIME = TIME * 365;
END;

END; /* DO CASE */
```

The first case of the example contains only ;. This is the PL/I null statement. You use the null statement wherever you want the program to do nothing, as when the interest is to be compounded yearly in the above program. The other cases show you how you can use DO groups instead of single statements in the DO CASE statement.

The only thing to watch out for if you build the DO CASE into your example is redeclaring INTEREST, NEWPRINC, and RATE with scales large enough to express the small daily rate of interest and the small daily interest the rate produces.

DO List

The DO list allows you to execute a loop a fixed number of times and to assign a different value to an index variable for each repetition. For instance:

```
DECLARE VAR CHARACTER(20) VARYING;
DECLARE INT FIXED BINARY;

INT = 1;
DO VAR = INT, INT + 1, "BUCKLE", "MY SHOE";
           PUT FILE(OUT) LIST(VAR);
END;
```

The output from this loop is

```
"1"   "2"  "BUCKLE" "MY SHOE"
```

The formal syntax of the DO LIST statement looks like this:

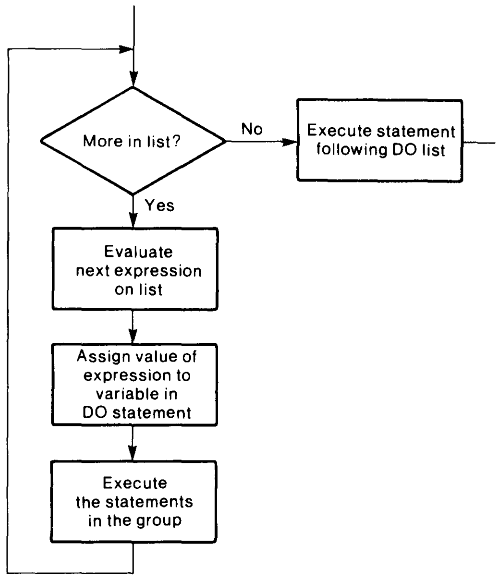
DO variable = exp-1 *[,exp-2, . . ., exp-n];*

statements

END;

As our example showed, the variable following the DO may have any data type. The list of expressions must contain at least one expression. The expression may have results of different data types, as long as the results can be converted to the data type of the variable following the DO.

The flowchart for the DO list looks like this:


More in list?
No
Execute statement following DO list
Yes
Evaluate next expression on list
Assign value of expression to variable in DO statement
Execute the statements in the group

SD-01021

Please note the following about the way PL/I executes the DO list:

1. Each expression is evaluated just before the loop uses it. Hence, the value of an expression on the list may depend on the evaluation of an expression that precedes it in the list, but not vice-versa.

2. As with other DO groups, all the statements in the group are executed once execution of the group begins. To stop execution, you must branch out of the group.

To give you a practical example of how DO list works, let's rewrite COMP_INT so that the user can input up to five different rates of interest for a given principal and time. The variables for the DO LIST loop and the loop itself look like this:

```
  DECLARE (RATE,RATE_1,RATE_2,RATE_3,RATE_4,RATE_5)
       FIXED DECIMAL(4,4);
       •
       •
    PUT FILE(OUT) LIST("INPUT PRINCIPAL AND TIME");
    GET FILE(IN) LIST(PRINCIPAL,TIME);

/* GET THE 5 RATES FOR THE INTEREST CALCULATION */

    PUT FILE(OUT) SKIP LIST("INPUT UP TO 5 RATES"!!
                            "OF INTEREST.  IF YOU INPUT"!!
                            "LESS THAN 5, USE 0 FOR THE"!!
                            "REST.");

    GET FILE(IN) LIST(RATE_1,RATE_2,RATE_3,RATE_4,RATE_5);

/****************************************/
/* DO LIST FOR UP TO 5 INTEREST RATES */
/****************************************/

    DO RATE = RATE_1,RATE_2,RATE_3,RATE_4,RATE_5;
       IF RATE = 0 THEN STOP;

/*********************************/
/* COMPOUND INTEREST CALCULATION */
/*********************************/

       NEWPRINC = PRINCIPAL;
       DO I = 1 TO TIME;
           INTEREST = ROUND(NEWPRINC * RATE,2);
           NEWPRINC = NEWPRINC + INTEREST;
       END; /* COMPOUND INTEREST LOOP */

       INTEREST = NEWPRINC - PRINCIPAL;

       PUT FILE(OUT) LIST("THE RATE IS", RATE,
                          "THE INTEREST IS", INTEREST);
  END; /* DO LIST */
```

Figure 4-2. COMP_INT with a DO List

As you can see, the DO increment loop for the interest calculation is nested in the DO list loop.

The above program stops after it has calculated the interest of the desired rates. How would you have to change it to make it return and ask for another set of rates? What would replace the STOP statement in the new version?

Stream Files and List-Directed I/O

Until now, we have used only two special files: an input file which was attached to the terminal keyboard and an output file which was attached to the terminal screen. In this section, we will show you how to set up and use other stream input and output files.

As you saw with the files we attached to the terminal, you determine the characteristics of a PL/I file when you open it. The OPEN statement for the terminal as an input file looked like this:

```
OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
```

This statement defined IN as a file with the following characteristics:

1. It is a stream file, so its data will be a stream of ASCII characters.

2. It is an input file, so you can only use it as a source of data for the program.

3. The TITLE keyword attaches it to the AOS dataset whose name appears in parentheses following the keyword, in this case, the generic dataset for the terminal keyboard, @INPUT.

You open other stream files exactly the same way. If you want a stream output file to output its data to the AOS generic @LPT dataset, the OPEN statement might look like this:

```
OPEN FILE (PRINTFILE) STREAM OUTPUT TITLE("@LPT");
```

If you use the TITLE keyword in the OPEN statement, you can attach your PL/I file to any AOS dataset. If you do not use the TITLE keyword, PL/I assumes that the AOS dataset and the PL/I file have the same name. The OPEN statement:

```
OPEN FILE(RESULTS) STREAM OUTPUT;
```

attaches the PL/I file RESULTS to an AOS dataset of the same name.

Why use the TITLE keyword at all? There are two situations where it is particularly useful:

1. When you want to use an AOS dataset in your program whose name contains characters which are illegal in PL/I. We could not simply write OPEN FILE (@INPUT) etc. in our programs because @ is an illegal PL/I character.

2. When the input file or output file may be one of several AOS datasets. When this is the case, you can use a character-string variable following the TITLE option and can assign the name of the dataset you want to use to the variable before you open the file. Take as an example:

```
FILENAMES:
            PROCEDURE;

            DECLARE FILENAME CHARACTER(20) VARYING;
            DECLARE(IN,OUT,RESULTS)FILE;

            OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

            PUT FILE(OUT) LIST("NAME THE FILE YOU"!!
                              "WANT YOUR RESULTS IN");
            GET FILE(IN) LIST(FILENAME);

            OPEN FILE(RESULTS) STREAM OUTPUT TITLE(FILENAME);

            PUT FILE(RESULTS) LIST("2 + @ =", 2 + 2);
END; /* FILENAME */
```

If you reply to the prompt by inputting an AOS dataset name, say SUM, and then do the CLI command TYPE SUM after FILENAMES is finished, you will find the output, "2 + 2 = " 4, in the AOS dataset SUM.

The above example also shows what happens when PL/I executes an OPEN statement for a file. If you opened the file as a stream input file, PL/I looks for an AOS dataset which has the name you gave with the TITLE keyword, or if you did not use the keyword, an AOS dataset which has the same name as the PL/I file. If there is no such dataset or if AOS cannot find the dataset, the program will raise the error condition.

If, as in the example, the file is opened as a stream output file, PL/I looks for an AOS dataset with the appropriate name. If it finds such a dataset, it deletes the dataset and creates a new one with the same name; otherwise, it creates a new dataset. Consequently, if you use the same dataset name over and over in executing our example, the dataset would be deleted and created over and over and would never contain more than

```
"2 + 2 =" 4
```

There are two exceptions to the above: when you open a PL/I stream output file which has as its dataset the AOS generic datasets @LIST or @OUTPUT, PL/I appends the new output to the dataset rather than deleting it. Hence, if you used @LIST as the dataset name in the above example, ran the program three times, and then printed the contents of your LIST file, you would get a series of outputs:

```
"2 + 2 ="   4
"2 + 2 ="   4
"2 + 2 ="   4
```

The CLOSE Statement

The CLOSE statement closes a PL/I file. We have not had to use the statement up to now because PL/I automatically closes all open files when it executes a STOP statement or a program's final END statement. We introduce it now because you must close a file with the CLOSE statement before you can use it again with a different set of file attributes.

The CLOSE statement has the syntax:

```
CLOSE FILE (file-exp);
```

If the file expression is the name of an open file, PL/I closes the file; otherwise, it continues on to the next statement. Note that you must write a separate CLOSE statement for every file you close.

To see how and where to use CLOSE, let's look at a program that opens a file as an output file, outputs data to it, closes the file, and then uses it as an input file:

```
OPENINGS:
          PROCEDURE;

          DECLARE (IN,OUT,MATERIAL) FILE;
          DECLARE (A,B,C,D) FIXED BINARY;

          OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
          OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

/* OPEN FILE MATERIAL AS AN OUTPUT FILE */

          OPEN FILE(MATERIAL) STREAM OUTPUT;

          PUT FILE(OUT) LIST("INPUT 4 INTEGERS");
          GET FILE(IN) LIST(A,B,C,D);
```

```
/* OUTPUT DATA TO FILE MATERIAL */

             PUT FILE(MATERIAL) LIST(A,B,C,D);

/* CLOSE FILE MATERIAL AND REOPEN AS AN INPUT FILE */

             CLOSE FILE(MATERIAL);
             OPEN FILE(MATERIAL) STREAM INPUT;

/* INPUT DATA FROM FILE MATERIAL */

             GET FILE(MATERIAL) LIST(A,B,C,D);
             PUT FILE(OUT) LIST("THE SUM IS", A+B+C+D);
END; /* OPENINGS*/
```

GET LIST and PUT LIST with Stream Files

AOS PL/I allows you to use three different sets of I/O statements with stream files: GET LIST and PUT LIST, GET EDIT and PUT EDIT, and READ and WRITE. Here, we will deal only with GET LIST and PUT LIST; we discuss the others in detail in Chapter 7.

I/O with GET LIST and PUT LIST is list-directed I/O, because the data in the file takes the form of a list of items. Each item on the list is followed by a terminator. In PL/I the terminators may be either commas or spaces.

When your program uses PUT LIST to output data to a file which it opened with the STREAM OUTPUT attributes, it automatically gives the data the proper format for list-directed I/O. If the item it is outputting is character-string data, it puts quotation marks around the item. If the item is bit-string data, it converts the bit string to a sequence of 1's and 0's, puts quotation marks around the sequence, and follows it with a "B". Otherwise, it converts the item to a string of ASCII characters. Each item is followed by a space. Hence, if you take five variables:

```
DECLARE NAME CHARACTER(20) VARYING;
DECLARE (PRINCIPAL,INTEREST) FIXED DECIMAL(7,2);
DECLARE RATE FIXED DECIMAL(4,4);
DECLARE TIME FIXED BINARY;
```

With these values:

```
"JOHN Q. CUSTOMER"
1000.00
120.00
.12
1
```

A PUT LIST statement like this:

```
PUT FILE(RESULTS) LIST(NAME,PRINCIPAL,INTEREST,RATE,TIME);
```

will write the following to the file RESULTS:

```
"JOHN Q. CUSTOMER" 1000.00 120.00 .12 1
```

Because PUT LIST formats data this way, you can use GET LIST to read files made by PUT LIST. Note, however, that there is one exception: if the file was opened with the STREAM OUTPUT PRINT attributes, PUT LIST formats the output for the lineprinter. As part of the formatting process, it removes the quotes from character strings and inserts tab characters into the output line. If the strings contain terminators, GET LIST will be unable to read the file correctly. For details on PRINT, see Chapter 7.

If you write your own files for GET LIST to read, you have to be sure to format them properly. Here are the rules:

1. Each data item must be followed by a space or a comma.

2. Arithmetic data items may not have commas or spaces within the data item. PL/I will not take a number written as 1,000.

3. If a character-string data item includes commas or spaces, the whole item must be enclosed in quotation marks. If you write JOHN SMITH instead of "JOHN SMITH", the computer will interpret JOHN SMITH as 2 items.

4. The last data item in the file must be followed by a terminator and a newline character.

Thus, if you want to write your own file which will hold data for the variables NAME, PRINCIPAL, RATE, and TIME of the previous example, it must look like this if you use spaces as terminators:

```
"JOHN SMITH" 3000 .10 5 <NEWLINE>
```

or like this if you use commas:

```
"JOHN SMITH",3000,.10,5,<NEWLINE>
```

You can, of course, use several spaces where you use a single space, you can surround commas with spaces, and you can mix commas and spaces:

```
"JOHN SMITH" , 300 .10 5 <NEWLINE>
```

As long as each item is followed by a comma or a space, you can also insert tabs or newline characters between items. You can see how GET LIST works from the format of the files it reads. When your program executes a GET LIST statement, it reads the file until it finds the next terminator. It then assigns everything between where it started and the terminator to the first variable in the input list. It then reads to the next terminator and assigns everything between the two terminators to the next variable in the input list. The process continues until there are no variables in the input list or until there is no more data in the file. For example:

```
GET FILE(DATA) LIST(NAME,PRINCIPAL,RATE,TIME);
```

would read "JOHN SMITH" and assign it to NAME, 3000 and assign it to PRINCIPAL, .10 and assign it to RATE, and 5 and assign it to TIME.

What happens if the GET LIST statement can find no more data?

That depends on the AOS dataset you are reading data from. here, as elsewhere, the AOS generic dataset for terminal input, @INPUT, works differently from other AOS datasets. If you use a GET LIST statement with a file opened as @INPUT and do not input enough data for all the variables on the input list, the GET LIST statement simply waits for the rest. With any other AOS dataset, if the GET LIST statement cannot find enough data for all the variables on the input list, the programs raises the endfile condition. Unless you use an ON-unit (see Chapter 13) to specify otherwise, the endfile condition outputs an error message and stops program execution.

To make sure you understand how list-directed I/O works, write a file that GET LIST can read and then write a program that reads data from the file and outputs it to another file. Experiment with the format of the input file and see what happens when there is too little data in the file.

After that program is working, try rewriting your interest program so that it gets its data from an AOS dataset other than @INPUT, and outputs it to an AOS dataset other than @OUTPUT.

End of Chapter

Chapter 5
Arrays and Structures

Arrays and structures are collections of variables.* When you use arrays and structures in programs, you can refer either to the collection as a whole or to parts of it.

Arrays are collections of variables in which all the variables have the same data type. The individual variables in an array are the elements of the array. To refer to an element, you use the array name with a subscript. For example, you might declare an array called NAMES which had 3 character varying elements like this:

```
DECLARE NAMES(3) CHARACTER(20) VARYING;
```

If you wanted to assign a value to the third variable in NAMES, you could do it like this:

```
NAMES(3) = "ADAM SMITH";
```

You can also use expressions for subscripts, so you could set all the variables in NAMES to "" like this:

```
DO I = 1 TO 3;
          NAMES(I) = "";
END;
```

If you want to refer to the whole array at once, you use the name without subscripts:

```
PUT FILE(OUT) LIST(NAMES);
```

This PUT LIST statement will output the values of all the variables in NAMES.

Structures are collections of variables in which the variables may have different data types. The variables belonging to a structure are the structure's members. The members of a structure may be elementary variables, arrays, or other structures. To refer to individual variables in a structure, you use structure-qualified names. For instance, you might declare a structure BORROWER which has an elementary variable member NAME, an array structure REFERENCES, and a structure member LOAN like this:

```
DECLARE 1 BORROWER,
          2 NAME CHARACTER(20) VARYING,
          2 REFERENCES(3) CHARACTER(20) VARYING,
          2 LOAN,
                   3 PRINCIPAL FIXED DECIMAL(8,2),
                   3 RATE FIXED DECIMAL(4,4),
                   3 TIME FIXED BINARY,
                   3 INTEREST FIXED DECIMAL(8,2);
```

*NOTE: PL/I also has arrays of label constants. See Chapter 11.

The variables PRINCIPAL, RATE, TIME, and INTEREST are members of the structure LOAN. To refer to a variable in a structure, you write the variable's name, preceded by the name of the structure of which it is member, preceded by the name of the structure of which that structure is a member, and so on, until you reach the name of the first structure. You separate the names with a ".". This form of referencing is called structure-qualified reference. A structure-qualified reference to the variable INTEREST in BORROWER looks like this:

```
BORROWER.LOAN.INTEREST
```

Hence, a simple interest calculation with the variables of the structure BORROWER.LOAN would look like this:

```
BORROWER.LOAN.INTEREST = BORROWER.LOAN.PRINCIPAL *
BORROWER.LOAN.RATE * BORROWER.LOAN.TIME;
```

If you want to refer to a whole structure at once, you use the name of the structure without the names of its members:

```
PUT FILE(OUT) LIST(BORROWER)
```

will output all the values in the structure BORROWER:

```
PUT FILE(OUT) LIST(BORROWER.LOAN)
```

will output all the values in the stucture LOAN of BORROWER.

Operators with Array and Structure Operands

AOS PL/I does not allow you to use arrays and structures as operands with the operators. Operands in AOS PL/I are restricted to expressions that yield a single value.

Arrays

Declaring Arrays

The declaration of the array NAMES is typical:

```
DECLARE NAMES(3) CHARACTER(20) VARYING;
```

As you can see, it looks exactly like the declaration of an elementary character varying variable except for the (3) following NAMES. The (3) is the dimension attribute for the array NAMES. The dimension attribute for an array gives the number of dimensions in the array and the lower and upper bounds of the subscripts in each dimension. As we will see later, the (3) declares NAMES as an array of one dimension with a lower subscript bound of 1 and an upper subscript bound of 3.

The declaration of an array consists of the array's name the dimension attribute, and the data type of the array's elements. You can write the dimension attribute two ways: as a set of parentheses immediately following the array's name, or as a set of parentheses immediately following the DIMENSION keyword.

If you use the DIMENSION keyword, you can put the dimension attribute anyplace in the array's declaration:

```
DECLARE NAMES CHARACTER(20) VARYING DIMENSION(3);
```

This declaration is exactly equivalent to the preceding one.

Arrays in AOS PL/I may have as many as 7 dimensions. The dimensions appear in the dimension attribute as a list separated by commas. For example, you would declare a two-dimensional array of integers called SQUARE like this:

```
DECLARE SQUARE_1 (2,2) FIXED BINARY;
```

Each dimension has a lower bound and an upper bound. The lower bound is the smallest value which a subscript for that dimension may have; the upper bound is the largest value the subscript may have. You may use any integer values for the bounds of a dimension as long as the lower bound is less than the upper bound. If you only use one value for the bounds of a dimension, PL/I assumes that the value is the upper bound and that the lower bound is 1. Each dimension in the array SQUARE above has a lower bound of 1 and an upper bound of 2.

If you want to give an array a lower bound other than 1, you write the lower bound followed by a colon, followed by the upper bound. SQUARE, declared with bounds of -1 and 0 in one dimension and 0 and 1 in the other, would look like this:

```
DECLARE SQUARE_2(-1:0, 0:1) FIXED BINARY;
```

The bounds of a dimension determine its extent. The extent of the dimension is the number of integers, including the bounds, between the lower bound and the upper bound. The single dimension of NAMES has an extent of 3, and each dimension of SQUARE_1 and SQUARE_2 has an extent of 2. The number of elements in an array is the product of the extents of all the dimensions. NAMES has 3 elements, while SQUARE_1 and SQUARE_2 each have 4 elements.

Subscripted References

As we saw above, you use subscripted references to refer to individual elements of an array. A subscripted reference has the form:

array-name(fixed-bin-exp-1 *[,fixed-bin-exp-2,. . .,fixed-bin-exp-n]);*

You can use any expression for a subscript which yields a fixed binary value. Note that AOS PL/I will not convert subscripts of other data types to fixed binary. Note also that expressions which have mixed fixed decimal and fixed binary operands yield fixed decimal results in AOS PL/I, and that you therefore cannot use such expressions as subscripts.

A subscripted reference must have as many subscripts as the array has dimensions. Consequently, references to elements of NAMES will have a single subscript, while references to elements of SQUARE will have to subscripts. Your program is in error if the value of a subscript is outside the bounds of that dimension of the array. For example, SQUARE_1 has the dimension attribute (2,2), so a reference to SQUARE(0,1) is in error. AOS PL/I does not perform run-time checks for subscript values which are out of bounds unless you compile with the /SUB switch. This switch generates a good deal of extra code, so you should only use it while you are debugging.

GET LIST and PUT LIST with Arrays

GET LIST treats an array as a collection of individual variables. When you use a subscripted name, it works exactly the same way as the name of an unsubscripted elementary variable. For example, if you have a file called ECONOMISTS which looks like this:

```
"ADAM SMITH" "KARL MARX" "JOHN M.KEYNES"
```

you can write the following loop:

```
DO I = 3 TO 1 BY -1;
            GET FILE(ECONOMISTS) LIST(NAMES(I));
END;
```

This loop will assign ADAM SMITH to NAMES(3), KARL MARX to NAMES(2), and so on.

If you use an unsubscripted array name in a GET LIST statement, PL/I will assign the items in the file to the elements of the array in row-major order, that is, so that the rightmost subscript will change most of and the leftmost least often. For instance, if you had a file of integers, 1,2,3,4,5 . . . and wrote the statement:

```
GET FILE(NUMBERS) LIST(SQUARE_1);
```

the program would assign 1 to SQUARE_1(1,1), 2 to SQUARE_1 (1,2), 3 to SQUARE_1(2,1), and 4 to SQUARE_1(2,2). Of course, if there are more elements in the array than there is data in the file, the program will raise the endfile condition.

PUT LIST treats an array as a collection of expressions. If you use a subscripted name in PUT LIST, the statement outputs the value of that element of the array; if you use the array name without subscripts, it outputs the values for the whole array in row-major order. The statement

```
PUT FILE(OUT) LIST(SQUARE)
```

would first output the value of SQUARE(1,1), then of SQUARE(1,2), and so on.

Implied DO Increment with GET and PUT LIST

When you want to use GET LIST or PUT LIST with part of an array, or when you want to input or output values in other than row-major oder, you can use an implied DO increment in the GET or PUT statement. For example, you can write a GET LIST statement which works the same way as the DO loop we used to read data from the file ECONOMISTS:

```
GET FILE(ECONOMISTS) LIST((NAMES(I) DO I = 3 TO 1 BY -1));
```

The syntax of the implied DO in the GET or PUT statement has several peculiarities

1. The DO and the variable it belongs to form a single item in the input list or output list.

2. You must enclose the variable and its DO in parentheses.

3. You can nest implied DO's. When you do, each of the DO's has a set of parentheses enclosing itself and the variable. For example, a GET LIST statement which assigns values to SQUARE_1 in column-major order might look like this:

   ```
   GET FILE(WHOLES) LIST(((SQUARE_1(I,J)DO I = 1 TO 2)DO J = 1 TO 2));
   ```

   The first item will be assigned to SQUARE_1(1,1), the second to SQUARE_1(2,1), and so forth.

Assignment with Arrays

Assignment to elementary variable elements of an array works the same way as assignment to any other elementary variable. If the element and the value assigned to it have different data types, precision, scale, or length, PL/I will convert the value to the data type and precision, scale, or length of the array element. Take the following as an example:

```
SQUARE_1(1,1) = 3.56;
```

Since the elements of SQUARE_1 have the fixed binary data type, the fractional part of 3.56 will be truncated and SQUARE_1(1,1) will have the value 3.

PL/I also allows you to assign one array to another. Unlike some other PL/I's, AOS PL/I will not perform conversions when one array is assigned to another. Consequently, you can use arrays in assignment statements only under the following conditions:

1. The elements of both arrays must have the same data types, and either the same precision and scale, or length.

2. Both arrays must have the same number of dimensions and the same bounds.

   For example:

```
DECLARE LIST(4) FIXED BINARY;
DECLARE TABLE(2,2) FIXED BINARY;
DECLARE SQUARE(2,2) FIXED BINARY;
.
TABLE = SQUARE;
```

   is a legal assignment, since both arrays have the same type of element, the same number of dimensions, and the same bounds in each dimension.

```
TABLE = LIST;
```

   on the other hand, is not, because the two arrays do not have the same number of dimensions.

Using Arrays: a Simple Sort Program

To pull together everything we've said about arrays so far, let's look at a simple program that inputs a list of names into an array, assigns that array to a second array, puts the names in the second array into alphabetic order, and then outputs both arrays.

```
SORT:
  PROCEDURE;

  DECLARE (U_NAMES,S_NAMES)(20) CHARACTER(20) VARYING;
  DECLARE NAME CHARACTER(20) VARYING;
  DECLARE I FIXED BINARY;
  DECLARE SW BIT;

  DECLARE (UNSORTED,SORTED) FILE;

  OPEN FILE(UNSORTED) STREAM INPUT;
  OPEN FILE(SORTED) STREAM OUTPUT;

  GET FILE(UNSORTED) LIST(U_NAMES);

/* ASSIGN U_NAMES TO S_NAMES */

  S_NAMES = U_NAMES;

/* BUBBLE SORT S_NAMES */

  SW = "1"B;
  DO WHILE(SW);
    DO I = 1 TO 19;
    SW = "0"B;
    IF S_NAMES(I) > S_NAMES(I + 1)
      THEN
      DO;
      NAME = S_NAMES(I + 1);
      S_NAMES(I + 1) = S_NAMES(I);
      S_NAMES(I) = NAME;
      SW = "1"B;
      END;

      END; /* DO-INCREMENT */
  END; /* DO WHILE*/

  PUT FILE(SORTED) LIST(S_NAMES,U_NAMES);
END; /* SORT */
```

Figure 5-1. SORT Gives You an Alphabetic List

If the file UNSORTED looks like this:

```
MADISON JEFFERSON LINCOLN TRUMAN EISENHOWER
HARDING JOHNSON ADAMS FORD GARFIELD
COOLIDGE ROOSEVELT KENNEDY NIXON WILSON
MCKINLEY GRANT CARTER MONROE WASHINGTON
```

the program will produce a file SORTED that looks like this:

```
"ADAMS" "CARTER" "COOLIDGE" "EISENHOWER" "FORD" "GARFIELD" "GRANT" "HARDING"
"JEFFERSON" "JOHNSON" "KENNEDY" "LINCOLN" "MADISON" "MCKINLEY" "MONROE" "NIXON"
"ROOSEVELT" "TRUMAN" "WASHINGTON" "WILSON" "MADISON" "JEFFERSON" "LINCOLN" "TRUMAN"
"EISENHOWER" "HARDING" "JOHNSON" "ADAMS" "FORD" "GARFIELD" "COOLIDGE" "ROOSEVELT"
"KENNEDY" "NIXON" "WILSON" "MCKINLEY" "GRANT" "CARTER" "MONROE" "WASHINGTON"
```

Try writing a program of your own that gets a list from a file and assigns it to an array and then does things with the array and its elements.

Structures

Declaring Structures

The declaration of the structure BORROWER in the introduction to this chapter is a typical PL/I structure declaration:

```
DECLARE 1 BORROWER,
              2 NAME CHARACTER(20) VARYING,
              2 REFERENCES(3) CHARACTER(20) VARYING,
              2 LOAN,
                          3 PRINCIPAL FIXED DECIMAL(8,2),
                          3 RATE FIXED DECIMAL(4,4),
                          3 TIME FIXED DECIMAL,
                          3 INTEREST FIXED DECIMAL(8,2);
```

The members of the structure BORROWER are the elementary variable NAME, the array REFERENCES, and the structure LOAN. The members of LOAN are the elementary variables PRINCIPAL, RATE, TIME, and INTEREST. Since LOAN is a member of another structure, it is called a substructure; BORROWER, which is not a member of another structure, is a main structure.

You must declare all variables which belong to a main structure in a single DECLARE statement. As you can see from the example, you declare the main structure by writing 1 followed by the main structure name. A space must separate the 1 from the main structure's name. Note that structure names do not have data type attributes, and that a comma follows the declaration:

```
DECLARE 1 BORROWER,
```

You must precede the declarations of the members of the main structure with an integer constant greater than 1. A space separates the integer from the member's name. You must follow each declaration with a comma. If a member is scalar variable, you must declare its data type, precision, and scale:

```
2 NAME CHARACTER(20) VARYING,
```

If a member is an array, you must give its dimension attribute along with the data type and precision, scale, or length of its elements.

```
2 REFERENCES(3) CHARACTER(20) VARYING;
```

If a member is a structure, you declare the structure the same way you declared the major structure name. You declare the members of the substructure the same way you declare the members of the major structure, except that you must precede the declarations of the members of the substructure with an integer constant greater than the integer which preceded the declaration of the substructure's name. The declaration of the last item in the structure ends with a semicolon.

The numbers which precede the declarations in a structure are called level numbers. The structure BORROWER has three levels: the first level is the main structure BORROWER; the second level is the members NAME and LOAN of BORROWER; the third level is the members PRINCIPAL, RATE, TIME, and INTEREST of LOAN. In our example, we preceded the first level with 1, the second with 2, and the third with 3, but only the first level has to have the number 1. We could have used any integer larger than 1 for the second level, and any integer larger than the we used for the second level for the third level. Hence, the level numbers could just as well have 1, 3, 6 as 1, 2, 3. This feature is sometimes useful when you want to leave room to add levels to a structure.

Structure-qualified References

As we saw in the introduction, you use structure-qualified references to refer to members of a structure. A structure-qualified reference has the form:

main-structure-name *[.level-2-structure-name. ... .*
level-n-structure-name].member-name

If the reference refers to an elementary variable member of a structure or substructure, qualified name will contain the elementary variable name preceded by the name of the structure to which it belongs, preceded by the name of the structure that structure is a member of and so on back to the main structure:

```
BORROWER.LOAN.RATE
```

If a reference to an array member refers to the entire array, you write it like the reference to an elementary variable:

```
BORROWER.REFERENCES
```

If the reference refers to an element of the array, then the array name will have a subscript. You would refere to the third element of REFERENCES like this:

```
BORROWER.REFERENCES (3)
```

If a reference refers to a substructure, you write it like the reference to an elementary variable:

```
BORROWER.LOAN
```

Of course, if you want to refer to a member of the substructure, you have to qualify the member's name with thhe substructure's name.

When a qualified reference gives all of the substructures between the last name in the reference and the main structure, it is called a fully-qualified reference. As long as no ambiguities result, you can also use partially-qualified references: for example, if no other substructure of the structure BORROWER has a member RATE, you can write:

```
BORROWER.RATE
```

In fact, if no ambiguities result, you can simply use the member name without qualification. Be careful, though, to give enough qualification. If the compiler finds a reference that may refer either to a variable in a structure or to a variable outside the structure, it will treat the reference as if it refers to the variable that does not belong to the structure.

GET LIST and PUT LIST with Structures

GET LIST treats a structure as a collection of individual variables. If you use an elementary variable member of a structure or an element of an array member in a GET LIST statement, the statement will assign a single value to the variable. If you use an array member, it will assign values to the array the same way it does to other arrays. If you use a structure member or the main structure, the GET LIST assigns values to all the elementary variables or array elements belonging to the structure's first member, then to those belonging to its second member, and so on until it has assigned values to all the elementary variables and elements in the structure. Take as an example a file named DEBTORS containing the following:

```
"J.D.MAMMON" "A.CARNEGIE" "J.D.ROCKEFELLER" "C.VANDERBILT"
1000 .12 5 7623.41
```

and you read this file with a

```
GET FILE(DEBTORS) LIST(BORROWER);
```

the GET LIST statement will assign "J.D.MAMMON" to BORROWER.NAME, "A.CARNEGIE" through "C.VANDERBILT" to BORROWER.REFERENCES, and the next four items to the members of BORROWER.LOAN.

PUT LIST treats the structure as a collection of expressions, but otherwise it works just like GET LIST: if you have assigned BORROWER the items described above,

```
PUT FILE(OUT) LIST(BORROWER);
```

will output "J.D.MAMMON" "A.CARNEGIE" "J.D.ROCKEFELLER" "C.VANDERBILT" 10000 .12 5 7623.41.

Assignment with Structures

When you assign a value to an elementary variable belonging to a structure, it is no different from assigning a value to any other elementary variable. If necessary, PL/I will convert the value to the data type and precision, scale, or length of the variable.

Similarly, when you assign an array to an array member of an structure, the same rules hold as for the assignment of one array to another: for the assignment to work, the elements of the array must have the same data type and precision, scale, or length, and the arrays must have the same number of dimensions and the same extents in each dimension.

PL/I also allows you to assign a structure to a structure. Here, as with arrays, AOS PL/I does not allow conversion, and as with arrays, structure assignment is only possible if each member of the one structure is exactly equivalent to the corresponding member of the other structure. The rules are as follows:

1. If the member is an elementary variable, its data type, and precision, scale, or length must be the same as that of the corresponding member of the other structure.

2. If the member is an array, the corresponding member must be an array whose elements have the same data type, and precision, scale, or length, and which has the same number of dimensions and equal extents in each dimension.

3. If the member is a structure, its elementary variable array, and structure members must all be equivalent to the members of the corresponding structure.

If you have the following set of declarations:

```
DECLARE 1 BORROWER,
            2 NAME CHARACTER(20) VARYING,
            2 REFERENCES(3) CHARACTER(20) VARYING,
            2 LOAN;
                    3 PRINCIPAL FIXED DECIMAL(8,2),
                    3 RATE FIXED DECIMAL(4,4),
                    3 TIME FIXED BINARY,
                    3 INTEREST FIXED DECIMAL(8,2);

DECLARE 1 BORROW_DUMMY,
            2 NM CHARACTER(20) VARYING,
            2 R(3) CHARACTER(20) VARYING,
            2 LN,
                    3 PR FIXED DECIMAL(8,2),
                    3 RT FIXED DECIMAL(4,4)
                    3 T FIXED BINARY,
                    3 INT FIXED DECIMAL(8,2);

DECLARE CUSTOMER CHARACTER(20) VARYING;

DECLARE REFS(3) CHARACTER(20) VARYING;

DECLARE 1 LOAN_AMT,
            2 PRINC FIXED DECIMAL(8,2),
            2 RTE FIXED DECIMAL(4,4),
            2 TI FIXED BINARY,
            2 INTRST FIXED DECIMAL(8,2);
```

then the following assignments are legal:

```
BORROWER.NAME = CUSTOMER;

BORROWER.REFERENCES = REFS;

BORROWER.LOAN = LOAN_AMT;

BORROWER.REFERENCES(1) = REFS(3);

BORROWER.LOAN.PRINCIPAL = LOAN_AMT.PRINC;

BORROWER = BORROW_DUMMY;
```

The following assignment would not be legal since the two structures are not equivalent:

```
BORROWER = LOAN_AMT;
```

Arrays of Structures

A PL/I array may have structures as its elements. To declare an array of structures, you include a dimension attribute in the declaration of the structure name. A declaration of an array of 100 structures like BORROWER might look like this:

```
DECLARE 1 BORROWER(100),
             2 NAME CHARACTER(20) VARYING,
             2 REFERENCES(3) CHARACTER(20) VARYING,
             2 LOAN,
                          3 PRINCIPAL FIXED DECIMAL(8,2),
                          3 RATE FIXED DECIMAL(4,4),
                          3 TIME FIXED BINARY,
                          3 INTEREST FIXED DECIMAL(8,2);
```

References with arrays of structures work like references with other arrays: if you use a name belonging to the structure without a subscript, it refers to that variable in all the elements of the array. For example, if you wrote

```
PUT FILE(OUT) LIST(BORROWER);
```

PL/I would output all the data in all 100 structures of the array. If you wrote:

```
GET FILE(IN) LIST(BORROWER.NAME);
```

PL/I would get 100 items from the file and assign the first item to BORROWER(1).NAME, the second to BORROWER(2).NAME, and so forth.

To refer to the variables in individual elements of an array of structures, you have to use subscripted names. For instance, you could refer to the stucture LOAN in the 50th element of BORROWER like this:

```
BORROWER(50).LOAN
```

If you want to refer to a single element of an array of belonging to a structure which is in turn an element of an array of structures, you have to include both the structure's subscript and the subscript of the array element:

```
BORROWER(500).REFERENCES(1)
```

refers to the first element of the array REFERENCES in the 500th element of the array of structures BORROWERS.

Your programs will be clearer if you write each subscript after the name to which it belongs, but PL/I also allows you to write them after any name in the reference:

```
BORROWER.REFERENCES(50,1)
```

This is the same as BORROWER(500).REFERENCES(1). The only restriction is that the subscripts must have the same order as the names they belong to.

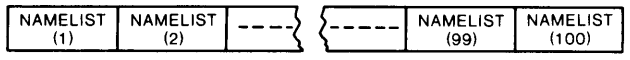
Connected and Unconnected Arrays

When you refer to members of individual structures in an array of structures, you use subscripted names, just like when you refer to elements of an array. However, the array formed by the members of structures which are elements of an array of structures is not the same as an array formed by variables which are not members of structures.

The difference lies in the way the computer stores the variables in the two arrays. If you have an array NAMELIST which you have declared like this:

```
DECLARE NAMELIST(100) CHARACTER(20) VARYING;
```

the variables making up the array will be stored in a single block staring with NAMELIST(1) and ending with NAMELIST(100):



SD-01022

The variables in the array formed by the member NAME of the structures in BORROWER, on the other hand, are not stored as a single block. The storage for each variable NAME is followed by the storage for the other variables in the structure:

```
BORROWER(1).NAME,BORROWER(1).REFERENCE(1),BORROWER(1).REFERENCE(2),

BORROWER(1).REFERENCE(3),BORROWER(1).LOAN.PRINCIPAL,

BORROWER(1).LOAN.RATE,BORROWER(1).LOAN.TIME,

BORROWER(1).LOAN.INTEREST,BORROWER(2).NAME...
```

The array BORROWER(100).NAME is unconnected.

The substructure array BORROWER(1).REFERENCE(3) is connected.

An array stored in a single block is called a connected array; one in which the elements are separated by other variables is an unconnected array. In AOS PL/I, unconnected arrays are not equivalent to connected arrays. Hence, you cannot assign the array NAMELIST to the array BORROWER.NAME. Instead, you have to assign single elements of the one array to single elements of the other array:

```
DO I = 1 TO 100;
            NAMELIST(I) = BORROWER(I).NAME;
END;
```

Using Structures

To give you a prctical example of how you can use structures and arrays of structures, we have rewritten our compound interest program using an arrray of structures to hold the data. The data is in a file called LOAN_APPS. The first item in the file is the number of loan applications in the file. The rest of the data is arranged as follows:

```
borrower's name principal rate time
```

For example:

```
"ADAMS, SAMUEL" 5000 .12 5
```

The program reads the file into an array of structures, calculates the compound interest for each loan, sorts the array into alphabetic order by the name of the applicant, and outputs the array into a file LOANS.

