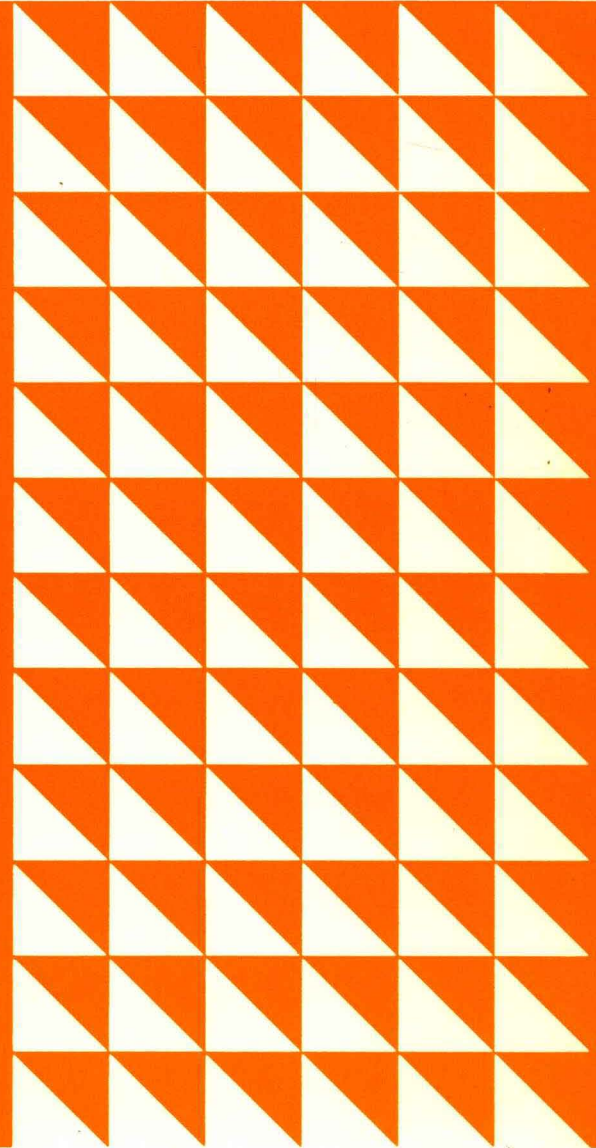


A Guide to PL/I for
Commercial Programmers



Student Text



A Guide to PL/I for
Commercial Programmers

Student Text

Revision of Form SC20-1651-1 (April 1968)

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

© Copyright International Business Machines Corporation 1966

PL/I is a multipurpose, high-level programming language that enables the programming not only of commercial and scientific applications but also of real-time and systems applications. It also permits a programmer to use the full power of his computer in an efficient manner and to program applications in a relatively machine-independent fashion. Because PL/I has a modular design, it provides facilities for programmers at all levels of experience. The novice need learn only those features of the language that satisfy his needs. The experienced programmer may use more advanced features and take advantage of the subtler aspects of the language so that he can program more efficiently and with greater ease than in other programming languages.

This publication presents those features of PL/I that apply to commercial data processing. It does not restrict itself to a particular aspect of commercial data processing but attempts to discuss all features of PL/I that may be used in the full spectrum of commercial applications. The nature of commercial data processing has become more complex in recent years. In the past, as business operations grew in size and complexity, commercial data processing problems were characterized by piecemeal solutions, such as the addition of more personnel to handle a problem or the increased use of equipment. Within the past few years, however, it has become apparent that total-systems methods, formal long-range planning, and scientific techniques are required to solve commercial data processing problems, in somewhat the same way as they have been required to solve production problems. Consequently, the role of the commercial programmer has grown. Besides being responsible for such activities as designing reports, producing payrolls, and solving accounting problems, the commercial programmer is becoming increasingly involved with more sophisticated problems, such as decision making, linear programming, and production forecasting. This means

that the commercial programmer occasionally is concerned with the scientific aspects of data processing; he may be required to process binary data and to perform arithmetic calculations in a floating-point format. Such considerations are discussed in this publication but are not developed in detail; the intent is to make the commercial programmer familiar with those features of PL/I that apply to advanced commercial applications.

This publication consists of five chapters and an appendix. Chapter 1 discusses language notations and deals with such matters as the language character set and the rules for forming identifiers. Chapter 2 describes the various types of data that are processed by PL/I programs. Chapter 3 contains a discussion of input and output. The structure of a PL/I program and the control of statement execution are presented in Chapter 4. Data manipulation and data editing are developed in Chapter 5. Appendix A contains sample PL/I programs.

Because of the many similarities between PL/I and COBOL (Common Business Oriented Language), each chapter in this publication concludes with a section showing comparisons of the two languages. These comparisons do not form a necessary part of this publication and may be skipped by those readers who do not have a knowledge of COBOL.

The material in this publication is quite comprehensive, but by no means is it a complete description of PL/I nor does it represent the definitive treatment of any language feature. A full description of PL/I is given in the publication *IBM Operating System/360, PL/I: Language Specifications*, Form C28-6571. Other publications which may be of interest to the reader are *A PL/I Primer*, Form C28-6808; *A Guide to PL/I for FORTRAN Users*, Form C20-1637, and *PL/I Reference Data, Keywords and Character Sets*, Form X20-1744.

Contents

| | | | |
|--|----|---|----|
| Chapter 1: Language Notation | 7 | Chapter 3: Input/Output | 24 |
| Introduction | 7 | Introduction | 24 |
| PL/I Format | 7 | External Storage Attributes | 24 |
| Words, Delimiters, Blanks, and Comments | 7 | FILE Attribute | 24 |
| Character Set | 7 | STREAM and RECORD Attributes | 24 |
| Delimiters | 8 | INPUT, OUTPUT, and UPDATE Attributes | 24 |
| Identifiers | 8 | PRINT Attribute | 24 |
| Statements | 8 | BUFFERED and UNBUFFERED Attributes | 24 |
| PL/I and COBOL Comparison: Language Notation | 9 | ENVIRONMENT Attribute | 24 |
| | | Opening and Closing of Files | 25 |
| | | OPEN Statement | 25 |
| | | CLOSE Statement | 25 |
| Chapter 2: Data Description | 10 | Data Transmission | 26 |
| Introduction | 10 | Record-Oriented Transmission | 26 |
| Data Types | 10 | Stream-Oriented Transmission | 27 |
| String Data | 10 | Edit-Directed Data Transmission | 27 |
| Character-String Data Items | 10 | Data Format Descriptions | 28 |
| Bit-String Data Items | 10 | Character-String Format Descriptions | 28 |
| Arithmetic Data | 10 | Bit-String Format Descriptions | 30 |
| Decimal Data Items | 10 | Picture Format Description | 30 |
| Binary Data Items | 10 | Repetition of Format Descriptions | 30 |
| Fixed-Point and Floating-Point Formats | 10 | Multiple Data Specifications | 30 |
| Label Data | 10 | Data Specifications for Structures and Arrays | 30 |
| Statement Labels | 10 | Control Format Descriptions | 31 |
| Names of Data Items | 11 | Spacing Format Description | 31 |
| Constants | 11 | Printing Format Descriptions | 31 |
| Character-String Constants | 11 | List-Directed Data Transmission | 32 |
| Bit-String Constants | 11 | List-Directed Data Lists | 32 |
| Decimal Fixed-Point Constants | 11 | Format of List-Directed Data | 32 |
| Binary Fixed-Point Constants | 11 | List-Directed Data Representation | 32 |
| Decimal Floating-Point Constants | 12 | Data-Directed Data Transmission | 33 |
| Binary Floating-Point Constants | 12 | The STRING Option | 33 |
| Label Constants | 12 | DISPLAY Statement | 34 |
| The DECLARE Statement | 12 | PL/I and COBOL Comparison: Input/Output | 34 |
| Arithmetic Attributes | 12 | | |
| DECIMAL and BINARY Attributes | 12 | Chapter 4: Program Structure | 35 |
| FIXED and FLOAT Attributes | 12 | Introduction | 35 |
| Precision Attribute | 12 | Blocks | 35 |
| String Attributes | 13 | Procedure Blocks | 35 |
| CHARACTER Attribute | 13 | Begin Blocks | 35 |
| BIT Attribute | 13 | Internal and External Blocks | 35 |
| PICTURE Attribute | 13 | Scope of Declarations | 36 |
| Fixed-Point Decimal Attributes with PICTURE | 14 | Nested Blocks | 37 |
| Floating-Point Decimal Attributes with PICTURE | 14 | EXTERNAL and INTERNAL Attributes | 37 |
| Character-String Attributes with PICTURE | 14 | Parameters and Arguments | 38 |
| Repetition of PICTURE Characters | 14 | Entry-Name Parameters and the ENTRY Attribute | 39 |
| Exclusivity of Attributes | 15 | Sequence of Control | 40 |
| The DEFINED Attribute | 15 | RETURN Statement | 40 |
| Default Attributes | 15 | Activation and Termination of Blocks | 40 |
| Data Aggregates | 16 | Dynamic Descendance of Blocks | 41 |
| Structures | 16 | GO TO Statement | 41 |
| Qualified Names | 17 | IF Statement | 42 |
| Arrays | 18 | Comparison Expressions | 42 |
| Subscripting | 20 | Comparison Expressions in an IF Statement | 43 |
| Structures Containing Arrays | 20 | DO Statement | 43 |
| ALIGNED and PACKED Attributes | 21 | ON Statement | 45 |
| LIKE Attribute | 21 | Use of the ON Statement | 45 |
| LABEL Attribute | 21 | Prefixes | 46 |
| INITIAL Attribute | 21 | Purpose of the Prefix | 46 |
| Factoring of Attributes | 22 | Scope of the Prefix | 46 |
| PL/I and COBOL Comparison: Data Description | 22 | ON Conditions | 47 |

| | |
|--|----|
| Storage Allocation | 48 |
| Static Storage | 48 |
| Automatic Storage | 48 |
| Controlled Storage | 49 |
| Based Storage | 49 |
| ALLOCATE Statement | 49 |
| FREE Statement | 50 |
| PL/I and COBOL Comparison: Program Structure | 50 |

| | |
|--|----|
| Chapter 5: Data Manipulation | 51 |
| Introduction | 51 |
| Assignment Statement | 51 |
| Conversion Between Data Types | 53 |
| Bit-String Data to Character-String Data | 53 |
| Character-String Data to Bit-String Data | 53 |
| Bit-String Data to Arithmetic Data | 53 |
| Character-String Data to Arithmetic Data | 53 |
| Arithmetic Data to Character-String Data | 53 |
| Arithmetic Data to Bit-String Data | 53 |
| Expressions Containing Operators | 54 |
| Arithmetic Expressions | 54 |
| Conversion of Arithmetic Data | 55 |

| | |
|--|----|
| Comparison Expressions | 55 |
| Bit-String Expressions | 56 |
| Concatenation Expressions | 57 |
| Array Expressions | 57 |
| Structure Expressions | 58 |
| Assignment BY NAME | 59 |
| Data Editing | 59 |
| PL/I and COBOL Comparison: Data Manipulation | 62 |

| | |
|--|----|
| Appendix | 63 |
| Problem 1 — A Book Pricing Problem | 63 |
| Solution to Problem 1 | 63 |
| Problem 2 — A Work Card Study | 64 |
| Solution to Problem 2 | 64 |
| Problem 3 — A File Search | 64 |
| Format of Code String | 64 |
| Solution to Problem 3 | 65 |

| | |
|--------------------|----|
| Index | 66 |
|--------------------|----|

Introduction

Programming languages may be classified as either computer-oriented or problem-oriented. The instructions used in a program written in a computer-oriented language are specified in notations that reflect the composition of machine instructions. Words, such as ADD; acronyms, such as TSX; "fields," such as address fields; might be contained in the instructions in such a program. The number of letters that can be used in a word, the set of acronyms, and the order of fields within an instruction are fixed in accordance with the order and size of the fields in a machine instruction. In short, a computer-oriented language is designed for a particular computer and is not intended for use on a different type of computer.

In a problem-oriented language, the notation reflects the type of problem being solved rather than the computer on which the program is to be run. In COBOL, the notations used to write a program resemble English; FORTRAN notations resemble the language of mathematics; PL/I notations combine the features of COBOL and FORTRAN notations.

Just as restrictions exist on the notation of English and of mathematics, there also exist restrictions in the notation of problem-oriented languages. Only a specified set of numbers, letters, and special characters may be used in writing a program; special rules must be observed for punctuation and for the use of blanks.

The remainder of this chapter is a discussion of the rules for the notations used in writing a PL/I program.

PL/I Format

PL/I allows the programmer to write his program in a free format, thus eliminating the need for coding on special forms or for punching items in particular columns of a card.

Depending on the particular machine configuration or the particular compiler, conventions can be established so that a program can be prepared for a computer through the medium of fixed-length records (for example, punched cards). If this is the case, certain predetermined fields in the records could be used for the program.

For example, columns 2 through 72 could be used for source text and columns 73 through 80 could be used as a sequence number field.

Words, Delimiters, Blanks, and Comments

All the elements that make up a PL/I program are constructed from the PL/I character set. There are two exceptions: character-string constants and comments. Character-string constants and comments may contain any character permitted for a particular machine configuration.

A PL/I program consists of words and/or delimiters. Words belong to one of two categories: identifiers or constants. Adjacent words can be separated by one or more delimiters and/or blanks. For example, CALLA is considered to be one identifier; CALL A is considered to be two identifiers; AB+BC is considered to be two identifiers separated by the delimiter + and is equivalent to AB + BC, where + is surrounded by blanks.

Comments may be used anywhere that a blank is permitted except within a character-string constant or a picture specification. Comments have the form:

```
/* comment */
```

Comments can consist of one or more of the characters permitted for a particular machine configuration. However, the character combination */ may not be contained within a comment since it signifies the termination of a comment.

Character Set

The PL/I character set comprises 60 characters. These characters are English language alphabetic characters, decimal digits, and special characters.

There are 29 characters known as alphabetic characters. The alphabetic characters are the letters A through Z, the currency symbol (written \$), the commercial at-sign (written @) and the number sign (written #).

There are 10 digits, 0 through 9.

There are 21 *special characters*. The names and graphics by which they are represented are:

| Name | Graphic |
|-----------------------------|---------|
| Blank | |
| Equal or Assignment symbol | = |
| Plus | + |
| Minus | - |
| Asterisk or Multiply symbol | * |
| Slash or Divide symbol | / |
| Left Parenthesis | (|

| | |
|---|---|
| Right Parenthesis |) |
| Comma | , |
| Decimal Point or Binary Point or Period | . |
| Quotation mark | ' |
| Percent symbol | % |
| Semicolon | ; |
| Colon | : |
| Not symbol | ¬ |
| And symbol | & |
| Or symbol | |
| Greater Than symbol | > |
| Less Than symbol | < |
| Break character | — |
| Question mark | ? |

Delimiters

There are two classes of delimiters: separators and operators.

The separators, their use in PL/I, and the graphics by which they are represented are:

| Name of Separator | Graphic | Use |
|-------------------|---------|---|
| comma | , | separates elements of a list |
| semicolon | ; | terminates statements |
| colon | : | follows statement labels and condition prefixes |
| period | . | separates name qualifiers |
| parentheses | () | used in expressions and for enclosing lists and specifying information associated with certain keywords |

The operators are divided into four classes: arithmetic operators, comparison operators, logical operators, and the concatenation operator. The operators are identified below; their use is discussed in Chapter 5.

The arithmetic operators are:

| | |
|----|---|
| + | denoting addition or a positive quantity |
| - | denoting subtraction or a negative quantity |
| * | denoting multiplication |
| / | denoting division |
| ** | denoting exponentiation |

The comparison operators are:

| | |
|----|-----------------------------------|
| > | denoting greater than |
| ¬> | denoting not greater than |
| >= | denoting greater than or equal to |
| = | denoting equal to |
| ¬= | denoting not equal to |
| <= | denoting less than or equal to |

| | |
|----|------------------------|
| < | denoting less than |
| ¬< | denoting not less than |

The logical operators are:

| | |
|---|--------------|
| ¬ | denoting not |
| & | denoting and |
| | denoting or |

The concatenation operator is:

||

Identifiers

An identifier can be a word created by the user to identify a file, a data item, or all or any part of his program; or it can be one of the words used to identify entities in the PL/I language, such as statements, attributes, and options. This latter type of identifier is called a keyword. An example of a keyword would be the word DECLARE when it is written as the first word in a DECLARE statement, because, as the first word, it identifies the statement as a DECLARE statement.

Throughout the remainder of this publication, keywords will be printed entirely in capital letters; identifiers assigned by the programmer will be printed in small capital letters.

Words used as keywords *may* also be used by the programmer to identify files, data items, or all or any part of his program. For example, one could write a DECLARE statement that specified within it a data item named DECLARE.

All identifiers, whether used as keywords or not, must be written according to the following rules:

1. An identifier can be composed of alphabetic characters, digits, and the break character. An identifier must begin within an alphabetic character.
2. Any number of break characters (—) are allowed within an identifier; consecutive break characters are permitted.
3. Identifiers must be composed of not more than 31 characters.