```
COMP_INT5:  /* WITH AN ARRAY OF STRUCTURES */
   PROCEDURE;

/**************************************************************/
/**************************************************************/
/*                        DECLARATIONS                        */
/**************************************************************/

/* BORROWER IS THE ARRAY OF STRUCTURES FOR THE DATA */

   DECLARE 1 BORROWER(100),
     2 NAME CHARACTER(20) VARYING,
     2 LOAN;
       3 PRINCIPAL FIXED DECIMAL(8,2),
       3 RATE FIXED DECIMAL(4,4),
       3 TIME FIXED BINARY,
       3 INTEREST FIXED DECIMAL(8,2);

/* DUM_BORROWER IS A SINGLE STRUCTURE EQUIVALENT TO */
/* AN ELEMENT OF BORROWER.  IT IS USED IN THE SORT  */

   DECLARE 1 DUM_BORROWER,
     2 NME CHARACTER(20) VARYING,
     2 LO,
       3 PR FIXED DECIMAL(8,2),
       3 RA FIXED DECIMAL(4,4),
       3 TI FIXED BINARY,
       3 INTER FIXED DECIMAL(8,2);

/* OTHER VARIABLES FOR THE COMPOUND INTEREST CALCULATION */

     DECLARE NEWPRINC FIXED DECIMAL(8,2);
     DECLARE INT FIXED DECIMAL(8,2);

/* A SWITCH VARIABLE FOR THE SORT */

     DECLARE SW BIT(1);

/* COUNTERS */

     DECLARE (I,J) FIXED BINARY:
     DECLARE LOAN_NO FIXED BINARY;  /* NO.OF ITEMS IN THE FILE */


/* FILES */

     DECLARE LOAN_APPS FILE; /* FILE OF LOAN APPLICATIONS */
     DECLARE LOANS FILE; /* SORTED FILE OF LOANS */

/**************************************************************/
/**************************************************************/
/*                      START OF PROGRAM                      */
/**************************************************************/

     OPEN FILE(LOAN_APPS) STREAM INPUT;
     OPEN FILE(LOANS) STREAM OUTPUT;

/*************************************************************/
/*  THE FIRST ITEM IN LOAN_APPS IS THE NUMBER OF LOAN       */
/*  APPLICATIONS IN THE FILE.  ASSIGN IT TO LOAN_NO         */
```

Figure 5-2. COMP_INT5 Reads into an Array of Structures

```
/*************************************************************/

     GET FILE(LOAN_APPS) LIST(LOAN_NO);

/* READ LOAN_APPS INTO THE ARRAY OF STRUCTURES */

     DO I = 1 TO LOAN_NO;
     GET FILE(LOAN_APPS) LIST
        (BORROWER(I).NAME,BORROWER(I).LOAN.PRINCIPAL,
         BORROWER(I).LOAN.RATE,BORROWER(I).LOAN.TIME);
     END;
/************************************************************/
/*  CALCULATE COMPOUND INTEREST FOR EACH LOAN IN THE ARRAY */
/************************************************************/

     DO I = 1 TO LOAN_NO;
     NEWPRINC = BORROWER(I).LOAN.PRINCIPAL;

       DO J = 1 TO BORROWER(I).LOAN.TIME;
       INT = ROUND(NEWPRINC * BORROWER(I).LOAN.RATE,2);
       NEWPRINC = NEWPRINC + INT;
       END;

     BORROWER(I).LOAN.INTEREST = NEWPRINC - BORROWER(I).LOAN.PRINCIPAL;

     END; /* INTEREST CALCULATION */

/************************************************************/
/* SORT THE ARRAY INTO ALPHABETIC ORDER BY THE NAME OF THE */
/*                       CUSTOMER                          */
/************************************************************/

   SW = "1"B; /* TURN ON THE SORTING LOOP */

   DO WHILE(SW);
   SW = "0"B;
     DO I = 1 TO LOAN_NO - 1;
     IF BORROWER(I).NAME > BORROWER(I + 1).NAME
       THEN
       DO;
       DUM_BORROWER = BORROWER(I + 1);
       BORROWER(I + 1) = BORROWER(I);
       BORROWER(I) = DUM_BORROWER;
       SW = "1"B;
       END;
     END;
   END; /* SORTING LOOP */

/*************************************************************/
/*   OUTPUT THE SORTED LIST OF LOANS TO THE OUTPUT FILE     */
/*************************************************************/

   DO I = 1 TO LOAN_NO;
   PUT FILE(LOANS) SKIP LIST(BORROWER(I));
   END;

END; /* COMP_INT5 */
```

Figure 5-2. COMP_INT5 Reads into an Array of Structures (continued)

For an input file like this:

```
4 "NELSON,G.E." 1000 .12 3 "SMITH,JOHN" 3000 .07 10 "WILLIAMS,PHILLIP"
7500 .10 2 "ADAMS,GEORGE" 5000 .14 5
```

you get an output file like this:

```
"ADAMS,GEORGE"    5000.00  0.1400  5   4627.07
"NELSON,G.E."     1000.00  0.1200  3   404.92
"SMITH,JOHN"      3000.00  0.0700  10  2901.40
"WILLIAMS,PHILLIP" 7500.00 0.1000  2   1575.00
```

Of course, the possibilities for adding to this program are infinite. You might want to modify it so that you could use many different AOS files for input or output, you might want to add a loop which catches bad data and outputs the bad data to an error file, or you might want to make a loop which calculates the average loan, the average interest, or the average time.

End of Chapter

Chapter 6
Blocks

Introduction

A block is a syntactical unit in a PL/I program in which an identifier has a unique meaning. The block structure of a PL/I program determines the area in which the references to a name refer to a single declaration of the name. Blocks also control the allocation of storage for the kind of PL/I variables we have been using up to now. These variables have PL/I's default automatic storage class. When your program begins executing a block, it sets up storage for each of the automatic variables declared in the block; when it finishes executing the block, it frees the storage belonging to the variables. PL/I's dynamic allocation of storage during program execution has several advantages: most importantly, it allows a program to use the same storage over and over for different values. It also lets you write programs that allocate only as much storage as is actually required for the data used in a given block activation. Finally, it lets you give the same name different meanings in different segments of the program.

PL/I has two types of blocks, BEGIN blocks and PRODECURE blocks. Both define the area in which a declaration has a specific meaning, and both may have their own storage allocations, but they differ in their syntax and the manner in which they are executed.

You have already used PROCEDURE blocks. BEGIN blocks are sequences of statements which begin with a BEGIN statement and end with an END statement. Like DO groups, BEGIN blocks count as a single statement; you can thus use them in THEN or ELSE clauses or as a statement in the DO CASE statement. When control encounters the BEGIN statement, it executes all of the statements between the BEGIN statement and the END statement as a single unit.

The following program can serve as an example both for BEGIN blocks and for block structure in general. The program uses PL/I's dynamic allocaton of automatic variables to set up an array of exactly the size needed to hold the data contained in an input file. The first piece of data in the file is an integer which gives the number of items in the rest of the file. The program uses the value of the integer to determine how much storage to allocate for the array that will hold the rest of the data in the file.

The block structure of the program is simple. The PROCEDURE block BLOCKS contains a single BEGIN block. The GET LIST statement which assigns the number of items in the file to ITEM_NO is in the PROCEDURE block. Since the value of ITEM_NO is already known when the storage for the variables declared in the BEGIN block is allocated, we can use ITEM_NO in the dimension attribute of the array LIST_ARRAY. When the computer allocates storage for the BEGIN block, it will evaluate ITEM_NO and give the array that many elements.

If the input file looks like this:

```
5 FIRST SECOND THIRD FOURTH FIFTH
```

then you will receive the output:

```
"FIFTH" "FOURTH" "THIRD" "SECOND" "FIRST"
```

```
        BLOCKS:
        PROCEDURE;

        DECLARE (WORDS,OUT) FILE;
        DECLARE ITEM_NO FIXED BINARY;

        OPEN FILE(WORDS) STREAM INPUT;
        OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

     /***************************************************************/
     /*   THE FIRST ITEM IN THE FILE IS AN INTEGER GIVING THE       */
     /*   NUMBER OF WORDS IN THE FILE.                              */
     /***************************************************************/

        GET FILE(WORDS) LIST(ITEM_NO);

     /***************************************************************/
     /*      THE BEGIN BLOCK HAS ITS OWN STORAGE AND                */
     /*                   VARIABLES                                 */
     /***************************************************************/

        BEGIN;

           DECLARE I FIXED BINARY;

     /***************************************************************/
     /* DECLARE AN ARRAY USING ITEM_NO IN THE DIMENSION ATTRIBUTE*/
     /* THIS ARRAY WILL BE EXACTLY THE SIZE NEEDED FOR THE DATA  */
     /***************************************************************/

           DECLARE LIST_ARRAY(ITEM_NO) CHARACTER(20) VARYING;

     /* SINCE THE ARRAY IS EXACTLY THE SIZE WE NEED, WE CAN USE */
     /* THE ARRAY NAME WITHOUT A DO LOOP TO GET THE DATA */

           GET FILE(WORDS) LIST(LIST_ARRAY);

     /* OUTPUT THE LIST IN REVERSE ORDER */

           DO I = ITEM_NO TO 1 BY -1;
              PUT FILE(OUT) LIST(LIST_ARRAY(I));
           END;
        END; /* BEGIN BLOCK */

     END; /* BLOCKS */
```

Figure 6-1. BLOCKS Contains a BEGIN Block

The Block Structure of BLOCKS

BLOCKS consists of a PROCEDURE block that contains a BEGIN block. A schematic of the program's block structure looks like this:

```
BLOCKS:
        PROCEDURE;

        BEGIN;
                  .
                  .
        END; /*   BEGIN BLOCK */

END; /* BLOCKS */
```

As you can see, you can nest blocks in other blocks the same way that you nest DO groups in other DO groups. The nesting follows the same rules in both cases: all of the statements in the nested block must be between the block's PROCEDURE or BEGIN statement and its END statement. Interleaved blocks are illegal. You may nest BEGIN blocks in PROCEDURE blocks and vice-versa. The only difference between the block types as regards nesting is that you must nest BEGIN blocks in other blocks, while PROCEDURE blocks may stand alone.

Scope in BLOCKS

When you declare a name in PL/I, you are defining the meaning of the name in an area of your program. This area is called the scope of declaration. The scope of a given declaration of a name is determined by the program's block structure. In general, the scope of a declaration is the block in which you declare the name and all the blocks nested in that block. If you redeclare the name in a nested block, the name will refer to the new declaration in that block and in any blocks nested in it. It is an error to refer to a name in a block that is not included in the scope of the name's declaration.

For example, in BLOCKS, the relationship between the block structure and the declarations looks like this:

```
BLOCKS:
        PROCEDURE;

        DECLARE(WORDS,OUT)FILE;
        DECLARE ITEM_NO FIXED BINARY;
                  .
                  .
        BEGIN;
        DECLARE I FIXED BINARY;
        DECLARE LIST_ARRAY(ITEM_NO) CHARACTER(20) VARYING;
                                  .
                                  .
        END; /* BEGIN BLOCK */

END; /* BLOCKS */
```

The scope of the file constants WORDS and OUT, and of the fixed binary variable ITEM_NO is both BLOCKS and the BEGIN block; consequently, we can use ITEM_NO in the declaration of LIST_ARRAY and can write GET and PUT statements in the BEGIN block which use the files declared in BLOCKS. The scope of the variable I and the array LIST_ARRAY is the BEGIN block only. It would be an error to refer to these variables in BLOCKS, outside of the block in which they are declared.

PROCEDURE Blocks

You can nest PROCEDURE blocks the same way you do BEGIN blocks, and PROCEDURE blocks determine the scope of declarations the same way BEGIN blocks do, but they differ greatly in the way they are executed. As we saw with BLOCKS, BEGIN blocks are executed when control encounters the BEGIN statement. PROCEDURE blocks, on the other hand, are not executed until the program invokes them. A program may invoke a procedure in two ways: with a CALL statement or with a function reference. Since the syntax of a procedure depends on how it is invoked, you cannot invoke the same procedure both with a CALL statement and a function reference.

To see how this works, let's take a look at the program called INVOCATIONS, which gets two numbers from the terminal and uses procedures to add them. The first procedure, ADD, is invoked by the

```
CALL ADD(NUM_1,NUM_2,SUM);
```

statement; the second procedure, ADD_FUNC, is invoked by the function reference ADD_FUNC(NUM_1,NUM_2) in the second PUT LIST statement:

```
PUT FILE(OUT) LIST(ADD_FUNC(NUM_1,NUM_2));
```

```
INVOCATIONS:
   PROCEDURE;

   DECLARE (NUM_1,NUM_2) FIXED DECIMAL(10,2);
   DECLARE SUM FIXED DECIMAL(11,2);

   DECLARE(IN,OUT) FILE;

   OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
   OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

   PUT FILE(OUT) LIST("INPUT TWO NUMBERS YOU WANT TO ADD");
   GET FILE(IN) LIST(NUM_1,NUM_2);

/* INVOKE THE PROCEDURE ADD WITH A CALL STATEMENT */

   CALL ADD(NUM_1,NUM_2,SUM);

   PUT FILE(OUT) LIST(SUM);

/* INVOKE THE PROCEDURE ADD_FUNC WITH A FUNCTION REFERENCE */

  PUT FILE(OUT) LIST(ADD_FUNC(NUM_1,NUM_2));

/************************************************************/
/*                    THE PROCEDURES                        */
/************************************************************/

ADD:
   PROCEDURE(N_1,N_2,S);

/* PARAMETER DECLARATIONS */

   DECLARE (N_1,N_2) FIXED DECIMAL(10,2);
   DECLARE S FIXED DECIMAL(11,2);
   S = N_1 + N_2;

END; /* ADD */

ADD_FUNC:
   PROCEDURE(NO_1,NO_2) RETURNS (FIXED DECIMAL(11,2));

/* PARAMETER DECLARATIONS */

   DECLARE(NO_1,NO_2) FIXED DECIMAL(10,2);

/* RETURN GIVES THE EXPRESSION TO BE RETURNED TO THE INVOKING BLOCK */

   RETURN(NO_1 + NO_2);

END; /* ADD_FUNC */

END; /* INVOCATIONS */
```

Figure 6-2. INVOCATIONS Shows Two Ways to Call Procedures -- with a CALL Statement, and a Function Reference

When the computer executes this program, it does the statements from

```
OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
```

through

```
GET FILE(IN) LIST(NUM_1,NUM_2);
```

in sequential order. When it comes to the

```
CALL ADD(NUM_1,NUM-2,SUM);
```

statement, it goes to the procedure ADD. It executes the procedure and then returns to the statement following the CALL statement, here, the

```
PUT FILE(OUT) LIST(SUM);
```

statement. After the program has executed this statement, it executes the next PUT LIST statement, which contains the function reference ADD_FUNC. ADD_FUNC invokes the procedure ADD_FUNC the same way that ADD in the CALL statement invoked the procedure ADD.

Function references are expressions, and you can use them the same way you use other expressions. When the computer evaluates a function reference, it invokes the procedure whose name appears in the function reference. When the procedure is executed, it returns a value to the function reference. After the procedure has returned the value, the program uses the value in the statement which contains the reference. Consequently, when INVOCATIONS executes the PUT LIST statement, it is really invoking the procedure ADD_FUNC, executing the procedure, and then outputting the value which the procedure returns.

PL/I uses arguments and parameters to transmit data to and from PROCEDURE blocks. In the

```
CALL ADD(NUM_1,NUM_2,SUM);
```

statement, a list of three variable names appear in parentheses following the procedure name ADD. These variables are the arguments. NUM_1 and NUM_2 contain the values we want to add, and the procedure will return the sum to the variable SUM.

If we look at the beginning of ADD, we see the following:

```
ADD:
            PROCEDURE(N_1,N_2,S);

/* PARAMETER DECLARATIONS */

            DECLARE(N_1,N_2) FIXED DECIMAL(10,2);
            DECLARE S FIXED DECIMAL(11,2);
```

The PROCEDURE statement contains the names of three of ADD's variables. If we look at how ADD uses N_1,N_2, and S, it is clear that N_1 is equivalent to NUM_1, N_2 to NUM_2 and S to SUM. The declarations for N_1,N_2, and S give the three variables the same data types, precisions, and scales as NUM_1, NUM_2, and SUM.

N_1, N_2, and S are the parameters of ADD. When the program invokes ADD, it gives N_1 the storage of NUM_1, N_2 the storage of NUM_2, and S the storage of SUM. Because the parameters share the storage of their arguments, any change in the value of a parameter is a change in the value of its argument. Therefore, when we invoke ADD with the arguments NUM_1, NUM_2 and SUM,

```
S = N_1 + N_2;
```

has exactly the same effect as the statement:

```
SUM = NUM_1 + NUM_2;
```

Of course, the invocation will work only if there are as many arguments in the invocation as there are parmeters in the invoked procedure, and you will get correct results only if you write the arguments in the order in which their parameters appear in the PROCEDURE statement for the procedure. For instance, if you write

```
CALL ADD(NUM_1,SUM,NUM_2);
```

ADD will use the storage of NUM_1 for N_1, the storage of SUM for N_2,. and the storage of NUM_2 for S. Since SUM does not have a value before ADD is executed, you will get useless results and will lose the value stored in NUM_2.

If we look at the PUT LIST statement which invokes ADD_FUNC

```
PUT FILE(OUT) LIST(ADD_FUNC(NUM_1,NUM_2));
```

and the PROCEDURE statement and declarations for ADD_FUNC

```
ADD_FUNC:
          PROCEDURE(NO_1,NO_2) RETURNS(FIXED DECIMAL(11,2));

/* PARAMETER DECLARATIONS */

DECLARE(NO_1,NO_2) FIXED DECIMAL(10,2);
```

we can see that the arguments and parameters work the same way in function references and procedures invoked as functions as they do in CALL statements and procedures invoked with CALL. The only difference is that ADD_FUNC returns the sum of NO_1 and NO_2 to the function reference, and therefore requires only two arguments and parameters.

Procedures invoked as functions define the data type of the value to be returned to the function reference with the RETURNS keyword in the PROCEDURE statement. The data type and either the precision and scale, or the length of the value to be returned appear in parentheses following the keyword:

```
ADD_FUNC:
PROCEDURE(NO_1,NO_2) RETURNES(FIXED DECIMAL(11,2));
```

ADD_FUNC thus returns a fixed decimal value with a precision and scale of 11,2. The value itself is defined in an expression in parentheses following the RETURN statement:

```
RETURN(NO_1 + NO_2);
```

Since NO_1 and NO_2 have NUM_1 and NUM_2 as their arguments, ADD_FUNC returns the sum of NUM_1 and NUM_2 to the function reference ADD_FUNC in the PUT LIST statement. If you make your own version of INVOCATIONS and run it, you will see that

```
PUT FILE(OUT) LIST(SUM);
```

and

```
PUT FILE(OUT) LIST(ADD_FUNC(NUM_1,NUM_2));
```

output the same value.

The Syntax of Blocks

We saw in BLOCKS and INVOCATIONS that you can nest both PROCEDURE and BEGIN blocks in other PROCEDURE and BEGIN blocks. While BEGIN blocks must be nested, PROCEDURE blocks may stand by themselves.

PROCEDURE blocks that are nested in other blocks are called internal procedures; those that are not nested, like BLOCKS, are called external procedures. A PL/I program must have at least one external procedure. In AOS PL/I, you must write each external procedure on a separate source-text file and must separately compile each of the procedures. With programs having more than one external procedure, you combine the separate external procedures into a single executable program file when you bind the program.

Terminology for Block Structure

PL/I uses a special terminology to describe the relationship of blocks to one another and of statements to a block.

1. **Containment**

   Block B is contained in block A if block B is between the PROCEDURE or BEGIN and END statements of block A. For example:

```
OUTER_BLOCK:
                PROCEDURE;

                DECLARE OUTER_VAR FIXED BINARY;
                BEGIN /* 1 */

                            INNER_BLOCK:
                                          PROCEDURE;
                                          DECLARE INNER_VAR CHARACTER(20);
                            END; /* INNER_BLOCK */

                END; /* BEGIN 1 */

                BEGIN; /* 2 */

                END; /* BEGIN 2 */

END; /* OUTER_BLOCK */
```

   BEGIN block 1, BEGIN block 2, and the PROCEDURE block INNER_BLOCK are all contained in the PROCEDURE block OUTER_BLOCK. INNER_BLOCK is also contained in BEGIN block 1.

2. **Immediate Containment**

   Block B is immediately contained in block A if it is contained in block A but in no other block contained in block A. In the above example, BEGIN 1 and BEGIN 2 are immediately contained in OUTER_BLOCK. INNER_BLOCK is immediately contained in BEGIN 1.

3. **Statements Internal to a Block**

   A statement is internal to block A when it belongs to block A but not to any of the blocks nested in block A. In the example, the statement

   ```
   DECLARE OUTER_VAR FIXED BINARY;
   ```

   is internal to the PROCEDURE block OUTER_BLOCK; the statement

   ```
   DECLARE INNER_VAR CHARACTER(20);
   ```

   is internal to INNER_BLOCK, but not the BEGIN 1 or OUTER_BLOCK.

4. **Parallel Blocks**

When two blocks are immediately contained in the same block, they are parallel. In the example above, BEGIN 1 and BEGIN 2 are parallel.

Scope

In PL/I, an identifier may have either internal or external scope. When an identifer has internal scope, each declaration of the identifier has a different meaning. If you have declared a variable in one block, and then you redeclare the variable in a block contained in the first block, the second declaration has a different meaning. References to the variable in the block in which it is redeclared will refer to the new declaration. All of the variables we have used so far have had internal scope. Statement labels and the names of internal procedures also have internal scope.

When an identifier has external scope, identical declarations of the identifier have the same meaning no matter where they are in the program. File constants, the names of external procedures, and variables declared with the STATIC EXTERNAL storage class and scope attributes have external scope.

Details on the scope of procedure names follow; for details on the scope of file constants, see Chapter 7. The scope of statement labels is discussed in Chapter 11 and the scope of static internal variables in Chapter 12.

The following program illustrates the properties of names with internal and external scope.

```
SCOPE:
        PROCEDURE;

        DECLARE I FIXED BINARY;

                I = 0;

        BEGIN; /* 1 */

                DECLARE I FIXED BINARY;
                DECLARE OUT FILE;

                OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

                I = 7;
                PUT FILE(OUT) LIST(I);
        END; /* BEGIN 1 */

        BEGIN; /* 2 */

                DECLARE OUT FILE;

                PUT FILE(OUT) LIST(I);
        END; /* BEGIN 2 */

END; /* SCOPE */
```

```
Output: 7            0
```

SCOPE has three blocks: the external procedure block SCOPE and the two parallel BEGIN blocks, BEGIN 1 and BEGIN 2. The program also has two declare statements for the fixed binary variable I and two declare statements for the file constant OUT.

The two variables I both have internal scope. I is first declared in the PROCEDURE block SCOPE. Therefore, I's scope is the block SCOPE and all blocks contained in SCOPE (except those blocks in which it is redeclared). I is redeclared in the block BEGIN 1, so the scope of the I declared in SCOPE is SCOPE and BEGIN 2. The scope of the I declared in BEGIN 1 is that block only. The output for the program shows that the two variables I are in fact distinct; the first PUT LIST statement for the variable I is in BEGIN 1 and outputs the value 7 assigned to the I declared in that block. The second PUT LIST statement is in BEGIN 2, which is within the scope of the I declared in SCOPE. It outputs the value 0 assigned to that variable.

The file constant OUT, on the other hand, has external scope. All

```
DECLARE OUT FILE;
```

statements in the program are declarations of the single file constant OUT. The scope of the file constant OUT is thus both blocks in which it is declared, or in other words, BEGIN 1 and BEGIN 2. Because OUT refers to the same file in both blocks, we can open it in BEGIN 1 and use it again without reopening it in BEGIN 2.

Block Activation

When the computer executes a BEGIN statement or invokes a PROCEDURE block, it activates the block. A block remains active until the program returns from the block to the block from which it was activated. Since blocks may contain other blocks, several blocks may be active at once. In SCOPE, for example, the operating system activates the block SCOPE. When the first BEGIN statement is executed, that BEGIN block is activated. SCOPE remains active, so at this point there are two blocks active. The BEGIN block remains active until the program reaches its END statement. The program then returns to the block SCOPE, but immediately activates the second BEGIN block. This block remains active until its END statement is executed. Control then returns to SCOPE. At this point, SCOPE is the only active block, but since the only statement left in the procedure is the END statement, control returns immediately to the operating system.

PL/I uses the terms predecessor and successor to describe the relationship between active blocks. Block A is a predecessor to block B if it is active when block B is activated. Block B in turn is the successor of block A. Of course, a block may have more than 1 predecessor and successor. In SCOPE, the block SCOPE is the predecessor of both BEGIN blocks, and when they are active they are SCOPE's successors. Note, however, that neither BEGIN block is a predecessor or successor of the other.

When PL/I activates a block, it sets up storage for all the automatic variables which are declared in the block. If the block is a PROCEDURE block with parameters, it gives the parameters the storage of the invocation's arguments. The storage remains allocated until the flow of control returns to the block's pedecessor. The return to the predecessor terminates the block's activation. The parameters no longer share the storage of their arguments. The storage belonging to the block's automatic variables is freed, so that the automatic variables lose their values.

Variables and Expressions in the Declarations of Automatic Variables

As we saw in BLOCKS, the fact that storage is set up for a block when the block is activated allows you to use variables or function references in the dimension attributes of arrays. You may also use variables and functions in the length attributes of character-string and bit-string variables. Expressions that appear in dimension attributes and length attributes are called extent expressions. There are two limitations on the use of extent expressions:

1. The extent expression must yield a fixed binary value. AOS PL/I does not convert values of other data types.

2. The program must be able to evaluate the expression when it activates the block to which the DECLARE statement belongs.

There are some corollaries to these limitations:

1. Because expressions with mixed fixed binary and fixed decimal operands have fixed decimal results in AOS PL/I, you cannot use them as extent expressions.

2. Because the program must be able to evaluate the expression when it activates the block, variables in extent expressions must be parameters or have a scope which includes the block in which the DECLARE statements containing the variables are internal. Neither of these conditions is possible in the first block to be activated; consequently, the extent expression in the first block must be integer constants.

In BLOCKS, the identifier LIST_ARRAY only had a single dimension, so the DECLARE statement looked like this:

```
DECLARE LIST_ARRAY(ITEM_NO) CHARACTER(20) VARYING;
```

You can also use extent expressions to specify the upper and lower array bounds. For example, the declaration of a two-dimensional array with variable upper and lower bounds might look like this:

```
DECLARE EXPAND_ARRAY(FIRST_LO:FIRST_HI,SEC_LO:SEC_HI) FIXED BINARY;
```

Of course, variable extent expressions must follow the same rules as constant extent expressions: if you only give a single bound for a dimension, the expression must always have a value greater than 0, and if you use expressions for both bounds, the lower bound may not have a greater value than the upper bound.

Since the length of a character- or bit-string variable is an extent expression, you could use a variable to give the length of the elements of LIST_ARRAY as well:

```
DECLARE LIST_ARRAY(ITEM_NO) CHARACTER(CHAR_MAX) VARYING;
```

If CHAR_MAX had a value of 10 when storage for LIST_ARRAY was allocated, the strings would have a maximum length of 10 characters.

For an example of what you can do with extent expressions, take a look at the following program. MAKE_STRINGS makes sets of single- character character strings. It asks for the character, the length of the string, and the number of strings, and then constructs the strings and outputs them.

The program consists of an external PROCEDURE block, MAKE_STRINGS, which contains a BEGIN block. The PROCEDURE block gets the information about the character to be used in the strings, the size of the strings, and the number of the strings. The BEGIN block then uses the variables with the values for the size of the strings and the number of the strings in its declaration of an array of character varying variables:

```
DECLARE STRING_ARRAY(STRING_NO)_ CHARACTER(STRING_LEN)VARYING;
```

It uses the values again in the set of nested DO loops that actually constructs the array of strings.

If you input "A", 5, and 3 to this program, you will get the output:

```
"HERE ARE YOUR STRINGS" "AAAAA" "AAAAA" "AAAAA"
```

Block Structure and Program Efficiency

Blocks allow more efficient use of storage and make a program easier to construct and understand, but there is some overhead involved: each time a program enters a block, PL/I executes the routines required for the block activation, and when it leaves the block, it executes the routines that terminate a block activation. Consequently, you should not use blocks where the logic of your program does not require them. In particular, you should use non-iterative DO groups instead of BEGIN blocks in situations that do not require storage allocation. For example:

```
IF S_NAMES(I) > S_NAMES(I + 1)
            THEN
            DO;
            NAME = S_NAMES(I + 1);
            S_NAMES(I + 1) = S_NAMES(I);
            S_NAMES(I) = NAME;
            END;
```

```
MAKE_STRINGS:
   PROCEDURE;

   DECLARE (STRING_NO,STRING_LEN) FIXED BINARY;
   DECLARE STRING_CHAR CHARACTER(1);

   DECLARE (IN,OUT) FILE;

   OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
   OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

   PUT FILE(OUT) LIST("INPUT THE CHARACTER YOU WANT"!!
                      "IN THE STRING");
   GET FILE(IN) LIST(STRING_CHAR);

   PUT FILE(OUT) LIST("INPUT THE NO. OF CHARACTERS"!!
                     "IN THE STRING");
   GET FILE(IN) LIST(STRING_LEN);

   PUT FILE(OUT) SKIP LIST("INPUT THE NUMBER OF STRINGS");
   GET FILE(IN) LIST(STRING_NO);

   BEGIN;
     DECLARE STRING_ARRAY(STRING_NO) CHARACTER(STRING_LEN)VARYING;
     DECLARE (I,J) FIXED BINARY;

     DO I = 1 TO STRING_NO;
     STRING_ARRAY(I) = "";
       DO J = 1 TO STRING_LEN;
       STRING_ARRAY(I) = STRING_ARRAY(I)!!STRING_CHAR;
       END;
     END;

     PUT FILE(OUT) LIST("HERE ARE YOUR STRINGS",STRING_ARRAY);
   END; /* BEGIN BLOCK */
END; /* MAKE_STRINGS */
```

Figure 6-3. MAKE_STRINGS Constructs an Array

is more efficient than:

```
IF S_NAMES(I) > S_NAMES(I + 1)
          THEN
          BEGIN;
          NAME = S_NAMES(I + 1);
          S_NAMES(I + 1) = S_NAMES(I);
          S_NAMES(I) = NAME;
          END;
```

Procedures

When you use procedures in a PL/I program, observe the following:

1. The order of the arguments in the invocation and the relationship of the arguments to the parameters.

2. The type of invocation and the syntax of the procedure it invoked.

3. Whether the procedure is an internal procedure or an external procedure.

We will discuss each of these areas below.

Parameters and Arguments

Declaring Parameters

As we saw in the program INVOCATIONS, the DECLARE statements for parameters look like the DECLARE statements for any other variable. You indicate that a variable is a parameter when you include its name in the list of parameters following the PROCEDURE keyword:

```
ADD:
            PROCEDURE(N_1,N_2,S);

/* PARAMETER DECLARATIONS */

            DECLARE (N_1,N_2) FIXED DECIMAL(10,2);
            DECLARE S FIXED DECIMAL(11,2);
```

Note that the DECLARE statement for the parameter must contain the parameter's data type and precision, scale, or length, but that the parameter appears in the PROCEDURE statement as a simple reference.

Asterisk Extent Expressions in Parameter Declarations

Since PL/I knows the extents of the arguments when it invokes a procedure, it can automatically give parameters the extents of their arguments. When you want PL/I to do this, you use the "*" character in the extent expressions of the parameters. For example, if you wanted to write a procedure which could sort one-dimensional arrays of any size consisting of character varying variables of any length, you could write the PROCEDURE statements and the array parameter like this:

```
STRING_SORT:
            PROCEDURE(ARRAY);

            DECLARE ARRAY(*) CHARACTER(*) VARYING;
```

If you invoke STRING_SORT with an array argument which you declare like this:

```
DECLARE NAME_LIST(200) CHARACTER(30) VARYING;
.
.
CALL STRING_SORT(NAME_LIST);
.
```

PL/I will automatically give the parameter ARRAY a dimension attribute of 200 and a length attribute of 30.

Of course, if an array has more than one dimension, you will need more than one *. If you used the two-dimensional array TABLE as a parameter, you might declare it like this:

```
DECLARE TABLE (*,*) FIXED BINARY;
```

Note that if you do not use * in the extent expressions of parameters, you must use integer constants. You may not use variables or other expressions.

The Array and String Built-in Functions DIM, HBOUND, LBOUND, and LENGTH

The advantages of using * extents in parameter declarations are obvious; the only problem is knowing the value of the extent expressions for a given invocation of the procedure. To solve this problem, PL/I has four built-in functions: DIM, HBOUND, LBOUND, and LENGTH.

DIM returns the extent of a single dimension of an array. The function has the format:

```
DIM(array-name,dimension-no)
```

The dimension-no must be an integer constant. If we wanted to find out the extent of the single dimension of the array parameter ARRAY, we could do it like this:

```
DIM(ARRAY,1)
```

If the procedure STRING_SORT was invoked with the argument NAME_LIST, DIM will return the value 200.

HBOUND returns the upper bound of a single dimension of an array and LBOUND returns the lower bound of a single dimension. Their formats are like the format of DIM:

```
HBOUND(array-name,dimension-no)
LBOUND(array-name,dimension-no)
```

We could use these functions in the DO increment statement of the sort:

```
DO I = LBOUND(ARRAY,1) TO HBOUND(ARRAY,1);
```

If STRING_SORT was invoked with NAMELIST, LBOUND(ARRAY,1) will have the value 1 and HBOUND(ARRAY,1) will have the value 200.

LENGTH returns the length of a character-string or bit-string data item. If the item is a character varying variable, it returns the current length of the value contained in the variable; otherwise, it returns the length of the bit-string variable or constant. LENGTH has the format:

```
LENGTH(string-exp)
```

You can use LENGTH to check for strings which are too long to be output. PL/I cannot output strings longer than 132 characters, so you could write a check like this:

```
DO I = LBOUND(ARRAY,1) TO HBOUND(ARRAY,1);
IF LENGTH(ARRAY(I)) <= 132
            THEN PUT FILE(OUT) LIST(ARRAY(I));
            ELSE PUT FILE(OUT) LIST("STRING TOO LONG");
END;
```

Of course, LENGTH has many other uses in string handling. For examples, see Chapter 8.

Parameter Passing in PL/I

In the example program INVOCATIONS, the parameters of ADD shared the storage of the variables used as arguments in ADD's invocation. Any change in the value of the parameters was a change in the value of the arguments. When a program passes data between arguments and parameters by having the parameters share the argument's storage,, it is passing parameters by reference. PL/I also allows you to pass parameters by value. When a program pases parameters by value, the parameters do not share storage with variables in the invoking procedure; instead, they share temporary storage, which is set up at the time of invocation and freed when the invocation ends. Since the parameters do not share storage with variables of the invoking procedure, changes in the parameters do not result in changes to variables of the invoking procedure.

Rules for Passing by Value and Passing by Reference

How data is passed between an argument and its parameter depends on the argument. Parameters must be variables but you can use constants, variables, function references, and compound expressions as arguments. In general, PL/I passes constants, compound expressions, functions, and variables that are not equivalent to their parameters by value. Only variables that are equivalent to their parameters are passed by reference.

1. Parameters are passed by value when:

   a. An elementary variable argument does not correspond to its parameter. In this case, the argument is converted to the data type, and precision, scale, or length of the parameter.

   b. The argument is enclosed in parentheses. You can use this convention to force passing by value even when the argument and parameter are corresponding elementary variables.

   c. The argument is a literal constant or a named constant.

   d. The argument is a function reference.

   e. The argument is a compound expression.

   Note that you cannot pass data aggregates by value.

2. Parameters are passed by reference when the argument is a variable that is exactly correspondent to its parameter. An argument is correspondent to its parameter when:

   a. An argument that is an elementary arithmetic variable has a parameter of the same data type, precision, and scale.

   b. An argument that is an elementary character-string or bit-string variable has the same data type and length as its parameter. If you use * in the parameter's length attribute, the parameter is equivalent to an argument of any length.

   c. Arguments which are single elements of arrays of elementary variables or elementary variable members of structures are passed by reference when their parameters are correspondent elementary variables. If you have an array declaration like this:

      ```
      DECLARE INTEGERS(10)FIXED BINARY;
      ```

      and a parameter declared like this:

      ```
      DECLARE INT FIXED BINARY;
      ```

      then INTEGERS(1) will be passed by reference if it is used as an argument for INT.

   d. An argument which is an array has elements of the same data type and has the same number of dimensions and the same bounds in each dimension as its parameter. If you use * for the extent expressions in an array parameter, the parameter is correspondent to an argument with any extents.

   e. An argument which is a structure has the same types of members in the same positions as its parameter. You may use the * in the length and dimension attributes of structure parameters.

Take the following procedure as an example for these rules:

```
ARGS:
            PROCEDURE;

            DECLARE(ARG_1,ARG_2) FIXED BINARY;
            DECLARE OUT FILE;

            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

            ARG_1 = 1;
            ARG_2 = 1;

            CALL PARAMS(ARG_1,(ARG_2));

            PUT FILE(OUT) LIST(ARG_1,ARG_2);

PARAMS:
            PROCEDURE(VAL_1,VAL_2);

            DECLARE(VAL_1,VAL_2) FIXED BINARY;
            VAL_2 = VAL_1 + 1;
            VAL_1 = VAL_1 +_VAL_2;
            END; /* PARAMS */

END; /* ARGS */
```

Output: 3 1

In the example, ARG_1 has the parameter VAL_1 and ARG_2 has the parameter VAL_2. Both arguments are correspondent to their parameters, but we put parentheses around ARG_2, which forces PL/I to pass by value. As you can see, ARG_1 changes its value when VAL_1 does, but changes in the value of VAL_2 have no effect on ARG_2.

Types of Procedures

Procedures Invoked with Call

The CALL statement itself is simple enough. It has the form:

```
CALL entry-expression[(arg-1[,arg-2, ... ,arg-n)];
```

So far, we have simply used the names of procedures in the statement. Procedure names are entry constants. You can use any expression in CALL which has an entry constant as a value. Aside from entry constants, PL/I has entry variables, and you may write functions which return entry values. For now, we will continue to use entry constants in CALL. For details on the other possibilities see Chapter 13.

As we saw in INVOCATIONS, the procedure name must be followed by as many arguments as the invoked procedure has parameters. The arguments may of course be constants, variables, compound expressions, or function references.

The PROCEDURE statement for procedures invoked with CALL has the format:

```
identifier: PROCEDURE[(param-1][, . . . , param-n])];
```

You must precede PROCEDURE statements with single label prefixes. The label prefix is the procedure name. A colon separates it from the PROCEDURE statement. When you write a label prefix ahead of a procedure statement, you are declaring the prefix as an entry constant. The scope of the declaration will depend on whether the procedure is an internal or external procedure.

The PROCEDURE statement must have as many references to parameters as the CALL statement has arguments. If the procedure has no parameters, there can be no parentheses following PROCEDURE. When the program invokes the procedure, it gives each parameter the storage of its corresponding argument.

Returning from a Called Procedure

In INVOCATIONS, the program returned from ADD after it had executed the procedure's END statement. You can also use the RETURN statement to return from CALLED procedures. RETURN in called procedures has the form:

```
RETURN;
```

When the program encounters the RETURN statement, it does the same thing as when it executes the procedure's END statement: it returns to the statement following the CALL statement which invoked the procedure.

Since END and RETURN do the same thing, you need use RETURN in a called procedure only when you want a branch to return control to the invoking procedure. Take as an example the procedure ROOT in the program CALL_INVOKE. ROOT uses RETURN to return control to the invoking procedure if it can perform the calculation. Otherwise, it outputs an error message and executes a STOP statement.

```
CALL_INVOKE:
          PROCEDURE;

          DECLARE(IN,OUT)FILE;
          DECLARE INTEGER FIXED BINARY;
          DECLARE RESULT FLOAT BINARY(53);

          OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
          OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

          PUT FILE(OUT) LIST("INPUT AN INTEGER");
          GET FILE(IN) LIST(INTEGER);

          CALL ROOT(INTEGER,RESULT);
          PUT LIST("THE SQUARE ROOT OF", INTEGER, "IS", RESULT);

ROOT:
          PROCEDURE(NUMBER,RES);

          DECLARE NUMBER FIXED BINARY;
          DECLARE RES FLOAT BINARY(53);

          IF NUMBER >=0
          THEN
                   DO;
                            RES = SQRT(NUMBER);
                            RETURN;
                            END;
                            PUT  FILE(OUT)  LIST("SQRT  OF  NEGATIVE  NUMBERS  NOT
                            DEFINED");
                            STOP;
          END; /*  ROOT */
END; /* CALL_INVOKE */
```

Procedures Invoked with Function References

Function references are expressions. You can use a function reference in any context that allows an expression, but since an expression in PL/I must represent a single value, you cannot use a function reference to return an array or structure.

Function references look like this:

```
entry-expression([arg-1, . . ., arg-n]);
```

The entry expression must have an entry constant as a value. As with the entry expression in the CALL statement, the expression may be an entry constant, an entry variable, or a function which returns an entry value.

Note that the parentheses following the entry expression are not optional in a function reference. You may write functions which take no arguments, but the compiler cannot identify an identifier as a function reference if it does not have the parentheses. Take the following as an example:

```
FUNKY:
            PROCEDURE;

            DECLARE OUT FILE;

            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");
            PUT FILE(OUT) LIST(GET_FIVE());

GET_FIVE:
            PROCEDURE RETURNS(FIXED BINARY);

            RETURN(5);
            END; /* GET_FIVE */

END; /* FUNKY */
```

Even though the function GET_FIVE always returns 5 and therefore takes no arguments, the function reference must have the form GET_FIVE( ).

FUNKY also illustrates the syntax of procedures invoked with function references. The parameters work the same way as they do in procedures invoked by CALL. Note that if there are no arguments, you cannot write parentheses after the PROCEDURE statement. The PROCEDURE statement for these procedures must have the RETURNS attribute:

```
identifier: PROCEDURE[(parameter-1, . . . ,parameter-n)]
RETURNS(data-type-attribute);
```

The data type attribute gives the data type of the value returned to the function reference. You can only use integer constants in the data type attribute's length attributes.

The procedure itself must contain a special form of the RETURN statement:

```
RETURN(exp);
```

When the program executes this statement, it converts the value given in the expression to the data type specified after the RETURNS keyword in the PROCEDURE statement and returns the value to the function reference. Note that the RETURN(exp) statement is not optional in a procedure invoked as a function. If you attempt to return from the function in any other way, the expression containing the function reference will not be evaluated.

Returning from a Block with a GOTO

PL/I allows you to use a GOTO to return from a block, but the returns via the END statement and the RETURN statement generally make the use of the GOTO unnecessary. For details on exiting from a block via the GOTO, see Chapter 11.

Side Effects with Procedures

If a procedure is contained in another block, the scope of the variables in the block includes the procedure. In consequence, you can use these variables in the procedure. Of course, anytime you change the value of the variable in the contained procedure, you have also changed its value in the containing block. Such changes are called side effects. For example:

```
SIDE_EFFECT:
            PROCEDURE;

            DECLARE VAL FIXED BINARY;
            DECLARE OUT FILE;

            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

            VAL = 2;
            CALL EFFECT;
            PUT FILE(OUT) LIST(VAL);

            EFFECT:
                       PROCEDURE;

                       VAL = 7;
                       END; /* EFFECT */

END; /* SIDE_EFFECT */
```

The program will output 7 because the invocation of EFFECT changed the value of VAL. In this example, the side effect is merely misleading, but side effects can cause serious problems.

For example, if you have a DO increment statement like this:

```
DO I = 1 TO HI(B) BY INTERVAL;
```

and the execution of the function HI(B) either depends on or changes the value of the variable INTERVAL, your program will have unpredictable results, since you do not know whether PL/I will execute HI(B) before or after it evaluates INTERVAL.

The best way to avoid problems with side effects is to declare all the variables you use in a procedure in the procedure. That way, the scope of all the variables will only be the procedure itself and the execution of the procedure will only affect the values of the arguments with which you invoke it.

The Scope of Procedure Names

Like other PL/I declared names, the label prefixes attached to PROCEDURE statements have scope. The scope of a procedure name determines the area of the program from which you can invoke the procedure. The names of internal and external procedures have different scopes.

The Scope of Internal Procedure Names

Until now, we have only invoked internal procedures. You declare an internal procedure name when you write the PROCEDURE statement to which it belongs. Internal procedure names have internal scope. The scope of the declaration is the block which immediately contains the PROCEDURE block and all of the blocks contained in that block, unless you redeclare the name in one of the contained blocks.

If a program has the following block structure:

```
OUTER:
          PROCEDURE;
          .
          .
          BEGIN; /* 1 */
                    .
                    .
INNER_1:
          PROCEDURE;
                    .
                    .
END; /* INNER_1 */
INNER_2:
          PROCEDURE;
                    .
                    .
END; /* INNER_2 */
                    .
          END; /*   BEGIN 1 */
END; /* OUTER */
```

then the scope of both INNER_1 and INNER_2 is the BEGIN block and all the blocks contained in it. You can thus invoke both INNER_1 and INNER_2 from each other and from the BEGIN block, but not from OUTER. Note that you can also invoke a procedure from itself or from one of its successors. This is called recursive invocation and is discussed in Chapter 13.

External Procedures

External procedures are procedures which are not contained in any other blocks. In AOS PL/I, you must write each external procedure in a separate source-text file and must compile each of the source-text files separately. To make a set of external procedures into a single executable program, you bind them with a single PL1BIND macro.

Since each external procedure has its own source-text file and is separately compiled, large programs are easier to write, compile, debug, and change when you design them as a set of external procedures.

The Scope of External Procedure Names

External procedure names have external scope. When you write the PROCEDURE statement for an external procedure, you are declaring the procedure name in an imaginary outer block that contains all the external procedures in the program.

The scope of an external procedure name thus includes itself and all the other external procedures in the program. However, you cannot invoke an external procedure from another external procedure unless you have declared the procedure name as entry data in a DECLARE statement whose scope includes the block from which you want to invoke the external procedure. For example, if we write the procedure ADD as an external procedure and invoke it from another external procedure, EXT_INVOKE, it looks like this:

```
EXT_INVOKE;
            PROCEDURE;
            DECLARE ADD ENTRY(FIXED DECIMAL(10,2),FIXED DECIMAL(11,2));
            DECLARE(IN,OUT) FILE;
            DECLARE(NUM_1,NUM_2)FIXED DECIMAL(10,2);
            DECLARE SUM FIXED DECIMAL(11,2);

            OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");
            PUT FILE(OUT) LIST("INPUT TWO NUMBERS YOU WANT TO ADD");
            GET FILE(IN) LIST(NUM_1,NUM_2);

            CALL ADD(NUM_1,NUM_2,SUM);
            PUT FILE(OUT) LIST(SUM);

END; /* EXT_INVOKE */

ADD:
            PROCEDURE(N_1,N_2,S);

            DECLARE(N_1,N_2)FIXED DECIMAL(10,2);
            DECLARE S FIXED DECIMAL(11,2);
            S = N_1 + N_2;
END; /* ADD */
```

If you compare this with INVOCATIONS, you will see that there are only two differences in the way ADD is handled here and there: there, ADD was contained in INVOCATIONS; here it is not; there, there was no DECLARE statement for ADD in INVOCATIONS; HERE, EXT_INVOKE has the

```
DECLARE ADD ENTRY(FIXED DECIMAL(10,2), FIXED DECIMAL(10,2),
FIXED DECIMAL(11,2));
```

statement. A program cannot invoke ADD unless the block from which it is invoked is within the scope of a DECLARE statement like the one above.

External procedure names also have external scopes; as with file constants, identical external entry constants refer to the same external procedure. For instance:

```
OUTER:
       PROCEDURE;
       .
       BEGIN; /* 1 */
              DECLARE ADD ENTRY(FIXED DECIMAL(10,2),FIXED DECIMAL(10,2),
                                          FIXED DECIMAL(11,2));
              .
              CALL ADD(3,5.60,SUM_1);
       .
       END; /* BEGIN 1 */
       .
       BEGIN; /* 2 */
              DECLARE ADD ENTRY(FIXED DECIMAL(10,2),FIXED DECIMAL(10,2),
                                          FIXED DECIMAL(11,2));
              .
              CALL ADD(-15.32,112,SUM_2);
              .
              END; /* BEGIN 2 */
                    .
END; /* OUTER */
```

Both of the above invocations of ADD will refer to the single external procedure ADD.

The scope of an external procedure name is thus the external procedure to which it belongs and all other blocks within the scope of a DECLARE statement for the name.

The DECLARE Statement for External Procedure Names

As you can see from the examples, the DECLARE statement for an external procedure is quite complicated. It has the form:

```
DECLARE external-entry-constant ENTRY
            [(parameter-attribute-1 [, . . ., parameter-attribute-n])]
            [RETURNS(data-type-attribute)];
```

The ENTRY declaration for a procedure name is similar to the name's PROCEDURE statement: if the procedure has parameters, there must be as many parameter attributes following the ENTRY keyword as the procedure has parameters, they must be in the same order, and the attributes themselves must be the same as in the declarations of the parameters. Here, as in parameter declarations, you can use * in extent expressions. If the procedure is a function, the ENTRY declaration must have the RETURNS attribute with the data type of the value returned by the function. As you would expect, you cannot use asterisks in the data type attribute.

If you had written ADD_FUNC as an external procedure, the ENTRY declaration for the name would look like this:

```
DECLARE ADD_FUNC ENTRY(FIXED DECIMAL(10,2),FIXED DECIMAL(10,2)
RETURNS(FIXED DECIMAL(11,2));
```

The parameter attributes for array and structure parameters are even more complicated than for elementary variable parameters. With the array parameter, the dimension attribute must precede the attributes of the elements, unless you use the DIMENSION keyword. If you had written the sort routine STRING_SORT as an external procedure, the ENTRY declaration for STRING_SORT would be as follows:

```
DECLARE STRING_SORT ENTRY((*) CHARACTER(*) VARYING);
```

or

```
DECLARE STRING_SORT ENTRY(CHARACTER(*) VARYING DIMENSION(*));
```

With structure parameters, the structure names have no data types. They are represented by their level numbers only. They array and elementary variable members are preceded by their level numbers, but are otherwise treated like other elementary variable and array parameter attributes. If you made an external procedure called STRUC_SORT to sort arrays like BORROWER, the ENTRY declaration for STRUC_SORT would look like this:

```
DECLARE STRUC_SORT ENTRY(1 (*),2 CHARACTER(*) VARYING,
            2,3FIXED DECIMAL(8,2),3 FIXED DECIMAL(4,4),
3 FIXED BINARY,3 FIXED DECIMAL(8,2));
```

Writing, Compiling, and Binding Programs with Several External Procedures

In AOS PL/I, each external procedure must have its own source text file. After you have written the external procedures on their separate source text files, you compile each source file separately, just as you compiled the source text files of other procedures. You assemble your external procedures into a single program by binding the .OB files produced by the compiler into a single .PR file.

When you use the PL1BIND macro to bind the program, the main procedure comes first and the other external procedures follow. The PL1BIND command for EXT_INVOKE and ADD looks like this:

```
PL1BIND EXT_INVOKE ADD
```

Binding PL1ECIS into your Program

If your Eclipse-line computer has the commercial instruction set, you will have received the file PL1ECIS with your PL/I compiler. When you bind your program with this file, you can take advantage of the commercial instruction set's extra speed and error checking.

To bind with PL1ECIS, add PL1ECIS to your PL1BIND. For example:

```
PL1BIND EXT_INVOKE ADD PL1ECIS
```

COMP_INT with Internal Procedures

As a practical exercise, let's rewrite our COMP_INT program using a BEGIN block to set up an array of structures exactly the size needed for the loan applications. Then, we'll use called procedures to get the data, calculate the interest, sort the array, and output the data.

The only part of this which should pose any problems at all is inputting and outputting the data. We have to pass a file constant as an argument to the called procedures which handle input and output. Parameters, on the other hand, must be variables, so we have to use a new data type, the file variable. You declare a file variable with the keywords FILE VARIABLE. If you have DECLARE and CALL statements in the invoking procedure which looks like this:

```
DECLARE (LOAN_APPS, LOANS) FILE;
               .
               .
CALL INPUT (LOAN_APPS, BORROWER);
```

the parameter declarations in the procedure INPUT will look like this:

```
INPUT:

PROCEDURE(IN_FI,DEBOT);

DECLARE IN_FI FILE VARIABLE;
.
.
```

The complete text of the program follows. As you can see, the procedure COMP_INT6 has been reduced to a sequence of procedure invocations. In consequence, the reader can understand the program's basic logic at a glance.

If you want to play around with this program, try adding other called procedures which detect errors.

You should also be able to turn the internal procedures in this program into external procedures. The only difficulty here is, of course, the ENTRY declarations in COMP_INT6. To get you going, here is the ENTRY declaration for the procedure INPUT:

```
DECLARE INPUT ENTRY(FILE VARIABLE,1(*),2 CHARACTER(20) VARYING,
               2,3 FIXED DECIMAL(8,2),3 FIXED DECIMAL(4,4),
3 FIXED BINARY, 3 FIXED DECIMAL(8,2));
```

```
COMP_INT6:  /* USING INTERNAL PROCEDURES */
   PROCEDURE;

   DECLARE (LOAN_APPS, LOANS) FILE;
   DECLARE APP_NO FIXED BINARY; /* NUMBER OF LOANS IN LOAN_APP */

   OPEN FILE(LOAN_APPS) STREAM INPUT;
   OPEN FILE(LOANS) STREAM OUTPUT;

/* GET THE NUMBER OF LOAN APPLICATIONS IN THE FILE */

   GET FILE(LOAN_APPS) LIST(APP_NO);

/**************************************************************/
/*   THE BEGIN BLOCK ALLOWS DYNAMIC STORAGE ALLOCATION       */
/*          BASED ON THE VALUE OF APP_NO                     */
/**************************************************************/

   BEGIN; /* 1 */

     DECLARE 1 BORROWER(APP_NO),
       2 NAME CHARACTER(20) VARYING,
       2 LOAN,
         3 PRINCIPAL FIXED DECIMAL(8,2),
         3 RATE FIXED DECIMAL(4,4),
         3 TIME FIXED BINARY,
         3 INTEREST FIXED DECIMAL(8,2);

     DECLARE I FIXED BINARY;

/* GET THE INPUT FOR BORROWER */

     CALL INPUT(LOAN_APPS,BORROWER);

/* CALCULATE THE INTEREST FOR EACH ELEMENT OF BORROWER */

     DO I = A TO APP_NO;
       CALL INT_CALC(BORROWER(I).LOAN);
     END;

/* SORT BORROWER BY BORROWER.NAME */

     CALL SORT(BORROWER);

/* OUTPUT THE RESULTS */

     CALL OUTPUT(LOANS,BORROWER);

   END; /* BEGIN_1 */

/*************************************************************/
/*                    PROCEDURES                             */
/*************************************************************/

INPUT:  /* READS A FILE INTO AN ARRAY */
   PROCEDURE(IN_FI,DEBTOR);

/* PARAMETERS */
```

Figure 6-4. COMP_INT6 Uses Internal Procedures

```
   DECLARE IN_FI FILE VARIABLE,
   DECLARE 1 DEBTOR(*),
     2 NA CHARACTER(20) VARYING,
     2 LN,
       3 PRINC FIXED DECIMAL(8,2),
       3 RTE FIXED DECIMAL(4,4),
       3 TI FIXED BINARY,
       3 INT FIXED DECIMAL(8,2);

/* VARIABLES BELONGING TO INPUT */

   DECLARE I FIXED BINARY;

   DO I = 1 TO HBOUND(DEBTOR,1);
       GET FILE(IN_FI) LIST(DEBTOR(I));
   END;

END; /* INPUT */

INT_CALC:  /* CALCULATES COMPOUND INTEREST */
   PROCEDURE(LN);

/* PARAMETER */

   DECLARE 1 LN,
     2 PRINC FIXED DECIMAL(8,2),
     2 RTE FIXED DECIMAL(4,4),
     2 TME FIXED BINARY,
     2 INT FIXED DECIMAL(8,2);
/* VARIABLES BELONGING TO INT_CALC */

   DECLARE NEWPRINC FIXED DECIMAL(8,2);
   DECLARE INT FIXED DECIMAL(8,2);
   DECLARE I FIXED BINARY;

   NEWPRINC = LN.PRINC;

   DO I = 1 TO LN.TME;
     INT = ROUND(NEWPRINC * LN.RTE,2);
     NEWPRINC = NEWPRINC + INT;
   END;

   LN.INT = NEWPRINC - LN.PRINC;

END; /* INT_CALC */

SORT:  /* SORTS THE LOANS BY BORROWER NAME */
   PROCEDURE(DEBTOR);

/* PARAMETER */

   DECLARE 1 DEBTOR(*),
     2 NA CHARACTER(20) VARYING,
     2 LN,
       3 PRINC FIXED DECIMAL(8,2),
       3 RTE FIXED DECIMAL(4,4),
       3 TI FIXED BINARY,
       3 INT FIXED DECIMAL(8,2);

/* VARIABLES BELONGING TO SORT */

   DECLARE I FIXED BINARY;
   DECLARE SW BIT;
   DECLARE I DUM_DEBTOR,
     2 N CHARACTER(20) VARYING,
     2 L,
```

Figure 6-4. COMP_INT6 Uses Internal Procedures (continued)

```
       3 PR FIXED DECIMAL(8,2),
       3 R FIXED DECIMAL(4,4),
       3 T FIXED BINARY,
       3 ITRST FIXED DECIMAL(8,2);

   SW = "1"B;

   DO WHILE (SW);
   SW = "0"B;
     DO I = TO HBOUND(DEBTOR,1) - 1;
     IF DEBTOR(I).NA > DEBTOR(I+1).NA
       THEN
       DO;
       DUM_DEBTOR = DEBTOR(I + 1);
       DEBTOR(I + 1) = DEBTOR(I);
       DEBTOR(I) = DUM_DEBTOR;
       SW = "1"B;
       END;
     END;
   END; /* SORT LOOP */
END; /* SORT */

OUTPUT:  /* OUTPUT THE SORTED ARRAY */
   PROCEDURE(OUT_FI,DEBTOR);

/* PARAMETERS */

   DECLARE OUT_FI FILE VARIABLE;
   DECLARE 1 DEBTOR(*),
     2 NA CHARACTER(20(VARYING,
     2 LN,
       3 PRINC FIXED DECIMAL(8,2),
       3 RTE FIXED DECIMAL(4,4),
       3 TI FIXED BINARY,
       3 INT FIXED DECIMAL(8,2);

/* VARIABLE FOR OUTPUT */

   DECLARE I FIXED BINARY

   DO I = 1 TO HBOUND(DEBTOR,1);
     PUT FILE(OUT_FI) SKIP LIST(DEBTOR(I));
   END;

END; /* OUTPUT */

END; /* COMP_INT6 */
```

Figure 6-4. COMP_INT6 Uses Internal Procedures (continued)

End of Chapter

Chapter 7
Input/Output

A program does I/O when it transmits data between memory and an external source or destination. The source or destination for the data is called a data set. The data set may be an I/O device, such as a terminal, tape unit, card reader, or lineprinter, or it may be a disk file. Data sets are represented in PL/I programs by named constants called file constants. Before you can perform I/O in a PL/I program, you must declare the file constant and open the file. When you open a file, you associate the file constant with a data set and define how the program will use the file. Once you have opened the file, you can use I/O statements to transmit data. To give you a concrete example of the above, let's look at a program that does nothing but read a single data item from one file and write it to another:

```
I_O:
            PROCEDURE;

            DECLARE(SOURCE,DEST) FILE;
            DECLARE WORD CHARACTER(20) VARYING;

            OPEN FILE(SOURCE) STREAM INPUT;
            OPEN FILE(DEST) STREAM OUTPUT;

            GET FILE(SOURCE) LIST(WORD);
            PUT FILE(DEST) LIST(WORD);

            CLOSE FILE(SOURCE);
            CLOSE FILE(DEST);
END; /* I_O */
```

The first DECLARE statement in the program declares two file constants, SOURCE and DEST. The OPEN statements then open the files. The OPEN statements for SOURCE and DEST define them both as stream files; that is, as files in which the data is a continuous sequence of ASCII characters. SOURCE is further defined as an input file; that is, a file from which data will be read. DEST is defined as an output file; that is, one to which data will be written. Neither OPEN statement explicitly specifies a data set name, so PL/I assumes that the data set associated with the file SOURCE will be an AOS data set called SOURCE and that the data set associated with the file DEST will be an AOS data set of that name. When the program executes the OPEN statement for SOURCE, it looks for an AOS data set file of that name; if none exists, the program raises the error condition. When it executes the OPEN statement for DEST, it looks for an AOS data set of that name. If it finds one, it deletes it and creates a new data set of the same name; if none exists, it creates a new data set.

Once the file is open, you can use the file name in I/O statements. The kinds of I/O statements you can use with a file depend on how it was opened. In I_O, a GET LIST statement reads the data from the file SOURCE and a PUT LIST statement outputs it to DEST. You can use GET LIST only with stream input files and PUT LIST only with stream output files.

The CLOSE statements close the files. After a file has been closed, the file constant no longer represents a data set. We have used CLOSE statements in the example to show you what they look like, but they are not necessary; your program will automatically close all open files when it executes a STOP statement of the final END statement.

Of course, L_O will not work unless the data set SOURCE exists before you run the program. If you want to run this program, create a dataset named SOURCE with a text editor or the CREATE/I CLI command and input a word. Because GET LIST requires terminators, you must follow the word with a space or comma. The data set as a whole must end with a newline character. Once you have created the data set and input the word, you can run the program. If you type the CLI command TYPE DEST after it is finished, you will see that DEST now contains the word you wrote in SOURCE.

File Data

A file data item is a data item which may represent an external data set. PL/I has file constants and file variables, and you may also write functions that return

File Constants

You must declare all file constants with a DECLARE statement (except for SYSIN and SYSPRINT). The DECLARE statement for a file constant looks like this:

```
DECLARE identifier FILE;
```

File constants have external scope. Hence, any reference to a file constant within the scope of any declaration for the constant will refer to the same file.

File Variables

File variables are variables which may have file constants as values. You declare a file variable like this:

```
DECLARE identifier FILE VARIABLE;
```

File variables are like other variables: they have the default automatic storage class and internal scope, but you can also give them other storage classes.

A common use for file variables is as parameters in procedures which take file values as arguments. For example, a procedure that reads data from a file might look like this:

```
                .
                DECLARE NAME_LIST FILE;
                DECLARE NAMES(100) CHARACTER(20) VARYING;

                .

                CALL NAME_GET(NAME_LIST,NAMES);L

NAME_GET:
                PROCEDURE(FI_NAME,CHAR_ARRAY);

                DECLARE FI_NAME FILE VARIABLE;
                DECLARE CHAR_ARRAY(*) CHARACTER(*) VARYING;
                          .
                          .
```

When the CALL statement invokes NAME_GET, the file variable parameter FI_NAME will refer to the file control block belonging to its argument NAME_LIST.

Assignment and Comparison with FILE data

PL/I does not define conversions between FILE data and other data types. Therefore, you may only assign file constants to file variables or one file variable to another.

You can use file data operands with the = and ^= comparison operators. Two file data items are equal if they refer to the same file control block. For instance, if you assign a file constant to two different file variables, the variables will be equal:

```
DECLARE NUMBERS FILE;
DECLARE FI_VAR_1 FILE VARIABLE;
DECLARE FI_VAR_2 FILE VARIABLE;

.
FI_VAR_1 = NUMBERS;
.
FI_VAR_2 = NUMBERS
```

If you have this combination of declarations and assignments, then

```
FI_VAR_1 = FI_VAR_2
```

is true.

AOS PL/I File Types

When you open a PL/I file, you determine what kind of file it is. AOS PL/I has three basic file types: stream files, record sequential files, and record direct files. A file's type determines how data is stored in the file, how data is transmitted to or from the file, and how the data may be accessed.

Stream files are files in which the data is stored as a sequence of ASCII characters. When a program reads data from a stream file, it converts the ASCII representation contained in the file to the data type of the variable it assigns the data to. When it outputs data to a stream file, it converts the data to its ASCII representation. The conversions follow the rules for conversions to and from character-string data. For details, see the reference manual.

A program may only access a stream file sequentially. On input, this means that once the file has been opened, the program begins inputting data with the first item in the file. The items are read one after another until the program reaches the end of the file or closes it. When a program outputs to a stream file, the first item output is the first item in the file. The next item follows that item, and so on, until the program closes the file. PL/I uses the GET LIST, GET EDIT, and READ statements to input data from stream files; it outputs data to stream files with the PUT LIST, PUT EDIT, and WRITE statements.

With record direct and record sequential files, the data is stored in its internal representation. When a program inputs data from a record file, it copies the sequence of bits that represent the data in the file directly into the storage for the variable that appears in the input statement; when a program outputs data to a record file, it copies the sequence of bits that represent a variable in memory directly into the file. Since no conversion takes place, record I/O is faster than stream I/O.

The two types of record files differ in the manner in which a program may access them. Record sequential files must be accessed sequentially. Sequential access with record sequential files works in a similar manner as with stream files: when the program reads a record sequential file, it starts at the beginning of the file and reads one item after another; when it writes to a record sequential file, it starts at the beginning and writes one item after another. You use the READ statement without a key to get data from a record sequential file and the WRITE statement without a key to output data to a record sequential file.

Record direct files, on the other hand, allow random access. The data in record direct files is stored in blocks of 256 words. Each block has a number called a key, and you must use a block's key when you want to transmit data to or from it. You can use any key which has a record, and can thus access the blocks in any order. The record direct I/O statements are READ and WRITE with keys and the REWRITE and DELETE statements.

File Attributes

A file's type and how it is used in your program are defined by its attributes. PL/I files have three types of attributes: usage attributes, access attributes, and function attributes. A file's usage attribute defines how data is stored in the file; if the data is stored as a sequence of ASCII characters, the file has the stream usage attribute; if the data is stored in its internal representation, it has the record usage attribute.

The access attribute describes how data may be accessed; if it may be accessed sequentially, the file has the sequential access attribute; if it may be accessed randomly, it has the direct access attribute. Hence, stream files have the stream usage and sequential access attributes, record sequential files have the record usage and sequential access attributes, and record direct files have the record usage attribute and direct access attributes.

The function attribute describes how the program may use the file. There are three function attributes: input, output, and update. When you open a file with the input attribute, you may use is only as a source of data; when you open it with the output attribute, you may use it only as a destination; when you open it with the update attribute, you may use it as either. You may open all of the basic AOS PL/I file types as input or output files; you may open only record direct files as update files.

PL/I File Types and AOS Record Types

The different PL/I file types require different AOS record types. When PL/I creates an AOS data set, it automatically gives it the proper AOS record type; when you use AOS data sets as INPUT and UPDATE files in your programs, you must be sure that the data set has the right record type for the PL/I file to which it is attached. The following table gives the record types required by the PL/I file types:

PL/I file type	**AOS record type**
STREAM	data sensitive
RECORD SEQUENTIAL	variable
RECORD DIRECT	256-word fixed-length record

Opening PL/I Files

You may open a PL/I file either explicitly or implicitly. You open a file explicitly with an OPEN statement; you open it implicitly when you use the file in an I/O statement without opening it first with an OPEN statement. In both cases, a program does the following when it opens a file:

1. It defines the file's usage, access, and function attributes.
2. If the file is an input file, it associates the file with a data set. The error condition arises if there is no data set.
3. If the file is an output or update file and its data set does not exist, the program creates the data set and associates the file with it.
4. If the file is an output file and the data set already exists, the program deletes the data set, creates it again, and associates the file with it. Note that this means that you lose the contents of a data set when you open it as an output file. There are two exceptions: when you open files associated with the AOS generic data sets @OUTPUT and @LIST, the program will not delete the data sets, but merely append the new output.

Opening a File with the OPEN Statement

The OPEN statement has the following syntax:

OPEN FILE(file-exp) *[file-attributes];*

The file attribute are keywords which specify the file's usage attribute, access attribute, and function attribute. The keywords may also specify the data set associated with the file. In the case of stream output files, the manner in which the output will be formatted.

You need not give a complete set of attributes to define a file. Some attributes imply others, and if an attribute is lacking, PL/I will supply a default attribute. If the attributes you give are contradictory or if they imply contradictory attributes, your program is in error and you will receive a compile-time error message. In the following, we will give sets of attributes which completely define each of the AOS PL/I file types. For the default and implied attributes, see below.

Specifying the Data Set: The TITLE Keyword

As we saw in the program I_O, if you do not explicitly specify the name of a data set in the OPEN statement, PL/I assumes that the AOS data set has the same name as the PL/I file. To explicitly specify a data set, you use the TITLE keyword. It has the form:

TITLE(char-string-exp)

The character-string expression is a character-string constant, variable, or function which has an AOS data set name as its value. The TITLE keyword is particularly valuable when you want to use an AOS data set whose name is illegal in PL/I, or when you want to be able to choose a file when you execute the program. For example, the AOS generic data set names all have the " @ " character, which you cannot use in a PL/I identifier. Consequently, if you want to open the AOS generic data set for terminal input, @INPUT, as a stream input file, you have to do it like this:

```
OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
```

If you want to choose which data set the program uses when you execute it, you can use a character-string variable following the TITLE keyword and assign the AOS data set name to the variable:

```
            DECLARE (DATA,IN,OUT) FILE;
            DECLARE FILE_NAME CHARACTER(32) VARYING;
            .
/* OPEN THE TERMINAL AS AN INPUT AND OUTPUT FILE */

            OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

/* GET THE NAME OF THE SYSTEM FILE FROM THE TERMINAL */

            PUT FILE(OUT) LIST("WHAT FILE DO YOU WANT TO USE?");
            GET FILE(IN) LIST(FILE_NAME);

/* ASSOCIATE THE FILE DATA WITH THE SYSTEM FILE */

            OPEN FILE(DATA) STREAM INPUT TITLE(FILE_NAME);
            .
```

This program fragment associates the PL/I file DATA with the AOS dataset whose name you assign to the variable FILE_NAME.

Keywords for STREAM Files

The STREAM keyword specifies that a file is a stream file and is to be accessed sequentially. Since STREAM specifies both the access type and as the usage type, you cannot use the keyword SEQUENTIAL with the keyword STREAM.

You may use stream files as input files or as output files. The keyword for the input function attribute is INPUT, and the keyword for the output function attribute is OUTPUT.

With stream output files, you may use keywords to format your output. The keyword

```
LINESIZE(fixed-bin-exp)
```

determines the length of lines in the file. The fixed binary expression gives the length. The

```
PRINT
```

keyword formats the output for the lineprinter. With stream output print files, you may use the keyword

```
PAGESIZE(fixed-bin-exp)
```

to specify the number of lines in the page. The fixed binary ex- pression following the keyword gives the number of lines. For details on how PRINT, LINESIZE, and PAGESIZE work, see below.

The keyword combinations which fully define stream files look like this:

Usage and Access	Function	Format
STREAM and	INPUT	(none for Input files)
	or	
	OUTPUT may take	LINESIZE and/or PRINT may take PAGESIZE

Examples of OPEN statements for stream files:

```
OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");
OPEN FILE(PRINTER) STREAM OUTPUT PRINT LINESIZE(30)
PAGESIZE(20) TITLE("@LPT");
```

The last OPEN statement opens the AOS generic data set for the lineprinter. The output will be formatted for the printer, with lines 30 characters long and pages 20 lines long.

Keywords for Record Sequential Files

Record sequential files require two keywords to define their usage attribute and their access attribute. The RECORD keyword defines the file as a record file and the SEQUENTIAL keyword gives it the sequential access type. Record sequential files may have either the INPUT or the OUTPUT function attribute keyword.

The keyword combinations for record sequential files thus looks like this:

Usage	Access	Function
RECORD and	SEQUENTIAL and	INPUT or OUTPUT

Examples of OPEN statements for record sequential files:

```
OPEN FILE(SOURCE) RECORD SEQUENTIAL INPUT;
OPEN FILE(DESTINATION) RECORD SEQUENTIAL OUTPUT;
```

Keywords for Record Direct Files

As with record sequential files, you define the usage type with the RECORD keyword; the DIRECT keyword defines the access type. Record direct files may have the update function attribute as well as the input or output function attributes. The keyword for the update function attribute is UPDATE. The following are the keyword combinations for record direct files:

Usage	Access	Function
RECORD and	DIRECT and	INPUT or OUTPUT or UPDATE

Examples of OPEN statements for record direct files:

```
OPEN FILE(ACCOUNTS) RECORD DIRECT INPUT;
OPEN FILE(CLIENTS) RECORD DIRECT UPDATE;
OPEN FILE(BALANCE) RECORD DIRECT OUTPUT;
```

Default and Implied File Attributes

As we mentioned above, you need not give a complete set of attribute keywords in the OPEN statement. Some keywords imply attributes, and if the keywords neither give nor imply an attribute, the compiler supplies a default attribute.

Implied Attributes

Attribute Keyword	Implied Attribute Keywords
DIRECT	RECORD
SEQUENTIAL	RECORD
PRINT	STREAM OUTPUT
UPDATE	RECORD

Default Attributes

If the keywords in the OPEN statement neither specify nor imply a usage attribute, an access attribute, or a function attribute, the compiler provides these attributes as follows:

1.	No usage attribute:	STREAM
2.	No access attribute:	SEQUENTIAL
3.	No function attribute:	INPUT

Default Data Sets

If you do not use the TITLE keyword in the OPEN statement, the data set name will be the same as the PL/I file name.

Default Line Size with Stream Output Files

If you do not use the LINESIZE keyword in the OPEN statement for a stream output file, the default line size is 80 characters.

Default Page Size with Stream Output Print Files

If there is no PAGESIZE keyword in the OPEN statement for a stream output print file, the default page size is 60 lines.

Examples of Open Statements with Implied Attribute Keyword Lists

OPEN Statement	Complexed Keyword List
OPEN FILE(MATERIAL)	STREAM INPUT TITLE("MATERIAL")
OPEN FILE(OUT) OUTPUT;	STREAM OUTPUT LINESIZE(80) TITLE("OUT")
OPEN FILE(WRITER) PRINT;	STREAM OUTPUT PRINT LINESIZE(80) PAGESIZE(60) TITLE("WRITER")
OPEN FILE(INFO) RECORD;	RECORD SEQUENTIAL INPUT TITLE("INFO")
OPEN FILE(INFILE) SEQUENTIAL;	RECORD SEQUENTIAL INPUT TITLE("INFILE")
OPEN FILE(INFILE) DIRECT;	RECORD DIRECT INPUT

Opening Files with the GET, PUT, READ, and WRITE Statements

If you use a file constant in a GET, PUT, READ, or WRITE statement without explicitly opening the file with an OPEN statement, PL/I will open the file implicitly when it executes the first GET, PUT, READ, or WRITE statement for the file. The file's attributes are determined by the statement which opens it:

Statement	File Attributes
GET LIST, GET EDIT	STREAM INPUT
PUT LIST, PUT EDIT	STREAM OUTPUT
READ	RECORD SEQUENTIAL INPUT
WRITE	RECORD SEQUENTIAL OUTPUT

When you implicitly open a file, PL/I assumes that the data set name is the same as the file name.

Note that you cannot implicitly open files with READ and WRITE statements with keys or with REWRITE and DELETE statements. With READ and WRITE statements with keys, the system will open the file as a record sequential file, and will raise the error condition when it cannot find any keys.

The Default PL/I Files SYSIN and SYSPRINT

You may use GET and PUT statements in your program without file names. When you do this, PL/I supplies the default file name SYSIN with GET and the default file name SYSPRINT with PUT. If you have declared these file names and opened them with an OPEN statement, PL/I will give them the attributes you specify. Otherwise, it will implicity open SYSIN and SYSPRINT as follows:

SYSIN will be a stream input file associated with the AOS generic data set @DATA.

SYSPRINT will be a stream output print file associated with the AOS generic data set @LIST.

For example:

```
DEF_FI:
          PROCEDURE;

          PUT LIST("HELLO");

END; /* DEF_FI */
```

This minimal PL/I program will output

```
HELLO
```

to @LIST. HELLO has no quotation marks because SYSPRINT is a stream output print file.

File Openings and the File Control Block

Until a program opens a PL/I file, the file control block belonging to a file constant contains only the information that the file is closed. When the program opens the file, it changes the file's status to open and adds information that describes the data set the file is associated with and the file's usage, access, and function attributes. It also sets up the file so that the program can transmit data to and from it.

1. File Control Blocks for Stream Files

   File control blocks for stream files contain the current column position, that is, which character in the file is the next to be read or written. When a program opens a stream input or output file, it sets the column position to 1. Each time it inputs or outputs an ASCII character, it adds 1 to the column position. If the file is a stream output file, the file control block will also contain the file's line size and the current line number. If the file is a stream output print file the file control block also contains the file's page size and the current page number. When the file is opened, the current line number and the current page number are set to 1. Each time the program outputs a line mark, it increases the current line number by 1 and resets the current column position to 1. Each time the program outputs a page mark, it increases the current page number by 1 and resets the current line number and the current column position to 1.

2. File Control Blocks for Record Sequential and Record Direct Files

   File control blocks for record sequential and record direct files contain the current record. That is, the position of the next record to be read or written. When the program opens a record sequential input file,it sets the current record to the beginning of the file. As it reads data from the file, it sets the current record to the next record to be read. When it opens a record sequential output file, it sets the current record to null. As the program writes data to the file, PL/I sets the current record to the next record to be written.

   With record direct files, the value of the current record is the key which appears in the input or output statement.

The ENDFILE Condition

When an input statement attempts to read past the end of the data in a file, PL/I raises the endfile condition for that file. If there is no ON unit for the endfile condition for that file, PL/I writes a message to the terminal executing the program and signals the error condition. See Chapter 13 for details about ON units and an example of an ON unit for the endfile condition.

The ENDPAGE Condition

When an output statement attempts to write more lines to a stream output print file than the file's page size allows, PL/I raises the endpage condition. If there is no ON unit for the endpage condition for that file, PL/I simply outputs a pagemark, increases the page number by 1, sets the current line number and the current column position to 1, and continues with the output. For details about ON units for endpage, see Chapter 13.

I/O with Stream Files

Types of I/O with Stream Files

All stream files are sequences of ASCII characters. On input, PL/I reads through a stream file like this:

1. Each time it reads a character, it increases the column position by 1.

2. When it reads a linemark or pagemark, it resets the column position to 1.

PL/I writes a stream file like this:

1. Each time it writes a character, it increases the column position by 1.

2. When the current column position is one greater than the linesize, it outputs a linemark and increases the line number by 1.

3. When the current line number is one greater than the page size, it outputs a pagemark and resets the column postion and line number to 1.

AOS PL/I has three types of I/O with stream files: list-directed I/O, edit-directed I/O, and line-directed I/O. The differences between the three types are a consequence of how PL/I reads or writes the data in the stream file.

With list-directed I/O, the file is read or written as a list of data items. Each item on the list is followed by a space or comma terminator. You use GET LIST statements for list-directed input and PUT LIST statements for list-directed output.

With edit-directed I/O, the file is simply a sequence of ASCII characters. The manner in which the file is read or written is determined solely by format specifiers in the I/O statements. You use GET EDIT statements for edit-directed input and PUT EDIT statements for edit-directed output.

With line-directed I/O, the file is a series of lines. Each line is terminated by a linemark or a pagemark, and the I/O statements read or write the file a line at a time. You use special forms of the READ and WRITE statements for line directed I/O.

List-Directed I/O

When you use PUT LIST to output data to a stream output file, PL/I automatically gives the data the proper format for GET LIST; if you use other stream input files, you must make sure that PL/I can read them. The stream of data items must look like this:

1. A terminator must follow each data item, including the last item in the file. Terminators may be blanks or commas.

2. Character-string data items which contain commas or spaces must be enclosed in quotation marks. PL/I will read "JOHN SMITH" as one item, but JOHN SMITH as two.

You may follow terminators with spaces, tabs, linemarks or pagemarks. A linemark by itself is not a terminator.

The GET LIST Statement

The syntax of the GET LIST statement looks like this:

GET *[FILE(file-exp)] [SKIP[(fixed-bin-exp)]]* LIST(input-list);

The value of the file expression must be a file that the program has not yet opened or that has been opened as a stream input file. If the file has not been opened, PL/I will open it as a stream input file.

You use the SKIP keyword if you want the statement to skip over one or more lines of the input file before it begins reading data. The fixed binary expression gives the number of lines. The default is one line.

The input list is a list of variables separated by commas. The variables may be arrays and structures as well as elementary variables. Note that the input list may also include an implied do-increment statement. For details, see below.

The GET LIST statement starts reading data at the current column position for the file. If the statement has the SKIP keyword, it passes over as many linemarks as the expression with SKIP calls for before it begins reading data. When it begins reading, it reads until it comes to a terminator. It ignores tabs, linemarks, and pagemarks, but assigns all other characters from where it started reading to the terminator to the first variable on the input-list. If there are more variables in the output list, it reads data in this manner until it has assigned data to all the variables in the list or until it has reached the end of the file. In the latter case, the statement raises the endfile condition.

GET LIST treats data aggregates as collections of elementary data items. If a variable on the input list is an array, GET LIST assigns items to all the elements in the array. The items are assigned in row-major order. If a variable is a structure, GET LIST assigns items to all the members of the structure in the order in which they were declared.

For example, if you have a set of declarations like this:

```
DECLARE TAB_SUM FIXED BINARY;
DECLARE TABLE(2,2) FIXED BINARY;
DECLARE NUMBERS FILE;

OPEN FILE(NUMBERS) STREAM INPUT;
```

and a file which contains the following sequence of numbers:

```
. . . . 20,1,2,3,4 . . . .
```

The GET LIST statement

```
GET FILE(NUMBERS) LIST(TAB_SUM, TABLE);
```

will assign 20 to TAB_SUM, 1 to TABLE(1,1), 2 to TABLE(1,2), and so forth.

The PUT LIST Statement

The PUT LIST statement looks like this:

PUT *[FILE(file-exp)][SKIP[(fixed-bin-exp)]] [PAGE(fixed-bin-exp)]]* LIST(output-list);

The value of the file expression must be a file which the program has not yet opened or which it has opened as a stream output file. If the program has not yet opened the file, PL/I will open it as a stream output file.

If the FILE expression is not given, PL/I asumes that the file is the stream output print file SYSPRINT.

You use the SKIP keyword when you want the statement to output linemarks before it begins outputting data. The fixed binary expression gives the number of linemarks. The default is 1. You can use the PAGE keyword only if you opened the file as a stream output print file. The keyword gives the number of page marks the statement will output before it begins outputting data. The default number is 1. Whenever PUT LIST outpus a linemark, it sets the column position to 1; whenever it outputs a pagemark, it sets the column position and the line number to 1.

The output list is a list of variables or expressions. The variables may be elementary variables or data aggregates.

PUT LIST works by converting the value of each of the expressions or variables in the output list to character strings and then outputting them to the file. The conver- sions follow the rules for conversion to character-string data. If a variable is an array, PUT LIST outputs the values of the elements in row-major order; if it is a structure, it outputs the values of the members in the order in which the members were declared.

If a PUT LIST statement reaches the end of a line while it is outputting data, it outputs a linemark and continues the output on the next line. If it reaches the end of a page, it raises the endpage condition.

The format of PUT LIST's output depends on whether the output file has the print attribute. If it does not, PUT list outputs data like this:

1. If the item being output is a character-string value, it encloses the value in quotation marks.

2. If it is a bit-string value, it encloses the value in quotation marks and appends a "B".

3. It appends a space to the value.

If the output file does have the PRINT attribute, PUT LIST does the following:

1. It outputs character-string values without quotation marks.

2. After it outputs a value, it outputs one blank and then as many blanks as are required to reach the next tab postions. It then begins the output of the next value. The tabs are at column positions 1,9,17,25, and so on.

Note that because PUT LIST removes the quotation marks from around character-string values when it writes to a stream output print file, you may not be able to use the file as an input file.

For example:

```
PUT_LIST:
          PROCEDURE;

          DECLARE(OUT_FI,PRINT_FI)FILE;
          DECLARE NUM FIXED DECIMAL(5,2);
          DECLARE ARRAY(4)FIXED BINARY;
          DECLARE I FIXED BINARY

          DO I = 1 TO 4;
                      ARRAY(I) = I;
          END;

          NUM = 213.50;

          OPEN FILE(OUT_FI) STREAM OUTPUT;
          OPEN FILE(PRINT_FI) STREAM OUTPUT PRINT;

          PUT FILE(OUT_FI) LIST("2 * NUM =",2 * NUM,"ARRAY =",ARRAY);
          PUT FILE(PRINT_FI) LIST("2 * NUM =",2 * NUM,"ARRAY=",ARRAY);
END; /* PUT_LIST */
```

The output in OUT_FI, which did not have the print attribute, looks like this:

```
"2 * NUM =" 427.00 "ARRAY =" 1 2 3 4
```

The output in PRINT_FI, which did, looks like this:

```
2 * NUM =          427.00            ARRAY =            1 2
3                  4
```

The Implied DO Increment in Input Lists and Output Lists

PL/I allows you to incorporate DO increment loops into input and output lists. This feature is useful for inputting values to arrays or outputting them from arrays. In input lists, you use the control variable as a subscript or as part of a subsequent expression. For instance, a program fragment which gets data for nine elements of a 10-element array might look like this:

```
DECLARE OUTGO FILE;
DECLARE BILLS(10) FIXED DECIMAL(6,2);
DECLARE I FIXED BINARY;
.
OPEN FILE(OUTGO) STREAM INPUT;
GET FILE(OUTGO) LIST((BILLS(I) DO I = 1 TO 9));
.
```

In output lists, you can use the control variable as a subscript or as part of an expression. For example, you could output the numbers 1 to 10 like this:

```
PUT FILE(OUT) LIST((I DO I = 1 TO 10));
```

As with the DO increment statement, the control variable and the three expressions all must have fixed binary values. If you omit BY exp-3, the control variable will be increased by 1 on each execution. Note that when you use a DO increment with an input or output item, you must enclose the entire item, including the DO, in parentheses.

You can also nest implied DO increments in input or output lists. Each of the DOs has its own parentheses. The outermost DO goes to the end of the set of DOs and the innermost DO immediately follows the reference containing the control variable. For example, a GET LIST statement which assigned values to an array in column-major order might look like this:

```
DECLARE NUMBERS FILE;
DECLARE ARRAY(2,2) FIXED BINARY;
DECLARE (I,J) FIXED BINARY;
.
GET FILE(NUMBERS) LIST(((ARRAY(I,J) DO I = 1 TO 2)
DO J = 1 TO 2));
```

These nested implied DOs are the equivalent of the nested DO loops

```
DO J = 1 TO 2;
          DO I = 1 TO 2;
                   GET FILE(NUMBERS) LIST(ARRAY(I,J));
          END;
END;
```

Both will assign values to ARRAY in the order ARRAY(1,1),ARRAY(2,1), ARRAY(1,2),ARRAY(2,2).

Edit-directed I/O

Edit-directed I/O uses the GET EDIT and the PUT EDIT statements. These statements have input-lists and output-lists like GET LIST and PUT LIST, but the manner in which the file is written or read is determined by a list of format items. For example, if we wanted to start at the middle of a line and output five letters, each one preceded by 3 spaces, we could do it with a PUT EDIT statement like this:

```
PUT FILE(OUT) EDIT("A","B","C","D","E")(COLUMN41),
5(X(3),A(1)));
```

The format list follows the output list. The first item in the list, COLUMN(41) starts the output at the 41st character in the line. The second item, 5(X(3),A(1) repeats the list in parentheses, X(3),A(1) five times. X(3) outputs three blank spaces and A(1) outputs a character value one character long. Thus, 5(X(3),A(1)) outputs three spaces followed by a letter five times.

As you can see from the example, there are two kinds of format items on the format list. COLUMN(41) and X(3) determine where the output is to begin in the file, and A determines what kind of data will be output. COLUMN(41) and X(3) are control format items, while A(1) is a data format item. The integers following the data format items give the number of characters the item will read or output; the number of characters is called the item's field width.

In the following, we will discuss the general syntax of the GET EDIT and the PUT EDIT statements and then deal with the format specifiers and the FORMAT statement.

The GET EDIT Statement

The GET EDIT statement has the form

GET *[FILE(file-exp)][SKIP[(fixed-bin-exp)]]* EDIT(input-list)(format-list)

Except for the EDIT keyword and the format list, the syntax of GET EDIT is like the syntax of GET LIST. Note that you must give a field width with all data format specifiers when you use them with GET EDIT.

GET EDIT starts at the current column position for the file. If it has the SKIP keyword, it passes over as many linemarks as required before it begins evaluating the items in the format list. If the first item on the list is not a data format item, it evaluates control format items, moving forward into the file as they specify, until it reaches the first data format item. It then reads the number of characters specified in the data format item and assigns the value to the first variable in the input-list. It continues reading the file in this fashion until it reaches the end of the input list or the end of the file. When GET LIST encounters a linemark, it sets column position to 0, and continues reading characters from the next line. The linemark is not a terminator.

If GET LIST reaches the end of the file, it raises the endfile condition. If it reaches the end of the input list before it has reached the end of the format list, the remaining items in the format list will not be executed. If it reaches the end of the format list before it reaches the end of the input list, it returns to the beginning of the format list and starts over.

For example, if you have a file which contains five-character strings, each of which is preceded by five spaces, you can read three strings with a GET EDIT statement like this:

```
DECLARE (STR_1,STR_2,STR_3) CHARACTER(5);
GET FILE(STRINGS) EDIT(STR_1,STR_2,STR_3) (X(5),A(5));
```

The GET EDIT statement will first move forward 5 spaces, then read five characters as a character-string and assign it to STTR_1. Since it has reached the end of the format list, but not the end of the end of the input list, it will start over at the beginning of the format list, move forward five spaces, read five characters, assign them to STR_2, and so on.

Because PL/I stops executing the format list after it has assigned a value to the last item in the input list, you have to be careful about how you order the data format items and the control format items in the list. For instance, if the data you want in the file is followed by spaces, you have to add a dummy variable to your input list and a data format item to get data for it in order to make GET LIST move 5 spaces after it has assigned a value to STR_3.

```
DECLARE DUM_STR CHARACTERS(5);
      .
      .
GET FILE(STRINGS_2) EDIT(STR_1,STR_2,STR_3,DUM_STR)
(2(A(5),X(5)),A(5),A(5));
```

The PUT EDIT Statement

PUT EDIT's syntax looks like this:

PUT *[FILE(file-exp)] [SKIP(fixed-bin-exp)]] [PAGE[(fixed-bin-exp)]]* EDIT(output-list)(format-list);

Except for the EDIT keyword and the format list, PUT EDIT's syntax is like PUT LIST's syntax.

The format list with PUT EDIT works the same way it does with GET EDIT, except that the control format items write blanks or control characters instead of skipping characters and the data format items output values instead of reading them. If the format list begins with format items, the PUT LIST statement outputs blanks or control characters as required by the format items until it reaches the first data format item. It then evaluates the first expression in the output list, converts it to the data type required for the data format item, and outputs the number of characters specified by the item. It continues writing the file like this until it has reached the end of the output list. If it reaches the end of the format list before it reaches the end of the output list, it starts over at the beginning of the format list. It does not execute items remaining in the format list after it has finished the output list.

PUT EDIT treats line ends and page ends the same as PUT LIST. If it reaches the end of a line, it outputs a linemark and continues the output on the next line. If it reaches the end of a page, it raises the endpage condition.

Note that PUT EDIT does not put quotation marks around character-string values when it writes to stream output files. Data output with PUT EDIT has the same format regardless of whether the file has the print attribute; the only difference PRINT makes with PUT EDIT is that it allows you to specify the page size.

The following PUT EDIT statements write files which the example GET EDIT statements can read. The first PUT EDIT outputs a sequence of five blanks and a five-character string three times:

```
PUT FILE(STRINGS_1) EDIT(STR_1,STR_2,STR_3) (X(5),A);
```

The X(5) format specifier outputs the five blanks and the A outputs the five-character string. Since PL/I can determine the length of the string from the length of the variable, you need not give a field width with A.

The second PUT EDIT outputs a five-character string followed by five blanks three times. Here, as with GET EDIT, you need a dummy output value and an extra data format item to get PL/I to output the last set of blanks. The dummy output value is a string of five blanks:

```
DECLARE (STR_1,STR_2,STR_3) CHARACTER(5);
            .
            .
            PUT FILE(STRINGS_2) EDIT(STR_1,STR_2,STR_3,"     ")
(2(A,X(5)),A,A);
```

The FORMAT Statement

In PL/I, you can write a format list once and then use it in many GET and PUT EDIT statements. You do this with the FORMAT statement. It has the form:

```
identifier:FORMAT(format-list);
```

The identifier is a format constant, and when you write it ahead of a format statement, you are declaring it in the block which contains the statement. The format list is exactly the same as the format lists in GET and PUT EDIT statements. You use the format statement's format list in a GET or PUT EDIT statement by writing the remote format specifier R followed by the format constant in parentheses. For instance, if we have the format statement:

```
SPACES:FORMAT(X(5),A(5));
```

we can rewrite the statement

```
GET FILE(STRINGS_1) EDIT(STR_1,STR_2,STR_3) (S(5),A(5));
```

like this:

```
GET FILE(STRINGS_1) EDIT(STR_1,STR_2,STR_3) (R(SPACES));
```

When the computer executes this GET EDIT statement, it will use the format list in the format statement SPACES where R(SPACES) appears as a format item.

Note that PL/I executes format statements only when the statement's format constant appears in a format list. Otherwise, control simply passes over them. Consequently, you can write FORMAT statements, like DECLARE statements, anywhere in your program. As with DECLARE statements, it will be easier for others to understand your program if you put all your format statements in one place.

The Format List

The syntax of the format list looks like this:

([integer] format-item , . . . , *[integer]format-item])*

Each format item may be one of the data or control format specifiers, or it may be another format list. The optional integer preceding the format item gives the number of times the item is to be repeated. The integer's values range from 0 to 255. If the integer is 0, the item will be ignored. If there is no integer, the item will be executed once each time the format list is executed.

Note that AOS PL/I does not allow the use of variables or expressions other than integers in the format list.

The Format Items

AOS PL/I has data format items for five types of data:

Format Item	Data Type
F(integer[,integer]	For fixed-point arithmetic values
E(integer[,integer]	For floating-point or arithmetic values
A[(integer)]	For character-string values
B[(integer)], B1[(integer)], B2[(integer)], B3[(integer)], B4[(integer)]	For bit-string values
P''picture specifier''	For picture values

It has six control format items. You can use the first three of these with any stream file; you can use the last three only with a stream output print file.

Format Item	Description
COLUMN(integer)	Gives the column where input or output is to continue.
X(integer)	Indicates the number of characters to be skipped or blanks to be output.
SKIP(integer)	Gives how many lines are to be skipped or linemarks are to be output.

These require stream output print files:

Format Item	**Description**
LINE(integer)	Gives the line on the page where output is to continue.
TAB(integer)	Gives the number of tabs to be skipped before the next output begins.
PAGE	Outputs a pagemark.

The F Format - The F format specifier converts ASCII characters to fixed decimal values on input and arithmetic values to ASCII character strings representing fixed decimal values on output. F will work on input only if the character string it reads contains only blanks and a sequence of characters which PL/I can interpret as an arithmetic value. For example, F will read 23, -555.42, or 3.459E-01, but it will not read 1,000 or $5.95. If the string contains only blanks, F will convert it to the value 0 (blanks are treated as zeros). Similarly, F will work on output only if PL/I can convert the value it is outputting to an arithmetic value. You can output the character-string "123.5" with F, but not "ABC". The F format has the form

F(integer *[,integer]*)

The first integer is required. It gives the format specifier's field width. The second integer is optional. It gives the number of characters to the right of the decimal point. Note that field width is not the same as precision: the character string representation of a fixed decimal value may contain a decimal point, a sign, and a leading 0 as well as the digits. Consequently, if you want to be sure that a F format specifier will input or output all the values a variable can hold, you should make the field width 3 greater than the variable's precision. If the value to be input or output is greater than the field width will allow, the program will not read or write the correct value.

When you use F to read a file, the format specifier reads as many characters as specified in its field width. If it can interpret the characters as an arithmetic value, it converts the value to a fixed decimal value and then assigns it to the variable. The scale of the converted value depends on the character string and on the F format specifier. If the character string contains a decimal point, the decimal point determines the value's scale; if it does not, the second integer in the F format determines the scale. Leading blanks are ignored.

Take the following as an example:

Character String	Format	Fixed Decimal Value
□1.00	F(5)	1.00
□□100	F(5)	100
□00	F(3,2)	1.00
□21.5	F(5,2)	21.5
□□□□□	F(5)	0

When you use the F format specifier to output fixed decimal values, the second integer specifies the scale of the output value. If there is no second integer, the scale is 0.

The F format specifier rounds the values it outputs unless they have a precision of 16. It rounds the values by converting them to fixed decimal values with a precision and scale 1 larger than that specified in the F format specifier and then adding 5 (-5 with negative values) to the last digit. Since 16 is the maximum precision for fixed decimal values, the F format cannot round them, but simply converts them directly to character-strings.

The character string produced by the F format specifier has as many digits to the right of the decimal point as required by the specifier, the decimal point, the integer digits, and a sign if the value is negative. The format specifier suppresses all leading 0's but the one immediately preceding the decimal point. It pads the resulting string on the left with enough blanks to fill its field width. If the string is larger than the field width, your program will not output the correct value.

Examples:

Fixed Decimal Value	**F Format**	**Character String**
16.276	F(8,2)	□□□16.28
.006	F(8,2)	□□□□0.01
-.5	F(4)	□□-1

The E Format - The E format specifier converts ASCII characters to float binary values on input and arithmetic values to ASCII characters which express the values in scientific notation on output. Like F, E only works on input if it can interpret the sequence of characters it reads as an arithmetic value. Similarly, it will only output values it can convert to arithmetic values.

The E format's syntax looks like this:

E(integer *[,integer]*)

The first integer, which is required, gives the specifier's field width. The optional second integer has no effect on input; on output, it gives the number of digits the float decimal number will have to the right of the decimal point.

When you read a file with E, the format specifier reads the number of characters specified by its field width. If it can interpret the characters as an arithmetic value, it converts the value to a float binary value with a precision of 53 and assigns it to the variable.

When you output a value with E, the format specifier converts the value to a float binary value and then converts the float binary value to a character string representing a float decimal number. The second integer in the format specifier determines how many digits the float decimal number will have to the right of the decimal point.

When you use E on output, you must be careful to make the field width wide enough to allow for all the characters in the float decimal number. The exponent requires three characters, and the E, the decimal point, the digit to the left of the decimal point, and the sign each require one. Hence, you must make the field width at least 7 wider than the number of digits to the right of the decimal point. For example, 4 digits to the right will require the E specifier E(11,4). The minimum E format that will work on output is E(6,0). This allows for the exponent, the E, the sign, and the digit.

If you do not specify the number of digits, PL/I gives the string enough digits to the right of the decimal point so that it expresses the full float decimal precision of the value. For details on how to calculate a value's float decimal precision, see the reference manual. Here, we will only note that a field width of 13 will express any single-precision float binary value and that a field width of 22 will express any double-precision float binary value.

The string produced by the E format looks like this: on the right, there is the exponent, which consists of two digits preceded by a sign and an E. Then come the digits specified by the second integer in the format specifier. These are preceded by a decimal point, a single digit, and a sign if the value is negative. If the string is shorter than the E format specifier's field, it is padded on the right with blanks; if it is longer, your program will raise the error condition.

Examples:

Arithmetic Value	Float Decimal Precision	E Format	Character String
25	6	E(6,0)	□2E+01
25	6	E(10,3)	□2.500E+01
-25	6	E(14)	□□-2.50000E+01

The A Format - The A format specifier reads and writes data as character strings. On input, A simply reads the number of characters specified by its field width and assigns them to the variable. On output, it converts the value it is outputting to a character string as specified by the rules for conversion to character-string data.

A *[(integer)]*

When you use A to read a file, you must give a field width greater than 0. When you use it to write to a file, you may omit the field width. Without the field width, A outputs a character string of the length required for the value. If you give a field width and the character string is longer than the field width, A truncates from the right. If the string is shorter than the field width, A pads from the right.

Examples:

Character-string Value	A Format	Character String
"ABC"	A(1)	A
"ABC"	A(3)	ABC
"ABC"	A(5)	ABC□□

The B Format - The B formats convert ASCII characters to bit-string data on input and bit-string values to ASCII character strings on output. There are five B format specifiers:

B[(integer)] or *B1[(integer)]*	Converts to or from the B1 format
B2[(integer)]	Converts to or from the B2 format
B3[(integer)]	Converts to or from the B3 format
B4[(integer)]	Converts to or from the B4 format

We will use only the B[(integer)] format. For a description of the other forms, see the *PL/I Reference Manual,* (093-000204).

Like the E and F format specifiers, these specifiers will work on input only if they can interpret the sequence of characters they read as a bit-string value.

When you use the B formats to read a file, you must give a field width. The B format reads as many characters as specified by its field width and interprets them as bit-string data. If the format specifier finds any characters other than 1 or 0, it will raise the error condition. PL/I interprets the characters it reads with the B formats exactly the same way it interprets bit-string literal constants; for details, see the reference manual.

Examples for input:

Input String	**B Format**	**Bit-string Value**
101	B(3)	101

When a B format outputs a value, it first converts the value to a bit-string according to the rules for conversion to bit-string data. It then expresses the bit string as a character string of 1's and 0's. If the format specifier has no field width, PL/I will simply output the resulting character string. If it does have a field width and the character string is shorter than the field width, it will pad the string on the left with blanks If the character string is longer than the field width, it will raise the error condition.

Examples for output:

Bit-string Value	**B Format**	**Character String**
"00"B	B	00
"1"B	B(3)	□□1

The P Format - The P format specifier allows you to convert strings containing characters other than digits and the decimal point to fixed decimal values on input and to convert fixed decimal values to character strings containing characters such as $., CR and DB on output.

The P format specifier has the form

P"picture-specifier"

For details on the picture specifier, see Chapter 9.

Each character in the picture specifier other than the "V" character represents a character in the pictured value; hence, the field width of a P format specifier is the number of characters other than V. For instance, the P format specifier P"999V.99" has a field width of 6.

When a P specifier reads a file, it reads the number of characters required by its field width and converts the digits in the string to a fixed decimal value with the precision and scale called for by the picture specifier. The specifier does the conversion by removing all the non-digit characters from the string and giving the digits the scale called for by the picture specifier. Unless the string contains -, CR, or DB, the fixed decimal value will be positive If the string contains no digits, the fixed decimal value will be 0.

Examples:

Character String	P Format	Decimal Value
$1,000,000	P"99999999V99"	100000.00
1.00	P"9999"	100
$1,000.85	P"$99999V.99"	1000.85
AAA3AB%	p"9999V.99"	.35
3,500.00DB	p"99999V.99DB"	-3400.00

Note that here, as with picture specifiers in declarations, the V character alone determines the position of the decimal point in the fixed decimal value. The specifier ignores decimal points in the character string.

When a program outputs a value with a picture specifier, it first converts the value to a fixed decimal value. It then converts the fixed decimal value to the charcter string specified by the picture specifier. If the picture specifier allows fewer digits to the left of the decimal point than are required for the value, it will truncate from the left; if the scale is less, the specifier will truncate from the right. After the fixed decimal value has the proper precision and scale, the specifier inserts the other characters required and outputs the value.

Examples for output:

Fixed Decimal Value	P Format	Character String
25.00	p"$,$$9V.99"	↑↑$25.00
-25.00	P"S$,$$9V.99"	-↑↑$25.00

Note that the picture specifier will not put a - sign in front of negative values unless it contains a - or S character.

The Control Format Items

The COLUMN Format Item - The COLUMN format item lets you specify the column where input or output is to begin. It looks like this:

```
COLUMN(integer)
```

The integer gives the number of the column. It must be greater than 0.

COLUMN works like this with an input file:

1. If the program has not yet reached the column position indicated by the integer, it goes to that position.
2. If it has passed the column position, it goes to the next linemark and then goes to the column position in the new line.
3. If the integer is greater than the number of columns in the line, the program goes to the beginning of the next line.

For example, if you have a file like this:

```
12345678
12345
1234
123
```

and a GET EDIT statement like this:

```
GET FILE(COL_FI)EDIT(N,O,P)
(COLUMN(5),F(1));
```

the statement will read the 5 in the first row of numbers, the 5 in the second row of numbers, and the 1 in the fourth row of numbers.

When you use COLUMN in a PUT EDIT statement, it works like this:

1. If the program is at a column position less than that given with COLUMN, the PUT EDIT statement outputs blanks until it reaches that column.
2. If the current column position is greater than that given with COLUMN, the statement outputs a linemark. It then outputs blanks until it reaches the specified column position in the next line.
3. If the specified column position is larger than the file's line size allows, the statement outputs a linemark and goes to the beginning of the next line.

For example, if you open a file like this:

```
OPEN FILE(COL_FI) STREAM OUTPUT LINESIZE(10);
```

and have a PUT EDIT statement like this:

```
PUT FILE(COL_FI) EDIT("A", "B", "C")
(2(COLUMN(7),A(1)),COLUMN(12),A(1));
```

the output file will look like this:

```
                A
                B
C
```

The X Format Specifier - The X data format specifier moves you forward into the file by the number of characters specified in the integer:

X(integer)

The integer must be greater than 0.

When you use X with GET EDIT, it skips over the number of characters specified. If it encounters a linemark, it ignores the linemark and continues skipping characters on the next line.

With PUT EDIT, X outputs the number of blanks specified by the integer. If it reaches the end of the line, it outputs a linemark and continues outputting blanks.

The SKIP Format Specifier - The SKIP format specifier works like the SKIP keyword in GET and PUT statements. It has the form:

SKIP *[(integer)]*

Note that SKIP cannot have an expression when you use it as a format item. The integer must be greater than or equal to 0. The default is 1.

With GET EDIT, the integer must be greater than 0. SKIP skips over as many linemarks as indicated by the integer. With PUT EDIT, it outputs as many linemarks as indicated by the integer. If the integer is 0, and the file is a PRINT file, it outputs a carriage return character. This enables you to print more than one line of output on the same line. In all cases, it resets the column position to 1.

The LINE Control Format

You can only use LINE when you are outputting data to a stream output file opened with the PRINT attribute. It specifies the line on a page where the next output is to begin. LINE looks like this:

LINE(integer)

The integer integer gives the line number. It must be greater than 0.

What the LINE control format does depends on the relationship between the line number in the format item and the file's current line number:

1. If the file's current line number is the same as the line number in LINE, the format item does nothing.

2. If the file's current line number is less than the line number in LINE, the format item outputs linemarks until the file's current line is the same as the number in LINE.

3. If the file's current line number is greater than the number in LINE, the format item raises the endpage condition. It also raises the endpage condition if the number in LINE is greater than the number of lines on a page of the file.

The PL/I default action for the endpage condition is to output a pagemark, reset the line number and column positions to 1, and continue with the output. You can, of course, specify other actions. For details, see Chapter 13.

If you open a file like this:

```
OPEN FILE(PRINT_FI) STREAM OUTPUT PRINT TITLE("@LPT")
PAGESIZE(10);
```

and use the file in a PUT EDIT statement like this:

```
PUT FILE(PRINT_FI) EDIT("WORD_1", "WORD_2", "WORD_3")
(2(LINE(8),A),LINE(12),A);
```

the program will output "WORD_1" 8 lines down the first page of the output. The second execution of LINE(8) and the execution of LINE(12) both raise the ENDPAGE condition, so "WORD_2" appears on line 1 of the second page and "WORD_3" on line 1 of the third page.

The TAB Format Item - TAB requires an output print file. You write it like this:

```
TAB(integer)
```

TAB moves the output over the number of tab stops specified by the integer. In AOS PL/I, the tab stops are at column positions 1, 9, 17, 25, and so forth. TAB works by outputting blanks until the current column position is as many tab stops beyond the last column position as required by the specifier. If TAB reaches the end of a line, it outputs a linemark and continues to output blanks until it has moved over the proper number of tab stops.

If you open a file like this:

```
OPEN FILE(TAB_FI) STREAM OUTPUT PRINT LINESIZE(80);
```

and have a PUT EDIT statement like this:

```
PUT FILE(TAB_FI) EDIT("A", "B", "C")
(2(TAB(3),A),TAB(6)A);
```

the output in TAB_FI will look like this:

```
                A           B
C
```

The PAGE Format Item - You may only use PAGE with a stream output print file. When PL/I executes a PAGE format item, it outputs a pagemark, increases the page number by 1, and sets the current column position and the current line number to 1.

The R Format Item - You use R to transfer control to a format list in a format statement. R looks like this:

```
R(format-constant)
```

When PL/I executes an R format item, it transfers control to the FORMAT statement whose label prefix appears in the format item. An R format item in one FORMAT statement may transfer control to another FORMAT statement. However, once the remote format is executed, control returns to the statement that contained the R format item.

GET and PUT as Format Control Statements

You can use GET and PUT to skip lines and pages. If you write

```
GET FILE(MATERIAL) SKIP(2);
```

PL/I will move ahead two linemarks in the file MATERIAL. If you write:

```
PUT FILE(PRINTER) PAGE;
```

PL/I will output a pagemark to the file PRINTER.

When you use SKIP and PAGE this way, they work exactly like they do in a full GET or PUT statement. You can use fixed binary expressions with both. SKIP will work with any stream file, but PAGE requires a stream output print file.

Line-Directed I/O

AOS PL/I allows you to use the READ and WRITE statements to read and write stream files a line at a time. The READ statement reads the characters between one linemark and the next and assigns them to a character varying variable; the WRITE statement writes the value of a character varying variable and appends a linemark to it.

Note that you cannot use READ and WRITE with stream files unless the statements contain character varying variables.

READ with Stream Input Files

When you use READ with stream input files, it has the form:

```
READ FILE(file-exp) INTO(character-varying-var);
```

READ will not work unless the current column position is 1. If it is, READ reads to the next linemark (or pagemark), assigns all the characters from column 1 up to the linemark to the character varying variable, and sets the current length of the variable to the number of characters assigned to it. It then sets the current column position to column 1, that is, to the character following the linemark.

Take a set of declarations and an OPEN statement like this:

```
DECLARE LINES FILE;
DECLARE LINEREAD CHARACTER(80) VARYING;

OPEN FILE(LINES) STREAM INPUT;
```

LINES contains a sequence of characters like this:

```
...<LINEMARK>THERE IS A TIDE IN THE AFFAIRS OF MEN,<LINEMARK>...
```

when the last READ statement has read up to the first linemark, this statement:

```
READ FILE(LINES) INTO (LINEREAD);
```

will assign this string to LINEREAD, and set LINEREAD's current length to 39:

```
THERE IS A TIDE IN THE AFFAIRS OF MEN,<LINEMARK>
```

WRITE with Stream Output Files

WRITE with stream output files must have the form:

```
WRITE FILE(file-exp) FROM(character-varying-var);
```

The output from WRITE can begin at any column position. WRITE writes the characters in the variable to the file, writes a linemark after the last character, and sets the current column position to 1. Note that the statement does not insert a linemark if the number of characters is larger than the files linesize.

The following program shows how WRITE works with stream files:

```
WRITE_TEST:
            PROCEDURE;

            DECLARE PROVERBS FILE;
            DECLARE LINE CHARACTER(80) VARYING;

            OPEN FILE(PROVERBS) STREAM OUTPUT;

            LINE = "HASTE MAKES WASTE";
            WRITE FILE(PROVERBS) FROM(LINE);

            LINE = "HE WHO HESITATES IS LOST";
            WRITE FILE(PROVERBS) FROM (LINE);

END; /* WRITE_TEST */
```

The output in the file PROVERBS looks like this:

```
HASTE MAKES WASTE
HE WHO HESITATES IS LOST
```

Interactive Terminal I/O under AOS

AOS has two special generic files, @INPUT and @OUTPUT, for interactive terminal I/O. These generic files have several special features which simplify interactive I/O in PL/I:

1. Input and output are synchronized on @INPUT and @OUTPUT so that the value of the column position for both data sets corresponds to the position of the cursor on the console screen.

2. When you use GET EDIT with @INPUT, GET EDIT removes the linemark ending the input and pads the input with blanks until it is 80 characters long. This has two consequences:

    a. If the input is shorter than the GET EDIT data format item which reads it, the blanks will keep the statement from raising the endfile condition.

    b. If you use several separate GET EDIT statements to read data from @INPUT, each statement after the first must have the SKIP keyword or the SKIP format item. Otherwise, the statement will simply read the blanks which were used to pad the input for the first GET EDIT statement.

3. When a GET LIST statement reads a line from @INPUT, it replaces the linemark with a blank. Consequently, the user need not insert a terminator after the last input item.

4. When a PUT LIST statement writes data to @OUTPUT the data appears on the screen as soon as the PUT LIST statement has been executed. With other data sets, the data is written only when the line is full.

5. You can use CTRL D to signal ENDFILE from @INPUT.

6. If @OUTPUT is opened as a PRINT file, the pagesize is infinite. ENDPAGE will never be signalled.

Take the following program as an example. It accepts values written as dollars and cents as input from the terminal, adds them, and outputs the sum to the terminal:

```
FINANCE:
        PROCEDURE;

        DECLARE(IN,OUT) FILE;
        DECLARE(VAL_1,VAL_2) FIXED DECIMAL(8,2);
        DECLARE SUM FIXED DECIMAL(9,2);

        OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
        OPEN FILE(OUT) STREAM OUTPUT PRINT TITLE("@OUTPUT");

        PUT FILE(OUT) EDIT("THIS PROGRAM ADDS AMOUNTS OF UP TO"!!
                           "$9,999.99.", "INPUT THE FIRST AMOUNT")
                           (A,SKIP,A);
        GET FILE(IN) EDIT(VAL_1)(P"$9,999V.99");

        PUT FILE(OUT) SKIP EDIT("INPUT THE SECOND AMOUNT")(A);
        GET FILE(IN) SKIP EDIT(VAL_2)(P"$9,999V.99");

        SUM = VAL_1 + VAL_2;

        PUT FILE(OUT) SKIP EDIT("HERE IS THE SUM", SUM)
                                                (A,X(5),P"$$$,$$9V.99");

END; /* FINANCE */
```

Input and Output with Record Files

The data in record files is a copy of the data as it is stored in memory. When a WRITE statement outputs the value of a variable to a record file, it simply copies the sequence of bits in the variable's storage onto the file. When a READ statement reads data from a record file into a variable, it copies as many bytes from the file into the variable as are required to fill the variable's storage.

Because record I/O copies data from memory to a file with no conversion, it is faster than stream I/O. Data stored in record files also generally takes up less room than data stored in stream files. On the other hand, record I/O has none of the safeguards of stream I/O: for example, if you use a READ statement with a fixed binary variable to read a part of a file where character data was stored, PL/I will assign 1 word of the character data to the fixed binary variable. In general, record I/O will only work properly if the variables you use to read data have exactly the same attributes as the variables you used to write the data and if you read the data in exactly the same order in which you wrote it.

For example, if a program fragment like this writes a record file:

```
DECLARE N FIXED BINARY;
DECLARE NAME_LIST(20) CHARACTER(20) VARYING;
DECLARE NAMES FILE;

OPEN FILE(NAMES) RECORD SEQUENTIAL OUTPUT;

.
WRITE FILE(NAMES) FROM(N);
WRITE FILE(NAMES) FROM(NAME_LIST);

.
```

To read the file, you will probably want a fragment like this:

```
DECLARE NO FIXED BINARY;
DECLARE NAME_CATALOG(1000) CHARACTER(20) VARYING;
DECLARE NAMES FILE;

OPEN FILE(NAMES) RECORD SEQUENTIAL INPUT;
.
READ FILE(NAMES) INTO(NO);
READ FILE(NAMES) INTO(NAME_CATALOG);


.
```

Illegal Variables with Record I/O

The manner in which AOS PL/I copies data to and from record files places the following limitations on the variables you can use in record I/O statements:

1. If the variable is a data aggregate, its storage must be connected.
2. The variable cannot be bit-aligned. Hence, you cannot transmit data to and from bit elementary variables or data aggregates consisting solely of bit data. (Unless they are word aligned. See Chapter 10.)
3. The storage for the variable must consist of an integer number of bytes. This means that if a structure contains bit data items, the bits in the items must add up to an integral number of bytes.

The following declarations and record I/O statements show illegal data types and variables:

```
DECLARE 1 BORROWER(100);
          2 NAME CHARACTER(20) VARYING,
          2 LOAN,
                 3 PRINCIPAL FIXED DECIMAL(8,2),
                 3 RATE FIXED DECIMAL(4,4),
                 3 TIME FIXED BINARY,
                 3 INTEREST FIXED DECIMAL(8,2);

          .
          READ FILE(LOANS) INTO(BORROWER.LOAN);
```

The READ statement is illegal because BORROWER.LOAN is an unconnected array.

```
DECLARE TABLE(3,3) BIT;
.
WRITE FILE(GAME) FROM (TABLE);
```

The WRITE statement won't work because TABLE is bit-aligned.

Record Sequential I/O

When you input data from a record sequential file, you use a READ statement without a key; when you output data to a record sequential file, you use a WRITE statement without a key. If you use READ or WRITE with keys on a record sequential file, your program will raise the error condition when it executes the READ or WRITE statement.

The READ Statement with Record Sequential Files

When you use READ on record sequential files, it looks like this:

```
READ FILE(file-exp) INTO(variable);
```

If the file in the file expression has not been opened, the READ statement will open it as a RECORD SEQUENTIAL INPUT file. If it has been opened, it must have the record sequential input attributes. The variable must have the kind of storage required for record I/O.

The READ statement begins at the record in the file following the one read by the last READ. It reads as many bytes of data from the file as are required to fill the READ statement's variable. If there are not enough bytes in the record to fill the variable, the remainder of the variable is filled with 0 bits. If there are not enough bytes left in the file, the statement raises the endfile condition.

For instance:

```
DECLARE FLOAT_VAL FLOAT BINARY(53);
DECLARE NUMBERS FILE;
.
OPEN FILE(NUMBERS) RECORD SEQUENTIAL INPUT;
.
READ FILE(NUMBERS) INTO (FLOAT_VAL);
.
```

The READ statement will read four words (8 bytes) of the file NUMBERS into the double-precision float binary variable FLOAT_VAL.

The WRITE Statement with Record Sequential Files

WRITE with record sequential files looks like this:

```
WRITE FILE(file-exp) FROM(variable);
```

The file constant contained in the file expression must belong to an unopened file or to a file which the program opened as a record sequential output file. The variable must not violate any of the rules for variables with record I/O. When it outputs data, the WRITE statement starts at the beginning of the next record and outputs as many bytes as are contained in the variable's storage.

The WRITE statement in the following fragment outputs 1 word (2 bytes);

```
DECLARE INTEGER FIXED BINARY;
DECLARE INT_FILE FILE;

OPEN FILE(INT_FILE) RECORD SEQUENTIAL OUTPUT;
.
WRITE FILE(INT_FILE) FROM(INTEGER);
.
```

Record Direct I/O

Record direct files are made up of 256-word fixed length records. Each record must be referred to with an integer called its key. The key of the first record in the file is 0 and the following records are numbered consecutively. When you write an I/O statement for a record direct file, you must give the key of the record you are transmitting data to or from. The statement will then go to the proper record and transmit data between the record and the variable the statement specifies.

You may use record direct files, not only as input or output files, but as update files as well. When you open a record direct file as an update file, you can change individual records without having to delete and recreate the whole file.

The READ statement with a key lets you read data from a record direct input or update file; you can write data to record direct output files with the WRITE statement with a key, and you can write data to record direct update files with the WRITE or REWRITE statements. The DELETE statement deletes individual records in a ecord direct update file. Like the READ and WRITE statements, the REWRITE and DELETE statements require keys.

The READ Statement with Record Direct Files

The syntax for READ with a key is as follows:

```
READ FILE(file-exp) INTO(variable) KEY(fixed-bin-exp);
```

The file in the statement must have been opened with an OPEN statement and must have the record direct input or update attribute and the variable must not violate the rules for variables with record I/O. The fixed binary expression following the keyword KEY gives the key for the record from which the READ statement will read its data. The expression must yield 0 or a positive value. If the value of the expression is larger than the largest key in the file, the READ statement will raise the endfile condition.

If the record contains no data, the statement will fill the variable's storage with 0 bits.

Take the following:

```
DECLARE ACCT_NO FIXED BINARY;

DECLARE 1 BORROWER,
            2 NAME CHARACTER(20) VARYING,
            2 LOAN,
                        3 PRINCIPAL FIXED DECIMAL(8,2),
                        3 RATE FIXED DECIMAL(4,4),
                        3 TIME FIXED BINARY,
                        3 INTEREST FIXED DECIMAL(8,2);

DECLARE LOANS FILE;

.
OPEN FILE(LOANS) RECORD DIRECT UPDATE;
.
READ FILE(LOANS) INTO(BORROWER) KEY(ACCT_NO);
.
```

The statement will go to the record whose key is the value of ACCT_NO and read the first 35 bytes of the record into BORROWER.

The WRITE Statement with Record Direct Files

When you use a key with WRITE, it looks like this:

```
WRITE FILE(file-exp) FROM(variable) KEYFROM(fixed-binary-exp);
```

The program must have opened the file in the statement with an OPEN statement that gave it the update or output attributes. The variable must be a legal variable for record I/O. The fixed binary expression following KEYFROM gives the record to which the data is to be output. If the record for the key already exists, the statement rewrites the record using the data in the variable. If it does not exist, the statement writes the record for the key with the data from the variable. It creates empty records for all the keys between the key in the statement and the key for the last record in the file.

The following program fragment shows how this works:

```
DECLARE(LAST_KEY, KEY_VAL)FIXED BINARY;

DECLARE 1 BORROWER,
            2 NAME CHARACTER(20) VARYING,
            2 LOAN,
                        3 PRINCIPAL FIXED DECIMAL(8,2),
                        3 RATE FIXED DECIMAL(4,4),
                        3 TIME FIXED BINARY,
                        3 INTEREST FILED DECIMAL(8,2);
DECLARE LOANS FILE;

.
OPEN FILE(LOANS) RECORD DIRECT OUTPUT;
.
WRITE FILE(LOANS) FROM (BORROWER) KEYFROM(KEY_VAL);
IF KEY_VAL > LAST_KEY
            THEN
            DO;
            LAST_KEY = KEY_VAL;
            WRITE FILE(LOANS) FROM(LAST_KEY) KEYFROM(0);
            END;
```

If the last record written had the key 3 and KEYVAL now has the value 5, this fragment will create records 4 and 5, fill them with zeros, output the data in BORROWER to record 5, and rewrite record 0 with the value of KEYVAL.

The REWRITE Statement

You can use REWRITE only with record direct update files. With these files, it works exactly like WRITE. REWRITE's syntax looks like this:

```
REWRITE FILE(file-exp) FROM(variable) KEY(fixed-bin-exp);
```

The preceding example rewritten with REWRITE would look like this:

```
DECLARE(LAST_KEY, KEY_VAL)FIXED BINARY;

DECLARE 1 BORROWER,
            2 NAME CHARACTER(20) VARYING,
            2 LOAN,
                        3 PRINCIPAL FIXED DECIMAL(8,2),
                        3 RATE FIXED DECIMAL(4,4),
                        3 TIME FIXED BINARY,
                        3 INTEREST FILED DECIMAL(8,2);
DECLARE LOANS FILE;
```

```
.
OPEN FILE(LOANS) RECORD DIRECT UPDATE;
.
REWRITE FILE(LOANS) FROM (BORROWER) KEY(KEYVAL);
IF KEY_VAL > LAST_KEY
            THEN
            DO;
            LAST_KEY = KEY_VAL;
            REWRITE FILE(LOANS) FROM(LAST_KEY) KEY(0);
            END;
```

The DELETE Statement

You may only use the DELETE statement with record direct update files. It has the form:

```
DELETE FILE(file-exp) KEY(fixed-binary-exp);
```

DELETE deletes the record whose key is given in the fixed binary expression by writing it full of 0 bits.

For example:

```
DECLARE LOANS FILE;
.
OPEN FILE (LOANS) RECORD DIRECT UPDATE;
.
DELETE FILE(LOANS) KEY(5);
```

will delete the fifth record in LOANS.

The Program: STATEMENT

STATEMENT shows you some possible uses for the different PL/I file types and I/O statements. The program outputs financial statements which give the customer's name and address, his current balance, and the transactions since the last statement.

STATEMENT works with three files: ACCOUNT_LIST, CUSTOMER_LIST, and PRINT_FI.

ACCOUNT_LIST contains the data for the statements. It is a record direct file because the business using the file must be able to refer to the accounts and change them in any order.

The record sequential file CUSTOMER_LIST contains the names of the customers along with the keys for their records in ACCOUNT_LIST. Since the names and keys do not require much storage, you can read all of the data in CUSTOMER_LIST into an array of structures like this:

```
DECLARE 1 CUSTOMER(100),
            2 CUST_NAME CHARACTER(40) VARYING,
            2 C_NO FIXED BINARY;
```

If you want to use a single customer's record, you only have to find the element of the array which contains his name. That element will also contain his record number, and you can use the record number to read or write to his record.

Of course, to make finding names in the array easier, you may want to keep the array and the file CUSTOMER_ LIST in alphabetic order. If CUSTOMER_LIST is in alphabetic order, STATEMENT will output the statements in alphabetic order.

PRINT_FI is a stream output print file which will take the statements made from the data in ACCOUNT_LIST.

The program works like this: first, it reads the first item in CUSTOMER_LIST. This item gives the number of customers in the file. Then the program uses the value of the first item in a DO loop which reads the rest of the file into the array CUSTOMER. Once it has the customers' names and the numbers of their records in CUSTOMER, the program uses another DO loop to get the keys one at a time from the array, get the record belonging to that key, input the record to the structure ACCOUNT, and output the data to PRINT_FI with PUT EDIT statements.

The program looks like this:

```
STATEMENT:
   PROCEDURE;

   /* FILE DECLARATIONS */


   DECLARE (CUSTOMER_LIST, /* CONTAINS NAMES IN ALPHA ORDER AND
                              KEYS FOR ACCOUNT_LIST            */

           ACCOUNT_LIST,   /* CONTAINS THE ACCOUNTS            */

           PRINT_FI) FILE;      /* FILE FOR THE STATEMENTS     */

   /* ACCOUNT HOLDS THE DATA FROM ONE RECORD OF ACCOUNT_LIST   */

      DECLARE 1 ACCOUNT,
                  2 ACC_NAME,
                     3 NAME CHARACTER(40) VARYING,
                     3 STREET CHARACTER(40) VARYING,
                     3 CITY CHARACTER(20) VARYING,
                     3 STATE CHARACTER(2),
                     3 ZIP CHARACTER(5),
                  2 BALANCE FIXED DECIMAL(9,2),
                  2 TRANSACTIONS,
                     3 TRANS_NO FIXED BINARY,
                     3 ACTIONS(50) FIXED DECIMAL(9,2);

   /* CUST_NO IS THE NUMBER OF CUSTOMERS IN CUSTOMER_LIST.  IT */
   /*           IS THE FIRST ITEM IN THE FILE                  */

      DECLARE CUST_NO FIXED BINARY;

   /* THE ARRAY CUSTOMER HOLDS THE LIST OF CUSTOMERS AND RECORD */
   /*              NUMBERS FROM CUSTOMER_LIST                   */


      DECLARE 1 CUSTOMER(100),
                  2 CUST_NAME CHARACTER(40) VARYING,
                  2 C_NO FIXED BINARY;

   /* COUNTER VARIABLES                                        */

      DECLARE I FIXED BINARY;

      OPEN FILE(CUSTOMER_LIST) RECORD SEQUENTIAL INPUT;
      OPEN FILE(PRINT_FI) STREAM OUTPUT PRINT LINESIZE(60);

   /* READ THE FILE CUSTOMER_LIST INTO THE ARRAY CUSTOMER */

      READ FILE(CUSTOMER_LIST) INTO(CUST_NO);

      DO I = 1 TO CUST_NO;
         READ FILE(CUSTOMER_LIST) INTO (CUSTOMER(I));
      END;
```

Figure 7-1. STATEMENT Illustrates Data Types and I/O Statements

```
     CLOSE FILE(CUSTOMER_LIST);
  /* USE THE KEYS IN CUSTOMER.C_NO TO READ RECORDS FROM          */
  /* ACCOUNT_LIST INTO ACCOUNT                                   */



          OPEN FILE(ACCOUNT_LIST) RECORD DIRECT INPUT;



          DO I = 1 TO CUST_NO;

             READ FILE(ACCOUNT_LIST) INTO(ACCOUNT)
                                        KEY(CUSTOMER(I).C_NO);

          /* OUTPUT THE CONTENTS OF ACCOUNT TO PRINT_FI


             PUT FILE(PRINT_FI) EDIT("STATEMENT FOR",ACCOUNT.
                                        ACC_NAME)
                                    (PAGE,A,X(1),A,SKIP(2),
                                    4(COLUMN(15),A));
             PUT FILE(PRINT_FI) EDIT("CURRENT BALANCE IS",
                                        ACCOUNT.BALANCE)
                                    (SKIP(2),COLUMN(20),A,X(5),
                                     P"SSS,SSS,SS9V.99");

             PUT FILE(PRINT_FI) EDIT("TRANSACTIONS:")
                                        (SKIP(2),COLUMN(25),A);
             PUT FILE(PRINT_FI) EDIT((TRANSACTIONS.ACTIONS(I)
                                DO I = 1 TO TRANSACTIONS.TRANS_NO))
                                (SKIP,COLUMN(43),P"SSS,SSS,SS9V.99");
          END; /* DO INCREMENT */

       END; /* STATEMENT */
```

Figure 7-1. STATEMENT Illustrates Data Types and I/O Statements (continued)

Each statement output by the program looks like this:

```
STATEMENT FOR MILLER,JOHN

             7 ST.PAUL ST.
             BROOKLINE
             MA
             02146

                  CURRENT BALANCE IS          + $10,000.00

                  TRANSACTIONS:
                                              +         $100.00
                                              +         $500.00
                                              +         $325.50
                                              +         $75.00
                                              -         $3.00
```

End of Chapter

Chapter 8
Working with Character and Bit Strings

String Data in PL/I

String data is data whose value is the sequence of characters it contains. For example, when the digits "71" arre treated as string data, they are taken as the ASCII character "7" followed by the ASCII character "1".

PL/I has two basic types of string data: character-string data and bit-string data. The values of character-string data are sequences of ASCII characters; the values of bit-string data are sequneces of bits. PL/I has three character-string data types and two bit-string data types. Character-string data may have the character, aligned character, or character varying data types.

In the first part of this chapter, we will introduce you to PL/I's string-handling operations, functions, and pseudo-variables, and show you how to use them in some simple text-handling programs. In the second part, we will deal with UNSPEC, which allows you to work directly with a data item's internal representation, and with BOOL, with which you can define your own logical operations.

Operators and Built-in Functions for String Handling

PL/I has the concatenate operator (!!) and a large number of built-in functions for working with strings. The following tables give the function or operator, its operand or argument types, and a short description of its use.

Operators and Functions for String Manipulation

Name	Argument Types	Function
!!	bit-string or character-string	Joins the second operand to the first operand.
SUBSTR	bit-string or character-string	References parts of strings. You may use SUBSTR to get the value of a part of a string or to assign a value to a part of a string.
TRANSLATE	character-string	Replaces a character in a string with another character.

Functions which Return Information About Strings

Name	Argument Types	Use
LENGTH	character-string or bit-string	Gives the length of a string data item. With character varying data, it gives the current length.
INDEX	character-string or bit-string	Gives the position of the first occurrence of a string in another string.
VERIFY	character-string	Gives the position of the first character in a string which does not appear in another string.

Functions Using the ASCII Character Sequence

Name	Arguments	Use
COLLATE	none	Returns a 256 character string consisting of characters whose decimal representations are 0 through 255. The first 128 characters are the ASCII character sequence in ascending order.
ASCII	fixed binary	Converts a fixed binary value to the ASCII character whose decimal value is MOD(value,256)
RANK	character	Converts a single ASCII character to an integer corresponding to its position in the ASCII sequence.

Finally, there is the STRING function:

Name	Arguments	Use
STRING	character or bit string aggregate	Converts the aggregate to a string by concatenating its members or elements.

An Example: the Program PIGLAT

To give you an idea what you can do with PL/I's character-handling routines, let's take a look at the program PIGLAT. The program transforms English words into their pig latin equivalents. As you probably recall, you make an English word into a pig latin word as follows:

1. If the English word begins with a vowel, you add "YAY" to it. "I" becomes "IYAY".

2. If the English word begins with consonants, you move the consonants to the end of the word until you come to the first vowel. Then you add "AY" to the end of the word. "STRING" becomes "INGSTRAY".

The first thing the program has to find out is whether the English word begins with a vowel. It does this in line 20 with the SUBSTR and VERIFY functions. SUBSTR has three arguments, the string that contains the substring, the position in the string where the substring begins, and the substring's length. The string we are working with is contained in the variable WORD. Hence, SUBSTR(WORD,1,1) will give us the first character in the string.

To find out whether this character is a vowel, we use the VERIFY function. VERIFY checks whether the characters of one string are contained in another. If they are, it returns the value 0, if not, it returns the position of the first character in the first string which is not contained in the second string. In PIGLAT, the first string is the character returned by SUBSTR; since we want to know whether the character is a vowel, the second string is a variable VOWELS which as the value ''AEIOU''. If the character returned by SUBSTR is a vowel, it will be contained in VOWELS and VERIFY(SUBSTR (WORD,1,1)) will return 0. This will cause the execution of the THEN clause and the program will append ''YAY'' to the word.

If the first letter in the word is not a vowel, the program has to move consonants to the end of the word until it comes to the first vowel. It does this with the DO WHILE loop:

```
DO WHILE (VERIFY(VOWELS,SUBSTR(WORD,1,1)) = 0);
            WORD = SUBSTR (WORD,2,(LENGTH(WORD) - 1))!!
                       SUBSTR(WORD,1,1);
END;
```

Let's look at the statement which moves the characters first:

```
WORD = SUBSTR(WORD,2,(LENGTH(WORD) - 1))!!
SUBSTR(WORD,1,1);
```

This statement uses SUBSTR(WORD,1,1) to get the first character in WORD and SUBSTR(WORD,2,(LENGTH(WORD) - 1)) to get the remaining characters in WORD. LENGTH(WORD) returns the current length of WORD, so 2,(LENGTH(WORD) - 1) will give us all the characters in WORD but the first. Now that we have the first character and all the rest in separate strings, we can concatenate the first character to the end of the remaining characters and assign the resulting value to WORD.

We will want to keep on doing this until the first character in WORD is a vowel. We could use the VERIFY function again to test whether the character is a vowel, but because this is an example program, we have used the INDEX function instead.

The INDEX function has two string arguments. It looks for the second string in the first string, and if it finds it, it returns the position in the first string where the second string starts. If it does not find it, it returns zero. Clearly, we can use INDEX like VERIFY to test for vowels:

```
DO WHILE (INDEX(VOWELS,SUBSTR(WORD,1,1)) = 0);
```

The DO WHILE loop will keep moving consonants from the front to the back of the word until it reaches the first vowel. If WORD contains no vowels, the DO WHILE loop will continue indefinitely.

Once we have the pig latin ''stem'' of the word, all we have to do is concatenate the ''AY'' ending to it. (line 39).

Here is the text of the whole program:

```
SOURCE FILE: PIGLAT.PL
COMPILED ON 8/9/77     AT 10:39:10    BY PL/I REV  00.23

                PIGLAT:
                   PROCEDURE;

                   DECLARE VOWELS CHARACTER(5);
                   DECLARE WORD CHARACTER(30) VARYING;

                   DECLARE (IN,OUT) FILE;

                   OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
                   OPEN FILE(OUT) STREAM OUTPUT PRINT TITLE("@OUTPUT");

                   PUT FILE(OUT) LIST("TYPE THE WORD YOU WANT TO"||
                                         " TRANSLATE");
                   GET FILE(IN) LIST(WORD);

                   VOWELS = "AEIOU";

                /* FIRST CASE: IF THE WORD BEGINS WITH A VOWEL, ADD YAY */

                   IF VERIFY(SUBSTR(WORD,1,1),VOWELS) = 0
                        THEN
                           WORD = WORD || "YAY";
                        ELSE

                   /* SECOND CASE: WORD BEGINS WITH A CONSONANT */

                        DO;

                      /* SWITCH LETTERS FROM THE FRONT TO THE BACK OF WORD  */
                      /* UNTIL THE FIRST LETTER IS A VOWEL                  */

                           DO WHILE (INDEX(VOWELS,SUBSTR(WORD,1,1)) = 0);
                              WORD = SUBSTR (WORD,2,(LENGTH(WORD) - 1))||
                                                SUBSTR(WORD,1,1);
                           END;

                     /* ADD AY TO THE TRANSFORMED WORD                      */

                           WORD = WORD || "AY";

                        END; /* SECOND CASE */


                   PUT FILE(OUT) LIST("EREHAY ISYAY OURYAY ORDWAY",WORD);

                END; /* PIGLAT */
```

Figure 8-1. The Compiler Listing for PIGLAT

Now that you have some idea of what you can do with PL/I's string- handling routines, let's look at the routines themselves more closely.

Concatenate

The concatenate operator has the form:

```
exp!!exp
```

Its result is the second expression appended to the first expression.

You may use concatenate with arithmetic, character-string, and bit-string operands. If both operands are bit-string values, the operator will have a bit-string result; otherwise, the operands will be converted to character strings according to the rules for type conversion.

Examples:

Operation	**Result**
"1001"B !! "111"B	"1001111"B
"ABC" !! "DEF"	"ABCDEF"
"PRODUCT" !! (3 * 2)	"PRODUCT 6"

There are, of course, many uses for concatenation. A special use worth mentioning is outputting long character-string constants. If a character-string constant contains a new-line character, PUT LIST will only output the characters preceding the new-line. Consequently, unless you use !!, you cannot output character string constants larger than 1 line with PUT LIST.