Statements

In PL/I the words, delimiters, blanks, and comments that have been discussed up to now are used to form statements. Statements are the basic program elements used to construct a PL/I program. They are used for the description of data, for the actual processing of data, and for the control of the execution sequence of other statements. All statements in PL/I terminate with a semicolon. Except for the Assignment statement discussed in Chapter 5 and the Null statement discussed in Chapter 4, all statements in PL/I begin with a keyword.

PL/I and COBOL Comparison: Language Notation

The following discussion compares the language notations of PL/I and COBOL. In general, both languages employ similar notations: programmer-defined words use alphabetical characters and decimal digits; expressions consist of sequences of names, constants, and operators; keywords identify language elements; punctuation characters separate elements; statements have an English-like appearance. However, the notation rules of both languages do differ; the following list contains some of the more significant differences.

1. In both PL/I and COBOL, keywords have pre-assigned meanings. In COBOL, keywords are reserved for their intended purpose and cannot be used for other purposes. In PL/I, however, keywords are not reserved for special purposes and may appear wherever a programmer-defined word is permitted; for example, the keyword READ may be used in a PL/I program as the name of a file. In PL/I, different meanings for the same word are determined from context. This permits keywords to be created for new language features. It also avoids having to reprogram old source programs in which programmer-defined words might conflict with new keywords.

2. COBOL requires a programming form, PL/I does not. In PL/I, punctuation characters determine the significance and grouping of language elements. This

permits PL/I programs to be treated as long strings of characters and to be transmitted to a computer by means of almost any input medium.

3. In COBOL, blank characters must surround arithmetic operators; this is not required in PL/I. In PL/I, blank characters are only required between successive words that are not separated by special characters such as parentheses, operators, and punctuation characters.

4. COBOL restricts comments to the Procedure Division and requires that they be written in a NOTE statement. PL/I allows comments to appear throughout the entire program and permits them to be used wherever blank characters may appear (except in a character-string constant or in a picture specification; these language features are discussed in Chapter 2).

5. The COBOL character set consists of 51 characters; PL/I uses 60 characters.

The following points show some of the less significant differences:

1. PL/I terminates all statements with a semicolon; COBOL terminates statements with either a period, a comma, or a semicolon.

2. Programmer-defined words in COBOL must not be longer than 30 characters; in PL/I, the limit is 31 characters.

3. PL/I uses the break character (an underscore) within programmer-defined words to improve readability; COBOL uses the hyphen.

Chapter 2: Data Description

Introduction

The discussions that follow deal with the types of data that may be employed in a PL/I programming application. The characteristics of the various data types and the methods provided for organizing data into aggregations such as arrays are also discussed. However, the discussions in this chapter are restricted to the description and organization of data within the internal storage of a computer. The description of data as it appears on external storage media is included with the discussion of data transmission in Chapter 3.

Data Types

The instructions that are executed by a computer may be divided into three general categories: logical instructions, arithmetic instructions, and control instructions. Logical instructions manipulate sequences of bits and characters. Arithmetic instructions process numeric data. Control instructions determine the order in which other instructions are executed. Similar categories may be used to classify the operations that are provided by PL/I. For each category of PL/I operations, there is a corresponding type of data that is used by the operations. In PL/I, these types of data are called: string data, arithmetic data, and label data.

String Data

In PL/I, string data is used primarily in logical operations and consists of sequences of characters or bits. String data items may be divided into two general categories: character-string data items and bit-string data items.

Character-String Data Items

A character-string data item consists of a sequence of characters. In PL/I, the characters in a character-string item may be any of the characters allowed in a particular computer.

Character-string data items are used primarily in operations such as comparing, editing, and printing.

Bit-String Data Items

A bit-string data item consists of a sequence of bits, each of which represents a series of "on" or "off" conditions. An "on" condition is represented by a 1 bit,

and an "off" condition is represented by a 0 bit. Within a PL/I program, bit-string data items are frequently used in logical operations, the results of which are used for control purposes.

Arithmetic Data

Arithmetic data represents numeric information and is used in arithmetic operations. There are two kinds of arithmetic data items: decimal and binary, each of which may have either a fixed-point representation or a floating-point representation.

Decimal Data Items

A decimal data item represents numeric information and consists of a sequence of decimal digits. A unique bit pattern is defined for each decimal digit.

Binary Data Items

A binary data item represents numeric information and consists of a sequence of bits. Although a binary data item uses bits, it is not equivalent to a bit-string data item. A binary data item represents a numeric value; a bit-string data item represents a sequence of "on" or "off" conditions.

Fixed-Point and Floating-Point Formats

Decimal data items and binary data items may have either a fixed-point format or a floating-point format. A floating-point format often results in a more compact form than does a fixed-point format. This is generally the case when a large number of zeros is required to fix the location of the decimal or binary point in a fixed-point format. For example, the fixed-point decimal fraction .000000009 requires eight zeros to establish the location of the decimal point. In floating-point format, the zeros are not needed because the location of the decimal point is specified by an integer exponent appearing within the floating-point data item.

Label Data

Statement Labels

In a PL/I program, data processing operations are specified by means of statements. Statements may be given labels so that the statements may be referred to by other statements such as control statements, which

alter the sequence of statement execution. A label may be used as a data item in certain statements (the use of label data items is discussed in Chapter 4, "Program Structure"). The value of a label is the location in a program of the statement that the label identifies.

Names of Data Items

In a program, it is often necessary to use a name to identify data items to be processed. A name that identifies a data item is called a data name. Data names conform to the rules established in Chapter 1 for forming identifiers.

At any specific time during the course of program execution, a data name has a value; that is, it identifies a data item that represents a value.

The different data items that may be identified by the same data name must have the same data characteristics. These characteristics are associated with the data name by writing the data name and the associated characteristics in a DECLARE statement (see "DECLARE Statement" in this chapter).

Constants

In a PL/I program it is not always necessary to name every data item. Data items may actually appear within a PL/I program and, consequently, are immediately available for use in the program.

A data item that appears in a PL/I program is called a constant because the information it represents, that is, its value, cannot change. The characteristics of a constant are inherent in the representation of the constant. For example, in a PL/I program, the constant 98.6 is a data item that represents a numeric value. This constant specifies that the data item is a decimal data item with two digits to the left of the decimal point and one digit to the right of the decimal point.

For each type of data permitted in a PL/I program there is a corresponding type of constant available for use in the program.

The following discussion describes the types of constants that may appear in a PL/I program and the manner in which they are written. Following the discussion of constants, there is a discussion of the DECLARE statement and the way in which it is used to describe data.

Character-String Constants

Character-string constants may contain any character that can be recognized by a particular computer. Character-string constants are enclosed in single quotation marks. For example:

```
'$123.45'  
'JOHN JONES'  
'45.62'
```

The quotation mark is not part of the constant. If it is desired to represent a quotation mark as part of the constant, a double quotation mark must be used. For example:

```
'IT' 's'
```

Repetition of a character-string constant may be indicated by preceding the constant specification with a decimal integer (enclosed in parentheses) indicating the number of repetitions. For example:

```
(3)'*_**'
```

is equivalent to writing

```
'**_**_**_**_**'
```

Bit-String Constants

Bit-string constants consist of a sequence of the digits 1 or 0, enclosed in single quotation marks and immediately followed by the letter B. For example:

```
'0100'B
```

Repetition may be specified for bit-string constants in the same manner as for character-string constants. Thus,

```
(10)'1'B
```

is equivalent to writing

```
'1111111111'B
```

Decimal Fixed-Point Constants

A decimal fixed-point constant is composed of one or more digits (0 through 9) and may contain a decimal point. For example,

```
72.192  
.308  
1965
```

Note that decimal fixed-point constants are not enclosed in quotation marks.

Binary Fixed-Point Constants

A binary fixed-point constant is composed of one or more of the digits 1 or 0 and may contain a binary point (discussed above); it is followed immediately by the letter B. For example:

```
11011B  
11.1101B  
.001B
```

Note that binary fixed-point constants are not enclosed in quotation marks.

Decimal Floating-Point Constants

A decimal floating-point constant consists of a decimal fixed-point constant that is immediately followed by the letter E which, in turn, is followed by an optionally signed exponent. The exponent is a decimal integer and represents a power of ten which determines the actual location of the decimal point. For example, the decimal floating-point constant 123.45E+5 is equivalent in value to the arithmetic expression $123.45 + 10^5$; this expression is equivalent in value to the decimal fixed-point constant 12345000. Other examples of decimal floating-point constants are:

317.5E-16
32.E-5

Binary Floating-Point Constants

A binary floating-point constant consists of a binary fixed-point constant that is immediately followed by the letter E which, in turn, is followed by an optionally signed exponent; the exponent is immediately followed by the letter B. The exponent is a decimal integer and represents a power of two which determines the actual location of the binary point. For example, the binary floating-point constant .101E+9B is equivalent in value to the binary fixed-point constant 10100000B. Other examples of binary floating-point constants are:

1.1011E-3B
1111.E+20B
10101E5B

Label Constants

A label constant identifies a statement. A label constant conforms to the rules established in Chapter 1 for forming an identifier. In PL/I, a statement is labeled by writing a label constant immediately to the left of the statement and following the label constant. An example of a label constant is the word MESSAGE in the following:

```
MESSAGE:DISPLAY('END OF JOB');
```

MESSAGE is the label constant that identifies the DISPLAY statement (see "DISPLAY Statement" in Chapter 3; the use of label constants is discussed in Chapter 4, "GO TO Statement").

The DECLARE Statement

In a PL/I program, explicit descriptions of data characteristics are written in the form of statements. The DECLARE statement is used to describe named data as it is represented within the internal storage of a computer. Those properties that characterize a data

item are called attributes. The attributes of a named data item are specified by keywords. These keywords may appear with the name of a data item in a DECLARE statement. The general form of a DECLARE statement may be written:

```
DECLARE name-1 attributes, ..., name-n  
attributes;
```

Blank characters separate attributes. A comma follows the last attribute appearing with a name except at the end of the DECLARE statement in which case the semicolon is used.

The following discussion presents those attributes that may be used in a DECLARE statement to describe named data.

The attributes for named data may be divided into three categories. Each category corresponds to one of the three general data types that were discussed previously: arithmetic attributes, string attributes, and label attributes.

Arithmetic Attributes

Arithmetic attributes are provided by PL/I to describe data items that have a numeric value. Five of these are of interest to the commercial programmer. They are DECIMAL, BINARY, FIXED, FLOAT, and the precision attribute.

DECIMAL and BINARY Attributes

The DECIMAL and BINARY attributes specify that the data item named in the DECLARE statement is of either the decimal or the binary data type. (An additional discussion of these attributes appears later in this chapter; see "Default Attributes.")

FIXED and FLOAT Attributes

The FIXED and FLOAT attributes specify that the data item named in the DECLARE statement is represented in either a fixed-point (FIXED) or a floating-point (FLOAT) format. (An additional discussion of these attributes appears later in this chapter; see "Default Attributes.")

Precision Attribute

The precision attribute specifies the number of digits that are to be maintained in data items assigned to a data name. For fixed-point data it also specifies the location of the assumed decimal point.

The precision attribute consists of either a single decimal integer enclosed in parentheses, for example, (12), or two decimal integers, separated by a comma and enclosed in parentheses, for example, (8,3). The precision attribute must be immediately preceded in the DECLARE statement by one of the attributes:

DECIMAL, BINARY, FIXED, or FLOAT. Only blanks may intervene.

For a floating-point data item, only a single integer is written in the precision attribute. It indicates the number of digits appearing before the E in the data item. For a fixed-point data item, two integers are usually written; the first indicates the number of digits appearing in the data item, the second indicates the number of digits to the right of the decimal or binary point. If there are to be no digits to the right of the point, that is, the second integer is zero, only the first integer need be written. Thus, the attribute (6,0) is equivalent to the attribute (6).

Consider the following DECLARE statement:

```
DECLARE SALARY DECIMAL FIXED (7,2),  
ESTIMATE FLOAT DECIMAL (10),  
COUNTER FIXED (5) BINARY,  
MEAN BINARY (10) FLOAT;
```

A data item assigned to SALARY would be a decimal fixed-point data item, composed of seven digits, with two of these digits assumed to be to the right of the decimal point (for example, 78921.43).

A data item assigned to ESTIMATE would be a decimal floating-point data item, with ten digits preceding the E (for example, .4325437894E5).

A data item assigned to COUNTER would be a binary fixed-point data item, composed of five digits, with none of these digits assumed to be to the right of the binary point (for example, 11001.B).

A data item assigned to MEAN would be a binary floating-point data item, with 10 digits preceding the E (for example, 1000111011E-3B).

String Attributes

The string attributes specify either character-string data items or bit-string data items. These attributes are: CHARACTER and BIT.

CHARACTER Attribute

The CHARACTER attribute specifies that the data name in the DECLARE statement represents character strings. The attribute is written in the following form:

```
CHARACTER(length)
```

The length specification is a decimal integer constant that specifies the number of characters in the data items.

When the character string is of varying length, the following is written:

```
CHARACTER(length) VARYING
```

The length specification indicates the maximum number of characters in a varying-length character string.

The following statement:

```
DECLARE HEADER CHARACTER(80),  
TITLE CHARACTER(40) VARYING;
```

specifies that the character-string data item called HEADER is of fixed length and consists of 80 characters, and that the number of characters in the character-string data item called TITLE may vary from 0 to 40.

BIT Attribute

The BIT attribute specifies that the data item named in the DECLARE statement is represented as a bit string consisting of a certain number of bits. The BIT attribute may appear in two forms:

```
BIT(length)  
BIT(length) VARYING
```

The length specification and VARYING attribute have the same meaning as described for the attribute CHARACTER, except that "length" indicates the number of bits in the bit-string data item.

PICTURE Attribute

The PICTURE attribute is used to specify the detailed characteristics of string data items and frequently is used to edit the format of character-string data items that are to be printed. The editing of a character-string data item may involve replacing certain characters, such as leading zeros, with other characters, and inserting, within the character string, characters such as the dollar sign and the period.

The PICTURE attribute is written in the following way:

```
PICTURE 'picture-specification'
```

As indicated, the picture specification must be enclosed in quotation marks; it consists of a sequence of characters called picture characters. The following discussion deals only with those picture characters that allow the PICTURE attribute to serve as an alternative for the arithmetic attributes and for the string attributes; these characters are: A, X, 9, V, K, S. The remaining picture characters are for editing (see Chapter 5, "Data Editing").

Although the PICTURE attribute may serve as an alternative for the arithmetic attributes and, thereby, specify the representation of a numeric value, the data item described by the PICTURE attribute is a character-string data item and not an arithmetic data item. Consequently, the efficiency of computer operations may be affected when performing arithmetic calculations on data items described by the PICTURE attribute. In most computers, this lack of efficiency results from being unable to perform arithmetic calculations directly on character-string data items. Conversion from character string representation to arithmetic representation is required before arithmetic calcula-

tions are performed on data items described by a PICTURE attribute (see Chapter 5, "Conversion of Arithmetic Data").

Fixed-Point Decimal Attributes with PICTURE

The picture characters 9 and V may be used to specify a character-string data item that has fixed-point decimal attributes. A sequence consisting of the picture character 9 in a picture specification indicates that corresponding character positions in the character-string data item always contain decimal digits (0 through 9). The character V specifies that a decimal point should be assumed at the corresponding position in the character-string data item; however, no decimal point is actually present in the character string. The character V may appear only once in a picture specification. If no V character is used with a sequence consisting of the picture character 9, a decimal point is assumed at the right-hand end of the character-string data item. Consider the following DECLARE statement:

```
DECLARE COUNT PICTURE '999',
        COST PICTURE '999V99',
        TAX PICTURE 'V999';
```

This statement describes the characteristics of the three character-string data items named COUNT, COST, and TAX. The character string data item named COUNT consists of three decimal digits; a decimal point is assumed at the right-hand end. COST identifies a character-string data item that contains five decimal digits; a decimal point is assumed before the two rightmost digits. There are three decimal digits in the character-string data item named TAX; a decimal point is assumed at the left-hand end.

Floating-Point Decimal Attributes with PICTURE

Floating-point decimal attributes may be specified for a character-string data item by using the picture characters 9, V, K, and S. The picture specification for floating-point decimal attributes consists of two parts. The first part contains the picture characters for a fixed-point decimal data item; the second part is immediately to the right of the first part and represents the floating-point exponent. The picture specification for a floating-point decimal exponent begins with the letter K and is followed by the optional character S, after which a sequence of the character 9 appears. The character K does not represent an actual character in the character-string data item; it specifies the beginning position of the exponent in the character-string. The character S indicates that either a + or a - sign appears in the corresponding position of the

character-string data item and specifies the sign of the exponent. When S is not used, the exponent is positive. Consider the following statement:

```
DECLARE AVERAGE PICTURE 'V999KS99';
```

The character-string data item named AVERAGE consists of six characters. The three rightmost characters in the data item represent a signed exponent; the three leftmost characters are decimal digits with a decimal point assumed at the left.