```
PUT FILE(OUT) LIST("THIS STRING IS LONGER THAN"!!
"A SINGLE LINE");
```

Since !! converts arithmetic values to character strings, you can use it to output arithmetic values with labels:

```
DECLARE (NUM_1,NUM_2,SUM) FIXED BINARY;
.
PUT FILE(OUT) LIST("THE FIRST NUMBER IS" !! NUM_1 !!
                                  "THE SECOND NUMBER IS" !! NUM_2 !!
                                  "THE SUM IS" !! SUM);
```

SUBSTR

You may use SUBSTR either as a built-in function or as a pseudo- variable. When you use it as a built-in function, it returns the characters in a section of a string; when you use it as a pseudo-variable, you can assign characters to a section of a string.

When SUBSTR is a function, it has the form:

SUBSTR(string-exp,fixed-bin-exp *[,fixed-bin-exp])*

The first argument gives the string the substring is to be taken from, the second argument gives the position in the string expression at which the substring is to start, and the third argument gives the number of characters in the substring. The third argument is optional; if you omit it, the substring will contain all the characters from the position given by the second argument to the end of the string.

Both fixed binary expressions must be greater than zero, the first expression must be no greater than the length of the string, and the substring specified must not extend past the end of the string.

Note that your program will not detect fixed binary expressions in SUBSTR which are out of bounds unless you compile with the /SUB switch.

Examples:

Expression	**Result**
SUBSTR("12345",2,2)	"23"
SUBSTR("ABCDEF",4)	"DEF"
SUBSTR("10011"B,6)	error

When SUBSTR is a pseudo-variable, it looks like this:

SUBSTR(string-variable,fixed-bin-exp *[,fixed-bin-exp])*

Note that you cannot use string expressions which are not variables in the pseudo-variable. The fixed binary expressions work the same way as with the function.

Examples:

Assignment Statement	**Value of the Variable**
DECLARE WORD CHARACTER(5);	
.	
WORD = "VWXYZ";	
SUBSTR(WORD,2,2) = "AA"	"VAAYZ"
SUBSTR(WORD,4,3) = "BB"	ERROR

LENGTH

You may use LENGTH with character-string and bit-string expressions. It takes the expression as an argument and returns the number of characters in the expression as a fixed binary value:

LENGTH(string-exp)

With character varying variables, LENGTH returns the current length of the variable's value; with string constants and fixed-length string variables, it returns the length of the constant or variable.

Examples:

Function	**Value**
DECLARE STRING CHARACTER(5) VARYING;	
.	
STRING = "A";	
LENGTH(STRING)	1
LENGTH("100111"B)	6
DECLARE FIX BIT(3);	
.	
LENGTH(BIT)	3

Using SUBSTR and LENGTH to Read a Line One Character at a Time

The following program fragment shows how you can use SUBSTR and LENGTH together to read a line one character at a time:

```
DECLARE LINE CHARACTER(80) VARYING;
DECLARE LETTER CHARACTER(1);
DECLARE I FIXED BINARY;

DECLARE INPUT FILE;

.
.
READ FILE(INPUT) INTO LINE;
.
DO I = 1 TO LENGTH(LINE);
          LETTER = SUBSTR(LINE,I,1);
          .
          .
END;
```

Each time the program executes the iterative DO, it reads the next character. The example editing program at the end of this chapter shows one use for this kind of loop.

VERIFY

VERIFY takes two character strings as arguments. It checks to see whether all of the characters in the first string occur in the second. If they do, it returns the fixed binary value 0; if they do not, it returns a fixed binary value giving the position of the first character in the first string that does not occur in the second string:

```
VERIFY(char-string-exp,char-string-exp)
```

Examples:

Function	**Value**
VERIFY("A","AEIOU")	0
VERIFY("AB","AEIOU")	2

A common use for the function is to check whether a character is one of a group of characters. For example, if you wanted to check whether an ASCII character was an upper-case letter before you assigned it to a variable, you could do it like this:

```
DECLARE LINE CHARACTER(80) VARYING;
DECLARE LETTER CHARACTER(1);

.
IF VERIFY(SUBSTR(LINE,I,1),"ABCDEFGHIJKL" !!
                                      "MNOPQRSTUVWXYZ") = 0
          THEN
LINE,I,1);               LETTER =  SUBSTR(
.
.
```

The second argument of VERIFY is all the upper-case letters, and the function will have the value 0 only if SUBSTR(LINE,I,1) is one of these letters.

INDEX

INDEX takes two character-string or bit-string arguments:

```
INDEX(string-exp,string-exp)
```

It returns a fixed binary value which gives the first occurrence of the second string within the first. If either string is a null string, or if the second string is not contained within the first, INDEX returns the value 0.

Examples:

Function	**Value**
INDEX("ABCDE","CD")	3
INDEX("AEIOU","B")	0

The following procedure shows how you can use INDEX along with SUBSTR to shift ASCII characters from upper-case to lower-case. It works by assigning the upper-case ASCII characters to the variable UC_STRING and the lower-case characters to LC_STRING. Since the characters in both strings are in the same order, a lower-case character will be in the same position in LC_STRING as its upper-case equivalent in UC_sTRING. Hence, all that is needed for the transformation is to use the character's location in UC_STRING as an argument in a SUBSTR function that extracts one character from LC_STRING:

```
LCSHIFT: /* WITH INDEX */
            PROCEDURE(CHAR);

            DECLARE CHAR CHARACTER(1);
            DECLARE (LC_STRING,UC_STRING) CHARACTER(26);

            LC_STRING = "abcdefghijklmnopqrstuvwxyz";
            UC_STRING = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

            CHAR = SUBSTR(LC_STRING,INDEX(UC_STRING,CHAR),1);

END; /* LCSHIFT WITH INDEX */
```

TRANSLATE

TRANSLATE lets you replace characters in one string by characters given in another string. It has three character-string arguments:

TRANSLATE(char-string-exp,char-string-exp *[,char-string-exp])*

The first argument gives the string containing the characters the function it is to transform. If this argument is a null string, the function returns to a null string. The second and third arguments define the transformations. The third argument gives the characters which are to be transformed if they occur in the first argument, and the second argument gives the characters they are to be transformed into.

Obviously, the second and third arguments must be the same length if the function is to work. If the second argument is longer than the third, PL/I ignores the extra characters on the right. If the second argument is shorter than the third, PL/I pads the second argument on the right with blanks. For example:

Arguments as Given	**Arguments as Interpreted by PL/I**
"XYZ", "12"	"ZY", "12"
"XY", "123"	"XY□", "123"

The function works like this: whenever a character in the third string occurs in the first string, the function finds which character in the second string occupies the same position in that string as the character to be changed does in the third string. It then changes the character in the first string to the character in the second string. When the function is finished, it returns the transformed string.

Examples:

Function	**Resulting String**
TRANSLATE("ABC", "Z", "A")	"ZBC"
TRANSLATE("1212", "456", "123")	"4545"
TRANSLATE("XYZ!!ASCII(0)", "X")	"□□□X"

In the last case, no third string is given, so PL/I uses the 256 character sequence generated by COLLATE and pads the second string with 255 blanks. Consequently, PL/I translates ASCII 0 into "X" and replaces the other characters with blanks.

You can use TRANSLATE to translate upper-case ASCII characters to lower-ASCII characters if you make the character you want translated the first argument, the lower-case characters the second argument, and the upper-case characters the third argument:

```
LCSHIFT: /* WITH TRANSLATE */
            PROCEDURE(CHAR);

            DECLARE CHAR CHARACTER(1);

            DECLARE (L_CASE,U_CASE) CHARACTER(26);

            L_CASE = "abcdefghijklmnopqrstuvwxyz"
            U_CASE = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

            CHAR = TRANSLATE(CHARM,L_CASE,U_CASE);
END; /* LCSHIFT */
```

COLLATE

The COLLATE function takes no arguments. You may write it with or without parentheses:

```
COLLATE[()]
```

All it does is generate a string 256 characters long which gives all the ASCII characters in ascending order. For example,

```
PUT LIST(SUBSTR(COLLATE(),66,26);
```

will output the 26 upper-case letters, which begin at the 66th character in the string produced by COLLATE.

In the sequence produced by COLLATE, each lower-case ASCII character is 32 characters further up the sequence than its upper-case equivalent. Consequently, you can use COLLATE, SUBSTR, and INDEX to shift an upper-case character to its lower-case equivalent:

```
LCSHIFT: /* WITH COLLATE AND INDEX */
            PROCEDURE(CHAR);

            DECLARE CHAR CHARACTER(1);

            CHAR = SUBSTR(COLLATE(),INDEX(COLLATE(),CHAR) + 32,1);

END; /* LCSHIFT WITH COLLATE AND INDEX */
```

RANK

RANK takes a one-character character-string as an argument and returns the character's rank in the sequence of ASCII characters:

```
RANK(1-char-string)
```

The first character in the sequence has the rank 0, so the integer returned by RANK will always be one less than that returned by INDEX(COLLATE,char) for the character.

For example,

```
RANK("A")
```

will have the value 65. For an example of how you can use RANK, see the following discussion of ASCII.

ASCII

ASCII is the reverse of RANK. It takes a fixed binary expression as an argument and returns a single ASCII character:

```
ASCII(fixed-bin-exp)
```

It works by dividing the fixed binary expression by 256, taking the remainder, adding 1 to it, and using that value as an argument in SUBSTR:

```
ASCII(X) = SUBSTR(COLLATE(),MOD(X,256) + 1,1)
```

Hence, ASCII(65) is "A".

You can use ASCII and RANK together to change upper-case characters to lower-case characters. RANK finds the position of the upper-case character in COLLATE, and if you add 32 to that value, you will have the position of the lower-case character:

```
LCSHIFT: /* WITH RANK AND ASCII */
             PROCEDURE(CHAR);

             DECLARE CHAR CHARACTER(1);

             CHAR = ASCII(RANK(CHAR) + 32);

END; /* LCSHIFT WITH RANK AND ASCII */
```

STRING

You may use STRING either as a function or as a pseudo- variable. As a function, it converts a data aggregate consisting wholly of bit data or character data to a bit string or character string. As a pseudo-variable, it assigns a character- or bit-string value to a character- or bit-string aggregate. For either use, it has the form:

```
STRING(char-or-bit-aggregate)
```

There are some limitations:

1. The aggregate's storage must be connected.
2. The aggregate must consist of bit or character data only. It may not contain aligned bit, aligned character, or character varying data.

When you assign values to the pseudo-variable STRING, they are padded or truncated as follows:

If the character- or bit-string is longer than the storage space for the aggregate, the string is truncated from the right; if it is shorter, it is padded on the right. Character strings are padded with blanks and bit strings are padded with "0" bits.

For example:

```
DECLARE LETTERS(20) CHARACTER(1);
.
STRING(LETTERS) = "TWENTY LETTERS";
```

The assignment statement will assign the characters of "TWENTY LETTERS" to LETTERS(1) through LETTERS(14) and assign blanks to the remaining elements. Though it is a bit clumsy, you can also use STRING in an upper-case to lower-case shift program:

```
LCSHIFT: /* WITH STRING */
          PROCEDURE(CHAR);

          DECLARE LETTERS(52) CHARACTER(1);
          DECLARE CHAR CHARACTER(1);
          DECLARE I FIXED BINARY;
          DECLARE SW BIT;

          STRING(LETTERS) = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" !!
                            "abcdefghijklmnopqrstuvwxyz";
          I = 0;
                   SW = "1    "B;

          DO WHILE(SW);
                   I = I + 1;
                   IF CHAR = LETTERS(I)
                            THEN
                            DO;
                                     CHAR = LETTERS(I + 26);
                                     SW = "0"B;
                            END;
                   END;

END; /* LCSHIFT WITH STRING */
```

An Example: A Simple Editor

You can use most of the string-handling functions we have discussed in an editor that takes upper- case ASCII text and leaves upper-case letters preceded by a " \ " character upper-case, but shifts all other letters to lower-case.

The program asks the user for the names of his input and output files, opens the files, and reads the input file a line at a time. It then scans the input line a character at a time. What happens depends on the character:

1. If the character is neither a letter nor \ , the program appends it to the output line.
2. If the character is \ , the program reads the next character and outputs it unchanged to the output line.
3. If the character is a letter not preceded by \ , the program shifts the character and adds it to the output line.

The program text does not include the shift procedure. You could use any of the shift procedures from the preceding examples.

```
     LC_EDIT:
        PROCEDURE;

/***********************************************************************/
/*                       DECLARATIONS                                  */
/***********************************************************************/

        DECLARE FILENAME CHARACTER(31) VARYING; /* HOLDS
                                 ACS FILE NAME      */

        DECLARE LINE CHARACTER(80) VARYING; /* LINE FOR INPUT
                                                       TEXT */
        DECLARE OUT_LINE CHARACTER(80) VARYING; /* LINE FOR OUTPUT
                                                       TEXT */

        DECLARE LETTER CHARACTER(1);/* HOLDS LETTER TO  BE SHIFTED */

        DECLARE LCSHIFT ENTRY(CHARACTER(1)); /* SHIFT ROUTINE */

        DECLARE I FIXED BINARY;

        DECLARE (IN, OUT, TEXT, ED_TEXT) FILE;
         /* IN AND OUT ARE THE TERMINAL  */
         /* TEXT IS THE INPUT FILE       */
         /* ED_TEXT IS THE OUTPUT FILE   */

/*******************************************************************/
/*  OPEN THE TERMINAL AS AN INPUT AND OUTPUT FILE AND THEN       */
/*  GET THE NAMES OF THE INPUT AND OUTPUT FILES FOR THE TEXT     */
/*******************************************************************/

        OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
        OPEN FILE(OUT) STREAM OUTPUT PRINT TITLE("@OUTPUT");

        PUT FILE(OUT) LIST("WHAT FILE DO YOU WANT EDITED?");
        GET FILE(IN) LIST(FILENAME);
        OPEN FILE(TEXT) STREAM INPUT TITLE(FILENAME);

        PUT FILE(OUT) SKIP LIST("WHAT DO YOU WANT TO CALL THE" ||
                         " EDITED FILE?");
        GET FILE(IN) LIST(FILENAME);
        OPEN FILE(ED_TEXT) STREAM OUTPUT PRINT TITLE(FILENAME);

/***********************************************************************/
/*  STOP THE PROGRAM WHEN IT HAS FINISHED READING THE INPUT FILE    */
/***********************************************************************/

        ON ENDFILE(TEXT)
           BEGIN;
           PUT FILE(OUT) SKIP LIST("FILE "|| FILENAME || " FINISHED");
           STOP;
           END;

/***********************************************************************/
/*      BEGIN PROCESSING.  READ THE INPUT FILE A LINE AT A TIME     */
/*      READ THE LINE A CHARACTER AT A TIME, SHIFT THE UNMARKED     */
/*      LETTERS, MAKE AN OUTPUT LINE AND OUTPUT THE LINE TO         */
/*      THE OUTPUT FILE.                                            */
/***********************************************************************/
```

Figure 8-2. This Program is a Simple Text Editor

```
/* LOOP TO READ THE INPUT FILE UNTIL THERE IS NO MORE DATA    */

   DO WHILE("1"B);

    /* READ A LINE OF THE INPUT FILE; SET OUTPUT LINE TO ""  */

    READ FILE(TEXT) INTO(LINE);
    OUT_LINE = "";


   /* READ AND PROCESS LINE ONE CHARACTER AT A TIME         */

     I = 1;
     DO WHILE(I <= LENGTH(LINE));

     /* FIRST CASE: THE CHARACTER IS THE SHIFT CHARACTER Δ  */

        IF SUBSTR(LINE,I,1) = "Δ"
            THEN

            /* MOVE FORWARD 1 CHARACTER AND OUTPUT THE      */
            /* CHARACTER WITHOUT SHIFTING IT DOWN           */

            DO;
            I = I + 1;

            /* THE IF THROWS OUT "Δ" IF IT IS THE LAST CHAR   */
            /* IN THE LINE                                    */

            IF I ↑> LENGTH(LINE)
                THEN
                OUT_LINE = OUT_LINE || SUBSTR(LINE,I,1);
                I = I + 1;
            END;
        /* SECOND CASE: THE CHARACTER IS AN UPPER CASE LETTER */

            ELSE
            IF VERIFY(SUBSTR(LINE,I,1),"ABCDEFGHIJKL" ||
                                       "MNOPQRSTUVWXYZ") = 0
                THEN
             /* CHANGE THE CHARACTER TO A LOWER-CASE CHAR     */
                DO;
                LETTER = SUBSTR(LINE,I,1);
                CALL LCSHIFT(LETTER);
                OUT_LINE = OUT_LINE || LETTER;
                I = I + 1;
                END;

                 /* THIRD CASE: THE CHARACTER IS SOME OTHER   */
                 /*         CHARACTER                         */

                ELSE

                  /* OUTPUT THE CHARACTER UNCHANGED           */

                DO;
                OUT_LINE = OUT_LINE || SUBSTR(LINE,I,1);
                I = I + 1;
                END;

        END; /* CHARACTER - READING LOOP */

        WRITE FILE(ED_TEXT) FROM(OUT_LINE);

   END; /* FILE-READING LOOP */

   END; /* LC_EDIT */
```

Figure 8-2. This Program is a Simple Text Editor (continued)

Bit-string Functions

As we saw in the preceding discussion, you can use the string operator !! and the built-in functions SUBSTR, LENGTH, INDEX, and STRING with bit-string arguments as well as with character-string arguments. PL/I has two other functions for working with bit-strings. The first is UNSPEC, which converts arguments of any data type to a bit-string giving their internal representation. The second is BOOL, which allows you to define your own logical operators.

When you use these operators and functions along with the bit- string operators (↑ , !, &) you can work directly with your data's internal representations.

UNSPEC

You may use UNSPEC either as a function or as a pseudo-variable. For both uses, it has the form:

```
UNSPEC(variable)
```

The variable may be an elementary variable or an aggregate, and it may have any data type, but its storage must be connected.

As a function, UNSPEC returns a bit string which is the internal representation of the variable; when you use UNSPEC as a pseudo-variable, you can assign a string of bits directly to the variable's storage. If the variable's storage has fewer bits than the bit string, the string will be truncated from the right; if it has more, the string will be padded from the right with zero-valued bits.

For example, if you have a program fragment like this:

```
DECLARE ONES(3) FIXED BINARY;
.
DO I = 1 TO 3;
          ONES(I) = 1;
END;
```

then

```
UNSPEC(ONES)
```

will return the bit string "00000000000000010000000000000001 0000000000000001"B, which is the internal representation of an array of 3 fixed binary elements, each one with the value 1.

You can write a version of the character-shifting procedure LCSHIFT with UNSPEC and the bit-string operator !. It takes advantage of the fact that the internal representations of lower-case letters are the same as the internal representations of their upper-case equivalents, except that the third bit of the byte is 1 instead of 0. For example, the internal representation of "A" is 01000001, while the internal representation of "a" is 0110001. Conseqently, all we have to do to transform an upper-case letter to a lower-case letter is change the third bit to 1. We can use UNSPEC and ! to do it like this:

```
LCSHIFT: /* WITH UNSPEC */
          PROCEDURE(CHAR);

          DECLARE CHAR CHARACTER(1);

          UNSPEC(CHAR) = UNSPEC(CHAR) ! "00010000"B;

END; /* LCSHIFT WITH UNSPEC */
```

Since the result of ! is "1"B when either operand has a "1"B in that position, UNSPEC(CHAR) ! "00100000"B will change the third bit of CHAR from 0 to 1 and the character from upper-case to lower-case.

BOOL

BOOL allows you to define your own Boolean (logical) operators. The function takes three arguments. The first two, which may be bit-strings of any length, are like the operands of a bit-string operator. The third, which is a bit-string constant 4 bits long, defines the results of the operation:

```
BOOL(bit-exp-1,bit-exp-2,bit-exp-3)
```

The function works like a bit-string operator. It starts at the left of the first argument and commpares the argument's first bit with the corresponding bit of the second argument. The comparison produces a "1"B or a "0"B as determined by the third argument. The comparisons between the two strings continue until all the bits have been compared, and the result is a bit string that is as long as the longer of the first two arguments and that contains the sequence of bits determined by the third argument.

The third argument determines the result of the comparison like this:

- If both bits compared are 0, the first bit in the third argument gives the result.
- If the bit in the first argument is 0 and the bit in the second 1, the second bit in the third argument gives the result.
- If the bit in the first argument is 1 and the bit in the second argument is 0, the third bit in the third argument gives the result.
- If both arguments have "1" bit, the fourth bit in the third argument gives the result.

Or expressed as a table:

Values of Bits in Operands		**Bit in Third String with Result**
Bit in Operand 1	**Bit in Operand 2**	
0	0	bit 1
0	1	bit 2
1	0	bit 3
1	1	bit 4

For example, if you wanted to define an exclusive "or", that is, an "or" which would have a result of "1"B when either but not both of the operands had a "1"B, the third string would be "0110"B.

You might use "0110"B as the third argument in an expression like this:

```
BOOL("100101"B,"100001"B,"0110"B)
```

Since "0110"B says that the string resulting from the comparison will have a "1"B only where one but not both of the arguments has a "1"B, the result of the expression will be "000100"B.

BOOL returns a null bit string if either of the first two arguments is a null string. If one of the first two arguments is shorter than the other, the function pads the shorter argument on the right with 0 bits before it begins the comparison.

You can use BOOL to make your own logical functions. For example, a function called XOR, which took two 1-bit arguments and returned the result of the logical operation exclusive ''or'' might look like this:

```
XOR:
        PROCEDURE(ARG_1,ARG_2) RETURNS(BIT(1));

        DECLARE(ARG_1,ARG_2) BIT(1);

        RETURN(BOOL(ARG_1,ARG_2, "0110"));

END; /* XOR */
```

An invocation for XOR might look like this:

```
IF XOR(LOWER < 0, UPPER > 10) THEN ...
```

End of Chapter

Chapter 9 Pictured Data

Pictured data has fixed decimal values but is stored as a character string containing the digits of the value plus characters such as $, . , /, CR, and DB. You declare the pictured variables with the PICTURE keyword. The keyword is followed by a picture specifier. The picture specifier is a sequence of special characters surrounded by quotation marks. It determines the precision and scale of the fixed decimal value and defines the form of the character string:

```
DECLARE STRING PICTURE "$9V.99";
```

The picture specifier for STRING defines it as a pictured value whose fixed decimal value has a precision of 3 and a scale of 2, and whose character string will include $ and . as well as the digits. The following example shows the effect of the picture specifier:

```
DEPICTER:
          PROCEDURE;

          DECLARE STRING PICTURE "$9V.99";

          STRING = 5.01;
          PUT LIST(STRING);

          STRING = 2;
          STRING = 2 * STRING;
          PUT LIST(STRING);

          STRING = 0;
          PUT LIST(STRING);

END; /* DEPICTER */

Output:          $5.01          $4.00          $0.00
```

AOS PL/I uses the following characters in picture specifiers:

```
9 Z * , . / B - + S V $ DB CR
```

All of these characters but V specify positions in the pictured value's character string that will be occupied by characters. The V gives the location of the decimal point, and thus determines the value's scale. "$9V.99" has three characters which specify digits, so STRING has a precision of 3. Two of the digit specifiers are to the right of the V, so STRING has a scale of 2. As with fixed decimal variables, the maximum precision is 16, and the scale may range between 0 and the precision. The length of the value's character string is the number of picture characters other than V in it. Hence, STRING has a length of 5.

The characters 9, Z, and *, and under some circumstances, the characters $, S, +, and - specify positions in the string that may be occupied by digits of the fixed decimal value. The number of characters which specify digit positions determines the fixed decimal value's precision.

Using Pictured Variables in your Programs

AOS PL/I treats pictured variables like fixed decimal variables except in two cases:

1. If you use PUT LIST to output the value of a pictured variable, you will get the character string.

2. If you use a pictured variable as an argument for a fixed decimal parameter, the value of the variable will be passed by value.

The general rule that pictured variables are treated like fixed decimal variables has the following specific consequences:

1. You cannot assign a value to a pictured variable unless PL/I can convert the value to a fixed decimal value. The following are all legal assignments:

```
DECLARE PIC_VAL PICTURE "$9,999V.99";
.
PIC_VAL = 99.99;
PIC_VAL = 3.5E+2;
PIC_VAL = "53.42";
```

   This next assignment, however, will not work, because PL/I cannot convert a string with non-numeric characteristics like $ and , into a fixed decimal value:

```
PIC_VAL = "$1,000.00"
```

2. When you assign a pictured value to another variable, you are assigning the fixed decimal value, not the character string:

```
DECLARE CHARS CHARACTER(10) VARYING;
PIC_VAL = 1456.22;
.CHARS = PIC_VAL;
.PUT FILE(OUT) LIST(CHARS);
```

   The PUT LIST statement will output 1456.22.

3. Precision and scale with pictured variables work like precision and scale with fixed decimal values if a program assigns to a pictured variable a value with more digits to the left or right of the decimal point than allowed by the picture, the extra digits on the left will be truncated from the left, and those on the right will be truncated from the right. The program will raise the error condition under only two circumstances: if the precision of the value exceeds 16 or if you are using the commercial instruction set. To use the commercial instruction set, you must bind PL1ECIS with your program.

4. You can use pictured data as operands in arithmetic operations and as arguments with the arithmetic built-in function, but you cannot use it as arguments with built-in functions requiring character-string arguments. For example,

```
ROUND(PIC_VAL * 2)
```

   is legal, but

```
LENGTH(PIC_VAL)
```

   is not.

   If you want to use a pictured value in contexts requiring a character-string value, you have to define a character variable onto the pictured variable;

```
DECLARE PIC_VAL PICTURE "$9,999V.99;

DECLARE PIC_STRING CHARACTER(9) DEFINED(PIC_VAL);
```

   For details on defined variables, see the next chapter.

5. Pictured variables follow the same rules for input as fixed decimal variables; they will only accept input that PL/I can interpret as arithmetic values. For example, if you use a pictured variable in a GET LIST statement, it will read values like 3.56, 2, or 1.5E+02, but not values like $1,899.95. If you want a pictured variable to read values containing such characters, you have to use GET EDIT with a P data format item. For details, see Chapter 7.

6. Only PUT LIST outputs the character-string value of a pictured variable to a stream file. Unless you use the P data format item when you output a pictured value with PUT EDIT, you will get the fixed decimal value:

   ```
   PIC_VAL = 166.32;
   PUT FILE(OUT) EDIT(PIC_VAL)(A);
   ```

   will output 166.32, not $0,166.32.

7. Pictured data is stored in record files in its internal representation. If the file DOLLARS is a record file, then no conversion takes place in the following:

   ```
   WRITE FILE(DOLLARS) FROM(PIC_VAL);
   ```

8. A pictured variable is equivalent to another pictured variable only if both have the same picture specifier.

   ```
   DECLARE PIC_1 PICTURE "$$,$$9V.99";
   DECLARE PIC_2 PICTURE "$$,$$9V.99";
   DECLARE PIC_3 PICTURE "$9,999V.99;
   ```

   PIC_1 and PIC_2 are equivalent to each other, but neither is equivalent to PIC_3.

The Picture Specifier

Table 9-1. Picture Specifier Characters

Specifier Character	Character it May Produce in the Character-String Value
9	digit
Z	digit or blank
*	digit or *
S	+, -, digit, or blank
+	+, digit, or blank
-	-, digit, or blank
$	$, digit, or blank
/	/
B	blank
,	,
.	.
DB	DB or 2 blanks
CR	CR or 2 blanks
V	No character; determines the decimal value's scale

How to Use the Specifiers

As you can see from the table of the picture specifier characters and the characters they produce, there are three basic types of picture specifiers: those that always specify the position of a digit, namely 9, Z, and *, those that may specify the position of a digit, namely S, +, -, and those that never specify the position of a digit, namely /, B, , , ., DB, and CR. Then there is the V character, which specifies the position of the decimal point, but does not insert a character there.

9, Z, and *

If the pictured value has a significant digit in a position marked by 9, Z, or *, the digit will appear in that position. If there is no significant digit, 0 will appear is you used 9, a blank if you used Z, or a * if you used *.

For example:

Input	Picture	Output
2	"9999"	0002
2	"ZZZ"	2
2	"***"	**2

Combinations: You cannot use Z and * in the same picture; if you use either with 9, you must put the Z's and the *'s to the left of the 9.

Input	Picture	Output
2	"*99"	*02

$, S, +, -

This group of characters, which may specify digit positions, is called the drifting picture specifiers. They work like this:

1. If one of the characters belonging to this group appears once in a picture specifier, the character specified by that character will appear in that position in the string.
2. If a sequence of more than one of the characters appears, all of the specifiers but the leftmost specify positions where a digit will appear if the value has a significant digit in that position. If there is no significant digit, the sequence of specifiers will output the character specified immediately to the left of the most significant digit and will fill the remaining positions with blanks.

The drifting picture specifiers output characters as follows:

$ outputs $ S outputs + if the value is positive and - if it is negative + outputs + if the value is positive and a blank if it is negative - outputs - if the value is positive and a blank if it is negative

Precision with drifting characters: When you use more than one drifting character, the leftmost character cannot represent a digit. Thus, if you want your picture to represent a number of up to three digits preceded by a sign, you must use four S characters: "SSSS".

Combinations: You can use only one of the sign characters in a picture: +, -, S, CR, DB. If you use +, -, or S in a floating manner, you must place them to the left of any 9's in the picture, and you cannot use them in the same picture as * or Z. If you use +, -, or S singly, you may place the character in either the rightmost or leftmost picture position. You may only place CR and DB in the rightmost position.

Examples:

Input	Picture	Output
2	"$999"	$002
2	"$$$$"	$2
-20	"+999"	020
+20	"+999"	+020
+20	"++++"	+20
+20	"-999"	020
-20	"S999"	-020
20	"SSSS"	+20
25	"SSS9"	+25
25	"S$$$9"	+25

/, B, , , .

You use these specifiers to insert characters between digits. "," inserts a comma, "/" a slash, "." a period, and B a blank. If the digit preceding the character is replaced by a blank or *, the character will be replaced by a blank or *. Note that these characters have no mathematical significance. The V character, and not the "." character determines where the value's decimal point is. If you want the decimal point to appear in a position, you have to use V. in your specifier.

Examples:

Input	Picture	Output
1256	"99/99"	12/56
1234	"99.99"	12.34
1234	"9,999"	1,234
7476	"9B9B99"	7476
0	"Z,ZZZ,ZZ9"	□□□□□□□ 0

CR and DB

CR and DB must be the rightmost characters in your picture specifier. When you specify either, the characters will appear if the pictured value is negative:

Input	Picture	Output
-120	"$$$$DB"	$120DB

V

V determines where the pictured value's decimal point will fall. As noted above, V does not insert the decimal point. If you want the point to appear, you must use V.:

Input	Picture	Output
12.56	"999V999"	012560
12.56	"999V.999"	012.560

End of Chapter

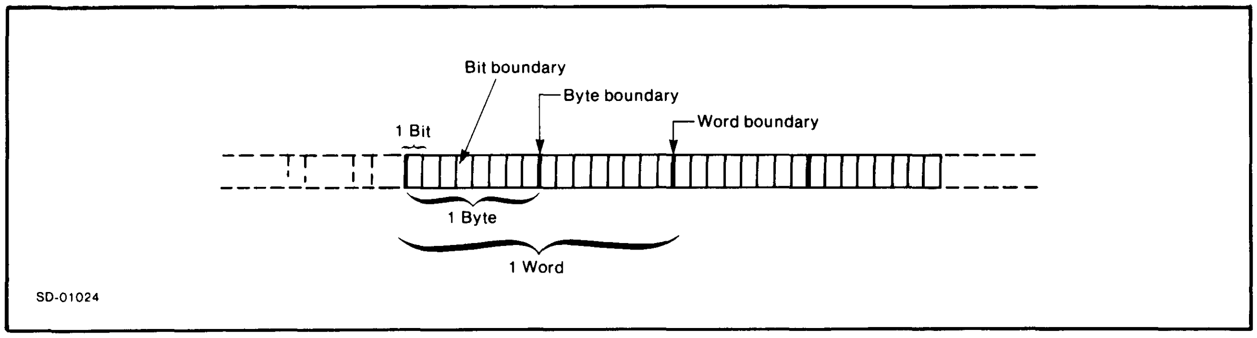
Chapter 10
Storage Classes, Based and Defined Variables

Introduction

For most purposes, you need not worry about how PL/I actually stores data in memory. The language does, however, have data attributes, statements, and functions that allow you to deal directly with storage. In order to use these features, you have to know how PL/I stores data. In this chapter, we discuss storage, PL/I's storage classes, and how you can work with storage in a PL/I program.

Data Types and Storage

Within memory, the sequences of bits which make up the internal representation of a data item are grouped into larger entities called bytes and words. In Data General computers, 8 bits make up a byte and 2 bytes make up a word. A schematic representation of an area of memory looks like this:



The junction between 2 bits of memory is called a bit boundary; that between two bytes is called a byte boundary, and that between two words is a word boundary. As you can see from the illustration, byte boundaries are also bit boundaries, and word boundaries are also byte and bit boundaries.

Where the storage for data of a given data type begins depends on the data type's alignment. If a data type is word aligned, for example, its storage must begin on a word boundary. As you will see later on in this chapter, the alignment of a data type determines what kind of pointer it may have and what other data types may share storage with it. The following table gives the alignments of the PL/I data types:

Alignment	**Data Type**
bit	BIT*
byte	CHARACTER FIXED DECIMAL PICTURE
word	FIXED BINARY FLOAT BINARY ALIGNED BIT ALIGNED CHARACTER CHARACTER VARYING POINTER LABEL FILE ENTRY

Arrays have the alignment of their elements; structures have the alignment on the member with the coarsest alignment. A structure with fixed binary and character variables, for example, is word- aligned because fixed binary data is word-aligned.

Aligned Data

In certain circumstances, the alignment of a data item will determine how you can use it. For this reason, AOS PL/I has two special aligned data types: aligned character and aligned bit data. These data types work exactly like character data and bit data, but their storage is word aligned instead of byte or bit aligned.

You declare aligned character variables with the

ALIGNED CHARACTER[(n)]

attribute keywords. The (n) gives the number of characters in the data item. The default value of n is 1. If you assign a character string with fewer than n characters to the variable, PL/I will pad the value on the right with blanks; if you assign a string which is longer than n, PL/I will truncate from the right. You can use aligned character data in any context which requires a character- string expression.

Declarations of aligned bit variables take the

ALIGNED BIT[(n)]

attribute keywords. (n) gives the number of bits in the data item. The default value is 1. If you assign a bit-string value to an aligned bit variable which is longer than the length of the variable, PL/I truncates from the right; if the bit-string value is shorter than the length of the variable, PL/I pads it on the right with zero-valued bits. You can use aligned bit data in any context requiring a bit-string expression.

* There is one exception to these general rules: bit variables with the defined and based storage classes are word-aligned. For details, see below.

PL/I's Storage Classes

All PL/I variables have a storage class. A variable's storage class determines how and when PL/I sets up storage for the variable. There are five storage classes: automatic, static, parameter, based, and defined. You give a variable a storage class when you declare the variable. The following table gives the keywords and other means used to declare the storage class:

Storage Class	**Method of Declaration**
automatic	AUTOMATIC of no keyword; automatic is the default.
static	STATIC keyword.
parameter	The parameter appears in the procedure's parameter list; see Chapter 6.
defined	DEFINED keyword.
based	BASED keyword.

Some sample storage class declarations:

```
CLASSES:
          PROCEDURE(NAME_LIST);

          DECLARE NAME_LIST(100) CHARACTER(20) VARYING;
          DECLARE REP_NO FIXED BINARY STATIC INTERNAL INIT(0);

          DECLARE NAME CHARACTER(20) VARYING BASED;
          DECLARE I FIXED BINARY;

          .
```

In the above set of declarations, NAME_LIST has the parameter storage class, REP_NO has the static storage class, NAME has the based storage class, and I has the default automatic storage class.

The Meaning of the Storage Classes

Depending on its storage class, a variable will either have storage of its own or share the storage of another variable. If it has its own storage, the storage class will determine whether PL/I allocates storage for the variable at the beginning of program execution, when it activates the variable's block, or when it executes an ALLOCATE statement for the variable.

The automatic storage class is the default storage class in PL/I. Storage for variables with the automatic storage class is allocated when the block in which the variable is declared is activated, and is released when the block activation ends. For details on automatic storage and its implications for PL/I programming, see Chapter 6.

Storage for variables with the static storage class is allocated at the beginning of program execution. It remains allocated until the program ends. You can thus use static storage to retain values between block activations. Static variables are also the only variables in AOS PL/I which may be given initial values in the DECLARE statement. For details, see Chapter 12.

Variables with the parameter and defined storage classes have no storage of their own; instead, they share the storage of other variables. Parameters share the storage of their arguments and defined variables share the storage of the variable which appears with the DEFINED keyword. For details on parameters, see Chapter 6; for defined variables, see below.

You use based variables when you want direct control over storage allocation or storage sharing. The following sections discuss based variables and their use.

Based Variables and Pointers

With non-based variables, the compiler generates code that automatically associates the variable name with a given generation of storage when the program is executed. When you declare a variable with the based storage class, the compiler keeps track of the variable's name and attributes, but does not associate it with any storage. In order to reference an area of memory with a based variable, you have to use it together with a pointer. A pointer is a variable whose value is simply the location of a byte or word in memory. The pointer and based variable together are the equivalent of the variables we have dealt with up to now: the pointer gives the location where the storage referred to by the based variable begins; the based variable determines how the computer will interpret the contents of the storage that begins at the location given by the pointer.

You may use based variables and pointers to refer to storage belonging only to the based variable or to storage that belongs to other variables. To give a based variable storage of its own, you use the ALLOCATE statement; to give it storage belonging to another variable, you use the based variable with a pointer that marks the location of the other variable's storage

The following program gives an example. It declares a based variable and a pointer, allocates storage for the variable, assigns a value to it, and outputs the value:

```
BASED_VAR:
            PROCEDURE;

            DECLARE ARROW POINTER;
            DECLARE INT_VAL FIXED BINARY BASED;

            ALLOCATE INT_VAL SET(ARROW);

            ARROW -> INT_VAL = 7777;
            PUT LIST(ARROW -> INT_VAL);

END; /* BASED_VAR */
```

Output: 7777

In the example, ARROW is the pointer variable and INT_VAL is the based variable. As you can see, the only difference between the declaration of a based variable and other variable declarations is the BASED keyword. When the computer executes the program, it sets up storage for ARROW, but none for INT_VAL.

The ALLOCATE statement,

```
ALLOCATE INT_VAL SET(ARROW);
```

sets up the single word of storage required for a fixed binary variable and assigns the location of the word to ARROW. Now that we have both the description and the location of the storage, we can use it in the program. To refer to storage belonging to based variables, you use a special form of reference called pointer-qualified reference. In pointer-qualified reference, you use a pointer expression, the - > symbol, and the based variable name:

```
ARROW -> INT_VAL
```

ARROW contains the address of INT_VAL, which in turn contains a fixed binary value.

As you can see from the assignment and PUT LIST statements in BASED_VAR, you use pointer-qualified references the same way you use other variable names:

```
ARROW -> INT_VAL = 7777;
PUT LIST(ARROW -> INT_VAL);
```

Now that you have a general idea how based variables and pointers look and work, we can go into the details.

Pointers

A pointer is a data item whose value is a location in memory. You may give variables the pointer data type, and you may write functions which return pointer values. To declare a pointer variable, you use the POINTER keyword:

```
DECLARE ARROW POINTER;
```

PL/I does not define conversions between pointers and another data types, so you cannot assign values of other data types to pointers and you cannot output pointer values to stream files. The only operations you can perform with pointer operands are = and ↑ =. Two pointers are equal when they represent identical locations in memory.

In AOS PL/I, there are two types of pointers: word pointers and byte pointers. Word pointers contain the location of a word in memory, while byte pointers contain the location of a byte. The pointer's type depends on the alignment of the variable whose address the pointer contains. For example, ARROW is a word pointer because its ALLOCATE statement allocated storage for the word-aligned variable INT_VAL.

Based Variables

We have already seen what the declaration for a based variable means and how you can use based variables and pointers together to refer to data stored in memory. The keyword for the based storage class takes the form:

BASED *[(pointer-exp)]*

If you do not give the pointer expression in parentheses following the keyword, you must always explicitly specify a pointer when you use the based variable in a reference. If you do give a pointer expression, PL/I will automatically associate that expression with the based variable and you can refer to the variable without specifying the pointer.

Extent Expressions with Based Variables

You can use expressions in the DECLARE statements for based variables the same way that you can use them in the DECLARE statements for automatic variables. There is, however, one important difference:

PL/I evaluates the variable's extent expressions each time you refer to the variable.

Consequently, if the values of the extent expressions change during execution, so will the size of the based variable. For details, see the discussion of pointer-qualified reference below.

Giving Storage to a Based Variable

In the example program BASED_VAR, we used the ALLOCATE statement to set up storage for our based variable. As we mentioned in the introduction, you can also give a based variable storage belonging to another variable. One way to do this is the ADDR built-in function. In the following, we will go into greater detail about ALLOCATE and discuss the FREE statement and the ADDR and NULL built in functions as well.

The ALLOCATE statement

The ALLOCATE statement allocates storage for a based variable and assigns the location of the storage to a pointer. Its syntax:

```
ALLOCATE based-variable SET (pointer-variable);
```

The based variable must be an scalar based variable or an entire based aggregate. You cannot allocate storage for part of an aggregate.

When the computer executes the ALLOCATE statement, it finds the current value of the extent expressions in the DECLARE statement for the BASED variable, examines the data type attributes, of the variable, allocates the amount of storage required, and assigns the location of the storage to the pointer which appears in parentheses following the SET keyword. Note that the amount of storage allocated for the variable depends on the current value of its extent expressions. If the values of the extent expressions change between allocations, the amount of storage allocated will change also.

Example:

```
DECLARE (AL_1,AL_2) POINTER;
DECLARE STRING CHARACTER(LEN);
DECLARE LEN FIXED BINARY;
.
LEN = 3;
.
ALLOCATE STRING SET(AL_1);
.
.
LEN = 10;
.
.
ALLOCATE STRING SET(AL_2);
.
```

Here, the first allocation of STRING will be 3 bytes long; the second will be 10 bytes long.

Multiple Allocations Of Storage

The ALLOCATE statement allocates new storage each time it is executed. Hence, you can allocate storage for a single based variable over and over. As long as the location of each allocation is contained in a different pointer, you can refer to any of them.

For example, you could use ALLOCATE to allocate storage for the same based variable 5 times and assign the location of each allocation to an element in an array of pointers. To refer to a given allocation, you need only use the element containing its pointer in your pointer-qualified reference:

```
MULT_ALL:
           PROCEDURE;

           DECLARE PT_S(5) POINTER;
           DECLARE INTEGERS FIXED BINARY BASED;
           DECLARE I FIXED BINARY;

           DO I = 1 TO 5;
           ALLOCATE INTEGERS SET(PT_S(I));
           PT_S(I) -> INTEGERS = 2*I;
           END;

           DO I = 1 TO 5;
           PUT LIST(PT_S(I) -> INTEGERS);
           END;

END; /* MULT_ALL */
```

```
Output: 2 4 6 8 10
```

In the first DO loop, the program allocates storage for INTEGERS, assigns the location of the storage to an element of PT_S, and then assigns a value to the storage. the program does this five times and then goes on to the second DO loop, where it outputs the value contained in the storage identified by each element of the array.

Note that if repeated allocations of a based variable have the same pointer, the pointer will only point to the last allocation and you will not be able to reference the earlier ones.

Freeing Storage Allocated to a Based Variable

Once you have allocated storage for a based variable, it remains allocated until you free it with a FREE statement or the program is finished. The FREE statement has the form:

```
FREE pointer-qualified-reference;
```

For example, a DO loop which freed the storage allocated to INTEGERS in the preceding example would look like this:

```
DO I = 1 TO 5;
            FREE PT_S(I) -> INTEGER;S
END;
```

WARNINGS

1. It is an error to reference freed storage. After you free a generation of storage, both the pointer and the contents of the storage are undefined. If you nevertheless attempt to reference freed storage, the consequences will be unpredictable.

2. It is an error to attempt to free storage which has not been allocated with the ALLOCATE statement.

Using Based Variables and Pointers to Reference Storage Belonging to Other Variables

When you use a based variable in a pointer-qualified reference, the based variable references whatever storage begins at the location given in the pointer. Hence, if you use the same pointer with several different based variables, you can reference the same storage with different pointer-qualified references. For example, if we add a second based variable to the program BASED_VAR, we can do this:

```
BASED_VAR:
            PROCEDURE;

            DECLARE ARROW POINTER;
            DECLARE INT_VAL FIXED BINARY BASED;
            DECLARE NUMBER FIXED BINARY BASED;

            ALLOCATE INT_VAL SET(ARROW);

            ARROW -> INT_VAL = 7777;
            PUT LIST(ARROW -> NUMBER);