Character-String Attributes with PICTURE

The picture characters X and A may be used to specify character-string data items. In a picture specification, the character X indicates that the corresponding character position in a character-string data item may contain any character that can be represented in the computer. The picture character A is similar to X, except that it is used to specify only letters of the alphabet and the blank character. Consider the following statement:

```
DECLARE NAME PICTURE 'AAAAA',
        SYMBOL PICTURE 'XXXXXXXXXX',
        CODE PICTURE 'AAXXX';
```

This statement describes the characteristics of the three character-string data items called NAME, SYMBOL, and CODE. The character-string data item identified by NAME consists of five characters, each of which may be any letter of the alphabet or a blank character. SYMBOL names a character-string data item consisting of 10 characters, each of which may be any character that can be represented in the computer. There are five characters in the character-string data item named CODE; each of the first two characters may be any letter of the alphabet or a blank character; each of the last three characters may be any character that can be represented in the computer.

Repetition of Picture Characters

Successive occurrences of the same character in a picture specification may be indicated by placing a decimal integer in parentheses before the character to be repeated. The value of the decimal integer specifies the number of repetitions. For example, the statement:

```
DECLARE GROSS PICTURE '(7)9V99',
        PART PICTURE '(6)A(5)X(2)9',
        FRACTION PICTURE 'V(8)9';
```

is equivalent to the statement:

```
DECLARE GROSS PICTURE '9999999V99',
        PART PICTURE 'AAAAAAXXXXX99',
        FRACTION PICTURE 'V99999999';
```

Exclusivity of Attributes

The following rules apply to the attributes described thus far:

BIT, CHARACTER, and PICTURE must not be used to describe the same data item. An item described with BIT, CHARACTER, or PICTURE must not also be described with FIXED, FLOAT, BINARY, or DECIMAL.

The DEFINED Attribute

It is often convenient to be able to refer to the same data item by different names, particularly when several programmers are involved with the same program and each is using a different name for the same data item. The DEFINED attribute may be used for that purpose; it is written in the following form:

```
new-name attributes DEFINED old-name
```

Consider the following statement:

```
DECLARE TITLE CHARACTER(80),
        HEADER CHARACTER (80) DEFINED TITLE;
```

This statement specifies that the data item identified by TITLE is a character-string data item consisting of 80 characters and that the same data item may be identified by the name HEADER. Note that the attributes for the data item are specified for the new name, even though they are identical to the attributes specified for the old name.

The DEFINED attribute may also be used with arithmetic data items, as illustrated by the following statement:

```
DECLARE SALARY FIXED DECIMAL(5,2),
        PAY FIXED DECIMAL (5,2) DEFINED
        SALARY;
```

This statement specifies that SALARY is the name of an arithmetic data item and that the same data item may be referred to by the name PAY.

In the case of a string data item, it is also possible to use the DEFINED attribute to apply a data name to a portion of a string. This is achieved by using a character-string attribute with the new name to specify the number of characters in the string to which the new name applies. For example, the following statement:

```
DECLARE DATE CHARACTER(6),
        MONTH CHARACTER(2) DEFINED DATE;
```

specifies that DATE is the name of a character-string data item consisting of six characters and that the first two characters of that character string are named MONTH.

When the new name applies to a portion of a string data item that does not begin with the first character in a character string or with the first bit in

a bit string, the DEFINED attribute is written with the following attribute:

```
POSITION (decimal-integer-constant)
```

The decimal integer constant in the POSITION attribute specifies the starting position of that portion of the string data item to which the new name refers. When the starting position is the first position of the string, the POSITION attribute is not required. Consider the following statement:

```
DECLARE DATE CHARACTER(6),
        MONTH CHARACTER(2) DEFINED DATE,
        DAY CHARACTER(2) DEFINED DATE
        POSITION (3),
        YEAR CHARACTER(2) DEFINED DATE
        POSITION(5);
```

This statement specifies that the character-string data item named DATE consists of six characters. The statement also specifies that the first two characters of the character string named DATE are a character string named MONTH, that the third and fourth characters are a character string named DAY, and that the fifth and sixth characters are a character string named YEAR.

The attributes appearing with the new name in a DEFINED attribute must be consistent with the attributes declared for the old name; for example, the new name cannot employ the BIT attribute when the old name employed the CHARACTER attribute. Furthermore, the POSITION attribute must specify a starting position that is consistent with the length indicated by the attributes of the new name. Consider the following statement:

```
DECLARE PART CHARACTER(10),
        MODEL CHARACTER(4) DEFINED PART
        POSITION(8);
```

This statement contains an inconsistency. The attribute POSITION(8) indicates that the name MODEL refers to a character string consisting of the eighth, ninth, tenth, and eleventh characters of the character string named PART. However, the character string named PART consists of 10, not 11, characters; therefore, the above statement is incorrect.

Default Attributes

In PL/I, there need not exist an explicit description in a DECLARE statement for every name in a program. The characteristics of some data names are understood from the context in which they appear and need not be detailed. The characteristics of other data names may be only partially described; certain attributes may be specified for these partially described data names and other attributes may be omitted.

When a data name is not explicitly described or when it is partially described, certain attributes are assumed to apply to the data name. These are called 'default' attributes.

The following is a set of default assumptions made:

1. When no explicit description of a data name has been given, if the name begins with the letters I through N, it is assumed to have the attributes **FIXED BINARY**; if the name begins with a letter other than I through N, it is assumed to have the attributes **FLOAT DECIMAL**.

2. If a name has the attribute **BINARY** or **DECIMAL**, it is assumed also to have the attribute **FLOAT**, unless otherwise specified.

3. If a name has the attribute **FIXED** or **FLOAT**, it is assumed also to have the attribute **DECIMAL**, unless otherwise specified.

4. The default specification of precision will be separately defined for each implementation of PL/I.

Data Aggregates

In PL/I, data items may be grouped together to form data aggregates. Two kinds of aggregates are provided by the language: structures and arrays. Aggregates or parts of aggregates may be referred to by a single name.

Structures

Data items that neither have identical sizes nor are of the same data type, but that possess a logical relation to one another, may be grouped into a hierarchy called a structure. The general concept of a structure is evident in many areas other than computing. The organization of a book illustrates one application of the structure concept. A book may be divided into several parts, and each part may consist of one or more chapters. The chapters may comprise several main topics, and each main topic may be composed of subtopics, etc. This hierarchy is usually shown in a table of contents, using indentations and different type faces, so that the organization of the book is evident at a glance.

Data in a PL/I program may be organized into a hierarchy much like that of a book. Consider a data structure that contains, as one item, an address. The address might be given the data name **ADDRESS**. However, if a programmer has to refer to the individual parts of the address, he has to name them. For example, **STREET**, **CITY**, **ZONE**, and **STATE**. In this case, **ADDRESS** is at a higher level than **STREET**, **CITY**, **ZONE**, and **STATE** and includes each of them. Once a subdivision of a data item has been specified, it may be further subdivided to permit more detailed references. The most basic subdivisions of a data structure, that

is, those that have not been further subdivided, are called elementary data items.

A system of level numbers is employed in PL/I to specify the organization of elementary data items into structures and, in turn, the organization of structures into more inclusive structures. Level numbers start at 1 for major structures; that is, structures not contained in other structures. Less inclusive structures, that is, minor structures, are assigned higher (not necessarily successive) level numbers. In general, the higher the value of the level number, the lower is its hierarchical level.

When a structure is described in a **DECLARE** statement, the level number associated with a data name must appear immediately before the data name. Attributes of elementary items in a structure appear after the name with which they are associated.

Note that the data names specified for all but the elementary items in a structure are the names of levels. A reference to such a name in the body of a program is, in fact, a reference to the elementary items subordinate to that name.

The effect of levels is illustrated in the following example.

```
DECLARE
1 CHECK_ACCOUNT_RECORD,
  2 NAME,
    5 LAST CHARACTER (15),
    5 FIRST CHARACTER (10),
    5 MIDDLE CHARACTER (10),
  2 ACCOUNT_NUMBER CHARACTER (9),
  2 ADDRESS,
    4 STREET CHARACTER (20),
    4 CITY CHARACTER (15),
    4 ZONE CHARACTER (5),
    4 STATE CHARACTER (15),
  2 BALANCE FIXED (7,2);
```

This is the description of a record in a master file containing the checking accounts of a bank. Each record contains information pertaining to one checking account. In the foregoing example, **NAME** is assigned the level 2. Immediately following are entries describing **LAST**, **FIRST**, and **MIDDLE**. These three entries are identified as being contained in **NAME** because they follow **NAME** and no other data name with a level number equal to or lower than that of **NAME** has intervened and because they have level numbers higher than that of **NAME**.

ADDRESS is also assigned the level 2. Immediately following are entries describing **STREET**, **CITY**, **ZONE**, and **STATE**. These four entries are identified as being contained in **ADDRESS** in the same way that **LAST**, **FIRST**, and **MIDDLE** are identified as being contained in **NAME**.

Consider another example. This is one portion of a payroll record:

```
DECLARE
  1 PAYROLL_RECORD,
    3 MAN_NUMBER CHARACTER (6),
    3 NAME,
      27 LAST CHARACTER (15),
      27 FIRST CHARACTER (15),
      27 MIDDLE CHARACTER (10),
    3 ADDRESS,
      12 STREET CHARACTER (20),
      12 CITY CHARACTER (15),
      12 ZONE CHARACTER (5),
      12 STATE CHARACTER (15),
    3 DATE_HIRED,
      4 MONTH FIXED (2),
      4 DAY FIXED (2),
      4 YEAR,
        14 DECADE FIXED (1),
        14 YR FIXED (1),
    3 RATE_OF_PAY FIXED (7,2);
```

The principal rules for assigning level numbers are illustrated in this example and may be summarized as follows:

1. Level 1 is reserved to identify a major structure.
2. A level with the name B is contained in a level with the name A if all of the following conditions are met:
 - a. B follows A.
 - b. B has been assigned a level number higher than that assigned to A.
 - c. A name with a level number equal to or lower than that for A does not appear between A and B. Thus, even though DECADE has a higher level number than STATE, it is not part of STATE, because data names with level numbers lower than 12 (for example, DATE_HIRED) appear between STATE and DECADE.
3. Level numbers need not be assigned consecutively. Thus, MONTH, DAY, and YEAR could be assigned any level number higher than that assigned to DATE_HIRED and lower than that assigned to DECADE.
4. A structure may include more than one level. However, all data names that make up a level within the same structure (for example, MONTH, DAY, and YEAR) must have the same level number.
5. When a data name is to have a lower level number than the one immediately preceding it, the level number must be chosen from the level numbers of the structures that include the preceding data name. Thus, ADDRESS must be assigned level 3, because that is the only level number

assigned to a structure that contains the preceding data name, MIDDLE. The data name ADDRESS could not be assigned the level number 1 because it is not a major structure. RATE_OF_PAY could be assigned one of two level numbers, 3 or 4, because the data name YR is contained in two structures, one with the level number 4 and the other with the level number 3. However, if it were assigned the level number 4, it would be treated as part of the structure called DATE_HIRED, which would probably not be the programmer's intent.

Qualified Names

When specifying names of elementary items or of minor structures, it is often convenient to use the same data name for items in different parts of the structure or to use the same data name for items in two different structures. For example, two major (level 1) structures called CARDIN and CARDOUT could each have an elementary item called PARTNO. If the item named PARTNO were referred to in the body of a PL/I program, it would be a non-unique reference; that is to say, it would not be clear whether PARTNO was the elementary item in the structure CARDIN or the elementary item in the structure CARDOUT. Therefore, whenever a non-unique data name is used, it must be associated with one or more of the names in the structure containing it in order to make the reference to it unique.

In PL/I, making a name unique is called qualification.

The name of an elementary item or of a minor (not level 1) structure is qualified by preceding it with the name of the structure to which the item belongs. These names are arranged, from left to right, in the order of increasing level number and are separated from one another, and from the name they qualify, by periods. The periods may be surrounded by blanks. The name of a major structure cannot be qualified.

For example, consider the following structures:

```
DECLARE
  1 CARDIN, 2PARTNO, 2DESCRIPTION,
  1 CARDOUT, 2PARTNO, 2DESCRIPTION;
```

The elementary data items can be referred to by the following qualified names:

```
CARDIN.PARTNO
CARDIN.DESCRPTION
CARDOUT.PARTNO
CARDOUT.DESCRPTION
```

A name need be qualified by only those structure names that make the name unique. Consider the following structure:

```

DECLARE
  1 MARRIAGE, 2 MAN, 3 NAME, 3 DATE,
    2 WOMAN, 3 NAME, 3 DATE;

```

This structure may have its elementary data items referred to as:

```

MAN.NAME OF MARRIAGE.MAN.NAME
WOMAN.NAME OF MARRIAGE.WOMAN.NAME
MAN.DATE OF MARRIAGE.MAN.DATE
WOMAN.DATE OF MARRIAGE.WOMAN.DATE

```

If the following structure:

```

DECLARE
  1 BIRTH, 2 WOMAN, 3 NAME, 3 DATE;

```

appears in the same program with the above structure, then the elementary data items in both structures may be referred to as:

```

MAN.NAME OF MARRIAGE.MAN.NAME
MARRIAGE.WOMAN.NAME
BIRTH.NAME OF BIRTH.WOMAN.NAME
etc.

```

The minor structures in this case are referred to as:

```

MARRIAGE.MAN
MARRIAGE.WOMAN
BIRTH.WOMAN

```

Arrays

Data items having the same characteristics, that is, of the same data type and of the same size (though not necessarily having the same value), may be grouped to form an array.

The simplest form of array is a sequence of data items. Such an array would be analogous to a structure with only one minor level. Thus, a structure consisting of 50 elementary items, the value of each item being the abbreviated name of 1 of the 50 states, could be specified by writing the following:

```

DECLARE
  1 STATE,
    2 ALABAMA CHARACTER(4),
    .
    .
    .
    2 WYOMING CHARACTER(4);

```

A simpler way of writing this grouping of data would be to specify it as an array of 50 elements. Each element would have to have the same size and be of the same data type as the others.

Such an array may be specified by immediately following the name of the array with a parenthesized decimal integer representing the number of elements in the array. This specification is then followed by the attributes of the items in the array. Thus, in the foregoing example, the major structure STATE and the elementary items under it could be specified as follows:

```

DECLARE STATE(50) CHARACTER(4);

```

Consider the following structure description:

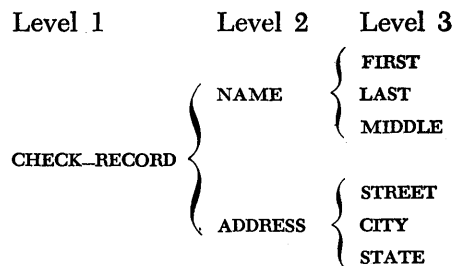
```

DECLARE
  1 CHECK_RECORD,
    2 NAME,
      3 LAST CHARACTER (15),
      3 FIRST CHARACTER (15),
      3 MIDDLE CHARACTER (15),
    2 ADDRESS,
      3 STREET CHARACTER (15),
      3 CITY CHARACTER (15),
      3 STATE CHARACTER (15);

```

Because all the elementary items in CHECK_RECORD have the same size and are of the same data type, this structure could be specified as an array. As an array, it would consist of only elementary items. These elementary items would be divided into two groups of three items each.

The grouping of items within the structure CHECK_RECORD could be visually represented as follows:



Such an array would be specified in a DECLARE statement as follows:

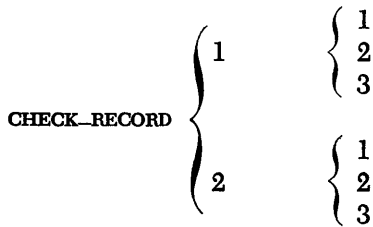
```

DECLARE CHECK_RECORD(2,3)
  CHARACTER(15);

```

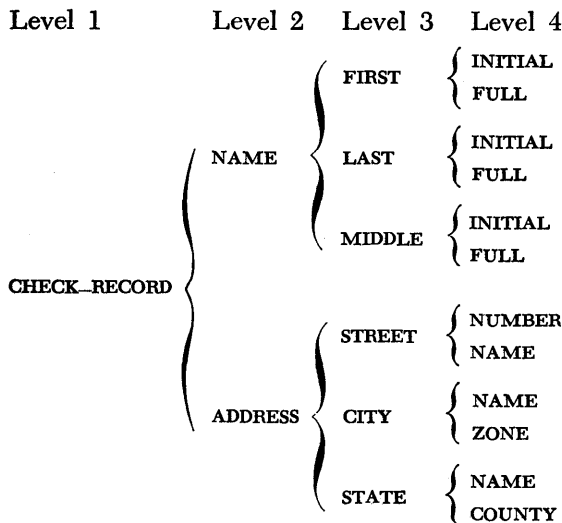
The parenthesized integers following CHECK_RECORD indicate the way items in this array are grouped. The first integer (2) indicates that there are two major groups of items. The second integer (3) indicates that each major group contains three items. Note that the structure CHECK_RECORD could not have been specified as an array if each major group had had a different number of items in it.