END; /* BASED_VAR */
```

The output will still be 7777, because NUMBER has the same pointer, and therefore the same storage as INT_VAL.

Of course, when a based variable refers to the same region in memory as another variable, it is sharing storage with that variable. As the example shows, when variables share storage, a change in the value of any of the variables is a change in all of them.

PL/I places limits on the kinds of variables based variables may share storage with. In general, the variables which share storage must be equivalent. A fixed decimal variable, for example, cannot share storage with a fixed binary variable. There are two exceptions to this general rule: character variables may share storage with character and picture variables which have different extents; and bit variables may share storage with bit variables with different extents. Consequently, a scalar character variable may share storage with a character variable of a different size or with a character aggregate.

The ADDR Built-in Function

PL/I's ADDR built-in function returns the location of the storage for any variable. Hence, you can use ADDR, pointers and based variables to refer to storage belonging to any variable in your program. ADDR looks like this:

```
ADDR(variable-name);
```

It returns the location of the variable that you gave as an argument.

Rules for ADDR

1. Type of Pointer Returned by the Function

   If the variable's data type is byte-aligned, ADDR returns a byte pointer. If the variable's data type is word-aligned, ADDR returns a word pointer.

2. ADDR and bit variables

   ADDR cannot be used on bit variables or bit aggregates. There is one exception: Storage for bit variables allocated by an ALLOCATE statement is word-aligned, and ADDR used with these variables will return a word pointer.

3. ADDR and Unconnected Storage

   ADDR cannot return the location of an unconnected array.

If you take the location returned by ADDR, assign it to a pointer, and use that pointer with a based variable, the based variable will share storage with the variable whose address was returned by ADDR.

Example:

```
BASED_SHARE:
          PROCEDURE;

          DECLARE POINT POINTER;
          DECLARE BWORD CHARACTER(3) BASED;

          DECLARE WORD CHARACTER(3);

          WORD = "YES";

          POINT = ADDR(WORD);
          POINT -> BWORD = "NO";

          PUT LIST(WORD);
END; /* BASED_SHARE */
```

Output: NO

BASED_SHARE uses ADDR to assign the address of WORD to the pointer POINT. When POINT is used with the based variable BWORD, BWORD shares storage with WORD. As the output shows, POINT - > BWORD and WORD have the same value.

The NULL Built-in Function

The NULL built-in function returns a pointer value which points to no location in storage. The function has the form:

NULL *[()]*

As you can see, it requires no arguments.

You can use NULL to mark unused pointers. For example, you might assign NULL to a pointer after you have freed its storage and then check whether it was set to NULL before you allocated new storage for it:

```
DECLARE 1 RESERVATION BASED,
              2 PASS_NAME CHARACTER(20) VARYING,
              2 FLIGHT FIXED BINARY,
              2 SEAT_NO FIXED BINARY;

DECLARE RES_POINTER POINTER;
              .
              .
FREE RES_POINTER -> RESERVATION;
RES_POINTER = NULL;
              .
              .
IF RES_POINTER = NULL
              THEN
              ALLOCATE RESERVATION SET(RES_POINTER);
              .
              .
```

WARNING:

Your program will not run correctly if you use a pointer which has been assigned NULL's value in a pointer-qualified reference.

Using Several Pointers to Refer to the Same Storage

Since pointers work like any other variable, you may assign the same location to several pointers. If you use a given based variable with any of these pointers, it will reference the same area of memory. One way you can take advantage of this fact is to construct arrays of pointers which let you access a single array of data in different ways.

For example, suppose we wanted to keep an array of names in the order in which they were input and at the same time be able to reference them in alphabetical order. We could solve the problem by setting up two arrays, one sorted and one unsorted, but this requires considerable storage space. A more economical way to solve the problem is to set up an array of pointers to the elements of the array. We can change the order of the pointers in the array of pointers and can thus set it up so that it will reference the original array in alphabetical order.

The program ALPHA_POINT shows how you can construct a subroutine to set up an array of pointers to reference another array in alphabetical order. The procedure has two parameters: a character varying array NAMELIST and an array of pointers ALPHA_PTRS. It leaves the character varying array unchanged and sorts the array of pointers so that they refer to the character varying array in alphabetical order.

In order to sort the pointers, we need a based variable to refer to the elements of NAMELIST and a pointer variable to hold a pointer when the sort swaps values. The based variable is NAME; it is equivalent to a single element of NAME_LIST. The pointer variable is DUM_PTR. The two variables I and SW, finally, control the sort.

The first thing the program does is assign the addresses of the elements of NAME_LIST to the elements of the array ALPHA_PTRS. Then it sorts the pointers according to the values of the variables whose location they contain. As you can see, the sort works exactly the same way as a sort involving ordinary variables:

```
ALPHA_POINT:
            PROCEDURE(NAME_LIST,ALPHA_PTRS);

            DECLARE NAME_LIST(*) CHARACTER(20) VARYING;

            DECLARE ALPHA_PTRS(*) POINTER;
            DECLARE NAME CHARACTER(20) VARYING BASED;
            DECLARE DUM_PTR POINTER;

            DECLARE I FIXED BINARY;
            DECLARE SW BIT(1);

/* ASSIGN THE LOCATION OF EACH ELEMENT IN NAME_LIST TO */
/* A POINTER IN ALPHA_PTRS */

            DO I = 1 TO HBOUND(NAME_LIST,1);
                        ALPHA_PTRS(I) = ADDR(NAME_LIST(I));
            END;

/* SORT THE POINTERS SO THAT THEY REFER TO THE ELEMENTS IN */
/* ALPHABETICAL ORDER */

            SW = "1"B;
            DO WHILE(SW);
                        SW = "0"B;
                        DO I = 1 TO HBOUND(NAME_LIST,1) -1;
                                IF ALPHA_PTRS(I) -> NAME < ALPHA_PTRS(I + 1) -> NAME
                                            THEN
                                            DO;
                                            DUM_PTR = ALPHA_PTRS(I + 1);
                                            ALPHA_PTRS(I + 1) = ALPHA_PTRS(I);
                                            ALPHA_PTRS(I) = DUM_PTR;
                                            SW = "1"B;
                                            END;
                        END;
            END; /*     OUTER SO  RT LOOP    */
END; /* ALPHA_POINT */
```

Pointer-qualified Reference

PL/I has two forms of pointer-qualified reference: explicit pointer-qualified reference, in which the reference contains both the pointer and the based variable name, and implicit pointer-qualified reference, in which the reference contains only the based variable name.

Explicit Pointer-qualified Reference

Explicit pointer-qualified refernces have the form:

```
pointer-exp -> based-variable
```

Since you can put pointer expressions as well as pointer variables to the left of the - > , you can write pointer- qualified references like this:

```
DECLARE VAL FIXED BINARY;
DECLARE INTEGER FIXED BINARY BASED;

ADDR(VAL) -> INTEGER = 7;
```

When the computer executes this assignment statement, it executes the ADDR built-in function to find the address of VAL and uses this address as INTEGER's pointer. The effect, of course, is to give INTEGER and VAL the same storage.

Implicit Pointer-qualified Reference

When you declare your based variable with the

```
BASED(pointer-exp)
```

keyword, you can refer to the based variable without giving the pointer expression. PL/I interprets each reference to the variable as a pointer-qualified reference using the pointer expression given in the variable's declaration. Note that you must still declare the pointer variable which appears with the BASED variable. If you allocate storage for the based variable with ALLOCATE, you must give the pointer in the ALLOCATE statement. Of course, the rules for implicit pointer-qualified reference are the same as those for explicit pointer-qualified reference.

BASED_VAR set up for implicit pointer-qualified reference would look like this:

```
BASED_VAR:
            PROCEDURE;

            DECLARE ARROW POINTER;
            DECLARE INT_VAL FIXED BINARY
                                  BASED(ARROW);

            ALLOCATE INT_VAL SET(ARROW);

            INT_VAL = 7777;
            PUT LIST(INT_VAL);

END; /* BASED_VAR */
```

As you can see, you still have to give the pointer in the ALLOCATE statement, but otherwise the computer interprets references to INT_VAL as references to ARROW - > INT_VAL.

Rules for Pointer-qualified Reference

When you write pointer-qualified references in your programs, you must make sure of two things: that the pointer in the reference is the proper type for the based variable, and that the current value of the based variable's extent expressions corresponds to the size of the storage you are working with. The following explains why.

Based Variable Types and Other Types

As mentioned in the discussion of pointers, AOS PL/I has byte pointers and word pointers. When it interprets pointer-qualified references, AOS PL/I assumes that word aligned based variables will have word pointers and that byte aligned based variables will have byte pointers. If you use a word pointer where PL/I expects a byte pointer or vice-versa, the reference will be invalid and your program will be in error. Neither the compiler nor the run-time routines will detect this kind of error in a pointer-qualified reference.

The following table gives the kind of pointer required for based variables of the various data types:

Pointer Type	**Data Type**
word	bit aligned bit fixed binary float binary aligned character character varying pointer file entry label
byte	character picture fixed decimal

An example of the kind of error discussed above is the following:

```
DECLARE LETTER CHAR(1);
DECLARE BYTE_STR BIT(8) BASED;
DECLARE P POINTER;
.
P = ADDR(LETTER);
P -> BYTE_STR = P -> BYTE_STR ! "00010000"B;
```

Since ADDR(LETTER) returns a byte pointer, while pointer-qualified references to bit variables require word pointers, the computer will not be able to interpret P - > BYTE_STR properly.

Extent Expressions and Pointer-qualified Reference

With all storage classes other than based, PL/I determines the variable's size when it allocates storage for the variable. All further references to the variable use the size it had when it was allocated. With based variables, PL/I evaluates the variable's extent expressions each time you refer to it in the program. If its extent expressions do not correspond to the size of the storage you wish to access, your program will be in error.

PL/I cannot detect such errors, and they can have serious consequences. For example, if you use the pointer-qualified reference in an expression, the expression may not have the proper value. If you assign a value to a pointer-qualified reference whose extent expressions are the wrong size, you may destroy the contents of adjacent storage.

For instance:

```
DECLARE PTR POINTER;
DECLARE CHAR_STRING CHARACTER(N) BASED;
DECLARE N FIXED BINARY;
.
N = 2;
ALLOCATE CHAR_STRING SET(PTR);
PTR -> CHAR_STRING = "AB";
.
N = 10;
PTR -> CHAR_STRING = "01234";
.
```

When the program allocates storage for CHAR_STRING, the value of N is 2, so CHAR_STRING is 2 bytes long. The string "AB" just fits. If PTR keeps the location the ALLOCATE statement set it to, the increase of N to 10 and the assignment of a character string to the variable will change the value of whatever was stored in the 8 bytes adjacent to the 2 bytes originally allocated for CHAR_STRING.

Fixing the Size of a Based Structure: the REFER Option

The REFER option lets you set up based structures in which one member of the structure determines the extent of other members. When you allocate storage for a based structure with the REFER option, PL/I uses the current value of the extent expression to determine the size of the allocation. Then it assigns the value of the extent expression to the structure member specified in the REFER option. In all future references to the variable, the value of the structure member, and not the current value of the extent expression, will determine the extents of the baserd structure.

The REFER option is part of the extent expressions of a member of a based structure. It takes the form:

```
fixed-bin-exp REFER(structure-member)
```

The fixed binary expresion is the value which determines the member's extents when storage for the structure is allocated. The structure member is the member which PL/I will use to determine the extents for all future references. The structure member must be declared in the structure before the reference to it in the REFER option occurs, it must have the fixed binary data type, and it may not be an element of an array.

For example, if you wanted to store character strings so that the storage for each string contained only the number of bytes needed to hold the characters in that string, you could set up a based structure which looked like this:

```
DECLARE L FIXED BINARY;

DECLARE 1 REF_STRUC BASED,
             2 LENGTH FIXED BINARY,
             2 STRING CHARACTER(L REFER
(REF_STRUC.LENGTH));
```

When an ALLOCATE statement for this structure is executed, the current value of the variable L will automatically be assigned to REF_STRUC.LENGTH. When you reference that allocation of REF_STRUC later on, the value of REF_STRUC.LENGTH, and not the current value of L, will determine how much storage the variable STRING refers to:

```
L = 10;
ALLOCATE REF_STRUC SET(PTR);

L = 20;

PTR -> REF_STRUC.STRING = "0123456789";
```

Even though N is given the value 20 before the reference to PTR - > REF_STRUC.STRING, REF_STRUC.LENGTH still has the value 10, and it is this value that determines the length of REF_STRUC.STRING. In consequence, the string "0123456789" will fit exactly and the assignment will not disturb adjacent storage, as it would have if REF_STRUC.STRING had had the new value of L for its length. For another example of the use of the REFER option, see the next program.

Cautions on the Use of Based Variables

Based variables and pointers are a powerful tool, but because they give you direct access to memory, they can be dangerous. When you use pointers and based variables, make sure that the based variable is of the proper size and data type and that the pointer has the right value. The following are some of the more frequent errors involving pointers:

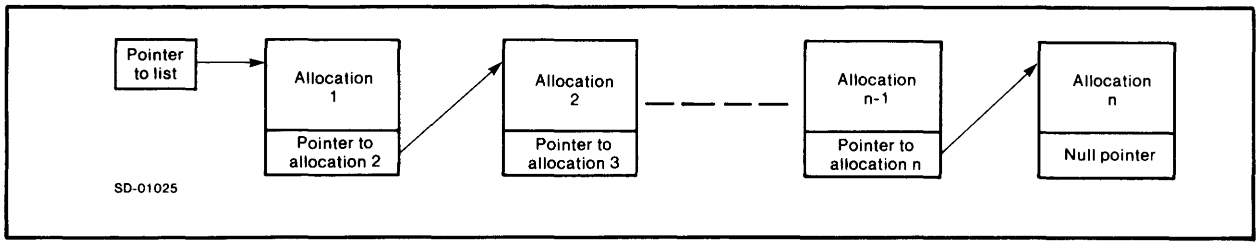
1. Using a pointer with a null value in a reference.
2. Using a pointer belonging to a based variable whose storage has been freed.
3. Using a pointer whose value was lost when the activation of the block to which it belonged ceased.

You can see from the list of errors that you must always take the scopes of the variables into account when you make a pointer-qualified reference. This is particularly the case with storage allocated with ALLOCATE: though such storage remains allocated until it is freed or until the program ends, the based variables and pointers referring to the allocation only remain valid as long as the block in which they were declared remains active. If the block activation ceases, you cannot access the storage.

An Example: Using Based Variables to Make a Linked List

A linked list is a list in which each allocation of storage for an item of the list contains a pointer to the storage for the next item. Linked lists allow you to allocate only as much storage as you actually need for the list at that execution of the program. If you make each element of the list a based structure, you can tailor the elements with the REFER option so that they are exactly as large as required to hold the data.

A schematic of a linked list looks like this:



Each element of the list contains a variable to hold the data and a pointer to the following element in the list. Since the last element has no following element, its pointer will have the value NULL.

When the program makes an element of the list, it has to do 4 things:

1. Allocate storage for the element.
2. Assign the data to the element.
3. Set the pointer to the next element to null.
4. Retrieve the preceding element and assign the location of the current element to its pointer.

The only complicated part is step 4. Since the program allocates storage for the elements one element at a time, we have to keep a pointer which refers to the preceding element while we allocate storage for the new element. We can then use this pointer to retrieve the preceding element and set its pointer to the location of the current element.

Once the list is made, we can retrieve it simply by using the pointer pointing to the whole list to get the first element and then using the pointer in each element in turn to retrieve the next element. We will know that we have reached the end of the list when an element has a pointer with the value NULL.

The program LINKED_LIST takes names and adds them to the linked list until you type "STOP". Then it retrieves the elements one at a time and writes them to your terminal:

```
LINKED_LIST:
   PROCEDURE;

   /* INNAME HOLDS THE NAME WHEN IT IS INPUT; L IS
      LENGTH(INNAME).  IT WILL BE USED IN THE REFER OPTION.    */

   DECLARE INNAME CHARACTER(60) VARYING;
   DECLARE L FIXED BINARY;

   /************************************************************/
   /*  NAME IS THE BASED STRUCTURE WITH THE REFER OPTION       */
   /*  WHICH WILL MAKE UP THE LIST. EACH ALLOCATION OF THE     */
   /*  STRUCTURE WILL CONTAIN THE NAME IN NME, THE VALUE       */
   /*  OF THE LENGTH OF NME IN LEN, AND A POINTER TO THE       */
   /*  NEXT ALLOCATION OF NAME IN NEXTPTR.                     */
   /************************************************************/

   DECLARE 1 NAME BASED,
             2 LEN FIXED BINARY,
             2 NME CHARACTER(L REFER(NAME.LEN)),
             2 NEXTPTR POINTER;
```

Figure 10-1. Using Based Variables to Make a Linked List

```
/*****************************************************************/
/*  POINTERS FOR THE LINKED LIST.  LISTPTR WILL HOLD THE        */
/*  POINTER TO THE FIRST ALLOCATION OF THE LIST; PTR WILL       */
/*  HOLD THE LOCATION OF THE CURRENT ALLOCATION; LASTPTR        */
/*  WILL HOLD THE LOCATION OF THE PRECEDING ALLOCATION          */
/*  UNTIL THE PROGRAM CAN ASSIGN THE LOCATION OF THE            */
/*  CURRENT ALLOCATION TO THE POINTER CONTAINED IN THE          */
/*  PRECEDING ALLOCATION.                                       */
/*****************************************************************/

DECLARE(LISTPTR,LASTPTR,PTR) POINTER;

DECLARE(IN,OUT) FILE;

OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

INNAME = "";
LISTPTR = NULL;

DO WHILE (INNAME ↑= "STOP");
PUT FILE(OUT) LIST("INPUT A NAME; INPUT STOP TO STOP");
GET FILE(IN) LIST(INNAME);
IF INNAME = "STOP" THEN

    /*****************************************************/
    /*  RETRIEVE THE LIST ONE ALLOCATION AT A TIME       */
    /*****************************************************/
   DO;
     IF LISTPTR = NULL THEN

      /* SPECIAL CASE: NO ALLOCATIONS OF NAME            */

        PUT FILE(OUT) LIST("NO LIST");
     ELSE

          /**********************************************/
          /*  GET THE LIST. SET PTR TO LISTPTR TO GET  **/
          /*  THE FIRST ALLOCATION AND THEN GET EACH    */
          /*  IN TURN UNTIL  NAME.NEXTPTR = NULL.       */
          /**********************************************/

        DO;
        PTR = LISTPTR;
           DO WHILE (PTR ↑= NULL);
           PUT FILE(OUT) LIST(PTR ->NAME.NME);
           PTR = PTR -> NAME.NEXTPTR;
          END; /* RETRIEVAL OF LIST */
        END;
    END;
ELSE

    /*****************************************************/
    /*  SET UP AN ALLOCATION OF NAME TO HOLD INNAME      */
    /*****************************************************/

    DO;
    L = LENGTH(INNAME);
    ALLOCATE NAME SET(PTR);
    IF LISTPTR = NULL THEN

        /* SPECIAL ACTION FOR THE FIRST ALLOCATION: ASSIGN
           PTR TO LISTPTR AND SET LASTPTR TO LISTPTR      */
```

Figure 10-1. Using Based Variables to Make a Linked List (continued)

```
            DO;
            LISTPTR = PTR;
            LASTPTR = LISTPTR;
            END;
         ELSE

             /*  SET NAME.NEXTPTR OF THE LAST ALLOCATION TO THE
                 VALUE OF THE CURRENT ALLOCATION'S POINTER      */

            LASTPTR -> NAME.NEXTPTR = PTR;

         /*****************************************************/
         /*  INPUT DATA TO THE CURRENT ALLOCATION OF NAME    */
         /*****************************************************/

         PTR -> NAME.NME = INNAME;
         PTR-> NAME.NEXTPTR = NULL;

         /* ASSIGN LASTPTR THE LOCATION OF THIS ALLOCATION      */

         LASTPTR = PTR;
         END; /* ALLOCATION */

      END; /* MAIN DO-WHILE LOOP */
   END; /* LINKED_LIST */
```

Figure 10-1. Using Based Variables to Make a Linked List (continued)

Defined Variables

Variables with the defined storage class have no storage of their own. Instead, they share storage with another variable. The keyword for the defined storage class is:

DEFINED(variable)

The variable name in parentheses is the variable that shares storage with the defined variable. As with other instances of storage sharing, any change in the value of either the defined variable or the variable it is defined onto changes the value of the other variable as well. For example:

```
DEFINER:
        PROCEDURE;

        DECLARE VAL_1 FIXED BINARY;
        DECLARE VAL_2 FIXED BINARY DEFINED (VAL_1);

        VAL_1 = 0;
        PUT LIST(VAL_2);
        VAL_2 = 5;
        PUT LIST(VAL_1);

END; /* DEFINER */
```

```
Output:            0        5
```

VAL_2 is defined onto VAL_1. As the output demonstrates, VAL_1 and VAL_2 share the same generation of storage. If the value of one of the variables changes, the value of the other changes also.

Rules for Defined Variables

1. You cannot define a defined variable onto another defined variable.
2. You can define a defined variable onto an aligned bit variable, but not onto a bit variable.
3. You cannot define a variable onto a variable with unconnected storage.
4. Defined variables follow the same rules for storage sharing as based variables; for details, see below.
5. Extent expressions with defined variables work like extent expressions with automatic variables: they are evaluated when the block in which the defined variable is declared is activated. Therefore, changes in the values of the extent expressions within the block do not affect the extents of the defined variable.

Storage Sharing in PL/I

Storage sharing occurs in PL/I when parameters share storage with their arguments, when based variables have pointers giving the locations of storage belonging to other variables, and when you define a variable onto another variable. The data type of a variable determines which other variables it may share storage with. For the rules governing storage sharing between arguments and parameters, see Chapter 6; for those governing storage sharing with defined and based variables, see below.

In the following discussion, we will first give the rules for storage sharing in standard PL/I and then the extensions of these rules in AOS PL/I. Programs that follow standard PL/I's rules for storage sharing are transportable; you will be able to compile them on any standard PL/I compiler. Programs that use AOS PL/I's extensions, on the other hand, may not generate correct code when compiled on other PL/I compilers.

Storage Sharing in Standard PL/I

With two exceptions, standard PL/I allows a based or defined variable to share storage with another variable only if the based or defined variable is equivalent to the elementary variable, array element, array, or structure member it is sharing storage with.

An elementary variable is equivalent to another elementary variable, an array element, or a structure member if both variables have the same attributes. Arithmetic variables must have the same data type and the same computational precision and scale. String variables must have the same data type and the same extents. Picture variables must have identical pictures.

Arrays are equivalent if they have equivalent elements, the same number of dimensions, and the same extents in each dimension.

Structures are equivalent if they have equivalent members in corresponding positions.

The exceptions:

Character variables may share storage with other character variables and with picture variables even if the two variables have different extents. The variable declarations cannot contain the ALIGNED or VARYING attributes. The same is true of bit variables that share storage with other bit variables.

Storage may be shared by two structures of different sizes if they have equivalent members in corresponding positions up to the end of the smaller structure, and the next member is the larger structure is a level 2 item.

Examples for Standard PL/I Storage Sharing

The following combinations of declarations, assignments, and references are legal:

```
DECLARE FIXED_1 FIXED DECIMAL(4,2);
DECLARE FIXED_2 FIXED DECIMAL(4,2) DEFINED(FIXED_1);

DECLARE INTEGERS(10) FIXED BINARY;
DECLARE INT FIXED BINARY DEFINED(INTEGER(1);

DECLARE 1 PERSON,
              2 NAME CHARACTER(30) VARYING
              2 STREET CHARACTER(40) VARYING,
              2 CITY CHARACTER(30) VARYING,
              2 STATE CHARACTER(2);

DECLARE P POINTER;
DECLARE NME CHARACTER(30) VARYING BASED;
.
P = ADDR(PERSON.NAME);
PUT LIST(P -> NME);

DECLARE WORD CHARACTER(5);
DECLARE WORD_LETS(5) CHARACTER(1) DEFINED(WORD);

DECLARE PIC_VAL PICTURE "$$,$$$,$$9V.99";
DECLARE PIC_CHAR CHARACTER(13) DEFINED(PIC_VAL);
```

Rules for Storage Sharing in AOS PL/I

Two variables may share storage in AOS PL/I if they have the same alignment. This rule is a consequence of the fact that you cannot use a byte pointer to reference a word-aligned variable and vice-versa. Additionally, the program must be compiled without optimization.

Bit variables are a special case. Bit variables with the based and defined storage classes are word- aligned and take word pointers; consequently, they can share storage with any word-aligned variable. Bit variables with other storage classes are bit-aligned. There are no bit pointers, and consequently you can neither define another variable onto such a variable nor find its location with ADDR.

Examples of storage sharing in AOS PL/I

```
DECLARE FLECHE POINTER;
DECLARE FL_INT FIXED BINARY DEFINED(FLECHE);

DECLARE LETTER ALIGNED CHARACTER;
DECLARE INT_STR ALIGNED BIT(8) BASED;
DECLARE P POINTER;

P = ADDR(LETTER);
P -> INT_STR = P -> INT_STR ! "00010000"B;

DECLARE PT POINTER;
DECLARE PT_STR BIT(16) BASED;
DECLARE P POINTER;

P = ADDR(PT);
PUT LIST(P -> PT_STR);

DECLARE PTR POINTER;
DECLARE PTR_STR DEFINED(PTR);
```

One frequent use for AOS PL/I's special storage sharing is system calls to AOS. AOS system calls often take an argument of one data type in an accumulator and return a result of another data type. The simplest way to handle this in PL/I is to give the system call arguments that fixed binary data type and to define variables of the other data types onto the arguments. For an example of this technique, see Chapter 14.

End of Chapter

Chapter 11
The GOTO Statement and Label Data

Introduction

The GOTO statement causes unconditional branching. When your program reaches a GOTO statement, control is transferred to the statement whose label you used in the GOTO statement. The GOTO may transfer control to a statement in any active block. The program DIVIDE shows how the GOTO works. The first GOTO is in the THEN clause of an IF THEN statement; it transfers control to the statement labelled ERROR. This statement outputs an error message. The statement that follows, another GOTO, transfers control back up to the top of the program to get more input. When the program does succeed in making the division, a third GOTO transfers control around the error message to the END statement:

```
DIVIDE:
            PROCEDURE;

            DECLARE (VAL_1,VAL_2) FIXED DECIMAL(8,2);
            DECLARE (IN,OUT) FILE;

            OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

TOP:
            PUT FILE(OUT) LIST("INPUT THE DIVISOR");
            GET FILE(IN) LIST(VAL_1);

            IF VAL_1 = 0 THEN GOTO ERROR;

            PUT FILE(OUT) LIST("INPUT THE DIVIDEND");
            GET FILE(IN) LIST(VAL_2);

            PUT FILE(OUT) SKIP LIST("THE QUOTIENT OF "!!VAL_2!!
                                    "DIVIDED BY "11VAL_1!!" IS "!!
                                    VAL_2 / VAL_1);

            GOTO LAST;

ERROR:
            PUT FILE(OUT) SKIP LIST("YOU CAN'T DIVIDE BY 0. TRY AGAIN");
            GOTO TOP;

LAST:
END; /* DIVIDE */
```

Label Data

The labels that appear in GOTO statements are label data. Label data is data whose value is the location of a statement in a PL/I program. PL/I has label constants and label variables. You may write functions that return label values.

Label Constants

You declare a label constant when you write a label prefix ahead of a statement. All statements but PROCEDURE, ENTRY, FORMAT, and DECLARE statements may take up to 7 label prefixes. When you use any of the prefixes in a GOTO statement, control will be transferred to the statement to which the prefix belongs. For instance:

```
L1:
L2:L3: STOP;
GO TO L1;
GO TO L2;
GO TO L3;
```

Each GO TO in the above example will transfer control to the STOP statement.

The single label prefixes preceding PROCEDURE and ENTRY statements are entry constants, and those preceding FORMAT statements are format constants. You may use neither of these constants in a GOTO. DECLARE statements may not take label prefixes.

Label constants have internal scope. With all label constants but those on BEGIN statements, the scope of the constant is the block in which you wrote it and all the blocks contained in that block. With constants on BEGIN statement, the scope of the constant includes the block which immediately contains the BEGIN block. Hence, you can GOTO a BEGIN block from its containing block. The program LAB_SCOPE shows how scope works with label constants:

```
LAB_SCOPE:
          PROCEDURE;

          GOTO LAB;

LAB:
          GOTO BEGIN_LAB;

BEGIN_LAB;
          BEGIN;

                    GOTO LAB;

          LAB:
          STOP;

          END; /* BEGIN */

END; /* LAB_SCOPE */
```

This program has two label constants LAB. The scope of the first constant is the PROCEDURE block but not the BEGIN block, where LAB is redeclared. The scope of the second LAB is the BEGIN block. The scope of BEGIN_LAB is both the BEGIN block and the PROCEDURE block.

Arrays of Label Constants

A label prefix that is a label constant may have a single integer constant subscript. When you use subscripts with label prefixes, you are declaring an array of label constants. The lower bound of the array is the smallest subscript that appears with the label; the upper bound is the largest; the extent of the array is the number of integers, including the bounds, between the lower and upper bounds. The array will have one element for each integer in the extent even if there is no label which corresponds to that element. It is an error to use a subscripted label constant in a GOTO which has no corresponding label in the program.

If you use subscripted labels in your program, you can make the transfer of control depend directly on the computed value of a subscript.

The following program gives a trivial example. It uses the LENGTH built-in function to measure the length of a character string that contains a number. The value of length then determines which statement the program will transfer control to. At that statement, the number in STRING is assigned to a fixed decimal variable which has the proper precision for it.

```
COMP_GOTO:
            PROCEDURE;

            DECLARE (IN,OUT) FILE;
            DECLARE STRING CHARACTER(5) VARYING;
            DECLARE DEC_1 FIXED DECIMAL(1);
            DECLARE DEC_2 FIXED DECIMAL (2);
            DECLARE DEC_3 FIXED DECIMAL(3);

            OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
            GET FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

            PUT FILE(OUT) LIST("INPUT A NUMBER");
            GET FILE(IN) LIST(STRING);

            IF LENGTH(STRING) > 3 THEN GOTO LAB(4);
            ELSE GOTO LAB(LENGTH(STRING));

LAB(1):
            DEC_1 = STRING;
            PUT FILE(OUT) LIST(DEC_1);
            GOTO EXIT;

LAB(2):
            DEC_2 = STRING;
            PUT FILE(OUT) LIST(DEC_1);
            GOTO EXIT;

LAB(3):
            DEC_3 = STRING;
            PUT FILE(OUT) LIST(DEC_3);
            GOTO EXIT;

LAB(4):
            PUT FILE(OUT) LIST("NUMBER TOO LONG",STRING);

EXIT:
END; /* COMP_GOTO */
```

Note that in AOS PL/I, the DO CASE statement is perhaps the best way to handle computed transfers of control.

Label Variables

Label variables are variables that may have label constants as values. You declare a label variable with the keyword

LABEL

Label variables are like other PL/I variables: they have internal scope unless you declare them with the STATIC EXTERNAL keywords. They may have any storage class. PL/I does not define conversions between label data and other data types. You can therefore only assign label constants to label variables and label variables to other label variables. The only operations you can perform with label constants and variables are = and ↑ =. Two label values are equal if they refer to the same statement in the same block activation.

Label variables are frequently used as parameters for label arguments. In the following example, the label constant argument ERROR and the label parameter ERR_LAB allow a transfer of control from the invoked procedure to the error message in the invoking procedure:

```
ERR_TRANS:
            PROCEDURE;

            DECLARE DIV ENTRY(FIXED DECIMAL(8,2),FIXED DECIMAL(8,2),
                              FIXED DECIMAL(16,12),LABEL);
            DECLARE  (NUM_1,N  UM_2) FIXED DECIMAL(8,2);
            DECLARE  QUOTIENT  FIXED DECIMAL(16,12);
            DECLARE  (IN,OUT)  FILE;

            OPEN FILE(IN) STREAM INPUT TITLE("@INPUT");
            OPEN FILE(OUT) STREAM OUTPUT TITLE("@OUTPUT");

            PUT FILE(OUT) LIST("INPUT THE DIVISOR");
            GET FILE(IN) LIST(NUM_1);
            PUT FILE(OUT) LIST("INPUT THE DIVIDEND");
            GET FILE(IN) LIST(NUM_2);

            CALL DIV(NUM_1,NUM_2,QUOTIENT,ERROR);

            PUT FILE(OUT) LIST("THE QUOTIENT IS," QUOTIENT);
            STOP;
ERROR:
            PUT FILE(OUT) LIST("YOU CAN'T DIVIDE BY 0");

END; /* ERR_TRANS */

/* DIV DOES THE DIVISION */

DIV:
            PROCEDURE(NUM_1,NUM_2,QUOT,ERR_LAB);

            DECLARE(NUM_1,NUM_2) FIXED DECIMAL(8,2);
            DECLARE QUOT FIXED DECIMAL(16,12);
            DECLARE ERR_LAB LABEL;

            IF NUM_1 = 0 THEN GOTO ERR_LAB;
            QUOT = NUM_2 / NUM_1;
END;
```

The GOTO Statement

The GOTO statement has the syntax:

```
GOTO label-exp;
          or
GO TO label-exp;
```

When the program executes the GOTO statement, it goes to the statement which has the label constant specified in the label expression. If the GOTO transfers control to a statement within its own block, it is called a local GOTO. If it transfers control to another active block, it is a non-local GOTO. Note that a GOTO within a BEGIN block that transfers control to the BEGIN statement is a non-local GOTO. When a non-local GOTO transfers control to another active block, PL/I ends the activation of the block containing the GOTO and of all the predecesors between it and the block containing the statement to which control is transferred. The following gives a schematic example:

```
OUTER:
          PROCEDURE;

                    BEGIN;

                              CALL INNER;

                    END;

          FINISH:
                              STOP;

INNER:
          PROCEDURE;

          GOTO FINISH;

          END; /* INNER */
END; /* OUTER */
```

When PL/I executes the GOTO FINISH statement in the procedure INNER, it ends the activation of both INNER and the BEGIN block from which INNER was invoked and transfers control to the STOP statement in OUTER.

Note that if you use the GOTO to exit from a procedure block invoked as a function, PL/I will not be able to finish evaluating the function. If you need the function's value to continue program execution, your program will be in error. The following is a trivial example of this kind of error:

```
BAD_GOTO:
            PROCEDURE;

            DECLARE (VAL,RES) FLOAT BINARY(53);
            DECLARE (VALUES,RESULTS) FILE;

            OPEN FILE(VALUES) STREAM INPUT;
            OPEN FILE(RESULTS) STREAM OUTPUT;

            GET FILE(VALUES) LIST(VAL);

            RES = ROOT(VAL);

NEXT:
            PUT FILE(RESULTS) LIST(RES);

/* ROOT FINDS THE SQUARE ROOT */

ROOT:
            PROCEDURE(NUM) RETURNS(FLOAT BINARY(53));

            DECLARE NUM FLOAT BINARY(53);

            IF NUM < 0 THEN GOTO NEXT;
            RETURN(SQRT(NUM));
END; /* ROOT */

END; /* BAD_GOTO */
```

If you give this program a negative value, the GOTO in ROOT transfers control to the PUT LIST statement. ROOT does not return a value to ROOT(VAL), no value can be assigned to RES, and the PUT LIST statement is meaningless.

Rules for the GOTO

1. The GOTO may not transfer control to the following statements:

   ```
   PROCEDURE
   ENTRY
   FORMAT
   DECLARE
   ```

   The label prefixes on PROCEDURE and ENTRY statements are entry constants, not label constants; the label prefixes on FORMAT statements are format constants. The DECLARE statement may not have a label prefix.

2. GOTO with statements in THEN, ELSE, and OTHERWISE clause statements that immediately follow the THEN, ELSE, and OTHERWISE clauses may not take label prefixes. Therefore, you cannot transfer control to them. Note, however, that if the clause is a non-iterative Do group or a DO CASE statement, you can transfer control to statements inside the group or DO CASE statement.

3. GOTO with DO Groups

You may GOTO the DO statement at the head of any kind of DO group. You may also GOTO any statement within a non-iterative DO group, but you may not GOTO a statement within an iterative DO group from outside the group.You may use the GOTO to exit from any DO group. The program LEGAL_GOTO demonstrates legal uses of the GOTO with DO groups:

```
LEGAL_GOTO:
                        PROCEDURE;

                        DECLARE I FIXED BINARY;

                        GOTO FIRST;

                        DO;
            FIRST:
            GOTO SECOND;
            END;

            SECOND:
            DO WHILE(I = 0);
                        THIRD:
                        I = I + 1;
                        PUT LIST(I);
                        IF I ↑= 5
                        THEN GOTO THIRD;
                        ELSE GOTO FOURTH;
                        END; /* DO WHILE */

            FOURTH:
            PUT LIST("ALL DONE");
LEGAL_GOTO /* END; */
```

Output:

```
1  2  3  4  5

ALL DONE
```

As the example shows, you can GOTO into a non-iterative DO group, but can only GOTO the label SECOND at the head of the DO WHILE group. Once you are in the group, you can GOTO statements in the group from other statements in the group and GOTO out of the group.

4. GOTO with DO CASE

You can GOTO the head of a DO CASE statement or to any of the cases. You may not GOTO the first statement following the OTHERWISE.

5. GOTO with Blocks

   A GOTO may only transfer control to an active block. It is an error to attempt to transfer control to an inactive block. The following gives a schematic example:

```
BAD_TRANS:
                        PROCEDURE;

                        DECLARE I FIXED BINARY;

                        IF I < 1 THEN GOTO ERMESS;

                        BEGIN;


              ERMESS:
              PUT LIST("VALUE OF I OUT OF BOUNDS");

              END;
BAD_TRANS /* END; */
```

   The GOTO ERMESS statement is illegal because the BEGIN block which contains the label is not active when the GOTO transfers control to it.

6. GOTO with the BEGIN Statement

   Strictly speaking, a BEGIN statement belongs to the block containing the BEGIN block the statement introduces and not to the BEGIN block itself. This has two consequences for GOTOs to BEGIN statements:

   a. A GOTO from the block containing the BEGIN block to the BEGIN statement is a local GOTO and not a non-local GOTO to an inactive block.

   b. A GOTO from inside the BEGIN block to the BEGIN statement is a non-local GOTO and terminates the BEGIN block's activation.

Compiler Detection of Errors with GOTO

Since you can use label variables and label functions in the GOTO statement, the compiler cannot check whether a program's GOTOs are legal. Programs that use illegal GOTOs will produce undefined results.

Proper Use of the GOTO in PL/I

PL/I's GOTO is powerful, but because the compiler cannot check its legality, it is also dangerous. In general, you should avoid it in your PL/I program and instead use DO groups, the DO CASE, procedure invocations, and the RETURN statement to transfer control. The compiler can check these constructs, and your program will be more readable.

There is, however, one place where the GOTO is the best solution in PL/I: namely, to transfer control out of an ON unit. ON units are BEGIN blocks which a program executes when an error condition arises. For details, see Chapter 13. For example, if you want to read a sequential file until it is empty and then transfer control to the part of the program which processes the data, you can do it like this:

```
FILE_READ:
          PROCEDURE;

          DECLARE CUSTOMERS FILE;
          DECLARE (I,N) FIXED BINARY;
          DECLARE 1 CUSTOMER(1000),
                    2 NAME CHARACTER(20) VARYING,
                    2 NUMBER CHARACTER(5) VARYING;

/* ON-UNIT TO TRANSFER CONTROL WHEN ENDFILE IS REACHED */

          ON ENDFILE(CUSTOMERS)
                    BEGIN;
                    GOTO START;
                    END;

/* READ THE FILE INTO THE ARRAY UNTIL ENDFILE IS RAISED */

          N = 1;
          DO WHILE ("1"B);
                    READ FILE(CUSTOMERS) INTO (CUSTOMER(I));
                    N = N + 1;
          END;

/* START OF PART OF PROGRAM WHICH PROCESSERS THE DATA */

START:
          DO I = I TO N;
                    .
                    .
```

End of Chapter

Chapter 12
Storage and Initialization of Static Variables

When a variable has the static storage class, PL/I sets up its storage at the beginning of program execution. The storage remains allocated until the program is finished. The fact that storage for static variables remains allocated throughout program execution has two important consequences: values assigned to static variables are not lost when an activation of the variable's block ceases, and you can give such variables initial values when you declare them.

The procedure MAX_CHECK demonstrates both properties. MAX_ CHECK keeps track of the largest positive value which has been passed to it as an argument. It does so by giving the static variable MAX the initial value of zero and then comparing the value of each argument with the current value of MAX. Since MAX has an initial value of zero, its value will remain zero until an argument with a value greater than zero is passed to the procedure. The procedure then assigns the argument's value to MAX and outputs the new maximum. As a static variable, MAX retains its value between invocations of the procedure; hence, each new argument will be compared with the value of the largest argument passed up to the current invocation. If the value of the new argument is larger, it is assigned to MAX. Otherwise, MAX retains its previous value.

```
MAX_CHECK:
      PROCEDURE(VAL);

      DECLARE VAL FIXED BINARY;
      DECLARE MAX FIXED BINARY STATIC INTERNAL INIT(0);

      IF VAL > MAX
            THEN
            DO;
            MAX = VAL;
            PUT LIST("NEW MAX =",MAX);
            END;

END; /* MAX_CHECK */
```

If the arguments for VAL have these execution time values:

```
-3,0,5,2,1,13,13,9,25,2,
```

then MAX_CHECK will output

```
NEW MAX = 5      NEW MAX = 13   NEW MAX = 25
```

The Scope of Static Variables

Variables with all storage classes but static have internal scope; you may give static variables either internal or external scope. You know from experience how variables with internal scope work: each time you redeclare a variable name in a different block, PL/I treats the new declaration as a new variable. The scope of a given declaration of a variable with internal scope is thus the block in which the variable is declared and all blocks contained in that block, unless you redeclare the variable in a contained block.

When a static variable has external scope, all of the declarations of the variable's name are declarations of the same variable. Static external variables thus work like file constants: if you declare the same static external variable in two different blocks, references to the variable in either block will refer to the same storage. For instance, the following two procedures, STAT_EXT_1, and STAT_EXT_2, both have the same static external variable WORD. Since WORD refers to the same storage in both procedures, we can assign it a value in STAT_EXT_2 and output the value in STAT_EXT_1:

```
STAT_EXT_1:
        PROCEDURE;

        DECLARE WORD CHARACTER(6) STATIC EXTERNAL;
        DECLARE STAT_EXT_2 ENTRY;

        CALL STAT_EXT_2;

        PUT LIST(WORD);
END;

STAT_EXT_2:
        PROCEDURE;

        DECLARE WORD CHARACTER(6) STATIC EXTERNAL;

        WORD = "STATIC";

END; /* STAT_EXT_2 */
```

Output:

```
STATIC
```

Of course, you can't use a name with external scope over again in a program unless you redeclare it with internal scope. The compiler cannot distinguish between two names with different data type attributes, so if you declare

```
DECLARE NUM FIXED BINARY STATIC EXTERNAL;
```

in one block and

```
DECLARE NUM FLOAT BINARY(53) STATIC EXTERNAL;
```

in another, the compiler will try to give both variables the same storage. Of course, if either declaration had not had the EXTERNAL keyword, it would have had internal scope, and PL/I would have treated it as a new declaration of the variable.

Declaring Static Variables

The preceding examples show how you declare static variables. If a static variable has external scope, you must use the

```
STATIC EXTERNAL
```

keywords. If it has internal scope, you use the

```
STATIC [INTERNAL]
```

keywords. As you can see, the default scope is internal. You cannot give a storage class to a part of an aggregate. When you declare a structure with the static storage class, the storage class keywords must follow the main structure name.

Extent Expressions with Static Variables

Because PL/I sets up the storage for static variables at the beginning of program execution, the extent expressions for static variables must be integer literal constants:

```
DECLARE WORD_ARRAY(100) CHARACTER(20) VARYING STATIC INTERNAL;
```

Initialization with Static Variables

You initialize a variable when you give it a value in the DECLARE statement. In AOS PL/I, you can only initialize variables with the static storage class. To do so, you use the INITIAL keyword in the variable's DECLARE statement. It looks like this:

```
INITIAL(constant-1[,constant-2, ... ,constant-n])
```

You use one constant for scalar variables and more than one for arrays. The constants are the values to be assigned to the variable. If the variable has an arithmetic data type, they will be arithmetic literal constants, if it has a bit-string data type, they will be bit-string literal constants, and so forth. You may also initialize pointer variables with the NULL built-in function. Here are some examples of initialization with scalar variables:

```
DECLARE LINE CHARACTER (80) VARYING STATIC INTERNAL INITIAL ("");

DECLARE SUM FIXED BINARY STATIC EXTERNAL INITIAL(0);

DECLARE PTR POINTER STATIC INITIAL(NULL);
```

Initializing Arrays

When you use INITIAL with an array, you must give a value for every element of the array. PL/I assigns the values to the elements in row-major order (i.c., the rightmost subscript varies most rapidly and the leftmost least rapidly).