The grouping of items within the array CHECK_RECORD could be visually represented as follows:

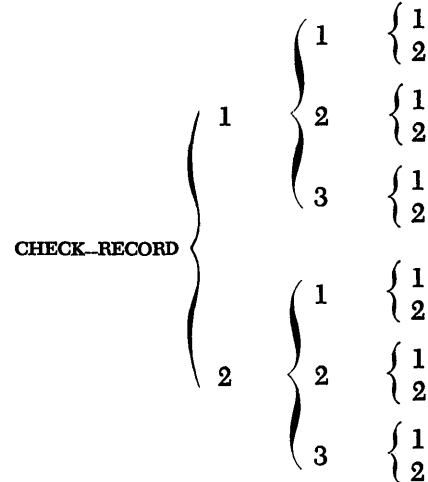


The way in which items are grouped in an array is analogous to the levels used in specifying a structure. Thus, in the example above, the groups of three items in the array are analogous to level 3 in the structure; the group of two items in the array is analogous to level two; and the single item CHECK_RECORD which is the name of the array is analogous to level 1 in the structure.

Just as the structure CHECK_RECORD could be expanded to contain another level, so also the array CHECK_RECORD could be expanded to contain another grouping. Consider the following expansion of the structure CHECK_RECORD.



This structure could be expressed as an array, thus:



This array would be specified in a DECLARE statement as follows:

```

DECLARE CHECK_RECORD(2,3,2)
CHARACTER(15);

```

The parenthesized integers following CHECK_RECORD would indicate that there were two major groups in the array; that each major group was divided into three minor groups and that each minor group contained two elementary items (a total of twelve elementary items). Each set of groups is called a dimension. In this example, there is a set of major, a set of minor, and a set of elementary groups; that is, a major, minor, and an elementary dimension.

In PL/I, arrays may have any number of dimensions. Thus, in the example above, the array CHECK_RECORD (2, 3, 2) could be expanded beyond the three dimensions specified by the parenthesized integers.

Expressions can be used instead of integers to indicate the number of items in a dimension. For example, one could write:

```

DECLARE TABLE (A+B, 2, 4);

```

The number of groups in the first dimension would be equal to the integral value of the sum of A and B (see expressions discussed in Chapter 5, "Assignment Statements").

Subscripting

A qualified name is used to refer to a minor structure or to an elementary item in a structure. A subscripted name is used to refer to an item in an array. Consider an array specified as follows:

```
DECLARE CHECK_RECORD(2,3)
CHARACTER(15);
```

In order to refer to the second item in this array, one would write `CHECK_RECORD (1,2)`. Each integer in the specification (1,2) is called a subscript. The first integer refers to the first (or major) grouping in the array; the second integer refers to the second item within the first grouping. The first of several subscripts always represents one of the groups in the first dimension in an array; the second of several subscripts represents one of the groups in the next most significant dimension in the array, and so on; the last of several subscripts represents the position of the elementary item within the least significant dimension in the array. One subscript is separated from another by a comma and all subscripts for a particular reference are enclosed in parentheses. For example, the items in the array `CHECK_RECORD` and the items to which they refer are:

- `CHECK_RECORD(1,1)` the first item in the first grouping
- `CHECK_RECORD(1,2)` the second item in the first grouping
- `CHECK_RECORD(1,3)` the third item in the first grouping
- `CHECK_RECORD(2,1)` the first item in the second grouping
- `CHECK_RECORD(2,2)` the second item in the second grouping
- `CHECK_RECORD(2,3)` the third item in the second grouping

A subscript need not be represented by a constant; it can be represented by a data name or by an expression. Thus the following subscripted names could be written:

```
CHECK_RECORD (A,2)
CHECK_RECORD (A,B)
CHECK_RECORD (A+1,2)
```

In the first example above, the value of `A` would be used as the value of the first subscript. In the second example, the value of `A` would be used as the value of the first subscript, the value of `B` as the value of the second subscript. In the third example, the value of the expression `A+1` (the sum of 1 and the value of `A`) would be used as the value of the first subscript.

Structures Containing Arrays

In PL/I, an item in a structure may be an array. For example, consider the following structure specification (written here without data attributes):

```
ROLL:DECLARE
1 CARDIN,
2 NAME,
2 WAGES(2),
3 REGULAR(3),
3 OVERTIME;
```

The parenthesized specification (2) indicates that the structure `WAGES` is an array consisting of two groups. Each group contains the items `REGULAR` and `OVERTIME`. The item `REGULAR` is also an array consisting of three items. The meaning of this structure may be described by an equivalent structure that does not contain arrays:

```
DECLARE
1 CARDIN,
2 NAME,
2 WAGES_1,
3 REGULAR_1,
3 REGULAR_2,
3 REGULAR_3,
3 OVERTIME,
2 WAGES_2,
3 REGULAR_1,
3 REGULAR_2,
3 REGULAR_3,
3 OVERTIME;
```

A combination of qualifying and subscripting may be used to refer to an item in an array contained in a structure. Thus, in order to refer to the second element of `REGULAR` in the first group of `WAGES` in the `ROLL` structure specification, one could write:

```
CARDIN.WAGES(1).REGULAR(2)
```

In the foregoing example, subscripts are written following the structure names to which they apply. This is the normal form of writing a combination of qualification and subscripting. As long as the order of the subscripts remains unchanged, subscripts may be moved to the right of the names to which they normally apply.

For example, the subscripted qualified name above could be written:

```
CARDIN.WAGES.REGULAR(1,2)
```

ALIGNED and PACKED Attributes

The ALIGNED and PACKED attributes are used only for structures or arrays composed of string data. They specify the arrangement in core storage of all of the character-string or bit-string elements that compose a particular array or structure. The PACKED attribute indicates that there is to be no unused core storage between any two elements of an array or between any two bit-string elements or any two character-string elements of a structure. The ALIGNED attribute indicates that the elements of an array or structure are to start at a particular core storage boundary and that there may, consequently, be unused core storage between the elements. The ALIGNED attribute often results in more efficient processing, particularly when a great deal of packing and unpacking of data would be required. (The storage boundary is defined individually for each implementation.)

When these attributes are specified, they apply to all the elements in a major structure or in an array. When they are written for a structure, they must be written for the data name level number 1. When they are written for an array, that array may not be part of a structure.

These attributes may be written in a DECLARE statement as follows:

```
DECLARE ISTRUCTURE PACKED, 2SUBSTR1
      BIT (4), 2SUBSTR2 BIT (5);
DECLARE ARRAY (4,2,4) ALIGNED;
```

LIKE Attribute

The LIKE attribute specifies that the name being declared is a structure with a substructure having elements with attributes and names identical to the names and attributes of the elements of the named structure. The attribute has the following format:

LIKE structure-name

Consider, for example, the major structures A and x, described with the following DECLARE statement.

```
DECLARE 1A,
      2FIELD1,
      3DTL1 PICTURE 'AA',
      3DTL2 CHARACTER (10),
      2FIELD2 CHARACTER (12),
1x,
      2FIELD1,
      3SUBFIELD1 LIKE A.FIELD1,
      2FIELD2 LIKE A.FIELD1
      2FIELD3 CHARACTER (5);
```

This specification of a major structure x is equivalent to writing:

```
1x,
      2FIELD1,
      3SUBFIELD1,
      4DTL1 PICTURE 'AA',
      4DTL2 CHARACTER (10),
      2FIELD2,
      3DTL1 PICTURE 'AA',
      3DTL2 CHARACTER (10),
      2FIELD3 CHARACTER (5);
```

The following should be noted:

1. The difference between the level number of the structure name following LIKE and the level numbers of items subordinate to structure name is maintained.
2. The names and attributes of items subordinate to the structure name become subordinate to the item with the LIKE attribute.
3. The level numbers of any items following the item with the LIKE attribute must be less than or equal to the level number of the item with the LIKE attribute.
4. The structure name following LIKE may be qualified but may not be subscripted.
5. Neither the structure name following LIKE, nor any item subordinate to it may be described with the LIKE attribute.

LABEL Attribute

The keyword LABEL may be used to specify that a name is a label name, representing a value that is a label constant. For example, the statement:

```
DECLARE POINTER LABEL;
```

specifies that the value of the label name POINTER is a label constant.

A label name may also identify an array of label constants. Consider the following statement:

```
DECLARE SWITCH (10) LABEL;
```

This statement specifies that SWITCH is the name of a 10-element array and that each element of the array is a label constant. The use of label names is discussed in Chapter 4, "GO TO Statement."

INITIAL Attribute

The INITIAL attribute specifies constants that are assigned to data names when computer storage is allocated (see "Storage Allocation" in Chapter 4). The INITIAL attribute may be written in the following form:

```
INITIAL (constant)
```

A constant appearing in the INITIAL attribute must conform to the rules established earlier in this chap-

ter for writing constants. Consider the following statement:

```
DECLARE MAXIMUM FIXED DECIMAL (4)
  INITIAL (1500);
```

This statement specifies that the data item identified by MAXIMUM is a fixed-point decimal integer consisting of four digits, and that the data item assigned to MAXIMUM, when computer storage is allocated, is equivalent to the constant 1500.

In the following statement:

```
DECLARE NAME CHARACTER (4)
  INITIAL ('JOHN'),
  SWITCH LABEL INITIAL (DEDUCTIONS);
```

the initial character-string data item assigned to NAME is the constant 'JOHN', and the initial constant assigned to the label name SWITCH is the label constant DEDUCTIONS.

The INITIAL attribute may also be used to assign constants to arrays. When used with arrays, the INITIAL attribute may be written in the following form:

```
INITIAL (constant-1, ..., constant-n)
```

Constants are assigned in successive order to successive positions of an array. Consider the following statement:

```
DECLARE PRICES (4) FIXED DECIMAL (4,2)
  INITIAL (10.99, 20.49, 75.39, 99.99),
  POINTER (3) LABEL
  INITIAL (OVERTIME, COMMISSION, BONUS);
```

In this statement, PRICES is declared to be a one-dimensional array that contains four fixed-point decimal data items. The initial data items assigned to the PRICES array are specified by the constants: 10.99, 20.49, 75.39, and 99.99. The one-dimensional array named POINTER contains three labels that are initially specified by the label constants: OVERTIME, COMMISSION, and BONUS.

Factoring of Attributes

In the DECLARE statement, one or more attributes can be associated with a set of names by a method known as factoring. Factoring is accomplished by grouping a set of names in parentheses before the associated attribute or attributes. Commas are used to separate the members of each set. For example, the statement:

```
DECLARE (YEAR, MONTH, DAY) CHARACTER
  (2);
```

associates the attribute CHARACTER of length 2 with the items named YEAR, MONTH, and DAY.

Attributes that apply to only one member of a factored set are written following that member. For example:

```
DECLARE (YEAR INITIAL (62), MONTH, DAY)
  CHARACTER (2);
```

A set of factored items and their associated attributes may themselves be factored. For example:

```
DECLARE ((YEAR, MONTH, DAY) CHARACTER
  (2), (HOUR, MINUTE) FIXED (2)) INITIAL
  (00);
```

Level numbers may be factored. When this is the case, the names associated with the level number appear in a parenthesized set following the level number. For example:

```
DECLARE 1RECORD, 2(NAME, 3(FIRST, MIDDLE,
  LAST), AGE, SALARY) CHARACTER(12);
```

PL/I and COBOL Comparison: Data Description

The following discussion compares the data description features of PL/I and COBOL. In general, both languages use similar methods for describing the characteristics of data items stored within a computer: programmer-defined words are used to name data items; keywords specify the characteristics of named data items; data items may be collected into aggregates; constants may be specified for each data type; data names may be assigned initial values; and a picture specification may serve as an alternative method for describing data. There are differences, however, between the data description features of both languages; the following points contain some of the more significant differences.

1. COBOL describes data in the Data Division; PL/I uses the DECLARE statement. The COBOL Data Division contains separate sections for different kinds of data; PL/I does not require a separation of the various data types in a DECLARE statement.
2. COBOL requires all programmer-supplied words to be defined in the Data Division. PL/I allows programmer-supplied words to be used in a program without being defined in a DECLARE statement; the meaning of such words is determined from context and a complete set of default rules is used to determine unspecified data characteristics.
3. In COBOL, the description of data on external storage media is specified in the Data Division. In PL/I, the input/output statements specify the description of externally stored data (see Chapter 3, "Input/Output").

4. Bit strings and label data are PL/I data types not contained in COBOL.
5. COBOL uses figurative constants; PL/I does not.
6. PL/I imposes no limit on the number of dimensions in an array or on the number of levels in a structure. COBOL limits a table (the COBOL equivalent of a PL/I array) to a maximum of

three dimensions, and a group (the COBOL equivalent to a PL/I structure) to a maximum of 49 levels.

7. The order of name qualifiers in COBOL is the reverse of the order used in PL/I. COBOL uses the keywords IN and OF as qualification operators; PL/I uses the period.

Chapter 3: Input/Output

Introduction

Before a computer can process data that is recorded on external storage media, the data must be reproduced within the computer; this reproduction process is called input. Likewise, when processing has been completed, the processed data is made available by reproducing it on external media; this reproduction process is called output. This chapter discusses two principal types of input/output.

External Storage Attributes

The discussions in the preceding chapter have dealt with the description of data items that are stored within a computer. However, most computer applications are also concerned with the representation of data on external storage media. It is by means of such media that data is presented to and received from a computer. When data is recorded on external media, it is organized into collections called files. For example, a collection of time cards may constitute a file in a payroll application. On magnetic tape, a file may consist of data that will eventually be used to produce a printed report.

A computer transmits data to and from a file by means of input/output statements. However, when attention is focused primarily on the data being transmitted, the environment characteristics of a file are of little concern. PL/I permits a file to be given a name and allows the characteristics of a file to be described with keywords called file attributes.

The following discussion presents the file attributes that are provided by PL/I.

FILE Attribute

The FILE attribute indicates that the associated identifier is a file name. In the following statement:

```
DECLARE MASTER FILE;
```

the identifier `MASTER` is declared to be a file name.

STREAM and RECORD Attributes

The STREAM and RECORD attributes describe the manner in which data in a file is to be treated. They indicate the type of data transmission (stream oriented or record oriented) that can be used in input/output operations for the file. (see "Data Transmission")

The STREAM attribute describes a file considered as one continuous stream of data items in character form.

The RECORD attribute describes a file containing a number of physically separate records, each consisting of one or more data items in any form.

INPUT, OUTPUT and UPDATE Attributes

One of these attributes (INPUT, OUTPUT, or UPDATE) may be specified to describe the type of data transmission that is permitted for a file. The INPUT attribute is used for files that are only to be read. The OUTPUT attribute is used for files that are to be created; OUTPUT files may only be written on. The UPDATE attribute is used for existing files that are to be read, or have new records added, or existing records altered or deleted. In the following statement:

```
DECLARE DETAIL FILE INPUT,  
REPORT FILE OUTPUT,  
MASTER FILE UPDATE;
```

DETAIL is the name of an input file; REPORT is the name of an output file; MASTER is the name of a file that is used both for input and for output.

PRINT Attribute

The PRINT attribute indicates that the data in a file will eventually be printed, that is, appear as output on a printed page.

The PRINT attribute may be declared only for an output file. A file with the RECORD attribute may not have the PRINT attribute.

BUFFERED and UNBUFFERED Attributes

The BUFFERED and UNBUFFERED attributes apply only to files that have the RECORD attribute. The BUFFERED attribute indicates that the data in a record being transmitted to and from a file is to be placed into a buffer (that is, an intermediate storage area). The UNBUFFERED attribute indicates that the data is to be assigned directly to and from a location specified by a data name.

ENVIRONMENT Attribute

The ENVIRONMENT attribute is used to specify the physical characteristics of a file that depends on the computer for which a PL/I program is written. The attribute is written in the following form:

ENVIRONMENT (option-list)

The option list will be defined individually for each PL/I compiler. Information such as file media, physical record format, buffering, and file disposition may be specified with the option list.

Opening and Closing of Files

Before a file is processed by a data transmission statement, certain file preparation activities must occur, such as checking for the availability of external storage media, positioning the media, and allocating appropriate programming support. This activity is known as opening a file. Similarly, when processing is completed, the file must be closed. Closing a file consists of releasing the facilities that were established during the opening of the file. PL/I provides two statements, OPEN and CLOSE, to perform these functions. These statements, however, are optional. If an OPEN statement is not executed for a file, the file is opened automatically when the first data transmission statement for that file is executed; in this case, the automatic file preparation consists of standard system procedures and uses information about the file contained in a DECLARE statement. Similarly, if a file has not been closed before completion of a program, the file is closed automatically upon completion of the program.

OPEN Statement

The OPEN statement acquires and prepares files for subsequent input/output operations. An OPEN statement has the following form:

OPEN FILE (filename) options;

Options may occur in any order and may be placed before and after the specification FILE (filename). Options include any of the external storage attributes (except the FILE and ENVIRONMENT attributes) that are discussed in Chapter 2; the IDENT option, which regulates the reading and writing of header label records; and the PAGESIZE and LINESIZE options, which control the overall layout of each page in a PRINT file. A discussion of the IDENT, PAGESIZE, and LINESIZE options follows.

IDENT (label information) Option

The IDENT option in an OPEN statement for an output file specifies that a header label record is to be placed in the file. For an input file, IDENT provides information for reading a header label record.

The label information, for an input file, is the name of a character string into which the header label is to be read. For an output file, the label information is either the name of a character string containing the

header label record to be written, or it is a character-string constant that is to be written as the header label record. For an update file, the IDENT option specifies label reading but does not specify label writing.

The format of label records is defined by the compiler for each implementation of PL/I. If the IDENT option is not specified, no label operations are performed.

Consider the following statement:

```
OPEN INPUT FILE (TABLES) IDENT (SERIAL);
```

This statement opens the input file called TABLES, positions the file at its logical beginning, and reads the header label record into the character string identified by SERIAL.

PAGESIZE(*w*) Option

The PAGESIZE option specifies the depth of a printed page. *w* indicates the number of lines, including skipped lines, that are contained on a page. For example, the specification PAGESIZE(45) means that no more than 45 lines are to be printed or skipped on a page. If an attempt is made to start a new line beyond line 45, a new page is started unless the programmer has specified another action in an ON statement (the ON statement is discussed in Chapter 4; also see the ENDPAGE condition in Chapter 4, "ON-Conditions").

The PAGESIZE option can be specified only for files that have the PRINT attribute.

LINESIZE(*w*) Option

The LINESIZE option specifies the maximum length of the lines on a printed page; *w* indicates the number of character positions available on a line. For example, the specification LINESIZE(120) means that a line extends from character position 1 to character position 120. If an attempt is made to place data to the right of character position 120, the data in question is placed on the next line, starting at character position 1.

The LINESIZE option can be specified only for files that have the PRINT attribute.

If an OPEN statement is encountered for a file already opened, the statement is ignored.

CLOSE Statement

The CLOSE statement releases facilities that were established during the opening of a file, repositions the file to its logical beginning, and causes proper disposition of the file. A CLOSE statement has the following form:

```
CLOSE FILE (filename) IDENT (label  
information);
```

The meaning of the IDENT option is the same as that for the OPEN statement, except that it handles trailer label records rather than header label records.

If a CLOSE statement is encountered and the file has not been opened, or has already been closed, the statement is ignored.

Data Transmission

The basic function of input and output is data transmission, getting the data items into the computer for processing and placing the results of the processing on external storage media. In PL/I two types of data transmission are available: record-oriented data transmission and stream-oriented data transmission. With record-oriented transmission, the data on the external medium is considered as a collection of physically separate records, each of which consists of one or more data items in any form. Data is transmitted, one record at a time, in the same form as it is recorded, either internally or externally; no conversion is performed. With stream-oriented transmission, the data on the external medium is considered as one continuous stream of data items in character form. Each data item is converted, if necessary, to and from its appropriate internal form, which is determined by the attributes of the data name to which a data item is assigned.

Record-Oriented Transmission

Record-oriented data transmission deals with files that are composed of a series of physically separate records. Each record is read or written as an entity, either into or from an addressable buffer or into or from a location specified by a data name (usually the name of a structure or an array).

In record-oriented transmission the principal statements used to transmit data to and from input and output files are the READ statement and the WRITE statement.

Files referred to in READ and WRITE statements must be declared to have the RECORD attribute. This specifies that the file is to be used with record-oriented statements. The UNBUFFERED attribute specifies that the data in the records does not go into an addressable buffer, but is assigned directly to or from the data name specified in the READ or WRITE statement.

For an unbuffered file, the READ statement has the following format:

```
READ FILE (filename) INTO (data name);
```

The WRITE statement has the format:

```
WRITE FILE (filename) FROM (data name);
```

When a READ statement is executed, causing a record to be read, there is no conversion of data types to conform to the attributes declared for the names. The data in the record should exactly match the declaration of the name to which it is assigned. Similarly, a record that is written has the same form externally as its internal representation.

When records are to be read into and written out of buffers instead of being directly assigned to data names, the READ statement has a different form:

```
READ FILE (filename)  
SET (pointer-variable);
```

The pointer variable is a name that is used to point to the location of the buffer. It is associated with a data name by means of a DECLARE statement. Such a data name is called a based variable. For example, consider the following statement:

```
DECLARE 1 DETAIL BASED (N),  
        2 ACCT_# CHARACTER (7),  
        2 PAYMENT DECIMAL FIXED (6,2),  
        2 NAME CHARACTER (40);
```

In this statement, DETAIL is a based variable and N is a pointer variable associated with DETAIL. As a based variable, DETAIL must be declared to have the BASED attribute (see Chapter 4, "Storage Allocation," for a discussion of the BASED attribute). The DECLARE statement indicates that N will point to a buffer into which and from which records will be read or written and that each record will consist of three data items with the same attributes as those declared for ACCT_#, PAYMENT, and NAME. The following READ statement, when executed, causes a record to be read into this buffer:

```
READ FILE (MASTER) SET (N);
```

The pointer variable N now points to the beginning of the record (in effect, the value of N is the address of the first storage location of the buffer). It is as if the record were assigned directly to DETAIL; a reference to ACCT_# becomes a reference to the first data item in the buffer, a reference to PAYMENT becomes a reference to the second data item in the buffer, a reference to NAME becomes a reference to the third data item in the buffer.

A record is written from this buffer (the buffer into which it was read) when the following WRITE statement is executed:

```
WRITE FILE (OUTPUT) FROM (DETAIL);
```

A record can also be written from a buffer created by a LOCATE statement. This statement has the form:

LOCATE based-variable FILE (filename) SET
(pointer-variable);

The LOCATE statement does not immediately cause data to be written. It indicates that a buffer, having the same attributes as the specified based variable, is to be allocated. The specified pointer variable will point to the buffer. Data is subsequently placed in the buffer to form a record. The record is written into the specified output file immediately before the next LOCATE statement or WRITE statement before the same file is executed or immediately before the specified file is closed.

For example, assume that a record containing an account number, the amount of payment, and a customer name has been read as described above. A record that contains only the customer name and the amount of payment is to be written in the output file. The following statements would be needed to set up a based variable and to allocate a new buffer:

```
DECLARE 1 BILL-RECORD BASED (P),  
        2 CUSTOMER CHARACTER (40),  
        2 AMOUNT DECIMAL FIXED (6,2),  
LOCATE BILL-RECORD FILE (OUTPUT);
```

The output record is created when the appropriate data items (referred to as NAME and PAYMENT) in the input buffer are moved into the output buffer (here they are referred to as CUSTOMER and AMOUNT). The move is accomplished by using the following assignment statements (the exact statements are shown here for the sake of example; the assignment statement is discussed fully in Chapter 5):

```
CUSTOMER = NAME;  
AMOUNT = PAYMENT;
```

Once the record has been created, it remains in the output buffer until the next LOCATE statement or WRITE statement for the file OUTPUT is about to be executed or until the file OUTPUT is about to be closed. The LOCATE statement does not require the SET (pointer variable) option because the BASED attribute for BILL-RECORD specifies a pointer variable (P), which is associated with the based variable.

Record-oriented transmission permits the use of update files. An update file is one that is both read from and written into. Data is written into an update file by means of the REWRITE statement. Each record that is read is rewritten into the update file, with or without change, before another record is read. If the update file is unbuffered, that is, if records from the file are assigned directly to data names, the REWRITE statement has the following form:

```
REWRITE FILE (filename) FROM (data  
name);
```

If the update file is buffered, that is, records are read into and written out of buffers, only the following statement is needed to cause a record to be written:

```
REWRITE FILE (filename);
```

There is no need to specify a data name because the REWRITE statement always refers to the buffer into which the record was read.

Stream-Oriented Transmission

Stream-oriented data transmission deals with files that are considered as one continuous stream of data in character form. Data items are assigned from the stream to data names or from data names into the stream. Stream-oriented transmission implies data conversion. All of the data items in the stream are in character form. On input, they are converted automatically to conform to the attributes of the data name to which they are assigned; on output, data items are converted, if necessary, to characters. Of course, only valid data conversions can be performed (see Chapter 5, "Conversion Between Data Types").

There are three modes of stream-oriented transmission: edit-directed transmission, list-directed transmission, and data-directed transmission. All three modes use the same statements for input and output, the GET statement and the PUT statement, respectively. Whichever mode is used, the following information is required for each GET or PUT statement:

1. The name of the file from which data is to be obtained or to which data is to be assigned.
2. A list of data names representing storage areas to which data items are to be assigned during input, or from which data items are to be obtained during output. Such a list is known as a data list.
3. The format of each data item.

In certain circumstances, all of this required information can be implied; in other cases, only a portion of it need be stated explicitly. If the file name is not specified, standard system files will be assumed; this applies to any of the three modes of stream transmission. In list-directed and data-directed transmission, the format of the data items is not specified. In data-directed input, not even the data list need be specified.

Edit-Directed Data Transmission

Edit-directed transmission specifies an explicit description of data items as they exist or are about to exist in the data stream. This explicit description is contained in a GET or PUT statement. The GET statement is used to transmit data from external to internal storage and the PUT statement to transmit data from internal to external storage. When used for edit-directed trans-

mission, the GET and PUT statements have the following form:

```
GET FILE (filename) EDIT (data list)
    (format list);
PUT FILE (filename) EDIT (data list)
    (format list);
```

The data list in an edit-directed GET or PUT statement is a list of data names to which or from which data items in external storage are to be assigned. The data names in a data list are separated by commas.

The format list is a list of format descriptions separated by commas. There are two types of format descriptions: data format descriptions and control format descriptions. Data format descriptions describe data items in the data stream; control format descriptions specify page, line, and spacing operations.

When an edit-directed GET or PUT statement is executed, the data names in the data list are paired with the data format descriptions in the format list. Data is transmitted in the following fashion:

1. If the statement is a GET statement, the data item described by the format description is assigned to the area identified by the data name with which the format description is paired. For example, in the statement:

```
GET FILE (FILE-A) EDIT (FIRST)
    (A(20));
```

the data name `FIRST` is paired with the format description `A(20)`. The data item described by `A(20)` is assigned to an area in internal storage identified by `FIRST`.

2. If the statement is a PUT statement, the data item assigned to the area identified by the data name in the data list is placed into the data stream. The representation of this value in the data stream depends on the format description with which the data name is paired. For example, in the statement

```
PUT FILE (FILE-A) EDIT (FIRST)
    (A(20));
```

the data name `FIRST` is paired with the format description `A(20)`. The data item identified by `FIRST` is placed into the data stream. The format description `A(20)` indicates that the data item is represented in the data stream as a character string 20 characters long.

Note that there is no necessary correlation between the way in which the same value is represented in internal storage and in the data stream. The representation of a value in internal storage depends on the attributes of the data name for which it is a value; the representation of a value in the data stream depends on its format description.

Data Format Descriptions

Data format descriptions are used to describe the representation and characteristics of data items in the data stream.

Character-String Format Descriptions

• $A(w)$

The format description $A(w)$ indicates that the value of the item in the data list with which $A(w)$ is associated is represented in the data stream as a character string. The specification w indicates the number of characters in the string.

Consider a data item called `UNIT` declared with the attribute `CHARACTER(20)`, meaning that its internal representation is a sequence of 20 characters. The value of `UNIT` may be written in a file called `UNIT_FILE` as a string consisting of 20 characters by using the following statement:

```
PUT FILE (UNIT_FILE) EDIT (UNIT) (A(20));
```

In this statement, the `A` indicates that the data is to be written as a character string, and the integer 20 specifies the length in characters of the string. If the format description `A(25)` had been used, 5 blank characters would have been added to the value of `UNIT` making the length of the item written 25 characters. Similarly, if the format description were `A(15)`, then only 15 characters would be written. These 15 characters would contain the value of the first 15 characters of `UNIT`.

For output, the length of the character string need not be specified with the `A` format. If omitted, the length is determined from the declared attributes of the name with which the format description is paired. Thus, the PUT statement above could have been written as follows:

```
PUT FILE (UNIT_FILE) EDIT (UNIT) (A);
```

Had `UNIT` been declared with the attribute `BIT(20)`, the value of `UNIT` would be converted from a 20-position bit string to a 20-position character string composed of the numeric characters zero and one and written out as 20 characters. Similar conversions apply to internally stored fixed-point and floating-point data items. If `UNIT` had been declared with the attribute `FIXED(10)`, the value of `UNIT` would be converted from internal fixed point to external character-string form and extended on the right with blanks, if necessary, before being written out as a 10-position character string.

The statement:

```
GET FILE (UNIT_FILE) EDIT (UNIT) (A(20));
```

causes the next 20 characters in the file called `UNIT_FILE` to be assigned to `UNIT`. The value is automatically

transformed from its character representation specified by the format A(20), to whatever representation is specified by the attributes declared for UNIT.

- F(w,d)

The fixed-point format description F(w,d) indicates that the value of the item in the data list with which F(w,d) is associated is represented in the data stream as a string of characters, with a decimal value and containing an assumed or actual decimal point.

The specification w indicates the number of character positions in the string. The d specification indicates that an assumed or actual decimal point appears d characters before the end of the string.

When a data item is written from inside the computer, its value is right-adjusted in the field specified by w in the format description. If d is greater than zero, an actual decimal point will appear in the field before the last d characters. If, in the data item being written, the number of significant characters to the right of the decimal point is less than d , trailing zeros will be supplied.

When the value of an item being written is less than zero, a minus sign character will be prefixed to the external character representation of the value.

No sign appears when the value is greater than or equal to zero and, therefore, the first position on the left will be a blank. The w specification must include a position for the minus sign and a position for the decimal point. Should the value of the item being written contain leading zeros, these will be changed to leading blanks; should the value of the item not fill the field specified by the format description, leading blanks will be supplied.

When a data item is being read into the computer, the value transmitted may appear anywhere in the field specified by w . A decimal point is *assumed* to appear in the field before the last d characters. The specification d need not be written if the value to be read is to have no decimal places. However, if an actual decimal point appears in the value read, and its position contradicts the position specified by d , the d specification will be ignored and the position of the actual decimal point will be used.

Consider a data item called AMOUNT which has the attribute FIXED(4,2) meaning that it is a four-digit, fixed-point decimal number with two decimal places. It can be written in a standard output file by the following statement:

```
PUT EDIT (AMOUNT) (F(6,2));
```

(6,2) specifies that a decimal point is to appear before the last two numeric characters and that the number be right-adjusted in a field of six characters. Leading zeros are changed to blanks, and, if necessary, a minus sign is placed to the left of the first numeric character.

If AMOUNT, as described above, were used in the statement:

```
GET EDIT (AMOUNT) (F(6,2));
```

the next six characters from a standard input file would be scanned for a fixed-point decimal number with two decimal places. When located, the number would be converted to a form compatible with the declared attributes of AMOUNT.

If AMOUNT had the attributes FIXED BINARY(24,7) indicating a 24-bit fixed-point binary number with seven binary places, the statement:

```
PUT EDIT (AMOUNT) (F(6,2));
```

would produce, in a standard output file, six characters containing the value of AMOUNT converted from internal fixed-point binary representation to a decimal number and represented as a string of decimal characters with a decimal point two characters before the end of the item. A GET statement would reverse the conversion. Similar conversions would apply if the attributes of AMOUNT were either floating-point decimal or floating-point binary. If AMOUNT had the CHARACTER attribute or a picture specification of a character string, conversion would still apply, provided, in the case of output, the character string represented a numeric value.

If the BIT attribute were declared for AMOUNT, the bit-string data would be treated as a binary integer, which would be converted to a decimal integer and represented as a string of decimal characters.

- E(w,d)

The floating-point format description E (w,d) indicates that the value of the item in the data list with which E(w,d) is associated is represented in the data stream as a character string. This string has a decimal value and contains both an assumed or actual decimal point and a signed exponent preceded by the character E.

The interpretation of w and d is the same as that for w and d in the format description F(w,d) with the following exceptions:

The specification w must include the number of characters occupied by the following:

1. The exponent and its sign
2. E
3. A sign for the value, should it be negative
4. An actual decimal point, if present

When this format description is used, d indicates the total number of characters between the decimal point and E. On output an actual decimal point will appear to the left of the significant digits. In the statement:

```
GET EDIT (GROSS_ESTIMATE) (E(8,2));
```

the value of the item in the data stream is assigned to `CROSS_ESTIMATE` and is automatically converted to the internal form specified by the attributes for `CROSS_ESTIMATE`.

Bit-String Format Descriptions

- `B(w)`

The bit-string format description `B(w)` indicates that the value of the item in the data list with which `B(w)` is associated is represented in the data stream as a character-string containing only the characters 1 and 0. The specification `w` indicates the number of characters in the string.

Consider a data item called `CODE` which has the attribute `BIT(40)`. The value of `CODE` may be written in a file called `CODE_FILE` as a string of 40 characters that represent bits by using the following statement:

```
PUT FILE (CODE_FILE) EDIT (CODE) (B(40));
```

If the format description `B(120)` had been used in the above statement, 80 zero characters would be appended to the right of the string.

For output, the length of the string need not be specified with the `B` format. If omitted, the length is determined from the declared attributes of the name with which the format description is paired. Thus, the `PUT` statement above could have been written as follows:

```
PUT FILE (CODE_FILE) EDIT (CODE) (B);
```

Picture Format Description

The picture format description has the following form:

```
P 'picture-specification'
```

The picture-specification is described in Chapter 2. Editing by means of a picture specification is discussed in Chapter 5.