```
DECLARE PRESIDENTS(5) CHARACTER(20) VARYING STATIC EXTERNAL
INITIAL("CARTER","FORD","NIXON","JOHNSON","KENNEDY");
```

In the above example, PRESIDENTS(1) would have the value "CARTER" and PRESIDENTS(5) would have the value "KENNEDY".

If you are initializing the array with arithmetic constants or the NULL built-in function, you can use a replication factor when you want to set several elements to the same value. A replication factor is a positive integer in parentheses preceding the value you want to assign to the elements. For example, if you wanted to set the first element of an array of 100 elements to 1 and all the rest to 0, you could do it like this:

```
DECLARE COLLECTION(10,10) FIXED BINARY STATIC EXTERNAL (1,(99) 0);
```

Initializing Structures

If a major structure has the static storage class, you can initialize the elementary variable and arrays it contains.

You can initialize elementary variable and array members of structures with the static storage class. To initialize a member, you simply give the INITIAL keyword with the initial value in the declaration of the member.

```
DECLARE 1 AGGREGATE STATIC INTERNAL,
        2 CASE_NAME CHARACTER(20),
        2 TABLE(5,5) FIXED BINARY INITIAL((25)0);
```

This declaration initializes all the elements of AGGREGATE.TABLE to 0.

End of Chapter

Chapter 13 Getting Into and Out of Blocks: Entry Data, the ENTRY Statement, Recursive Block Activation, and ON Units

Chapter 6 of PLAIN PL/I gives a general introduction to PL/I blocks, but it does not deal with some of the more sophisticated aspects of programming with blocks. In this chapter, we review entry constants and discuss entry variables, the ENTRY statement, recursive block activation, and ON units.

As their name implies, entry variables are variables that have entry constants as values; the ENTRY statement allows you to define secondary entry points into PROCEDURE blocks. Recursive block activation occurs when an already active block is activated again. On units are BEGIN blocks that define the actions a program is to take when certain errors occur.

Entry Data

Entry data is data whose values are the locations of PROCEDURE and ENTRY statements. PL/I has entry constants and entry variables, and you may write functions that return entry values. PL/I does not define conversions between entry data and other data types, so you cannot assign values of other data types to entry variables. The only operation you can perform with entry data are the comparison operations = and ↑ =. Two entry values are equal when they both refer to the same PROCEDURE or ENTRY statement in the same activation of that statement's block.

Entry Constants

You declare an entry constant when you write a label prefix ahead of a PROCEDURE or ENTRY statement. PL/I has two types of entry constants: internal entry constants and external entry constants.

An internal entry constant is a label prefix on a PROCEDURE or ENTRY statement whose procedure block is nested in another block. Internal entry constants have internal scope; hence, you can redeclare internal entry constant names in other blocks. The scope of the declaration of an internal entry constant is the block which immediately contains the PROCEDURE block to which the constant's PROCEDURE or ENTRY statement belongs, and all the blocks contained in that block, unless you redeclare the name in a contained block. You can use the entry constant to invoke the constant's procedure block anywhere within its scope.

For example:

```
SCOPES:
            PROCEDURE;

            .
            BEGIN;

            .
PROC_1:
            PROCEDURE;
            .
END; /* PROC_1 */

PROC_2:
            PROCEDURE;
            .
PROC_3:
            PROCEDURE;
            .
END; /* PROC_3 */
            .
END; /* PROC_2 */

            END; /* BEGIN-BLOCK :/
            .
END; /* SCOPES */
```

In this schematic example, the scope of the entry constants PROC_1 and PROC_2 is the BEGIN block and all the blocks contained in it. The scope of PROC_3 is the procedure PROC_2 and all the blocks contained in it. In consequence, none of the internal procedures may be invoked from a CALL statement or a function reference internal to SCOPES; the procedures PROC_1 and PROC_2 may be invoked from the BEGIN block, from each other, or from the procedure PROC_3. PROC_3 may only be invoked from PROC_2.

External Entry Constants

External entry constants are entry constants written on PROCEDURE or ENTRY statements belonging to external (non-nested) procedures. External entry constants have external scope; consequently, you can invoke an external procedure from any program location, as long as the location is within the scope of the original declaration of the external entry constant or of a DECLARE statement for the constant.

The DECLARE statement for an external entry constant has the form:

DECLARE name ENTRY *[(parameter-attribute-1, ... , parameter-attribute-n)][RETURNS(data-type-attribute)];*

The parameter attributes are attributes corresponding to the attributes belonging to the PROCEDURE or ENTRY statement's parameters. If the entry name is a function reference, the ENTRY declaration must include RETURNS with the data type of the returned value. When an entry's parameters include aggregates or entry values, the parameter attributes can become quite complicated. For details, see below.

The following program shows how external entry constants work:

```
OUTER_BLOCK:
            PROCEDURE;

INNER_1:
            BEGIN;

            DECLARE WRITE ENTRY(CHARACTER(*) VARYING);

            CALL WRITE("CALLED FROM INNER_1");
END; /* INNER_1 */

INNER_2:
            BEGIN;

            DECLARE WRITE ENTRY(CHARACTER(*) VARYING);

            CALL WRITE ("CALLED FROM INNER_2");
END; /* INNER_2 */

END; /* OUTER_BLOCK */

WRITE:
            PROCEDURE (STRING);

            DECLARE OUT FILE;
            DECLARE STRING CHARACTER (*) VARYING;

            OPEN FILE(OUT) STREAM OUTPUT TITLE ("@OUTPUT");
            PUT FILE(OUT) LIST(STRING);

END; /* WRITE */
```

Output:

```
"CALLED FROM INNER_1" "CALLED FROM INNER_2"
```

The entry constant WRITE belongs to an external procedure. It therefore has external scope and you can reference the procedure from any block which is within the scope of a DECLARE statement for WRITE.

Entry Variables

Entry variables are variables which have entry constants as values. A common use for entry variables is as parameters in procedures which take entry values as arguments. Like any other variables, entry variables may have any of the storage classes. If you do not declare them with the STATIC EXTERNAL keywords, they will have internal scope.

When you declare an entry variable, you use the VARIABLE ENTRY keywords. If the entry constants you assign to an entry variable have parameters, then the entry variable's declaration must include the parameter attributes. If the entry belongs to a procedure invoked as a function, the declaration must include RETURNS and the data type of the returned value:

DECLARE name VARIABLE ENTRY *[(parameter-attribute-1, parameter-attribute-n)][RETURNS(data-type-attribute)];*

The parameter attributes of the entry variable must match those of the entry constants you assign to it.

The following program shows how you can use entry variables as parameters. The program consists of two external procedures, CALLER, and INTEGRATE. CALLER contains the procedure SQUARE, which returns the square of its argument. INTEGRATE has an entry variable as a parameter. When CALLER invokes INTEGRATE, it uses the procedure named SQUARE as one of the arguments. Since the entry variable FUNC_N has the value SQUARE for this invocation of INTEGRATE, the function reference FUNC_N(L) works like the function reference SQUARE(L)and invokes SQUARE from INTEGRATE.

```
    CALLER:
         PROCEDURE;

         DECLARE INTEGRATE ENTRY(VARIABLE ENTRY(FLOAT)RETURNS(FLOAT),FLOAT,
                   FLOAT,FIXED) RETURNS(FLOAT);

         PUT LIST(INTEGRATE(SQUARE,0,10,100));


/* SQUARE RETURNS THE SQUARE OF ITS ARGUMENT                            */

      SQUARE:
         PROCEDURE(Y) RETURNS(FLOAT);

         DECLARE Y  FLOAT;

         RETURN( Y**2 );
      END; /* SQUARE */

      END; /* CALLER */

/*********************************************************************/
/*     INTEGRATE RETURNS THE INTEGRAL OF THE FUNCTION WHICH IS        */
/*     PASSED TO IT.  IT TAKES 4 ARGUMENTS: THE FUNCTION NAME,        */
/*     THE LOWER AND UPPER BOUNDS OF THE PORTION OF THE FUNCTION      */
/*     FOR WHICH THE INTEGER IS TO BE FOUND, AND THE NUMBER OF        */
/*     INTERVALS THE FUNCTION'S CURVE IS TO BE DIVIDED INTO.          */
/*********************************************************************/

      INTEGRATE:
         PROCEDURE(FUNC_N,LO,HI,NO) RETURNS(FLOAT);

         DECLARE FUNC_N VARIABLE ENTRY (FLOAT) RETURNS (FLOAT);
         DECLARE(L,LO,H,HI,A,AREA,INTERVAL) FLOAT;
         DECLARE (NO,I) FIXED BINARY;

         INTERVAL = (HI - LO) / NO;

         L = LO;
         AREA = 0;

         DO I = 1 TO NO;
         H = L + INTERVAL;
         A =ABS(((FUNC_N(L) + FUNC_N(H))/2) * INTERVAL);
         AREA = AREA + A;
         L = H;
         END;

         RETURN(AREA);

         END; /* INTEGRATE */
```

Output: 3.333452E+02

Figure 13-1. This Program Uses Entry Variables as Parameters

Examples of Parameter Attributes for Array, Structure and Entry Parameters

1. Array Parameter Attributes

   Parameter attributes for array parameters must contain the dimension attribute. For example, if the parameter were declared like this:

   ```
   DECLARE DATA_LIST ( * ) CHARACTER( * ) VARYING
   ```

   the attribute corresponding to that parameter in the DECLARE statement for the entry constant or variable would look like this:

   ```
   ,DIMENSION( * ) CHARACTER( * ) VARYING,
   ```

   Of course, you can also use the DIMENSION keyword:

   ```
   ,DIMENSION( * ) CHARACTER( * ) VARYING,
   ```

2. Structure Parameter Attributes

   Parameter attributes for structures must give the data types and level numbers of the structure members. If the parameter were declared like this:

   ```
   DECLARE 1 LIST_ELEMENT,
               2 PTR POINTER,
               2 NAME CHARACTER(10);
   ```

   the attribute in the DECLARE statement for the entry constant or variable would look like this:

   ```
   ..,1,2 POINTER,2 CHARACTER(10),...
   ```

   Note that since structures have no data type, the structure is represented in the attribute solely by its level number.

3. Parameter Attributes for Arrays of Structures

   The parameter attributes for arrays of structures are like those for structures, except that a dimension attribute follows the structure's level number. If you have a parameter like this:

   ```
   DECLARE 1 STRUC(100),
               2 NUM FIXED BINARY,
               2 TABLE(4,4) BIT(4);
   ```

   the parameter attribute in the DECLARE statement looks like this:

   ```
   ... 1(100),2 FIXED BINARY,2 (4,4) BIT(4),...
   ```

4. Parameter Attributes for Entry Data

   The parameter attribute for an entry variable parameter must include the entry variable's parameter attribute list. If you have a parameter like this:

   ```
   DECLARE FUNC VARIABLE ENTRY(FLOAT BINARY)
               RETURNS(FLOAT BINARY);
   ```

   the parameter attribute in the DECLARE statement looks like this:

   ```
   ...,VARIABLE ENTRY(FLOAT BINARY) RETURNS(FLOAT BINARY),...
   ```

The ENTRY Statement

The ENTRY statement lets you set up secondary entry points into procedure blocks. Like the PROCEDURE statement, the ENTRY statement must have a single label prefix and may have a parameter list. If the ENTRY statement belongs to a procedure that is invoked as a function, the statement must have the RETURNS keyword and the data type of the value it returns.

The label prefix on the ENTRY statement is an entry constant. When you use it to invoke the procedure to which the ENTRY statement belongs, control enters the procedure at the ENTRY statement and uses the parameters contained in it. If control has entered the procedure ahead of the ENTRY statement, it simply passes over the statement.

The following gives a simple example of how ENTRY statements work:

```
ENTREE:
            PROCEDURE;

            CALL ENTER_1;
            CALL ENTER_2;

ENTER_1:
            PROCEDURE;

            PUT LIST("ENTRY ENTER_1");

ENTER_2:
            ENTRY;

            PUT LIST ("ENTRY ENTER_2");

END; /* ENTER_1 */
END; /* ENTREE */
```

Output:

```
ENTRY ENTER_1        ENTRY ENTER_2        ENTRY ENTER_2
```

The procedure ENTER_1 has two entry points: the PROCEDURE statement and the ENTRY statement. The first invocation uses the entry constant ENTRY_1; as the output shows, all the PUT LIST statements in the program are executed. The second invocation uses ENTRY_2; since control enters the procedure after the first PUT LIST statement, it only executes the second PUT LIST.

Syntax of the ENTRY statement

The ENTRY statement has the form:

label-prefix:ENTRY *[(parameter-1, . . . ,parameter-n)] [RETURNS(data-type-attribute)];*

The label prefix, parameter list, and the RETURNS option work as they do for the PROCEDURE statement. If the PROCEDURE statement for the block to which the ENTRY statement belongs has the RECURSIVE keyword, you can invoke the ENTRY statement recursively.

Limitations on the ENTRY statement

1. The ENTRY statement must be internal to its PROCEDURE block.

2. You cannot use an ENTRY statement in a BEGIN block or in an iterative DO group.

An Example: the Procedure F_READ

F_READ is a procedure which reads a file. The first time it is invoked, it must open the file, so control enters at the PROCEDURE statement, which has as parameters a character varying variable for the AOS data set name and a structure for the data to be read from the file. Having entered at F_READ, control opens the file, reads the first 44 words of data into CLIENT, and returns.

The OPEN statement in F_READ used the file variable SOURCE as its file expression. Since the variable has the static storage class, it retains the location of the file control block between invocations of F_READ. Therefore, succeeding invocations need not re-open the file and can enter at NEXT_READ. Invocations of NEXT_READ require only a single argument: the structure into which the data is to be read. When control enters via NEXT_READ, it simply reads the file into CLIENT and returns.

```
F_READ:
            PROCEDURE(FILE_NAME,CLIENT);

            DECLARE FILE_NAME CHARACTER(32) VARYING;

            DECLARE SOURCE FILE VARIABLE STATIC INTERNAL;

            DECLARE 1 CLIENT,
                        2 NAME CHARACTER(20) VARYING,
                        2 STREET CHARACTER(30) VARYING,
                        2 CITY CHARACTER(20) VARYING,
                        2 STATE CHARACTER(2) VARYING,
                        2 ZIP CHARACTER(5) VARYING;

            OPEN FILE(SOURCE) RECORD SEQUENTIAL INPUT TITLE(FILE_NAME);

            READ FILE(SOURCE) INTO (CLIENT);
            RETURN;

NEXT_READ:
            ENTRY(CLIENT);

            READ FILE(SOURCE) INTO(CLIENT);
            RETURN;

END; /* F_READ */
```

The declarations for the external entry constants F_READ and NEXT_READ and the invocations of the procedure look like this:

```
DECLARE F_READ ENTRY(CHARACTER(32) VARYING,
          1,
                    2 CHARACTER(20) VARYING,
                    2 CHARACTER(30) VARYING,
                    2 CHARACTER(20) VARYING,
                    2 CHARACTER(2) VARYING,
                    2 CHARACTER(5) VARYING);

DECLARE NEXT_READ ENTRY
          (1,
                    2 CHARACTER(20) VARYING,
                    2 CHARACTER(30) VARYING,
                    2 CHARACTER(20) VARYING,
                    2 CHARACTER(2) VARYING,
                    2 CHARACTER(5) VARYING);
.
CALL F_READ("CLIENT_FILE",CL);

DO WHILE("1"B);
          CALL NEXT_READ(CL);
          PUT FILE(OUT) SKIP LIST(CL);
END;
```

Recursive Procedures

A recursive procedure is an already active procedure that a program invokes again. In PL/I, all recursive procedures must have the RECURSIVE keyword in their PROCEDURE statements. If the PROCEDURE statement has the RECURSIVE keyword, a program may use any entry point in the procedure for a recursive invocation.

Blocks in Recursive Invocation

When a program invokes an inactive block, it allocates storage for the block's automatic variables, gives the block's parameters the storage for the invocation's arguments, and if the procedure was invoked as a function, it sets up storage for the value returned by the function. When a program returns from an active block, it frees the storage for the automatic variables.

Exactly the same thing happens when a program invokes an already active block. The storage and the links between parameters and arguments set up for the first activation remain and new storage and new links are set up for the recursive activation. If the recursive activation leads to another recursive activation, the process is repeated. A recursive block can thus have many simultaneous activations. Control returns from a given activation of a recursive procedure the same way it returns from any other block activation: the storage for the block is released and control returns to the point in the block's predecessor where it was invoked or to the statement label contained in the non-local GOTO.

Because each activation of a recursive block has a separate allocation of storage for its automatic variables, the value of a variable in one activation may change without affecting the values of the variables in the other activations. Similarly, a reference to an automatic variable in a recursive procedure refers to the generation of storage allocated for the current activation of the procedure.

A program can pass values between successive recursive invocations of a block the same way that it passes them between a block and its successor. The parameters of the succeeding invocation will take their values from the arguments of the predecessor, and if the values were passed by reference, the values of the predecessor will change as the values in the successor change. If the recursive procedure is invoked as a function, each invocation will return a value to the function reference in its predecessor.

In the following example, each separate invocation of the recursive procedure FACT outputs values. The output thus shows the order in which the statements in FACT are executed and the value of the variable at each invocation.

Example:

```
RECURSIONS:
        PROCEDURE;

        DECLARE FACTORIAL FIXED BINARY;

        CALL FACT(3,FACTORIAL);
        PUT LIST(FACTORIAL);

FACT:
        PROCEDURE(INT,FCT) RECURSIVE;

        DECLARE (INT,FCT) FIXED BINARY;

        PUT LIST(INT);
        IF INT = 0 THEN FCT = 1;
        ELSE
                DO;
                CALL FACT(INT - 1,FCT);
                FCT = INT * FCT;
                PUT LIST(FCT);
                END;

        END; /* FACT */

END; /* RECURSIONS */
```

Output:

```
3       2       1       0       1
2       6       6
```

The example uses the recursive procedure FACT to calculate the factorial of 3 and return it to the invoking procedure RECURSIONS. The output from the PUT LIST statements in FACT shows how the recursive invocations work. On the first invocation of FACT, the value 3 is passed to INT. Since INT is greater than 0, control passes to the DO group. The first statement in the DO group, CALL FACT(INT-1)FCT), invokes FACT again. In this invocation, INT has the value 2, which is still greater than 0, so the CALL statement is executed again. The recursive invocations continue until INT = 0. This last invocation gives FCT the value 0. Control then terminates this invocation, returns to the next-to-the-last invocation, and executes the statements following the CALL statement with the values of the variables for that invocation.

Since FCT is a parameter, passed by reference, it will have the value it had in the last invocation, namely 1; as we know from the output, INT has the value 1 in this invocation, so FCT = FCT * INT gives FCT the value 1, which the PUT LIST outputs. When control reaches the END statement, it releases the next-to-last invocation's storage and then executes the statements following the CALL in the preceding invocation. This time, INT = 2 and FCT = 1, so FCT * INT assigns the value 2 to FCT. The PUT LIST outputs this value, and control releases this invocation's storage and returns to the first invocation of FACT. Here, INT = 3 and FCT = 2, so INT * FCT = 6, and this is the value that FCT's argument, FACTORIAL, has when control returns from the first invocation of FACT to RECURSIONS.

ON Units and Conditions

PL/I use ON statements and ON units to handle program interrupts. A program interrupt occurs when your program attempts to do something it cannot, such as read beyond the end of a file or convert the string ''ABC'' to a fixed decimal value. In PL/I terms, a program action that causes an interrupt raises a condition.

AOS PL/I recognizes three conditions: a general error condition, an endfile condition, and an endpage condition. The endfile condtion arises when the program attempts to read past the end of a file; and the endpage condtion arises when the program outputs more lines of data to a stream output print file than the file's page size allows for. A program's ON units tell PL/I what to do if any of the conditions arise during program execution. If there is no ON unit, PL/I takes a default action.

ON statements and ON units work like traps: just as you can set a trap at a certain time but can't predict when something will enter it, you can set an ON statement wherever you want in your program, and it won't be executed until the condition it was set for arises.

For instance:

```
ERR:
            PROCEDURE;

            DECLARE NUMBER FIXED BINARY;

            ON ERROR
                        BEGIN;
                        PUT LIST("I CAN'T DO THE CONVERSION");
                        STOP;
                        END;

            PUT LIST("FIRST STATEMENT EXECUTED");
            NUMBER = "ABC";
END; /* ERR */
```

Output:

```
FIRST STATEMENT EXECUTED            I CAN'T DO THE CONVERSION
```

The BEGIN block in this program is an ON-unit. The ON statement to which it belongs determines which kind of condition will cause the execution of the BEGIN block. If the condition arises in any statement following the ON statement and its BEGIN block, control will be transferred to the BEGIN block. In this program, the illegal assignment NUMBER = ''ABC''; will raise the error condition and transfer control to the BEGIN block of the ON unit. The program's output shows that it does in fact work that way.

In what follows, we will first discuss the general syntax of the ON statement and ON units, and then deal with the three conditions.

The ON Statement and the ON Unit

The ON statement defines both the condition that will activate the On unit and the ON unit itself. The ON statement is an executable statement; when the flow of control encounters it, it establishes the statement's ON unit for the part of the program following the statement. Henceforth, an error which raises the condition specified in the ON statement will transfer control to the statement's ON unit.

Syntax of the ON Statement and the ON Unit

In AOS PL/I, the ON unit must be a BEGIN block. Consequently, the ON statement has the form:

```
ON condition
          BEGIN;

          [statements]

          END;
```

1. The Condition

   The condition must be one of the three conditions allowed in AOS PL/I: ERROR, ENDFILE(file-expressions), and ENDPAGE (file-expression).

2. Rules for the ON Unit

   a. The BEGIN statement may not have a label. Hence, a local GOTO cannot transfer control to an ON unit.

   b. The BEGIN block may not have a RETURN or RETURNS(expression) statement internal to it.

   c. You cannot use the ON unit block to assign a value to a variable which is used in the statement that raised the condition. Neither can you close the file in the ON unit if the ON unit returns control to the I/O statement that raised the condition. For an example of how you can get around these restrictions, see the section on the error condition below.

   d. The meaning of declared names in the BEGIN block is defined by the location of the ON statement for the block, not by the location of the statement that causes the blocks's execution.

   For example:

```
ON_EX:
          PROCEDURE;

          DECLARE NUMBER FIXED BINARY;

          ON ERROR
                    BEGIN;
                    GOTO TRANS;
                    END;

          BEGIN;
          NUMBER = "ABC";
TRANS:
          PUT LIST("BEGIN BLOCK");
          END;
TRANS:
          PUT LIST("ON_EX");

END; /* ON_EX */
```

   Output:

```
ON_EX
```

   The ON unit in ON_EX is contained in the procedure, not in the BEGIN block. Consequently, the label constant TRANS refers to the label on the PUT LIST statement internal to ON_EX, not the one internal to the BEGIN block in which the program raises the condition.

Multiple ON-Units

You may write several ON statements and ON units for the same condition in your program. When the condition arises, the flow of control will transfer to the ON unit of the last ON statement that was executed.

What happens to the previous ON unit when a new On statement is e executed depends on whether the new ON statement is internal to the same block as the old one. If it is, the new ON unit replaces the old one. If the new ON statement is in a different block from the old one, information about the old ON statement and its ON unit is preserved in the storage allocated for the block. When control returns from the block containing the new ON unit, the old ON unit is restored.

The following program shows how this works:

```
ON_BLOCK:
            PROCEDURE;

            DECLARE BIT_STRING BIT(5);

            ON ERROR
                        BEGIN:
                        PUT LIST("ON UNIT 1");
                        GOTO FINISH;
                        END;

            BEGIN;

            ON ERROR
            BEGIN;
            PUT LIST("ON UNIT 2");
            GOTO LAST;
            END;

            BIT_STRING = "A";
LAST:
            END; /* BEGIN BLOCK */

            BIT_STRING = "B";
FINISH:
END; /* ON_BLOCK */
```

Output:

```
ON UNIT 2                ON UNIT 1
```

As the output shows, the second ON-unit is established only in the BEGIN block.

ON Units and Efficiency

You could conceivably write programs in which the actual data processing is done in an ON unit, but your programs will be more efficient if you keep your ON units as small as possible.

The SIGNAL and REVERT Statements

You use the SIGNAL statement when you want your program to execute an ON unit even though the condition which causes the execution of the unit has not arisen. SIGNAL has the form:

```
SIGNAL condition;
```

when control encounters the SIGNAL statement, it executes the current ON unit for the condition. If none is established, it executes the default action for the condition.

You use the REVERT statement to revoke the ON-unit established for the condition in the current block and re-instate the ON unit which preceded it. The syntax for REVERT is:

REVERT condition;

REVERT cannot revoke the default action for a condition. If your program does not have an ON unit established for the condition in the REVERT statement, the statement has no effect. Note also that REVERT cannot re-instate an ON unit which is internal to the same block as the one which was reverted.

Take as an example:

```
    SIGNAL_EX:
        PROCEDURE;

        DECLARE I FIXED BINARY STATIC INTERNAL INITIAL(0);

/*********** ON UNIT 1    *******************************************/

        ON ERROR
           BEGIN;
           PUT LIST ("ON UNIT 1");
             IF I = 0
                THEN
                DO;
                I = I + 1;
                GOTO INVOKE;
                END;

                ELSE
                STOP;
           END;  /* END ON UNIT 1 */

        SIGNAL ERROR;
    INVOKE:
        CALL SIG_PROC;

/******************************************************************/
/*        INTERNAL PROCEDURE SIG_PROC                             */
/******************************************************************/

    SIG_PROC:
        PROCEDURE;

    /*************** ON UNIT 2 ***************************************/

        ON ERROR
           BEGIN;
           PUT LIST ("ON UNIT 2");
           GOTO JUMP_1;
           END; /* ON UNIT 2 */

        SIGNAL ERROR;

    /***** ON UNIT 3      *******************************************/
```

Figure 13-2. SIGNAL_EX Illustrates the Use of ON Units

```
JUMP_1:
    ON ERROR
       BEGIN;
       PUT LIST("ON UNIT 3");
       GOTO JUMP_2;
       END; /* ON UNIT 3 */

    SIGNAL ERROR;
JUMP_2:
    REVERT ERROR;
    SIGNAL ERROR;
END; /* SIG_PROC */

END; /* SIGNAL_EX */
```

Output:

```
ON UNIT 1          ON UNIT 2          ON UNIT 3          ON UNIT 1
```

Figure 13-2. SIGNAL_EX Illustrates the Use of ON Units (continued)

The first thing SIGNAL_EX does is establish ON unit 1. The first SIGNAL ERROR statement transfers control to the ON unit, which in turn transfers control to the CALL SIG_PROC statement. SIG_PROC establishes a second ON unit. The new ON unit disestablishes ON unit 1, but since the first ON unit is in an active block, it is preserved. The second SIGNAL statement invokes the new ON unit, which transfers control to an ON statement which establishes a third ON unit. This ON unit is in the same block as the second ON unit, and therefore it replaces the second ON unit. The third signal statement invokes the new ON unit, which transfers control to the REVERT statement. The REVERT statement re-establishes the first ON unit, and the last SIGNAL statement transfers control to this ON unit, which stops the program on its second execution.

Conditions

In this section, we will discuss each of AOS PL/I's conditions in detail. We will explain the situations in which the condition may arise, PL/I's default action for the condition, and the manner in which ON units for the condition function. We will also introduce you to two built-in functions, ONCODE and ONFILE, which return information about conditions when they are raised.

The ERROR Condition

You specify the error condition in an ON statement with the ERROR keyword. PL/I's default action for most error conditions is to stop program execution, write an error message to the terminal from which the program is being run, close all open files, release the program's storage, and return to the operating system. The error messages that the default action returns to the terminal are in an appendix of the reference manual.

If your program has established an ON unit for the ERROR condition, PL/I will execute the ON unit. If a statement in the ON unit does not transfer control elsewhere, PL/I will attempt to return to the place where the error occurred when it finishes the ON unit. Such a return terminates program execution, so you must transfer control elsewhere if you want execution to continue.

For example, if you wanted a program to keep going even when it received a bad value for a mathematical function, you could write an ON unit like this:

```
                DECLARE NUMBER FLOAT BINARY(53);
                DECLARE ROOT FLOAT BINARY(53);
                DECLARE BAD_DATA BIT(1) STATIC INTERNAL INITIAL("0"B);
                .
                ON ERROR
                          BEGIN;
                          PUT LIST("BAD VALUE. NUMBER = ",NUMBER);
                          BAD_DATA = "1"B;
                          GOTO AGAIN;
                          END;
                .
AGAIN:
                IF BAD_DATA
                          THEN
                          DO;
                          NUMBER = 0;
                          BAD_DATA = "0"B;
                          END;

                ROOT = SQRT(NUMBER);
                .
```

When an input error occurs, the ON unit sets the bit variable BAD_DATA to "1"B and transfers control to an IF THEN statement whose THEN clause is executed only if the condition was raised. The DO group in the THEN clause assigns a legal value to NUMBER and resets BAD_DATA to "0"B. We have to use this indirect method because we cannot use an ON unit to assign a value to a variable used in the statement which raises the condition.

The following situations will raise the ERROR condition:

1. Subscripts out of range: If you compile with the /SUB switch, PL/I will raise the error condition when array subscripts or the arguments for SUBSTR are out of range.

2. DO CASE expressions out of range: If a DO CASE has no OTHERWISE clause and the value of the expression in the DO CASE is out of range, PL/I will raise the error condition.

3. I/O errors: The error condition arises when an error other than ENDFILE occurs during I/O or when your program has no ON unit for the ENDFILE condition for the file.

4. Conversion errors: Any conversion error, such as an illegal assignment or an attempt to output non-convertible data to a stream file, will raise the error condition.

5. Mathematical errors: Errors during the execution of the ** operator or the mathematical built-in functions raise the error condition.

6. Storage errors: If the program attempts to allocate more storage than is available, it will raise the error condition.

The ONCODE Built-in Function

ONCODE returns a fixed binary value which is the number of the error which raised the error condition. It has the form:

ONCODE[()]

As you can see, it takes no arguments. The numbers for the errors are listed in the appendix with the error messages.

You can use ONCODE to tailor the actions your ON units take for particular errors. For example, if you expected input errors in a program, you could do something like this:

```
          DECLARE VALUE FLOAT BINARY(53);
          .
          ON ERROR
                    BEGIN;
                    IF ONCODE = 28
                              THEN
                              DO;
                              PUT FILE(OUT) LIST("BAD DATA. CHECK YOUR INPUT"!!
                                                  "AND TRY AGAIN");
                              GOTO REPEAT;
                              END;

                              ELSE;
                              DO;
                              PUT FILE(OUT) LIST("ERROR NO", ONCODE);
                              STOP;
                              END;
                    END;
          .
 REPEAT:
          PUT FILE(OUT) LIST("INPUT A NUMBER");
          GET FILE(IN) LIST(VALUE);
```

28 is the error code for invalid conversions from character to arithmetic data, so the ON unit goes back for more data in this case, but otherwise outputs the error number and stops.

The ENDFILE Condition

PL/I raises the ENDFILE condition when a READ or GET statement attempts to read past the end of a file with sequential access or when a READ statement with a key has a key value greater than that of the last record in a file with direct access. Once a program has raised the endfile condition for a file, it cannot be read again until it has been closed. Repeated reads will only raise the endfile condition again.

The default action for the endfile condition is to raise the error condition. The ON statement for an ON unit for ENDFILE looks like this:

ON ENDFILE(file-expression)

After the program has executed the ON unit, it returns to the statement following the GET or READ statement which raised the condition, unless the ON unit transfers control elsewhere.

For instance,

```
DECLARE LINE CHARACTER(80) VARYING;
DECLARE TEXT FILE;

.
ON ENDFILE(TEXT)
        BEGIN;
        PUT LIST("NO MORE TEXT");
        STOP;
        END;

.
READ FILE(TEXT) INTO (LINE);
DO I = 1 TO LENGTH(LINE);
.
.
```

Without the STOP statement in the ON unit, control would go to the DO statement after the READ statement which caused the condition. This statement would use the preceding value of LINE.

The ONFILE Built-in Function

ONFILE returns a character string which is the name of the AOS data set the program was reading when the endfile or endpage condition arose. Like ONCODE it requires no arguments:

ONFILE[()]

You could use ONFILE with the procedure F_READ of the section on the ENTRY statement to return the name of the data set which the program was reading:

```
F_READ:
        PROCEDURE(FILE_NAME,CLIENT,OUT);
/* FILE_NAME IS THE AOS FILE NAME PASSED AS A PARAMETER */

DECLARE FILE_NAME CHARACTER(32) VARYING;

DECLARE SOURCE FILE VARIABLE STATIC INTERNAL;

/* OUT IS THE PARAMETER FOR THE TERMINAL FILE */

        DECLARE OUT FILE VARIABLE;

        DECLARE 1 CLIENT,
                2 NAME CHARACTER(20) VARYING,
                2 STREET CHARACTER(30) VARYING,
                2 CITY CHARACTER(20) VARYING,
                2 STATE CHARACTER(2) VARYING,
                2 ZIP CHARACTER(5) VARYING;

        OPEN FILE(SOURCE) RECORD SEQUENTIAL INPUT TITLE(FILE_NAME);

        ON ENDFILE(SOURCE)
                BEGIN;
                PUT FILE(OUT) LIST("NO DATA IN FILE "!!ONFILE());
                END;

        READ FILE(SOURCE) INTO (CLIENT);
        RETURN;
```

The ON unit here returns "NO DATA IN FILE" here because this is the first read; in the second half of F_READ, you would need another ON unit with a different error message.

The ENDPAGE Condition

Your program will raise the ENDPAGE condition if a PUT or a WRITE statement attempts to write beyond the end of a page of a stream output print file. If you do not have an ON unit for the endpage condition with a file, PL/I will simply output a page mark to the file, increase the current page number by 1, reset the current line and column numbers to 1, and continue outputting data.

The ON statement with ENDPAGE looks like this:

```
ON ENDPAGE(file-exp)
```

If the endpage condition arises for the file given in the file expression, PL/I transfers control to the ON unit. Unless a statement in the ON unit transfers control elsewhere, control will return from the ON unit to the statement which raised the condition and continue with the output.

When PL/I executes an ON unit for the ENDPAGE condition, it turns off the condition for the file. To reset the condition, you have to use a PUT PAGE statement or a PAGE control format item with PUT EDIT. For example, an ON unit which resets the condition, outputs the page number and a blank line, and then returns to the output statement looks like this:

```
DECLARE TEXT FILE;
DECLARE LINE CHARACTER(80) VARYING;

OPEN FILE(TEXT) STREAM OUTPUT PRINT LINESIZE(80)
                                        PAGESIZE(30);

ON ENDPAGE(TEXT)
        BEGIN;
        PUT FILE(TEXT) PAGE;
        PUT FILE(TEXT) EDIT("PAGE"!!PAGENO(TEXT),"")
                                        (COLUMN(65),A,SKIP(2),A);
        END;
```

The PAGENO(TEXT) is a built-in function which returns a fixed binary value which is the file's current page. You can also use PAGENO as a pseudo-variable: if you wanted the program fragment above to start numbering its pages with page 6, you could include a statement like this:

```
PAGENO(TEXT) = 6;
```

Of course, you can use PAGENO only with open stream output print files.

End of Chapter

Chapter 14
Calling AOS and Assembly-Language Routines from AOS PL/I

Making AOS System Calls

If you refer to the *AOS Programmer's Manual*, you will see that AOS system calls have three parts: the bit pattern, which specifies the system call to be made; arguments, which the call inputs directly to the computer's first three accumulators (accumulators 0 through 2); and a parameter packet. The parameter packet is a data aggregate that contains values passed to or from the system call. All AOS calls require the bit pattern; most receive values via the accumulators, and many requie the parameter packet. In calls requiring packets, accumulator 2 always receive the packet's address.

To make AOS system calls, you use the external procedure, SYS. SYS has four parameters: the bit-pattern for the AOS system call, and the values to be passed to accumulators 0, 1, and 2. You invoke SYS as a function; when the call works properly and has a normal return, SYS will have the value "0"B; if the call has an exceptional return, SYS will have the value "1"B. On exceptional returns, accumulator 0 will contain the error code for the cause of the exceptional return.

Since SYS is included in the PL/I library, you do not include it in the list of external procedures when you bind your program. You must, however, declare SYS as an external entry. The declaration of SYS is like that of any other external function: it contains the data types of the function's parameters, the RETURNS keyword, and the data type of the value the function returns. The first parameter is the bit-pattern for the system call; you must give it the aligned bit data type. The other parametersare for accumulators 0 through 2. These must be for data that is word-aligned and requires 1 word of storage. Therefore, you can use the fixed binary, aligned bit(16), or pointer data types. The function returns a BIT(1) VALUE. It is common practice to give the parameters for the accumulators the fixed binary data type. In this case, the declaration of SYS as an entry looks like this:

```
DECLARE SYS ENTRY(ALIGNED BIT(16),FIXED BINARY,FIXED BINARY,
FIXED BINARY) RETURNS(BIT(1));
```

When you invoke SYS, the first argument will be a bit-string constant that gives the proper bit-pattern for the system call; since the system call returns values to the accumulators, the arguments for the accumulators will generally be variables. The data types of the arguments must correspond to the data types of the parameters you gave SYS when you declared it as an entry. If you want to use SYS with arguments of different data types, the simplest way to do it is to define the variables for these arguments onto the variables you actually use as arguments.

A Simple Example: Calling ?DELETE to Delete a File

To show you how all this works in practice, let's take a look at a PL/I program that uses the AOS system call ?DELETE to delete a file.

The first thing we have to do is find out how ?DELETE works. When you look it up in the "AOS Programmer's Manual", you will see that ?DELETE has no parameter packet and requires only a single argument: a byte pointer to that name of the file to be deleted. The argument goes into accumulator 0. If the call is successful, it will not return any values; if it fails, it then returns an octal error code to accumulator 0.

How do we set up a PL/I program so that SYS will make the ?DELETE call. First, there are the declarations: we already know how to declare SYS as an external entry constant; we also need three fixed binary variables for the accumulators, a pointer variable for the AOS file name, and an aligned bit variable for the error code to be returned in AC0. If we call the variables for the accumulators AC0, AC1, and AC2, the variable for the pointer FILE_POINTER, and the variable fo the error code ERROR_MESS, our declarations will look like this:

```
DECLARE (AC0,AC1,AC2) FIXED BINARY;
DECLARE FILE_POINTER POINTER DEFINED (AC0);
DECLARE ERROR_MESS ALIGNED BIT(16) DEFINED(AC0);

DECLARE SYS ENTRY(ALIGNED BIT(16),FIXED BINARY,FIXED BINARY,
FIXED BINARY)RETURNS(BIT);
```

We also need a PL/I variable for the AOS file name. The system call requires a byte pointer, so the variable must have the character data type:

```
DECLARE AOS_NAME CHARACTER(32);
```

Before we can actually make the system call, we have to put the file name into a form that AOS can recognize, and assign the location of the file name to FILE_POINTER. All AOS file names hve the ASCII null character as a terminator; hence, we have to concatenate the terminator onto the character string for the file name. If the file name is a character-string constant such as "DATA.GN", this step looks like this:

```
AOS_NAME = "DATA.GN" !! ASCII(0);
```

To assign the location of AOS_NAME to FILE_POINTER, we use the ADDR built-in function:

```
FILE_POINTER = ADDR(AOS_NAME);
```

Now we are ready to make the call with SYS. The first argument is the bit pattern which actually calls ?SYSDELETE. You can get the bit pattern from the file PL1SYSID.PL1, which is part of your PL/I software. PL1SYSID.PL1 contains a series of %REPLACE statements which replace PL/I identifiers withthe bit patterns needed for the system calls. The identifers are like the AOS system calls, except that the are preceded by SYS_ instead of the question mark. The identifier for the bit pattern for ?DELETE is SYS_DELETE. The part of the file which contains the bit pattern looks like this:

```
%REPLACE SYS_DELETE BY "0001"B;
```

Right now, we are only interested in the bit pattern; later on, we will see how to use the identifier.

When we put everything together, the call looks like this:

```
IF SYS("0001" B4,AC0,AC1,AC2)
        THEN
        PUT SKIP EDIT("AS UNABLE TO DELETE_FILE",
                        SUBSTR(AOS_NAME,LENGTH(AOS_NAME)-1,
                        "THE OCTAL ERROR CODE IS", ERROR_MESS)
                        (Z,X(2),A,X(5)B3);

        ELSE
        PUT SKIP LIST("FILE" !! SUBSTR(AOS_NAME,1,LENGTH(AOS_NAME)-1)
                        !! "DELETED");
```

The function SYS returns the value "1"B if there is an exceptional return and "0"B if there is a normal return; hence, the ELSE clause will be executed when there is a normal return. The SUBSTR(AOS_NAME,1,LENGTH (AOS_NAME)-1) expression is necessary because AOS_NAME has the ASCII null terminator and PUT LIST will not.

Using %INCLUDE with PL1SYSID.PL

In the above example, we used PL1SYSID.PL1 merely as the source of the bit string for the fist argument SYS. Generally, however, you would simply write:

```
IF SYS(SYS_DELETE,AC0,AC1,AC2)
```

and let PL/I substitute the bit string for SYS_DELETE. In order to have the compiler make the substitution, you have to write the:

```
%INCLUDE "PL1SYSID.PL1";
```

compile-time command in your program before you use any of the constants in the file. %INCLUDE "data-set-name" instructs the compiler to include the contents of the data set given in %INCLUDE at the point in the program where the command appears. If you have used the %INCLUDE, PL/I will simply replace any occurrences of SYS_DELETE with the proper bit- string constant. For complete details on the %INCLUDE and %REPLACE compile-time commands, see the reference manual.

A Complete Subroutine to Delete a File

An entire PL/I subroutine which takes an AOS file name as a parameter and then deletes the file looks like this:

```
FILE_DELETE:
   PROCEDURE(FILE_NAME);

   DECLARE FILE_NAME CHARACTER(32) VARYING;
   DECLARE AOS_NAME CHARACTER(33);

   DECLARE (AC0,AC1,AC2) FIXED BINARY;
   DECLARE FILE_POINTER POINTER DEFINED (AC0);
   DECLARE ERROR_MESS ALIGNED BIT(16) DEFINED (AC0);

   DECLARE SYS ENTRY(ALIGNED BIT(16),FIXED BINARY,FIXED BINARY,
                  FIXED BINARY) RETURNS(BIT);

   %INCLUDE "PL1SYSID.PL1";

/* CONVERT THE FILE NAME INTO A FORM ACCEPTABLE TO ASO */

   AOS_NAME = FILE_NAME !! ASCII(0);

   FILE_POINTER = ADDR(AOS_NAME);

/* MAKE THE SYSTEM CALL */

   IF SYS(SYS_DELETE,AC0,AC1,AC2)
     THEN
     PUT EDIT("AOS UNABLE TO DELETE FILE",
              SUBSTR(AOS_NAME,1,LENGTH(AOS_NAME)-1),
              "THE OCTAL ERROR CODE IS",ERROR_MESS)
              (A,X(2),A,X(5),B3);

     ELSE
     PUT LIST("FILE", SUBSTR(AOS_NAME,1,LENGTH(AOS_NAME)-1),"DELETED");

   END; /* FILE_DELETE */
```

Figure 14-1. This Subroutine Deletes the File

A More Complicated Example: Using ?GTMES to Read a CLI Command

The AOS system call ?GTMES lets a program read switches and arguments from the command line which invoked it. If you look up ?GTMES in the 'AOS Programmer's Manual', you will see that it takes the address of a parameter packet in accumulator 2 and that accumulators 0 and 1 return values from the call.

The parameter packet for ?GTMES is four words long. The first word contains the request type. The request type is a bit pattern which tells AOS how it is to read the command line. How ?GTMES uses the remaining 3 words in the parameter packet depends on the kind of request. For instance, if the request requires an argument number, the number goes into the second word.

The bit patterns for the parameter packet are contained in the file PL1PARU.PL1. This file, which is part of your PL/I software, works exactly like PL1SYSID.PL1. As with PL1SYSID.PL1, you will get the right bit patterns for the parameter packets if you use a %INCLUDE statement with PL1PARU.PL1 in your program and then use identifiers of the form SYS_ < aos parameter name > wherever you need the bit pattern. For example, the AOS parameter name for the request which gets the number of arguments is ?GCNT, so wherever you need the bit pattern for that request in your program, you will use the identifier SYS_GCNT.

To show you how to use SYS to call ?GTMES from a PL/I program, we have written a procedure called COM_READ which reads the command line for a text formatting program called FORMAT. FORMAT's command line may have one or two arguments and a key-word switch. If it has a single argu- ment, the program will read the argument as the input file and make a temporary file called < input-filename > .TEMP for the output file. If the command line has two arguments, FORMAT will read the first argument as the input file and the second argument as the output file. The key-word switch is SHIFT= < ASCII character > . The switch is optional. If you use it, FORMAT treats the character in the switch as a shift character. The formal syntax of the command line thus looks like this:

```
XEQ FORMAT[/SHIFT=<ASCII character>] filename [filename]
```

The procedure COM_READ, which reads this command line, returns the input filename, the output filename, and the shift character to FORMAT. In order to return the information FORMAT needs, COM_READ has to do the following:

1. Find the number of arguments in the command line.
2. If there is one argument, read that argument as the input file and set up temporary output filename.
3. If there are two arguments, read the first argument as the input file and the second as the output file.
4. Find out whether FORMAT has a switch.
5. If there is a switch, check whether it is "SHIFT".
6. Read the shift character following SHIFT.

If you look at the information the various request types return to ?GTMES, you will see that ?GCNT will return the number of arguments, ?GARG will read an argument, ?GCMD will return the command line in a form which lets us check for switches, and ?GTSW will check a switch and return the shift character.

How do we set up our PL/I declarations for the system call? We can treat the arguments for the accumulators the same way we did with ?DELETE, namely, as fixed binary variables. As with the previous example, we will have to define other variables onto the variables for the accumulators. We need an aligned bit(16) variable which has been defined onto AC0 for the error messages, and we need a pointer variable which has been defined onto AC2 for the address of the parameter packet. Let's call this pointer PACK_POINTER. The first word of the parameter packet for ?GTMES must contain aligned bit data, the second fixed binary data, and the third and fourth pointer data. Hence, it is easiest to treat the packet as a PL/I structure, which we will call PARAM_PACK. We also need parameters for the file names and the shift character, some variables for the byte pointers in the parameter packet to point to, and a control variable. The whole set of declarations for COM_READ looks like this.