The `P` format description may be used to describe data values that are represented in the data stream in an edited or unedited character format. Consider the following statement:

```
GET EDIT (TITLE,COUNT) (P'AAAAA',P'9999');
```

When this statement is executed, the next nine characters in a standard input file are transmitted, and the first five (alphabetic) characters are assigned to `TITLE`; the integer located in the next four (numeric) character positions is assigned to `COUNT`. The internal representation of the data will conform to the attributes of `TITLE` and `COUNT`.

Repetition of Format Description

For abbreviation purposes a decimal integer can appear immediately before a format description to indi-

cate the number of times that format description is to be used before the next format description is selected. The statement:

```
GET EDIT (UNIT,ITEM,COST) (2A(10),F(4,2));
```

has the same effect as the statement:

```
GET EDIT (UNIT, ITEM, COST) (A(10), A(10), F(4, 2));
```

An expression enclosed in parentheses immediately before a format description can also be used to specify the number of times the format description is to be used. When the expression has a zero or negative value, the format description associated with it is not used.

A repetition factor may be applied to more than one format description by enclosing the format descriptions in parentheses and preceding the left parenthesis with the repetition factor.

Thus the statement:

```
PUT EDIT (UNIT, COST, ITEM, RATE) (2(A(10), F(4, 2)));
```

produces the same result as the statement:

```
PUT EDIT (UNIT, COST, ITEM, RATE) (A(10), F(4, 2), A(10), F(4, 2));
```

Multiple Data Specifications

In a `GET` or `PUT` statement, a data list and the format list associated with it are called an edit-directed data specification. Up to now, examples of `GET` and `PUT` statements have each contained one data specification. More than one data specification can be written in a `GET` or `PUT` statement. More than one is usually written in order to simplify the writing of statements employing long data lists and format lists. The relationship between a format item and a data name is more evident when data lists and format lists are short. Successive data specifications are separated by commas.

For example, the statement:

```
PUT EDIT (UNIT, COST, ITEM, RATE) (A(10), F(4, 2), A(10), F(4, 2));
```

might be more easily read as:

```
PUT EDIT (UNIT, COST)(A(10), F(4, 2)) (ITEM, RATE)(A(10), F(4, 2));
```

Data Specifications for Structures and Arrays

The names in a data list may be the names of structures and arrays. A structure or array name in a data list serves as a concise representation of all the elementary items in the structure or array.

Consider the following structure:

```
DECLARE 1 BILL,
        2 ITEM    CHARACTER(10),
        2 NUMBER  FIXED(5),
        2 COST    FIXED(5,2);
```

The use of `BILL` in the statement:

```
PUT EDIT(BILL)(A(10), F(5), F(5,2));
```

is equivalent to writing the following statement:

```
PUT EDIT (ITEM, NUMBER, COST) (A(10),
    F(5), F(5,2));
```

In the foregoing example, when the structure `BILL` is written, format descriptions are required for each elementary item in `BILL`, because `GET` and `PUT` statements transmit successive elementary items.

On output, a data list may specify expressions, including structure and array expressions. (Expressions are discussed in Chapter 5.) Each expression, when evaluated, is written in a format specified by an associated format item. For example, the statement:

```
PUT EDIT ('TOTAL IS ' || 2 * SUM) (A(15));
```

will convert the value of `2 * SUM` to a character string, append the string to the right of the constant `'TOTAL IS'`, and write the result as a 15-position character string in a standard output file.

Control Format Descriptions

The format descriptions discussed up to now are used to describe data items as they exist in the data stream. The format descriptions discussed below are not descriptions of data items; they describe spacing and printing operations. These control format descriptions are written in the format list of a `GET` or `PUT` statement together with the data format descriptions discussed previously. They are not, however, associated with a data name in the data list of the `GET` or `PUT` statement.

Spacing Format Description

The following format description is used in a format list to indicate spacing.

- `X(w)`

On input, the `X` format item specifies that the next `w` characters of the data in the stream are to be spaced over and not to be transmitted. On output, the `X` format specifies that `w` blank characters are to be inserted into the stream. For example, the statement:

```
GET EDIT (COUNT, DEDUCTION) (A(5), X(5),
    A(5));
```

specifies that the next 15 characters from a standard input file are to be treated as follows: the first five characters are assigned to `COUNT`, the next five charac-

ters are spaced over and ignored, and the remaining five characters are assigned to `DEDUCTION`.

The statement:

```
PUT EDIT (ITEM, TOTAL) (A(4), X(2), F(5));
```

places, in a standard output file, four characters that contain the value of `ITEM`, followed by two blank characters, followed by five characters that contain the value of `TOTAL`.

Printing Format Descriptions

The following format descriptions control printing operations. They are used only with files that have the `PRINT` attribute and, consequently, appear only in a `PUT` statement.

- `PAGE`

The format description `PAGE` specifies that the next data item written is to appear on a new page. For example:

```
PUT EDIT ('CONTINUED ON NEXT PAGE')
    (A(22)) ('MONTHLY SALES REPORT')
    (PAGE, A(20));
```

This `PUT` statement causes the phrase `CONTINUED ON NEXT PAGE` to be written on the current page and the heading `MONTHLY SALES REPORT` to be written on a new page.

- `SKIP(w)`

The format description `SKIP(w)` specifies that one or more lines are to be skipped before the next data item is written. `w` indicates the position of the next line relative to the current line. The number of lines skipped is equal to `w-1`. For example, the format description `SKIP(5)` means that four lines are to be skipped and the next data item is to be written on the fifth line following the current line.

- `LINE(w)`

The format description `LINE(w)` specifies that the next data item is to be written on a particular line. `w` indicates the number of the line on the page. For example, the format description `LINE(20)` means that the next data item is to be written on line 20 of the page.

- `COLUMN(w)`

The format description `COLUMN(w)` specifies horizontal positioning, giving the character position at which the next data item is to begin. `w` is the number of the character position for the first character of the data item. For example, the format description `COLUMN(35)` means that the first character of the next data item is to appear in character position 35 of the line.

Among the printing format descriptions, the SKIP format description specifies relative positioning, while LINE and COLUMN specify absolute positioning. The following example shows the use of printing format descriptions in combination with each other.

```
PUT EDIT ('MONTHLY BANK LOAN REPORT')
(PAGE, LINE(2), A(24));
PUT EDIT (LOAN_#, PRINCIPAL, INTEREST,
PAYMENT, BALANCE) (SKIP(3), A(7),
COLUMN(15), F(8,2), COLUMN(35),
F(3,3), COLUMN(45), F(6,2),
COLUMN(65), F(8,2));
```

The first PUT statement specifies that the heading MONTHLY BANK LOAN REPORT is to be written on line two of a new page. The second statement specifies that two lines are to be skipped and the value of LOAN_# is to be written, beginning at the first character of the fifth line; the value of PRINCIPAL, beginning at character position 15; the value of INTEREST at character position 35; the value of PAYMENT at character position 45; and the value of BALANCE at character position 65.

List-Directed Data Transmission

List-directed data transmission permits the programmer to specify the variables to which data is to be assigned (or from which data is to be acquired) without specifically stating a format for the data. The format is a standard one and is supplied by the compiler. List-directed transmission provides easy input/output operations for programmers who do not require a special format either on input or output, and who are interested only in a list of the results of the processing.

The elementary form of the GET and PUT statements, when used for list-directed input and output, is:

```
GET FILE (filename) LIST (data list);
PUT FILE (filename) LIST (data list);
```

The FILE (filename) is the *file specification*; LIST (data list) is the *data specification*. The file name and the data list must each be enclosed in parentheses. The two specifications need not appear in a particular order. If the file specification is omitted, it is assumed that one of the standard files is to be used. In list-directed transmission, the keyword, LIST, must always head the data specification.

List-Directed Data Lists

The data list in a list-directed GET statement is a list of variables (representing internal storage areas) to which data items in the data stream are to be assigned. The variables (data names) in a data list are separated

by commas. An example of a list-directed GET statement follows:

```
GET FILE (MASTER) LIST
(LOAN_#, PRINCIPAL, RATE);
```

The GET statement in the above example causes three data items from MASTER file to be assigned to the variables of the data list in the sequence in which they are listed; that is, the first data item is assigned to LOAN_#, the second to PRINCIPAL, and the third to RATE. Assignment stops at this point because the data list has been exhausted.

The data list in a list-directed PUT statement differs from that of a GET statement only in that a data item may be represented by an expression other than its name, for example, an arithmetic expression whose value is the item to be written. Once evaluated, the value represented by an expression is transmitted in the same way that the value represented by a variable is transmitted. Items in the data list (including expressions, if any) are separated by commas. An example of a list-directed PUT statement follows:

```
PUT FILE (OUT)
LIST (NAME,6.3*RATE,NUMBER-10);
```

The PUT statement in the above example causes three data items to be written in the file named OUT. The sequence in which the data items are written follows the sequence of the items in the data list; that is, the first data item is the value represented by the variable NAME, the second is the value resulting from the evaluation of the expression 6.3*RATE, the third is the value resulting from the evaluation of the expression NUMBER-10. Writing stops at this point.

Note that in list-directed input/output or in any form of stream-oriented transmission, it is the data list that determines the number of data items obtained from the stream or inserted into the stream.

Format of List-Directed Data

In list-directed input, successive data items on the external medium must be separated either by commas or blanks. On output, blanks are supplied between items automatically.

List-Directed Data Representation

The internal and external representation of a data item in list-directed transmission is determined by the attributes declared for it by the programmer. For example, a data item for which the attributes CHARACTER (10) have been declared would be recorded internally as a character string of length 10. On output, it would be written the same way. To better understand how this applies to list-directed GET and PUT statements,

assume that the standard input file contains the following data:

```
'NEW YORK', 'JANUARY', -6.5, 72.6
```

Assume, further, that the following two statements appear in the program:

```
DECLARE CITY CHARACTER (12), MONTH  
CHARACTER (9), MINTEM FIXED  
DECIMAL (4,2), MAXTEM FIXED  
DECIMAL (5,2);  
GET LIST (CITY, MONTH, MINTEM, MAXTEM);
```

The GET statement would cause the data items to be assigned as follows:

1. CITY is assigned the character string NEW YORK, left adjusted and padded on the right with four blanks.
2. MONTH is assigned the character string JANUARY, left adjusted and padded on the right with two blanks.
3. MINTEM is assigned the value -06.50.
4. MAXTEM is assigned the value 072.60.

The character strings are padded on the right with blanks to conform with the declared length of the strings; quotation marks are not maintained internally. The decimal fixed-point numbers are aligned on the assumed decimal point, to conform with the declared precision. Consider the result of the following PUT statement:

```
PUT, LIST (CITY, MONTH, MAXTEM,  
MINTEM, 'RANGE:', MAXTEM-MINTEM);
```

The record would be printed as:

```
NEW YORK JANUARY 72.6 -6.5 RANGE: 79.1
```

Note that if a character string is printed, the single quotation marks are not written, whether the string is specified as the value of a variable (CITY and MONTH) or is specified as a character constant ('RANGE:'). If a character string is written in a file that does not have the PRINT attribute, the enclosing quotation marks are *supplied, if necessary, and are written.*¹

Data-Directed Data Transmission

The elementary forms of the GET and PUT statements in data-directed transmission are written as follows:

```
GET FILE (filename) DATA;  
PUT FILE (filename) DATA (data list);
```

The data list need not be included in the GET statement because data items in the stream are accompanied by the data names to which they are to be

assigned. The data in the input stream might look like this:

```
A = 7.3 B = 'ABCDE' C(4,2) = 9876;
```

The data name, followed by an assignment symbol, followed by the value that is to be assigned, is called an assignment. The assignments in the input stream can be separated by commas or by blanks. The last assignment to be obtained by a single GET statement must be followed by a semicolon. If the above stream were part of the standard system input file, the statement:

```
GET DATA;
```

would cause values to be assigned to A, B, and C.

On output, the data list must appear to specify which data items are to be written into the stream. The PUT statement referring to the data items could be:

```
PUT FILE (OUT) DATA (A, B, C(4,2));
```

The assignments are separated by blanks in the output stream and a semicolon is written after the last item specified in the data list.

The STRING Option

One feature of stream-oriented data transmission is concerned with internal data transmission, rather than input and output. In either the GET statement or the PUT statement, the FILE (filename) option can be replaced by the STRING (string name) option. When the string option is specified, the statement has nothing to do with a file. In a GET statement, it indicates that the designated string is to be considered as a stream of input characters; in a PUT statement, it indicates that the designated string is to be considered as the output stream.

Although the string option can be used with any of the three modes of stream-oriented transmission, it is most practical in association with a format list since individual items in the string need not be separated by commas or blanks.

Consider the following example:

```
GET STRING (RECORD) EDIT (NAME, PAY_NO,  
HOURS, RATE) (A(12), A(7), F(2),  
F(4,2));
```

This statement specifies that the character string having the name RECORD, which is recorded in the internal storage area, is to be scanned. The first 12 characters of the string are to be assigned to NAME, the next 7 characters are to be assigned to PAY_NO, the next 2 characters are to be converted to decimal fixed-

¹ When a bit string is written by a list-directed PUT statement, the single quotation marks *do* appear, as does the letter B, even in a PRINT file. The binary digits are converted to the characters, 1 and 0.

point representation and assigned to `HOURS`, and the last 4 characters specified are to be converted to a fixed-point decimal number (with two digits after the decimal point) and assigned to `RATE`. If any characters remain in the string, they are to be ignored.

The `PUT` statement with a string option produces the reverse effect. Consider the statement:

```
PUT STRING (RECORD) EDIT (NAME, PAY_NO,  
HOURS*RATE) (A(12), A(7), P '$$99.99');
```

This statement specifies that the character value of `NAME` is to be assigned to the first 12 character positions of the string named `RECORD`, and that the character value of `PAY_NO` is to be assigned to the next 7 character positions of `RECORD`. The value of `HOURS` is to be multiplied by the value of `RATE` and the product is to be converted to a character string and assigned to the next 7 character positions of `RECORD`.

DISPLAY Statement

The `DISPLAY` statement causes a message to be displayed to the machine operator. A response may be requested. The device upon which the message is displayed will be specified for each implementation of `PL/I`.

The form of the `DISPLAY` statement is:

```
DISPLAY (expression) REPLY (data name);
```

Execution of the `DISPLAY` statement causes the expression to be evaluated and, when necessary, converted to a character string. This character string is the message that is displayed. The data name in the `REPLY` option must be declared as a character string. This data name receives the message supplied by the operator of the computer. Execution of the program is suspended until the operator's message is received, or, if the `EVENT` option is specified, execution continues and the reply is considered complete only after an associated `WAIT` statement specifying the event name is executed.

The `REPLY` specification is optional; when not used, execution continues without interruption.

The following statement:

```
DISPLAY ('WHICH IS THE NEXT FILE?') REPLY  
(NEXT_FILE);
```

displays the message: `WHICH IS THE NEXT FILE?`, and causes the computer to wait for the operator's reply. The operator's reply is assigned to `NEXT_FILE`.

The statement:

```
DISPLAY ('END OF PHASE-2');
```

displays the indicated message but does not interrupt computer execution.

PL/I and COBOL Comparison: Input/Output

The following discussion compares the input/output features of `PL/I` and `COBOL`. Both languages employ similar methods for transmitting data between internal and external storage areas to the extent that input/output statements process files and identification records (label records) in files may be processed both on input and on output. The languages differ, however, in input/output capabilities; the following points cover some of the more significant differences.

1. In `COBOL`, data transmission implies files that are composed of logical records. One or more data items form a logical record; data is transmitted one logical record at a time.

`PL/I` provides two types of data transmission. Record-oriented transmission, like `COBOL`, deals with logical records. Stream-oriented transmission handles individual data items; a file is thought of as one continuous stream of data items rather than as a collection of logical records.

2. `PL/I` provides control format specifications that regulate printing and spacing operations.

3. In `PL/I`, identification records (label records) are read or written as a result of the `IDENT` option in an `OPEN` or `CLOSE` statement. In `COBOL`, label records are specified by a file description entry in the Data Division, and special label procedures are specified in a `USE` statement.

4. `COBOL` permits a filename to be used as a name qualifier; `PL/I` does not.

5. In `PL/I`, the characteristics of a file are specified in a `DECLARE` statement or an `OPEN` statement. In `COBOL`, file characteristics are specified in a file description entry in the Data Division.

Chapter 4: Program Structure

Introduction

Some of the statements in a PL/I program read in and write out data, do calculations and perform the conversions from one data representation to another that are necessary to do these calculations. Some statements control the order in which other instructions in the program are executed. Because these statements control the order of execution in a program, they define the structure of a program. These statements specify transfers in the flow of program execution, prepare portions of a program and activate them for execution, specify iterative execution of certain statements and specify the control of program flow when program errors occur.

The remainder of this chapter discusses how a PL/I program is structured and explains the function of those statements that define the structure of a PL/I program and control the flow of program execution.

Blocks

In PL/I, a program consists of a set of external procedures each of which is composed of one or more blocks. A block is a collection of statements.

There are two kinds of blocks: *procedure* blocks and *begin* blocks.

Procedure Blocks

A procedure block — or, more briefly, a procedure — has the general form:

```
label: PROCEDURE;  
    statement 1  
    .  
    .  
    .  
    statement n  
END label;
```

A label must precede the PROCEDURE statement. An END statement need not contain a label; if it does, the label must be the same as that for the PROCEDURE statement.

Begin Blocks

A begin block has the general form:

```
label: BEGIN;  
    statement 1  
    .  
    .  
    .  
    statement n  
END label;
```

Two statements are required, a BEGIN statement and an END statement. A label may be written before the BEGIN statement to identify it as the start of a begin block. The END statement need not include a label; if it does, the label must be the same as that for the BEGIN statement.

Internal and External Blocks

Any procedure or begin block may include within it another entire procedure or begin block. One block must be completely included in another. Blocks may be contained within blocks. When one procedure block is not contained in any other block it is called an external procedure. A procedure block included in some other block is called an internal procedure. Begin blocks cannot be said to be external since every begin block must appear in some other block.

A program consists of a set of one or more external procedures which may contain internal blocks (nested blocks). The first external procedure block to be executed in a program must be identified by an implementation-defined keyword in the OPTIONS attribute. This keyword follows the keyword PROCEDURE in the PROCEDURE statement of the block and is required even when the program consists of only one external procedure.

An example of a program containing procedure blocks and begin blocks is the following:

```

A: PROCEDURE OPTIONS (MAIN);
   statement 1
B: BEGIN;
   statement 2
   statement 3
END B;
statement 4
C: PROCEDURE;
   statement 5
   D: BEGIN
       statement 6
       statement 7
   END D;
   statement 8
END C;
statement 9
END A;

```

In this example, block A is an external procedure and contains begin block B and internal procedure C. Block D is a begin block contained in procedure C.

The implementation-defined keyword MAIN identifies block A as the first block to be executed.

Although the begin block and the procedure block have a physical resemblance they differ in an important functional way. A begin block, like a single statement, is executed in the course of sequential program flow. With a procedure, however, program flow passes around the procedure, from the statement before the PROCEDURE statement to the statement after the END statement of that procedure.

The only way in which an internal procedure can be executed is by means of a CALL statement. The elementary form of a CALL statement is:

CALL entry name;

“Entry name” is the label of a PROCEDURE statement and, consequently, the name of a procedure. The CALL statement causes control of program flow to be transferred to the procedure named by “entry name.”

A CALL statement in a procedure or begin block that causes control of program flow to be transferred to another procedure is known as an “activating” CALL statement. The procedure to which control has been transferred is known as an “activated” procedure. When execution of an activated procedure is completed, control is sent to the statement following the activating CALL statement. Because a procedure can be activated only by using the name of the procedure as an entry name in an activating CALL statement, every PROCEDURE statement must have a label.

In the previous example, statement 1 could be a CALL statement that activated procedure C; it would be written as:

CALL C;

Upon completion of procedure C, control would be sent to the statement:

B: BEGIN;

because this statement would be the one following the activating CALL statement.

Scope of Declarations

The same name may appear in several blocks and be described with different attributes in each block. This allows a programmer to partition a program into many blocks and to use the same name in each block for different data items. For example:

```

PAYROLL: PROCEDURE;
  DECLARE TAX CHARACTER (5);
  .
  .
  .
REPORT: BEGIN;
  DECLARE TAX FIXED (4, 2);
  .
  .
  .
END REPORT;
  .
  .
  .
END PAYROLL;

```

PAYROLL is an external procedure and REPORT is a begin block internal to PAYROLL. The data name TAX is described differently in each of these blocks. TAX, with the attribute CHARACTER (5) applies to the entire PAYROLL procedure except for the begin block REPORT. In REPORT, the data name TAX has another meaning and applies to a different data item.

The region of a program within which the description of a name applies is called the scope of the declaration or the scope of the name established by the declaration.

In the above example, the scope of the declaration:

DECLARE TAX CHARACTER (5);

extends throughout the PAYROLL procedure but does not include the REPORT block. The scope of the declaration:

DECLARE TAX FIXED (4, 2);

is limited to the REPORT block.

In general, separate declarations of the same name imply unique names with distinct scopes that do not overlap.

It may be necessary, however, to use the same data item in different blocks. PL/I provides several ways

of accomplishing this: by nesting blocks, by using the EXTERNAL attribute, and by employing names as parameters and arguments. Each of these methods will be discussed in turn.

Nested Blocks

The same data item may be used in different blocks if the blocks are nested. When a data name is declared in an outer block by means of a DECLARE statement and is *not* redeclared in an internal block, the scope of the declaration includes the internal block.

Consider the following example:

```
x: PROCEDURE;
  DECLARE NUMBER FIXED (3), COST
    FIXED (4, 2);
  .
  .
  .
  GET LIST (NUMBER, COST);
  .
  .
  .
  CALL y;
y: PROCEDURE;
  DECLARE TITLE CHARACTER (10);
  .
  .
  .
  PUT LIST (TITLE, NUMBER, COST);
  .
  .
  .
  END y;
  .
  .
  .
  END x;
```

When procedure x is executed, NUMBER and COST are established as fixed point decimals and are assigned values by the GET statement. The scope of NUMBER and COST includes procedure y. The references to these two data names in the PUT statement employ the declaration established in procedure x. The scope of TITLE, however, is limited to procedure y; when control leaves procedure y, the data of TITLE is not accessible by statements in procedure x. Consequently, TITLE in procedure y must not be used by statements in procedure x.

Transfer of control from procedure x to procedure y must be made by means of a CALL statement. If procedure y had been written as a begin block, no CALL statement would be necessary; sequential flow of control would be allowed to proceed through the BEGIN statement of block y.

EXTERNAL and INTERNAL Attributes

If procedure x and procedure y in the previous example had not been nested, and the same values of NUMBER and COST were to be used in both procedures, then the means of indicating this would be to declare NUMBER and COST in both procedures with the attribute EXTERNAL. This is illustrated by rewriting the previous example as follows:

```
x: PROCEDURE;
  DECLARE NUMBER FIXED (3)
    EXTERNAL, COST EXTERNAL
    FIXED (4, 2);
  .
  .
  .
  GET LIST (NUMBER, COST);
  .
  .
  .
  END x;
y: PROCEDURE;
  DECLARE TITLE CHARACTER (10),
    NUMBER EXTERNAL FIXED (3), COST
    FIXED (4, 2) EXTERNAL;
  .
  .
  .
  PUT LIST (TITLE, NUMBER, COST);
  .
  .
  .
  END y;
```

This example has the same effect with respect to NUMBER and COST as did the previous example which employed nesting.

If the EXTERNAL attribute had not been used with NUMBER in both of the above procedures, then the NUMBER in procedure x and the NUMBER in procedure y would refer to two separate and distinct data items.

When the same data name is employed in two or more external procedures to refer to the same data item, the data name must be declared with the EXTERNAL attribute in each procedure. In all such declarations for the same data name, the attributes declared must be consistent, since the declarations involve a single data item.

The use of the EXTERNAL attribute is not restricted to procedure blocks; it may also appear in begin blocks that are external with respect to each other.

When the EXTERNAL attribute is not declared for a data name, the INTERNAL attribute is assumed.

However, file names and the label names of external procedure blocks are assumed to have the EXTERNAL attribute. (Label names of external procedures are discussed later under "Entry Name Parameters and the ENTRY Attribute.") The INTERNAL attribute specifies that the scope of a name is that block to which the declaration of the name is internal (a discussion of the scope of declarations appears earlier in this chapter). The EXTERNAL and INTERNAL attributes are called scope attributes and are also discussed later in this chapter under "Storage Allocation".

Parameters and Arguments

Another way of employing the same data items in different procedures is by specifying a parenthesized list of names in a PROCEDURE statement following the word PROCEDURE. The names in this list are called parameters.

The parameters appear in a PROCEDURE statement in the following way:

```
label: PROCEDURE (parameter 1, ...,
                parameter n);
        statement 1
        .
        .
        .
        statement n
END label;
```

Parameters are never permitted in a BEGIN statement. Parameters may be declared in a procedure with a DECLARE statement and may be data names, filenames, and entry names (the names identifying procedures). Parameters must be declared in the procedure to which they apply; they cannot be declared in any other blocks. If no explicit declaration is given, an implicit or contextual declaration is assumed that applies to the procedure containing the parameters.

Parameters permit the programming of procedures that require the values of unknown data names. Consider a procedure that is used by several other procedures to analyze a tax deduction. The person writing the procedure may not know what other programmers may call the tax deduction data item in their procedures. However, the writer of the tax deduction procedure can proceed by assigning whatever data name he pleases to the tax deduction item. He might, for example, use the data name TAX_DEDUCTION. This name is placed in the parameter list of the procedure and may also be used in the statements of the procedure. Another programmer may name the tax deduction item T_D in his procedure. Even though the tax deduction item has two different names in two different pro-

cedures, an association between T_D and TAX_DEDUCTION can be established. This is done by placing T_D in a parenthesized list of names in the CALL statement that transfers control to the tax deduction procedure. The names in this list are called arguments. The arguments appear in the CALL statement in the following way:

```
CALL entry name (argument 1,...,argument n);
```

Arguments may be data names, filenames, and entry names.

The parameters of an activated procedure and the arguments of the activating CALL statement are paired, from left to right, one by one. The number of arguments must equal the number of parameters.

During execution of the activated procedure the name of each parameter is made equivalent to the name of its associated argument. In general, the attributes of an argument must be the same as those of the corresponding parameter; arguments that identify arrays must correspond to parameters that identify arrays, and arguments that identify structures must correspond to parameters that identify structures.

Consider the following procedure:

```
OUTPUT: PROCEDURE (TITLE, NUMBER, COST);
        DECLARE TITLE CHARACTER (10),
                NUMBER FIXED (3), COST FIXED
                (4, 2);
        .
        .
        .
        PUT LIST (TITLE, NUMBER, COST);
        .
        .
        .
END OUTPUT;
```

This procedure contains three parameters: TITLE, NUMBER, and COST, the values of which are written in a standard file as list-directed output. Assume that the OUTPUT procedure is activated from the following procedure:

```
PROCESS: PROCEDURE;
        DECLARE NAME CHARACTER (10),
                COUNT FIXED (3), PRICE FIXED
                (4, 2);
        .
        .
        .
        CALL OUTPUT (NAME, COUNT, PRICE);
        .
        .
        .
END PROCESS;
```

When control is sent to the OUTPUT procedure by the CALL statement in the PROCESS procedure, the values of the arguments NAME, COUNT, and PRICE become the values of the corresponding parameters TITLE, NUMBER, and COST. When the OUTPUT procedure is completed, control returns to the statement following the CALL statement in the PROCESS procedure.

When an argument is the name of an array, the number of dimensions and the number of items in each dimension of the array must correspond to those of the associated parameter. When an argument is the name of a character or bit string, the length of the string must be the same as that of the corresponding parameter. However, the number of items in the dimensions of an array and the length of a string may not be known at the time a procedure is written. When this is the case, an asterisk (*) may be used for each dimension or for the string length in the declaration of the parameter. The asterisk then indicates that the string length or number of items in a dimension is the same as for the corresponding argument.

Consider the following example:

```
PRINT: PROCEDURE (TAX_TABLE)
        DECLARE TAX_TABLE(*) FIXED
           (4, 2);
        .
        .
        .
        PUT LIST (TAX_TABLE);
        .
        .
        .
END PRINT;
```

This procedure contains one parameter, TAX_TABLE, which is the name of a one-dimensional array. The asterisk in the declaration indicates that the size of the array is unspecified and will assume the size of the corresponding argument in a CALL statement. Assume that the PRINT procedure is activated from the following block:

```
BEGIN;
DECLARE WORK_TABLE (100) FIXED (4, 2);
.
.
.
CALL PRINT (WORK_TABLE);
.
.
.
END;
```

When control is sent to the PRINT procedure by the CALL statement in the begin block, the length of

TAX_TABLE becomes the length of WORK_TABLE, which in this case is 100, and the values of WORK_TABLE become the values of TAX_TABLE.

Each time the PRINT procedure is activated, TAX_TABLE can be associated with an argument of different size. However, the array named by the argument must be one-dimensional and the elements in the array must have the attribute FIXED (4,2).

In addition to data names, file names, and entry names, arguments may also be expressions. (Expressions are discussed in Chapter 5.) For example, the following statement:

```
CALL TAX (12*MONTH_SALARY);
```

could be used to execute a procedure called TAX that employed a parameter representing an annual salary. The value of the expression:

```
12*MONTHLY_SALARY
```

would then become the value of the parameter employed by the TAX procedure.

Entry-Name Parameters and the ENTRY Attribute

An entry-name has been defined as a label constant directly following the word CALL in a CALL statement. Entry-names may also be used in parameter lists and in argument lists. When an entry name is used as an entry-name in a CALL statement it is considered to be contextually declared as an entry name; when it is used in an argument list of a CALL statement, it is not considered to be contextually declared. (Contextual description is discussed in Chapter 2 under Default Attributes.) For example, consider the following procedure:

```
ROUTINE: PROCEDURE (x, y, z);
          DECLARE x FIXED DECIMAL
             (4), z LABEL;
          .
          .
          .
          CALL y;
          .
          .
          .
END ROUTINE;
```

The DECLARE statement in the procedure specifies that parameter x is a four-digit fixed-point decimal integer, and parameter z is a label name that may be used in the body of the procedure. y is contextually declared as an entry-name. If ROUTINE is activated by a CALL statement in the following procedure:


```

PROCESS: PROCEDURE;
      .
      .
      .
      CALL ROUTINE (COUNT, EDIT, L5);
      .
      .
      .
END PROCESS;

```

it is then necessary that argument COUNT have the attributes of parameter x, that EDIT be an entry name, and that L5 be a statement label. The appearance of EDIT as an argument in the CALL statement does not make it an entry name by context. Unless EDIT has been used as an entry-name in a CALL statement, it must be explicitly declared in the PROCESS procedure as an entry-name. This is accomplished by using the ENTRY attribute, as follows:

```

PROCESS: PROCEDURE;
      DECLARE COUNT FIXED DECIMAL
      (4), EDIT ENTRY, L5 LABEL;
      .
      .
      .
      CALL ROUTINE (COUNT, EDIT, L5);
      .
      .
      .
END PROCESS;

```

In the PROCESS procedure, the name ROUTINE is defined contextually as an entry name because it immediately follows the keyword CALL.

Sequence of Control

When a program is being executed, its sequence of control determines the order of execution of the statements.

Within a block, control normally passes sequentially from one statement to the next. If a DECLARE statement is encountered, control passes over it to the next statement. Sequential execution of statements is modified, however, by the following statements: CALL, PROCEDURE, END, RETURN, GOTO, IF, DO, and ON. The first three of these statements have been introduced. RETURN will be discussed in the following text. Then a discussion dealing with the activation and termination of blocks will follow, after which the GOTO, IF, DO, and ON statements will be discussed. Because the allocation of core storage is influenced by the sequence of control in a program, this chapter also includes, at the end, a discussion of storage allocation.

RETURN Statement

The PROCEDURE and END statements delimit a procedure block so that sequential control passes around an internal procedure block to the statement following the END statement of the internal procedure block. The CALL statement is used to send control to a specified procedure. When control reaches the END statement of a procedure, control returns to the statement following the CALL statement. It is possible to return control before reaching the END statement of a procedure. The RETURN statement serves that purpose. It is written:

```

RETURN;
Consider the following procedure:
TEST: PROCEDURE;
      .
      .
      .
      RETURN;
      .
      .
      .
      RETURN;
      .
      .
      .
END TEST;

```

Whenever either of the RETURN statements or the END statement is encountered in this procedure, control returns to the statement following the CALL statement that activated the TEST procedure.

Activation and Termination of Blocks

A begin block is said to be activated when control passes through the BEGIN statement for the block. A procedure block is said to be activated when a CALL statement transfers control to the PROCEDURE statement for the block.

There are several ways to terminate an active block:

1. A begin block is terminated when control passes through the END statement for the block.
2. A procedure block is terminated on execution of a RETURN statement or an END statement for the block.
3. A block is terminated on execution of a GO TO statement which transfers control to a point not contained in the block. (The GO TO statement is discussed later in this chapter).

Dynamic Descendance of Blocks

When a block is terminated, all the dynamic descendants of that block are terminated.

In the discussion that follows, consider the two blocks: block A and block B.

Block B is said to be an immediate dynamic descendant of block A if B is activated by a statement internal to A. A statement is internal to block A if it is contained in A but it is not contained in any other block which is itself contained in A.

Consider the following example:

```
R: PROCEDURE;
    statement 1
    statement 2
S: PROCEDURE;
    statement 3
    statement 4
T: BEGIN;
    statement 5
END T;
    statement 6
END S;
    statement 7
END R;
```

In this example, statements 1, 2, and 7 are internal to R because they are contained in R and are not contained in another block which is itself contained in R. Statements 3, 4, and 6 are not internal to R but are internal to S. Statement 5 is internal only to block T.