```
/* PARAMETERS */

            DECLARE I_VILE CHARACTER(32) VARYING;
                        /* INPUT FILE NAME */

            DECLARE O_FILE CHARACTER(32) VARYING;
                        /* OUTPUT FILE NAME */

            DECLARE S_CHAR CHARACTER(1);
                        /* SHIFT CHARACTER */

/* ENTRY DECLARATION FOR THE SYSTEM CALL */

            DECLARE SYS ENTRY(ALIGNED BIT,FIXED BINARY,FIXED BINARY,
                                    FIXED BINARY) RETURNS(BIT(1));

/* VARIABLES FOR THE SYSTEM CALL */

            /* ACCUMULATORS */

            DECLARE (AC0,AC1,AC2) FIXED BINARY;

            /* POINTER FOR ACCUMULATOR 2 */

            DECLARE PACK_POINTER POINTER DEFINED(AC2);

            /* ALIGNED BIT VARIABLE FOR THE ERROR MESSAGES IN AC0 */

            DECLARE ERR_MESS ALIGNED BIT(16) DEFINED(AC0);

            /* VARIABLES FOR THE VALUES RETURNED IN THE ACCUMULATORS */

            DECLARE NO_ARGS FIXED BINARY;
                        /* NUMBER OF ARGUMENTS */

            /* THE PARAMETER PACKET */

            DECLARE 1 PARAM_PACK,
                        2 REQ_TYPE ALIGNED BIT(16),
                                    /* KIND OF REQUEST TO GTMES */

                        2 ARG_NO FIXED BINARY,
                                    /* ARGUMENT THE CALL IS READING */

                        2 SW_LOC POINTER,
                                    /* POINTER TO THE STORAGE FOR THE KEYWORD
                                                SWITCH */

                        2 ARG_LOC POINTER;
                                    /* POINTER TO STORAGE FOR THE ARGUMENT */

            /* STORAGE FOR SWITCHES AND ARGUMENTS */

            DECLARE COM_LINE CHARACTER(80);
            DECLARE SWITCH CHARACTER(6);

            /* CONTROL VARIABLE */

            DECLARE I FIXED BINARY;
```

The %INCLUDE statements for PL1SYSID.PL1 and PL1PARU.PL1 should precede or follow the declarations.

The first thing the procedure has to do is find the number of arguments in the command line. The request type for this request is ?GCNT, so we will want to assign the identifer SYS_GCNT to PARAM_PACK.REQ_TYPE. We also have to assign the address of PARAM_PACK to PACK_POINTER. Now we can make the system call. If it is successful, ACO will contain the number of arguments following FORMAT. As you will see from the description of ?GCNT in the 'AOS Programmer's Manual', ?GCNT does not count either XEQ or the name of the invoking program. If the call does not succeed, AC0 will contain an error code. COM_READ has an internal procedure ERROR, which takes the error code as an argument, to handle errors. The system call looks like this:

```
PARAM_PACK.REQ_TYPE = SYS_GCNT;
PACK_POINTER = ADDR(PARAM_PACK);
IF SYS(SYS_GTMES,ACO,AC1,AC2)
            THEN
            CALL ERROR(ERR_MESS);

            ELSE
            NO_ARGS = ACO;
```

The program's next action will depend on the number of arguments. If there are no arguments or more than 2 arguments, COM_READ should issue an error message; if there is 1 argument, it should set up a file called < argument > .TEMP, and if there are two, it should read both arguments. The easiest way to handle this in PL/I is to make a procedure GET_ARG which reads arguments and then use this procedure in the cases of a DO CASE statement whose case number is the value ?GCNT returns to AC0.

GET_ARG has to parameters: the argument number and the character string returned by ?GTMES invoked with the ?GARG request type. When AOS numbers the arguments in a command line, it does not count XEQ, gives the name of the program following XEQ the number 0, and gives the following arguments the numbers 1, 2, and so forth. If there is only one argument following FORMAT, we will want to invoke GET_ARG with the argument number 1; if there are two arguments, we will invoke it twice, once with the argument number 1, and once with the argument number 2.

Within GET_ARG, we have to set up the parameter packet and the system call for the ?GARG rrequest type. If you once again refer to the *AOS Programmer's Manual,* you will see that ?GARG take the argument number in the second word of the parameter packet and the location of the storage for the command line argument in the fourth word. If the call is successful, ?GARG returns the argument to the storage given in the parameter packet and returns the number of characters in the argument in AC0. There is only one complication: the file name returned by GET_ARG should be contained in a character varying variable, so that it has no blanks, but character varying variables are word aligned. Consequently, we cannot use a pointer to a character varying variable in the parameter packet. To solve the problem, we use a character variable called ARGUMENT as storage for the argument and then use the argument length returned in AC0 to take the substring of ARGUMENT which contains the argument and assign it to the parameter FILE_NAME. The procedure looks like this:

```
GET_ARG:
            PROCEDURE(FILE_NAME,A_NO);

/* GET_ARG USES THE SYS_GARG REQUEST TO GET AN */
/* ARGUMENT FROM THE COMMAND LINE */

            DECLARE FILE_NAME CHARACTER(32) VARYING;
            DECLARE A_NO FIXED BINARY;
            DECLARE ARGUMENT CHARACTER(32);

            PARAM_PACK.REQ_TYPE = SYS_GARG;
            PARAM_PACK.ARG_NO = A_NO;
            PARAM_PACK.ARG_LOC = ADDR(ARGUMENT);
```

```
                IF SYS(SYS_GTMES,AC0,AC1,AC2)
                            THEN
                            CALL ERROR(ERR_MESS);

                            ELSE
                            FILE_NAME = SUBSTR(ARGUMENT,1,AC0);
  END; /* GET_ARG */
```

To test whether FORMAT has a switch, we use the ?GCMD request. This request returns an edited version of the command line to the location given in the fourth word of the parameter pack. In this edited version, each item in the command line is followed by a comma. If FORMAT has no switch, the seventh character in the command line will be a comma, so we can use the INDEX built-in function to check for an argument.

If there is a switch, we will use the ?GTSW request to check it and read the shift character. The parameter packet for ?GTSW looks like this:

word 1:	The request type
word 2:	The number of the argument whose switch is to be read.
word 3:	A location which contains a string to compare the switch with.
word 4:	A location to store the value contained in the switch.

If ?GTSW can't find the switch, it returns -1 in AC0; if it finds a non-keyword switch, it returns 0 in AC0; if it finds the switch, it returns the length of the switch in AC0. If the switch's value is a decimal number, it returns the number's binary equivalent in AC2; otherwise, it returns the value to the storage given in word 4. The only trick to using ?GTSW is that the string it compares the switch with must be terminated by the ASCII null character. The code for checking the switch in COM_READ looks like this:

```
PARAM_PACK.REQ_TYPE = SYS_GTSW;
PARAM_PACK.ARG_NO = 0; /* ARGUMENT 0 IS THE PROGRAM NAME */
PARAM_PACK.SW_LOC = ADDR(SWITCH); /* ARGUMENT HOLDS THE
                                        KEYWORD SWITCH */
PARAM_PACK.ARG_LOC = ADDR(S_CHAR);
SWITCH = "SHIFT" !! ASCII(0);

IF SYS(SYS_GTMES,AC0,AC1,AC2)
          THEN
          CALL ERROR(ERR_MESS);

          ELSE
          IF AC0 = -1
                    THEN
                    DO;
                    PUT LIST("THE ONLY LEGAL SWITCH IS " !!
                                        "/SHIFT=<SHIFT CHARACTER>");
                    STOP;
                                        END;

                    ELSE
                                        RETURN;
  END; /* DO */
```

The entire procedure COM_READ looks like this:

```
COM_READ:
   PROCEDURE(I_FILE,O_FILE,S_CHAR);

/* PARAMETERS                                                  */

   DECLARE I_FILE CHARACTER(32) VARYING;
        /* INPUT FILE NAME */

   DECLARE O_FILE CHARACTER(32) VARYING;
        /* OUTPUT FILE NAME */

   DECLARE S_CHAR CHARACTER(1);
        /* SHIFT CHARACTER  */

/* ENTRY DECLARATION FOR THE SYSTEM CALL */

   DECLARE SYS ENTRY(ALIGNED BIT,FIXED BINARY,FIXED BINARY,
                        FIXED BINARY) RETURNS(BIT(1));

/* VARIABLES FOR THE SYSTEM CALL                               */

   /* ACCUMULATORS                                             */

   DECLARE (AC0,AC1,AC2) FIXED BINARY;

   /* POINTER FOR ACCUMULATOR 2                                */

   DECLARE PACK_POINTER POINTER DEFINED(AC2);

   /* ALIGNED BIT VARIABLE FOR THE ERROR MESSAGES IN AC0   */

   DECLARE ERR_MESS ALIGNED BIT(16) DEFINED(AC0);

   /* VARIABLES FOR THE VALUES RETURNED IN THE ACCUMULATORS */

   DECLARE NO_ARGS FIXED BINARY;
        /* NUMBER OF ARGUMENTS */

   /* THE PARAMETER PACKET                                     */

   DECLARE 1 PARAM_PACK,
               2 REQ_TYPE ALIGNED BIT(16),
                 /* KIND OF REQUEST TO GTMES */

               2 ARG_NO FIXED BINARY,
                 /* ARGUMENT THE CALL IS READING */

               2 SW_LOC POINTER,
                 /* POINTER TO THE STORAGE FOR THE KEYWORD  */
                 /*   SWITCH                                 */

               2 ARG_LOC POINTER;
                 /* POINTER TO STORAGE FOR THE ARGUMENT     */

   /* STORAGE FOR THE COMMAND LINE AND THE SWITCH STRING      */

   DECLARE COM_LINE CHARACTER(80);
   DECLARE SWITCH CHARACTER(6);

  /*  CONTROL VARIABLE                                         */
```

Figure 14-2. The COM_READ Procedure

```
          DECLARE I FIXED BINARY;

          /* FILES WHICH REPLACE NAMES WITH BIT VALUES FOR THE        */
          /*       SYSTEM CALL                                         */

          %INCLUDE "PL1PARU.PL1";
          %INCLUDE "PL1SYSID.PL1";

/**********************************************************************/
/*      READ THE NUMBER OF ARGUMENTS WITH SYS_GCNT                    */
/**********************************************************************/

          PARAM_PACK.REQ_TYPE = SYS_GCNT;
          PACK_POINTER = ADDR(PARAM_PACK);
          IF SYS(SYS_GTMES,AC0,AC1,AC2)
             THEN
             CALL ERROR(ERR_MESS);

             ELSE
             NO_ARGS = AC0;

/**********************************************************************/
/* ACTIONS BASED ON THE NUMBER OF ARGUMENTS                           */
/*                                                                    */
/*    NO ARGUMENTS OR MORE THAN 2 ARGUMENTS: ERROR                    */
/*    1 ARGUMENT: SET UP A TEMPORARY FILE FOR THE OUTPUT FILE         */
/*    2 ARGUMENTS: FIRST ARG IS INPUT FILE, SECOND OUTPUT FILE        */
/**********************************************************************/

        DO CASE(NO_ARGS);

           /* CASE 1: MAKE A TEMPORARY OUTPUT FILE */

           DO;
           CALL GET_ARG(I_FILE,1);
           O_FILE = I_FILE || ".TEMP";
           END;

           /* CASE 2: FIRST ARG IS THE INPUT FILE,                    */
           /*         SECOND ARG IS THE OUTPUT FILE                    */

           DO;
           CALL GET_ARG(I_FILE,1);
           CALL GET_ARG(O_FILE,2);
           END;

           END; /* DO CASE */

        OTHERWISE
           DO;
           PUT SKIP LIST("FORMAT REQUIRES 1 OR 2 FILENAMES AS " ||
                             "ARGUMENTS");
           STOP;
           END;

        /*************************************************************/
        /* TEST WHETHER THE PROGRAM NAME HAS A SWITCH                */
        /*************************************************************/

        REQ_TYPE = SYS_GCMD;
        ARG_LOC =  ADDR(COM_LINE);
        IF SYS(SYS_GTMES,AC0,AC1,AC2)
           THEN
           CALL ERROR(ERR_MESS);
```

Figure 14-2. The COM_READ Procedure (continued)

```
      ELSE
      IF INDEX(COM_LINE,",") > 7
           THEN
              DO;

/******************************************************************/
/*  CHECK AND READ THE  SHIFT SWITCH WITH SYS_GTSW              */
/******************************************************************/

              PARAM_PACK.REQ_TYPE = SYS_GTSW;
              PARAM_PACK.ARG_NO = 0; /* ARGUMENT 0 IS THE PROGRAM NAME */
              PARAM_PACK.SW_LOC = ADDR(SWITCH); /* ARGUMENT HOLDS THE
                                     KEYWORD SWITCH */
              PARAM_PACK.ARG_LOC =  ADDR(S_CHAR);
              SWITCH = "SHIFT" || ASCII(0);

              IF SYS(SYS_GTMES,AC0,AC1,AC2)
                 THEN
                 CALL ERROR(ERR_MESS);

                 ELSE
                 IF AC0 = -1
                    THEN
                       DO;
                       PUT LIST("THE ONLY LEGAL SWITCH IS " ||
                            "/SHIFT=<SHIFT CHARACTER>");
                       STOP;
                       END;

                    ELSE
                    RETURN;
               END; /* DO */


         ELSE RETURN;

GET_ARG:
   PROCEDURE(FILE_NAME,A_NO);

/******************************************************************/
/* GET_ARG USES THE SYS_GARG REQUEST TO GET AN ARGUMENT FROM    */
/*        THE COMMAND LINE                                      */
/******************************************************************/

   DECLARE FILE_NAME CHARACTER(32) VARYING;
   DECLARE A_NO FIXED BINARY;
   DECLARE ARGUMENT CHARACTER(32);

   PARAM_PACK.REQ_TYPE = SYS_GARG;
   PARAM_PACK.ARG_NO = A_NO;
   PARAM_PACK.ARG_LOC = ADDR(ARGUMENT);

   IF SYS(SYS_GTMES,AC0,AC1,AC2)
        THEN
        CALL ERROR(ERR_MESS);

        ELSE
        FILE_NAME = SUBSTR(ARGUMENT,1,AC0);
END; /* GET_ARG */
```

Figure 14-2. The COM_READ Procedure (continued)

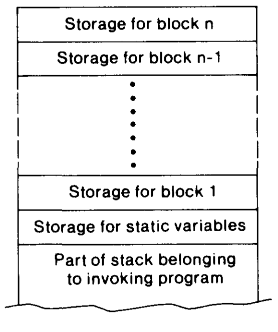
```
ERROR:
     PROCEDURE(ERR_CODE);

     DECLARE ERR_CODE ALIGNED BIT(16);
     DECLARE ERR_NO FIXED BINARY STATIC INTERNAL
             INIT(0);



        ERR_NO = ERR_NO + 1;
        PUT EDIT("ERROR NUMBER =",ERR_NO,"OCTAL ERROR CODE =",ERR_CODE)
                        (SKIP,A,F(6),X(1),A,X(1),B3);

     END;/* ERROR */

END; /* COM_READ */
```

Figure 14-2. The COM_READ Procedure (continued)

Calling Assembly-Language Routines from PL/I

PL/I treats assembly-language routines like PL/I external procedures. You may invoke assembly-language routines which return values to the entry with function references, and you may invoke routines which do not with the CALL statement. However, you must have declared the assembly-language entry name as an external entry constant in the calling procedure and you must include the assembly-language routine's .OB file with the PL/I .OB files when you bind your programs.

To write assembly-language routines for PL/I programs, you have to know something about the stack. Each process running under AOS has an area of storage which contains the program's static variables, the automatic variables for the currently active blocks, and the information PL/I needs to return from a block activation to its predecessor. Each time PL/I activates a block, it adds to the stack; each time it returns from a block, it frees the block's storage on the stack. A schematic for the stack of a PL/I program with n active blocks looks like this:

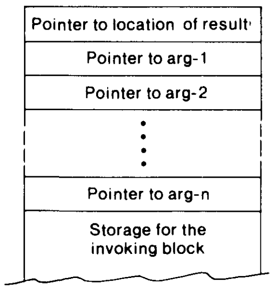

Storage for block n
Storage for block n-1
Storage for block 1
Storage for static variables
Part of stack belonging to invoking program
SD-01026

PL/I manages the stack for you, so you need not worry about it when you write PL/I procedures. In assembly-language routines, on the other hand, you reference storage in terms of its location on the stack. Hence, you must know what the stack looks like when you invoke an assembly-language routine from a PL/I program.

When you invoke an asembly-language routine, the stack will grow as follows. When PL/I executes the invocation it adds an argument block to the stack. The argument block contains pointers to the invocation's arguments. The pointer to the first argument is at the top of the argument block and the location of the last argument is at the bottom. If the assembly-language routine is a function, the argument block will have a pointer to the location where the function's result is to be stored. PL/I stores this pointer above the pointer to the last argument. If you have an invocation like this:

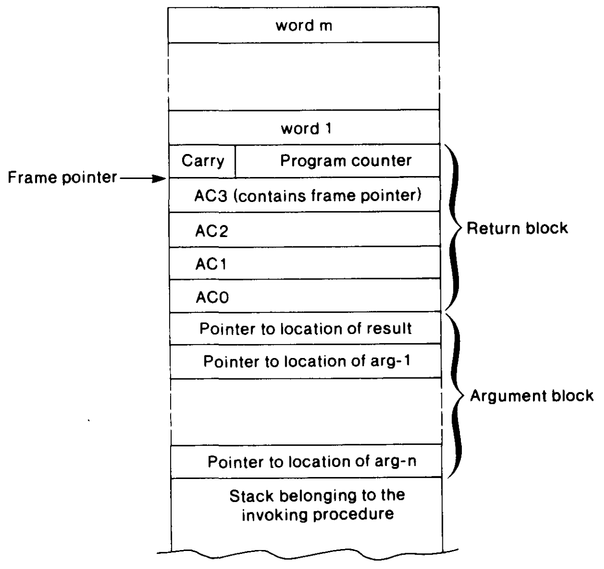
```
FUNC(arg_1,arg_2,. . . ,arg_n)
```

the top of the stack will look like this:



SD-01027

Your assembly-language routine will generally begin with the SAVE instruction. The SAVE instruction sets up the routine's storage and adds it to the stack above the argument block. SAVE first sets up a five-word return block, which contains the information the computer needs to return to the calling routine. Then it sets up m words of storage above the return block. After a SAVE instruction, the stack will look like this:



word m
word 1
Carry
Program counter
Frame pointer
AC3 (contains frame pointer)
AC2
AC1
AC0
Return block
Pointer to location of result
Pointer to location of arg-1
Pointer to location of arg-n
Argument block
Stack belonging to the invoking procedure

SD-01028

The frame pointer contains the location of the top-most word of the return block. The SAVE instruction stores this pointer in word AC3 of the return block. When you refer to storage within the assembly-language routine, you do so in terms of offsets from the location stored in the frame pointer.

Since the frame pointer points to the top word of the return block, positive offsets will refer to the locations above the return block and negative offsets will refer to locations in the return block and the argument block. For example, the pointer to the location in which the value returned from the routine is to be stored has the offset -5. For complete details on stack management, see the *Programmer's Reference Manual ECLIPSE-Line Computer.*

For subroutine calls, the pointer to the location of the first argument has offset -5, the second -6, etc. For function calls, the pointer to the location of the first argument has offset -6, the second -7, etc.

To illustrate the above, let's take a look at a simple assembly-language routine called SHIFT and a PL/I program which invokes it. SHIFT shifts the bits in a single word of a word-aligned data item a given number of places to the left or the right, and returns the shifted value to the invoking procedure. The assembly-language routine has two entry points: SHL, which shifts the value to the left, and SHR, which shifts the value to the right. The first argument in the invocation gives the value to be shifted; the second gives the number of places it is to be shifted. For example, if we have an aligned bit variable called BIT_WORD then the invocation:

```
DECLARE BIT_WORD ALIGNED BIT(16) STATIC INTERNAL INIT("FFFF"B4);
.
PUT LIST(SHL(BIT_WORD,1));
```

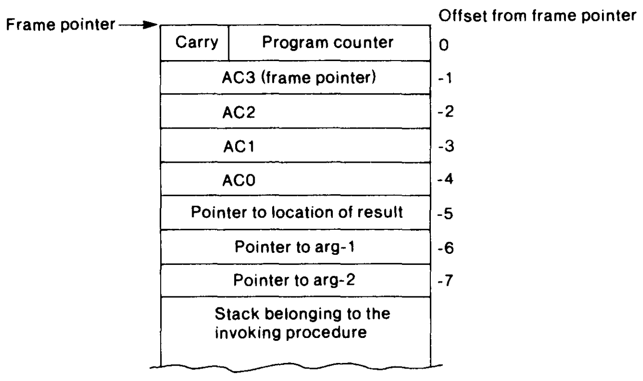
will shift the contents of BIT_WORD one bit to the left, so that the PUT LIST statement will output "1111111111111110"B.

The assembly-language routine SHIFT looks like this:

```
                .ENT        SHR,SHR         ;NAMES OF ENTRY POINTS ;SHL
                                            SHIFTS LEFT, SHR SHIFTS RIGHT

                .NREL       1

SHL:            SAVE        0               ;ADD THE RETURN BLOCK TO THE
                LDA         1,@-7,3         STACK
                JMP         COMON           ;LOAD ARG 2 INTO AC1
                                            ;JUMP TO THE LABEL COMON

SHR:            SAVE        0               ;ADD THE RETURN BLOCK TO THE
                LDA         1,@-7,3         STACK
                NEG         1,1             ;LOAD ARG 2 INTO AC1
                                            ;NEGATE ARGUMENT 2

COMON:          LDA         2,@-6,3         ;LOAD ARG 1 INTO AC2
                LSH         1,2             ;DO THE SHIFT
                STA         2,@-5,3         ;STORE THE SHIFTED VALUE
                RTN                         ;RETURN TO THE INVOKING
                .END                        PROCEDURE
```

When you invoke the routine, control enters at SHL for the left shift and SHR for the right shift. The actual shift is performed by a common section of code beginning at the label COMON. For the left-hand shift, the program simply gets the second argument and jumps to COMON. For the right-hand shift, it gets the argument and negates it.

The first instruction at both entries is SAVE 0. Since SHIFT is a function, the stack looks like this:



SD-01029

The LDA 1, @ -7,3 instruction puts the second argument into accumulator 1. It does this as follows:

a. It goes to accumulator 3 (the 3 in the instruction) and gets the frame pointer.
b. Then it goes to offset -7 from the frame pointer and gets the pointer to the second argument. (the -7 in the instruction)
c. It goes to the location indicated by the pointer and loads the value it contains into accumulator 1. (the 1, @ - in the instruction)

LDA 2, @ -6,3 loads the first argument, whose offset is -6, into accumulator 2.

LSH 1,2 shifts the contents of accumulator 2 by the amount given in accumulator 1. Positive values produce a left shift; negative ones produce a right shift.

STA 2, @ -5,3 stores the result of the shift (in accumulator 2) in the location given by the pointer stored in offset -5 of the stack.

RTN returns to the calling procedure.

The PL/I procedure SHIFT_INVOKE, which simply invokes SHIFT and outputs the result, looks like this:

```
SHIFT_INVOKE:
            PROCEDURE;

            DECLARE (SHL,SHR) ENTRY(ALIGNED BIT(16),FIXED BINARY)
                                      RETURNS(ALIGNED BIT(16));

           DECLARE BIT_WORD ALIGNED BIT(16) STATIC INTERNAL INIT("FFFF"B4);

           PUT LIST("LEFT SHIFT", SHL(BIT_WORD,1),"RIGHT SHIFT",SHR(BIT_WORD,1));

END;
```

Of course, before you can use SHIFT with SHIFT_INVOKE, you must bind the .OB files into a single .PR file. If you do this and then execute the program, SHIFT_INVOKE will output:

```
LEFT SHIFT          "1111111111111110"B       RIGHT SHIFT "0111111111111111"B
```

Note that if an argument to a procedure is byte aligned, the pointer in the argument list on the stack will be a byte pointer to the argument. Care must be taken to be sure the argument is referenced correctly.

End of Chapter

Index

Note: The primary reference appears first.

@ character 1-9
V 9-5, 9-1
DB 9-5, 9-1
CR 9-5, 9-1
. 9-5, 9-1
, 9-5, 9-1
B 9-5, 9-1
/ 9-5, 9-1
- 9-4, 9-1
+ 9-4, 9-1
$ 9-4, 9-1
* 9-4, 9-1
Z 9-4, 9-1
9 9-4, 9-1

A format 7-19, 7-16
access attributes 7-4
access a stream file 7-3
activating a block 6-9
accumulators 14-1
addition 2-8, 2-9
ADDR built-in function 10-8
ALIGNED BIT variables 3-8
aligned character variables 2-15
aligned data 10-2
alignment 10-2
ALLOCATE statement 10-6
allocation of storage 6-1
AND operator 3-9
AOS binder 1-5
AOS data set 4-12, 4-13, 4-15
AOS generic file 1-2
AOS record type 7-4
AOS system calls 14-1
arguments 6-5, 6-6, 6-12, 14-1
argument block 14-12
arithmetic data types 2-4, 2-1
arithmetic operators 2-8
array parameters 6-21
array operands 5-2
array parameter attributes 13-5
arrays 5-2, 5-1
arrays of label constants 11-3
arrays of structures 5-10
arrays of structure parameter attributes 13-5
ASCII 8-2
ASCII code 3-7
ASCII characters 1-9, 7-10
assembly-language routines 14-1, 14-11, 14-12
assignment statement 2-3
assignment of FILE data 7-3
assignment, with arrays 5-4
assignment with pictures 9-2
assignment, with structures 5-8
asterisk extent expressions 6-12
attribute 7-4
automatic storage 10-3
automatic storage class 6-1
automatic variables 6-1, 6-9

B format 7-20, 7-16
B1 format 7-16
B2 format 7-16
B3 format 7-16
B4 format 7-16
based aggregate 10-6
BASED keyword 10-11
based storage 10-3, 10-4
based structure 10-12
based variables 10-5, 10-1, 10-4, 10-11
BEGIN block 6-1, 6-3, 6-7
BINARY 2-5
binding 1-5, 1-1
binding PL1ECIS 6-22
binding programs with external procedures 6-21
bit 10-1
bit boundary 10-1
bit constants 3-8
bit pattern 14-1
bit-strings 8-1
bit-string data 3-8
bit-string expression 4-4
bit-string functions 8-14
bit variables 3-8
blocks 1-7
Block activation 6-9
block structure 6-1
blocks in recursive invocation 13-8
BOOL 8-15
Boolean operators 8-15
branching 3-1, 3-12
built-in functions 2-12, 6-12
byte 10-1, 10-11
byte boundary 10-1
byte pointer 10-11, 14-2

calculating single interest 2-14
CALL statement 6-3, 6-6
calling assembly-language routines 14-11
character data 2-1, 2-3
character-string 8-1
character-string constants 2-1, 2-15
character-string data 2-15
character-string expression 1-9
character-string literal constant 1-6
character-string operator 2-15
character-string variables 2-1
character variables 2-15
character varying data 2-1
character varying variables 2-15
CLOSE statement 4-13, 7-1
COLLATE 8-9, 8-2
COLUMN format 7-22, 7-16
comma 1-10
comments 1-6
Commercial Instruction Set 2-7
common type, standard PL/I 2-10
comparison 3-7, 3-11
comparison with FILE data 7-3
compile 1-3
compiler 1-1
compiler switches 1-3
compiling 1-1, 1-3
compiling programs with external procedures 6-21
computational precision 2-4, 2-5
concatenate operator 8-1, 8-5
concatention 2-15, 3-11
conditional branching 3-1
conditions 13-10, 13-14
connected arrays 5-10, 5-11
containment 6-7
control format items 7-22
control variable 4-1, 4-7
correspondent 6-14
conversion precision 2-4, 2-5
conversions 2-8, 2-1, 2-3, 2-4, 2-16, 5-8, 7-3
conversions with operations 2-10
converting standard PL/I 2-10
creating a program 1-1

data 1-9, 2-1
@DATA 7-8
data aggregate 14-1
data sensitive 7-4
data set 4-15, 7-1
data storage 10-1
data types 2-1, 10-1
data type attribute 6-17
DECLARE statement 1-2, 1-7, 7-1, 7-2
DECLARE statement, external procedures 6-21
declaration of aligned data 10-2
declaration of an array 5-2
declaration of automatic variables 6-9
declared name 1-6
declaring an array of structures 5-10
declaring arrays 5-9
declaring entry constants 13-1
declaring fixed binary variables 2-5
declaring float binary variables 2-7
declaring a name 6-3
declaring parameters 6-12
declaring static variables 12-2
declaring structures 5-6
declaring SYS 14-1
DECIMAL conversion function 2-13
DECIMAL keyword 2-6
decimal numbers 2-6
default data set 7-7
default file 7-8
default file attributes 7-7
default line size 7-7
default page size 7-7
default precision, fixed binary 2-5
default storage class, see automatic storage
defined storage class 10-3, 10-16
defined variables 10-1, 10-16
defining an array 5-1
defining a strucutre 5-1
DELETE statement 7-32
delimiters 1-6, 1-5
DIM built-in function 6-12
dimension attribute 5-3, 5-7
DIMENSION keyword 5-2
DIRECT keyword 7-7
DIVIDE built-in functions 2-9, 2-12
division 2-8, 2-9
DO CASE [OTHEWISE] statement 3-1, 3-12
DO increment statement 4-1, 4-6
DO increment, implied 7-13
DO list statement 4-1, 4-10
DO loop 4-2
DO WHILE statement 4-1, 4-3
double-precision 2-7
dynamic allocation 6-1

E format 7-18, 7-16
Eclipse-line computer 2-7
elements of arrays 5-1
ELSE clause 3-3, 6-1
edit-directed I/O 7-10, 7-13
EDIT keyword 7-14, 7-15
END statement 1-2, 1-7, 4-13, 6-1
ENDFILE condition 7-9, 13-16
ENDPAGE condition 7-9, 13-18
entry constant 2-1, 6-15, 6-17, 11-2, 13-1
entry data 2-1, 13-1
entry data parameter attributes 13-5
entry declaration 6-21
entry expression 6-17
ENTRY statement 11-2, 13-1, 13-6
entry variables 6-15, 6-17, 13-3
ERROR conditions 13-14
error message 1-4
executable program file 1-5

executing a program 1-5
exiting a block 6-18
explicit pointer-qualified reference 10-10
exponentiation 2-8, 2-10
expression 2-3, 6-5, 6-9
extent expressions 6-9, 6-10, 10-12
extent expressions with based variables 10-5
extent expressions with static variables 12-3
external entry constants 13-2
external procedures 6-7, 6-19, 14-11
external scope 6-8, 12-1

F format 7-17, 7-16
file attribute 7-4
file constant 1-8, 1-9, 1-10, 2-1, 6-8, 7-1, 7-2
file constant attributes 1-8
file control block 7-3, 7-9
file control blocks for record direct files 7-9
file control blocks for record sequential files 7-9
file control block for stream files 3-9
file data 2-1, 7-2
file data operands 7-3
file expression 1-9, 1-10
file openings 7-9
file type 7-3, 1-9, 7-4
file variables 7-2
FIXED 2-5
fixed binary 2-1
fixed binary data 2-5, 2-2
fixed binary precision 2-4
fixed decimal 2-1
fixed decimal data 2-6, 2-3
fixed-length record 7-4, 7-30
flag 4-4
float binary data 2-1, 2-7, 2-2
float binary precision 2-4
flow chart of DO increment 4-7
flow chart for DO list 4-10
format constants 11-2
format item 7-16
format-list 7-16, 7-14, 7-15
FORMAT statement 7-15, 11-2
freeing storage 10-7
fully-qualified reference 5-8
function attributes 7-4
function name 2-12
function reference 6-3, 6-5, 6-6, 6-9, 6-17, 8-14

generic data set 4-15, 7-8
generic file 1-2
GET as format control 7-25
GET EDIT statement 7-14, 7-13, 7-26
GET LIST statement 7-10, 1-2, 1-10, 4-15, 5-3, 7-1, 7-11, 7-26
GET LIST with stream files 4-14
GET LIST with structures 5-8
giving storage to a based variable 10-5
GOTO statement 3-1, 6-18, 11-1
GOTO with BEGIN 11-8
GOTO with blocks 11-8
GOTO with DO CASE 11-7
GOTO with DO groups 11-7
groups 1-7

HBOUND built-in function 6-12
high-level language 1-1

identifiers 1-5, 6-1
IF THEN 3-2
IF THEN ELSE 3-3, 3-1
illegal variables with record I/O 7-28
imaginary outer block 6-19
immediate containment 6-7
implicit pointer-qualified reference 10-10
implied DO increment 5-4, 7-13
implied file attributes 7-7
%INCLUDE 14-3
inclusive OR operator 3-9
INDEX 8-8, 8-2
individual elements 5-3
INITIAL keyword 12-3
initialization of arrays 12-3
initialization with static variables 12-1, 12-3
initialization of structures 12-3
input 1-8
@INPUT 1-2, 1-9, 4-15, 7-26
input from a record file 7-3
input list 1-10, 7-11
input/output 1-2, 7-1
input/output with record files 7-27
I/O 1-2, 7-1
I/O statement 7-4
I/O with record files 7-27
I/O with stream files 7-10
interactive terminal I/O 7-26
internal procedures 6-7
internal scope 6-8, 12-1
internal to a block 6-7
invocation 6-6
invocation of recursive procedure 13-9
invoking a procedure 6-3
invoking assembly-language routines 14-11
invoking with CALL 6-15
iterative DO groups 3-5, 4-1
iterative DO loop 4-2
iterative groups 1-7

keyword 1-6
keywords for record direct files 7-7
keywords for record sequential files 7-6
keywords for STREAM files 7-6

label constants 11-2
label data 11-2
label prefix 6-15, 6-18, 11-2, 11-3
label variable 11-2, 11-4
LBOUND built-in function 6-12
LENGTH built-in functions 8-6, 8-2, 6-12

level 5-7
level numbers 5-7
level of precedence 2-11
line-directed I/O 7-10, 7-25
LINE format 7-23, 7-17
list-directed I/O 7-10
@LIST 1-3, 7-4, 7-8
literal arithmetic constant 2-3
literal constants 1-6, 1-5
logical complement operation 3-9
logical operations 3-9
logical operators 3-9, 8-15
logical truth tables 3-10
loops 4-1
looping 3-1
lower bound 5-3, 6-10
@LPT 4-12

mantissa 2-7, 2-8
maximum precision, fixed decimal 2-3
member 5-1
members of a structure 5-1
minimal PL/I program 7-9
multiple allocations of storage 10-6
multiple branching 3-12
multiple ON units 13-12
multiple storage allocations 10-6
multiplication 2-8, 2-9

named constant 1-6, 7-1
names of external procedures 6-8
nested DO group 4-5
nested DO loop 4-5
nested IF THEN statements 3-4
nesting 6-3
nesting BEGIN blocks 6-3
non-iterative DO groups 3-5
non-iterative groups 1-7
NOT operator 3-9
NULL built-in function 10-9
null statement 4-9

.OB file 6-21
ON statement 13-10
ON units 2-12, 7-9, 13-1, 13-10
ON units and efficiency 13-12
ONCODE built-in function 13-16, 13-17
OPEN statement 1-2, 1-8, 7-4, 7-5
opening a file 1-8, 7-1
opening files with GET 7-8
opening files with PUT 7-8
opening files with READ 7-8
opening files with WRITE 7-8
opening PL/I files 7-4
operands 2-9
operations with bit-string data 3-11
OR (inclusive) operator 3-9
order of evaluation 3-11
order of precedence 2-11, 3-11

output 1-8
@OUTPUT 1-2, 1-9, 4-15, 7-4, 7-26
output-list 1-9, 7-11
output to a record file 7-3

P format 7-16
padded 2-4
padding 2-4
PAGE format 7-24, 7-17
parallel blocks 6-8
parallel DO group 4-5
parameters 6-6, 6-5, 6-9, 6-12
parameter attributes 13-5
parameters in procedures 7-2
parameter packet 14-1
parameter passing 6-13
parameter storage 10-3
parentheses 2-11
passing by reference 6-13, 6-14
passing by value 6-13, 6-14
picture data 9-1
picture data as operands 9-2
picture specifiers 9-3, 9-1
PL1 command 1-3
PL1BIND macro 1-5, 6-19, 6-21
PL1ECIS 6-22
PL1STSID.PL 14-3
PL/I file type 7-4
PL/I null statement 4-9
PL/I Reference manual (093-000204) 1-3
pointer-qualified reference 10-10, 10-12
pointer 10-5, 10-4
pointer types 10-11
.PR file 6-21
precedence 2-9, 2-11
precision 2-3, 2-4
precision, fixed decimal 2-6
precision with pictures 9-2
predecessor 6-9
prefix 11-2
procedure 6-11
PROCEDURE block 1-2, 6-1, 6-3, 6-7, 6-9
procedures invoked as functions 6-6
procedures invoked with CALL 6-6, 6-15
procedures invoked with function references 6-17
PROCEDURE statement 1-7, 6-17, 6-21, 11-2
program efficiency 6-10
program element 2-3
program execution 1-5
program interrupts 13-10
programming with blocks 13-1
PUT LIST statement 7-11, 1-2, 1-9, 5-3, 7-1, 7-26
PUT LIST with stream files 4-14
PUT LIST with structures 5-8
PUT EDIT statement 7-15, 7-13, 7-26
PUT as format control 7-25
pseudo-variable 8-14

quotation marks 1-9

R format 7-24
RANK 8-10, 8-2
READ statement with record direct files 7-30
READ statement with record sequential files 7-29
READ with stream input files 7-25
record direct file 1-9, 7-3, 7-30
record direct I/O 7-30
record direct update files 7-31, 7-32
record I/O 7-27, 7-31
RECORD keyword 7-6, 7-7
record sequential file 1-9, 7-3
record sequential I/O 7-29
record type 7-4
recursive block activation 13-1
recursive keyword 13-8
recursive procedures 13-8
REFER option 10-12
references 5-10
referencing 5-2
relational operators 3-7, 3-2
results 2-9
results of relational operators 3-8
RETURN statement 6-16
returning from a block with a GOTO 6-18
returning from a called procedure 6-16
RETURNS attribute 6-17
REVERT statement 13-12, 13-13
REWRITE statement 7-31
ROUND built-in function 2-13
routines 2-12
runtime library 1-1

scalar based variable 10-6
scalar variable 5-7
scale 2-3, 2-4, 2-6
scale with pictures 7-2
scientific notation 2-8
scope 6-8
scope attributes 6-8
scope of a declaration 6-3
scope of external procedure names 6-19, 6-20
scope of internal procedure names 6-19
scope of procedure names 6-18
scope of static variables 12-1
semicolon 1-7
SEQUENTIAL keyword 7-6
SIGNAL statement 13-12
single-precision 2-7
SKIP format 7-23, 7-16
SKIP keyword 7-11
source text 1-1
space 1-10
specifying a data set 7-5
stack 14-11, 14-12
standard PL/I 2-10, 10-17
standard PL/I compiler 10-17
statements 1-7
statements internal to a block 6-7
STATIC EXTERNAL storage class 6-8
static storage 10-3
static storage class 12-1
STOP statement 4-13
storage class 10-3, 10-1
storage class declaration 10-3
storage of static variables 12-1
storage sharing 10-17
storage sharing, standard PL/I 10-17
storage sharing, AOS PL/I 10-18
stream file 1-9, 7-3, 7-6, 7-10
STREAM keyword 1-9, 7-6
STRING 8-10, 8-2
string-handling functions 8-1
string-handling operations 8-1
string-handling pseudo-variables 8-1
structure 5-6, 5-1, 5-11
structure members 5-1
structure operands 5-2
structure parameters 6-21
structure parameter attributes 13-5
structure-qualified reference 5-2, 5-7
subexpressions 2-11
subexpressions and precedence 2-10
subscripted references 5-3
SUBSTR 8-5, 8-1
substructure 5-6
subtraction 2-8
successor 6-9
syntactical unit 6-1
syntax of blocks 6-7
syntax of DO increment 4-7
syntax of GET LIST 7-10
syntax of GOTO statement 11-5
syntax of ENTRY statement 13-6
syntax of iterative DO group 4-3
syntax of the ON statement 13-11
syntax of non-iterative DO groups 4-3
SYS 14-1
SYSIN 7-2, 7-8
SYSPRINT 7-2, 7-8
system calls 14-1

TAB format 7-24, 7-17
terminology for block structure 6-7
text editor 1-2
text-handling program 8-1
THEN clause 3-3, 6-1
TITLE keyword 1-9, 7-5
TO expression 4-7
TRANSLATE 8-8, 8-1
true 3-7
truncating 2-6
truth table 3-10
types of based variables 10-11
types of I/O with stream files 7-10
types of procedures 6-15

unary - (minus) 2-8
unary - operator 2-9
unary + (plus) 2-8
unary + operator 2-9
unconditional branching 11-1
unique meaning 6-1
unconnected arrays 5-10, 5-11
UNSPEC 8-14
UPDATE keyword 7-7
upper bound 5-3, 6-10
usage attributes 7-4
using built-in functions 2-12
using picture variables 9-2

variable 1-6, 2-3, 6-9, 7-1
VERIFY 8-7, 8-2

writing assembly-language routines 14-11
writing fixed binary constants 2-5
writing fixed decimal literal constants 2-6
writing float binary constants 2-8
writing programs with external procedures 6-21
WRITE statement with record direct files 7-31
WRITE statements with record sequential files 7-29
WRITE statements with stream output files 7-26
word 10-1, 10-11
word boundary 10-1
word pointers 10-11

X format 7-23, 7-16

DataGeneral

Software Documentation Remarks Form

How Do You Like This Manual?

Title ______________________________ **No.** ____________

We wrote the book for you, and naturally we had to make certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve our manuals. Please take a few minutes to respond.

If you have any comments on the software itself, please contact your Data General representative. If you wish to order manuals, consult the Publications Catalog (012-330).

Who Are You?

☐ EDP Manager
☐ Senior System Analyst
☐ Analyst/Programmer
☐ Operator
☐ Other ____________

What programming language(s) do you use? ____________

How Do You Use This Manual?

(List in order: 1 = Primary use)

_____ Introduction to the product
_____ Reference
_____ Tutorial Text
_____ Operating Guide
_____ ____________

Do You Like The Manual?

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the topic order easy to follow?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Does the manual tell you everything you need to know?

Comments?

(Please note page number and paragraph where applicable.)

From:

Name ____________ Title ____________ Company ____________

Address ____________ Date ____________

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Software Documentation

FOLD UP SECOND FOLD UP

SD-00742A

STAPLE