Block B can become an immediate dynamic descendant of A in the following ways:

1. B is a procedure block immediately contained in A (that is, B is not contained in another block that is itself contained in A) and is referred to by a CALL statement internal to A.
2. B is a procedure block not contained in A and is referred to by a CALL statement internal to A.
3. B is a begin block that is executed as the result of an interruption. (Interruptions are discussed with the ON statement later in this chapter.)

Block B itself may have an immediate dynamic descendant C, etc., so that a chain of blocks (A, B, C, D, ...) is created, in which all blocks are active. In this chain, blocks C, D, ... are dynamic descendants of A, but only B is an immediate dynamic descendant of A.

When block A is terminated, all active blocks contained in B are also terminated.

GO TO Statement

The GO TO statement transfers control to a specified statement. There are two forms of the GO TO statement:

```
GO TO label constant;
GO TO label name;
```

In the first form control is sent to the statement that has the label constant as its label. In the second form the GO TO statement has the effect of a multi-way switch; control is transferred to the statement identified by the current value of the label name. Because the label name may have different values at each execution of the GO TO statement, control may not always pass to the same statement. The label name may also be the name of an array of labels. The following example illustrates the use of a GO TO statement that effectively is a multi-way switch.

```
SWITCH: PROCEDURE;
    .
    .
    .
    DECLARE L LABEL (L1, L2)
    INITIAL (L2);
    GO TO MEET;
L1:    X = Y - 1;
        L = L2;
        GO TO MEET;
L2:    Y = X - 1;
        L = L1;
MEET:  CALL FUDGE (X, Y, Z);
        IF Z = LIMIT THEN GO TO L;
    .
    .
    .
END SWITCH;
```

When the value of L is L2, the GO TO statement sends control to the CALL statement. When L1 is the value of L, control is sent to the first assignment statement.

The value of a label name is changed to L1 by executing the assignment statement L=L1. (The assignment statement is discussed in Chapter 5.)

A GO TO statement may not pass control to a block that has not been activated or to a block that has been terminated.

A GO TO statement that transfers control from a block within a nest of blocks to an encompassing block at a higher level in the nest terminates all other blocks that are dynamic descendants of the block.

In the following example:

```
FIRST:  PROCEDURE;
        .
        .
        .
SECOND:  BEGIN;
        .
        .
        .
THIRD:   BEGIN;
        .
        .
        .
        CALL FOURTH;
        .
        .
        .
FOURTH:  PROCEDURE;
        .
        .
        .
        GO TO SECOND;
        .
        .
        .
END FOURTH;
        .
        .
        .
END THIRD;
        .
        .
        .
END SECOND;
        .
        .
        .
END FIRST;
```

the statement `GO TO SECOND;` sends control to the begin block called `SECOND`, and terminates the blocks `FOURTH` and `THIRD`.

IF Statement

It is often desirable to execute a statement or a series of statements only under certain circumstances. In such a situation, it might be convenient to evaluate an expression and, on the basis of this evaluation, select or reject the statement or statements to be executed. PL/I provides the IF statement for this purpose. It also provides eight comparison operators that are used to construct comparison expressions that may be either true or false. These expressions are used in the

IF statement to determine whether or not the statement or statements are to be executed.

Comparison Expressions

Comparison expressions use the following comparison operators:

| | |
|--------------------|----------------------------|
| <code>></code> | (greater than) |
| <code>¬></code> | (not greater than) |
| <code>>=</code> | (greater than or equal to) |
| <code>=</code> | (equal to) |
| <code>¬=</code> | (not equal to) |
| <code><</code> | (less than) |
| <code>¬<</code> | (not less than) |
| <code><=</code> | (less than or equal to) |

These operators are used with data names and constants to form comparison expressions. For example, the expression:

```
COUNT > 10
```

is a comparison expression in which the numeric value of `COUNT` is compared with the constant 10. If the value of `COUNT` is greater than 10 the expression is true; otherwise it is false.

In addition to numeric data items, character-string data items may appear in comparison expressions. The expression:

```
NAME < 'ROBERT'
```

is true when the character-string value of `NAME` is less than the character-string constant `'ROBERT'`; otherwise it is false. Character strings are compared from left to right, character by character. The comparison is based on an implementation-defined collating sequence. If the character strings are of different lengths, the shorter string is extended on the right with blanks.

Bit strings are permitted in comparison expressions. The following expression:

```
MASK ¬='1010101'B
```

is true when the bit-string value of `MASK` does not equal the bit-string constant `'1010101'B`. The expression involves a left-to-right comparison of binary digits. The binary digit zero is lower in comparison to the binary digit one. If the bit strings are of different lengths, the shorter is extended on the right with zero bits.

A more detailed discussion of comparison expressions appears in Chapter 5.

The result of a comparison is a bit string of length one; the value of a true comparison is the constant `'1' B`; a false comparison has the value `'0' B`.

Comparison Expressions in an IF Statement

An IF statement can assume one of 2 forms:

```
IF comparison expression THEN unit
IF comparison expression THEN unit 1 ELSE
unit 2
```

Each "unit" is either a single statement, a begin block, or a group. A group consists of a sequence of statements and/or blocks preceded by a DO statement and terminated by an END statement. An example of a group is the following sequence of statements:

```
DO; GET LIST (GROSS, NET);
CALL PROFIT; END;
```

This group may appear in an IF statement as follows:

```
IF SALES = 22500 THEN DO; GET LIST (GROSS,
NET); CALL PROFIT; END;
```

In this example, if the numerical value of SALES is equal to the constant 22500, the group of statements following the keyword THEN is executed; otherwise the group is skipped and control passes to the statement after the END statement of the group.

When the unit is a single statement, the DO and END statements need not be specified unless iteration by means of the DO statement (discussed later) is desired. The following statement is an example of a single statement "unit."

```
IF PROFIT < 0 THEN GO TO LOSS;
```

When the numeric value of PROFIT is less than zero, control is sent to the statement labeled LOSS; otherwise, the statement following GO TO LOSS; is executed.

In the second form of an IF statement, a true comparison causes the execution of the unit after the THEN and causes the unit after the ELSE to be skipped. When the comparison is false, the unit after THEN is skipped, and the unit after the ELSE is executed. Consider the following example:

```
IF SALES > 1000
THEN BEGIN;
    DECLARE GROSS FIXED (7, 2),
    NET FIXED (6, 2);
    GET FILE (DETAIL) EDIT (GROSS,
NET) (F(7,2), F(6,2));
    CALL ANALYSIS (GROSS, NET);
    PUT FILE (REPORT) LIST
    (PROFIT);
END;
ELSE DO;
    CALL TAX (SALES);
    GO TO ADJUSTMENT;
END;
```

When the numeric value of SALES is greater than 1000 the begin block after the THEN is executed and the DO group after the ELSE is skipped; otherwise the DO group is executed and the begin block is skipped.

In an IF statement the units following THEN and ELSE may also contain one or more IF statements. When this is the case, the units following THEN and ELSE are treated as pairs. Each ELSE unit 2 is paired with the immediately preceding unpaired THEN unit 1. It is, therefore, necessary to specify an ELSE unit 2 for each THEN unit 1. In this case, unit 2 can be a "null" ELSE statement, that is, the word ELSE followed by a semicolon.

The following example illustrates the use of nested IF statements, one of which employs a null ELSE clause.

```
IF AGE > 21
THEN
    IF WEIGHT < 150
    THEN IF HAIR = 'BROWN'
        THEN GO TO TYPE1;
        ELSE GO TO TYPE2;
    ELSE;
    ELSE GO TO TYPE3;
```

In this example, the effect of the null ELSE statement is to execute nothing, and to transfer control to the statement following GO TO TYPE3 should WEIGHT < 150 be false and AGE > 21 be true.

DO Statement

The DO statement, used together with an END statement, provides a means for grouping a set of statements. It also provides for the repeated execution of a sequence of statements and permits modification and testing of data items to control the repetition. Being able to repeat the execution of a set of statements a specified number of times generally results in a smaller and more efficient program. A DO statement can be specified in several ways.

Consider the following form of a DO statement:

```
DO WHILE (comparison expression);
statement 1
.
.
.
statement n
END;
```

This use of the DO statement causes the indicated sequence of statements to be executed repeatedly as long as the value of the comparison expression is true. (Comparison expressions were discussed briefly with the IF statement and will be discussed in greater de-

tail in Chapter 5.) When the comparison expression is false, control passes to the statement following the END statement. In the following example:

```
DO WHILE (CODE = 1);
  GET LIST (CODE, STRING);
  PUT EDIT (STRING) (A);
END;
```

when the DO statement is first encountered the numeric value of CODE is compared to 1. If the comparison is equal, the GET and PUT statements are executed, and the statement returns control to the DO statement where the value of CODE is again compared to 1. When the comparison is not equal, control is transferred to the statement following the END statement. Each time a record is read, the data in the first field of the record is assigned to CODE and should contain a numeric value of 1. The last record processed should contain a numeric value in the first field not equal to 1.

Another form of the DO statement is the following:

```
DO data name = expression 1 TO expression
  2 BY expression 3;
statement 1
.
.
.
statement n
END;
```

The expressions in this form of the DO statement are arithmetic expressions and represent numeric values. (Arithmetic expressions are discussed in detail in Chapter 5.) When the DO statement is first encountered, data name is initialized to this value of expression 1, and this value is compared to the value of expression 2; if the result of the comparison falls inside the range of values of expression 1 and expression 2, the sequence of statements is executed. The END statement then returns control to the DO statement where the value of data name is incremented by the value of expression 3. If the new value of data name falls outside the limit specified by expression 2, control is sent to the statement following the END statement; otherwise the sequence of statements is executed and the END statement again transfers control to the DO statement.

The following example illustrates this form of the DO statement:

```
DO COUNTER = 1 TO 10 BY 1;
  GET FILE (MASTER) EDIT (ITEM(COUNTER))
  (A(80));
END;
```

In this example, the data name COUNTER is used in the GET statement as a subscript to specify a position in an array called ITEM. When the DO statement is first encountered, the value of COUNTER is set to 1 and the GET statement is executed. The next 80 characters in the MASTER file are assigned to the first position in the ITEM array; then the END statement returns control to the DO statement. The value of COUNTER is incremented by 1 and tested to determine if it exceeds the range 1 to 10. The second execution of the GET statement assigns the next 80 characters in the MASTER file to the second position in the ITEM array. This process is repeated until the value of COUNTER exceeds 10.

The expressions in this form of the DO statement may have negative values. Because negative values are permitted, expression 2 need not be greater than expression 1, and the value of expression 3 may be used as a decrement rather than an increment.

The two previous forms of the DO statement may be combined as follows:

```
DO data name = expression 1 TO expression
  2 BY expression 3 WHILE (expression 4);
statement 1
.
.
.
statement n
END;
```

When specified this way, the sequence of statements is executed repeatedly until one of the following takes place: either the value of the data name falls outside the specified range, or the comparison expression associated with the WHILE becomes false.

The following example illustrates the combined form of the DO statement:

```
DO INDEX = 10 TO 1 BY - 1
  WHILE (DEDUCTION < ESTIMATE);
.
.
.
END;
```

In this example the value of INDEX goes from 10 to 1 in decrements of 1. When either the value of INDEX falls outside the range 10 to 1 or the comparison expression DEDUCTION < ESTIMATE proves false, control is sent to the statement following the END statement.

Control may transfer into a DO group from outside the DO group only if the DO group is delimited by a DO statement that does not specify repeated execution.

ON Statement

During the course of program execution any one of a certain set of conditions may occur that can result in an interruption. An interruption causes the suspension of normal program activities, in order to permit the execution of a special action. When the special action has been performed, program execution may or may not resume at the point where it was suspended. The place in a program where an interruption may occur is generally unpredictable.

For most conditions that can cause an interruption, the programmer may specify the special action to be performed. To do this, he may specify the condition in an ON statement; therefore these conditions are called ON-conditions. A discussion of individual ON-conditions appears later. Each ON-condition is given a unique name suggestive of the condition. For example, ZERODIVIDE names the condition resulting from an attempt to divide by zero. These names (hereafter called ON-conditions) are an intrinsic part of the language, but the programmer may also use them for other purposes (such as file names, data names, and label names) so long as no ambiguity exists.

The general forms of an ON statement are:

```
ON condition SNAP unit;  
ON condition SYSTEM;
```

The first form allows the programmer to state what action is to be performed when the specified condition occurs. The action is specified in the "unit"; it is written either as an unlabeled single statement or an unlabeled begin block. Note that a RETURN statement may not appear in the unit. The keyword SNAP is optional; if specified, it produces a listing of information relevant to the status of the program when the specified condition occurs.

The second form specifies that a standard system action is to be performed. (The standard system action for each condition is discussed later.)

The following example:

```
ON ZERODIVIDE CALL ANALYSIS;  
CALL ANALYSIS;
```

specifies that the statement:
be executed when division by zero is attempted. If a standard system action is to be performed when division by zero is attempted, the following statement may be employed:

```
ON ZERODIVIDE SYSTEM;
```

Use of the ON Statement

The ON statement is an executable statement. When executed, an ON statement internal to a block (for example, block B) establishes an ON-action. If any

of the statements executed after the execution of the ON statement and before termination of block B (including execution of statements in all dynamic descendants of block B) is interrupted by the occurrence of the condition specified in the ON statement, the statement or begin block appearing in the ON statement is executed as though it were activated as a procedure block. Control normally will be returned to the point following the interruption.

If, after a given action is established by execution of an ON statement, and while this action specification is still effective, another ON statement specifying the same condition is executed, then the latter ON statement will take effect as described above, so that its specified action will determine the interruption action for the given condition. If both of these ON statements (old and new) are internal to the same block, the effect of the old ON statement is completely nullified. When the new ON statement is in a block dynamically descended from the block containing the old ON statement, the effect of the old ON statement is temporarily suspended. The effect of the old ON statement is restored upon termination of the block containing the new ON statement.

The standard system action for ON conditions is established automatically. In some situations, the programmer may want to specify his own action for a given condition, for part of the execution of the program, and then to have this specification nullified and allow the standard system action to occur.

The following example illustrates how this may be done.

```
A: PROCEDURE;  
.  
.  
.  
ON ZERODIVIDE CALL ANALYSIS;  
.  
.  
.  
ON ZERODIVIDE;  
.  
.  
.  
ON ZERODIVIDE SYSTEM;  
.  
.  
.  
END A;
```

When control is transferred to procedure A, the ZERODIVIDE condition is enabled by the system; any ZERODIVIDE condition that occurs before the first ON ZERODIVIDE statement is executed will

result in an interruption, with standard system action. However, the execution of the first ON ZERODIVIDE statement establishes the action specified by the CALL statement. Any ZERODIVIDE interruptions will cause this action to be executed until the second ON ZERODIVIDE statement is executed. The action specified here is a null statement; any subsequent ZERODIVIDE interrupts will effectively be ignored until control reaches the third ON ZERODIVIDE statement, which re-establishes the standard system action.

Prefixes

A prefix is a list of condition names, separated by commas and enclosed in parentheses. A prefix may be attached to a statement. When attached to a statement, the prefix precedes the entire statement, including any possible label for the statement, and is followed by a colon to separate it from the rest of the statement.

A statement with a prefix has the general form: (condition name 1,..., condition name n): label: statement

The following condition names may appear in a prefix:

UNDERFLOW
 OVERFLOW
 ZERODIVIDE
 FIXEDOVERFLOW
 CONVERSION
 SIZE
 SUBSCRIPTRANGE
 CHECK (identifier-list)

Each condition name may be preceded by the word NO; for example, NOOVERFLOW can be specified in the prefix list.

Purpose of the Prefix

The conditions named in the prefix of a statement may occur during program execution of a statement lying within the scope of the prefix (the scope of the prefix is discussed below). If one of these conditions actually does occur, the appearance in the prefix of the corresponding condition name — or its negation with the word NO — determines whether or not an interruption action will then take place.

A condition may or may not cause an interruption depending upon whether or not the condition is enabled. Enabling of the conditions named UNDERFLOW, OVERFLOW, ZERODIVIDE, FIXEDOVERFLOW, and CONVERSION is provided automatically by PL/I; any occurrence of one of these conditions will cause an interruption unless the enabling has been negated through the use of a prefix containing the

condition name preceded by the word NO. The programmer must himself enable the conditions named SIZE and SUBSCRIPTRANGE through the use of a prefix. For example, no interrupt will occur for a SIZE error, unless the error occurs in a statement within the scope of a SIZE prefix.

Scope of the Prefix

The portion of a program for which a condition is enabled by means of a prefix is known as the scope of the prefix. The scope of the prefix depends upon the statement to which it is attached.

If the statement is a PROCEDURE or BEGIN statement, the scope of the prefix is the block defined by this statement, including all nested blocks, except those for which the condition is respecified by means of a prefix. The scope does not include external procedure blocks that are dynamic descendants of the blocks within this scope.

When the statement is an IF statement or an ON statement, the scope of the prefix does not include the units (blocks or groups) that are part of the statement. Any such unit may itself have an attached prefix whose scope rules conform to the rules stated below.

For any statement other than IF, ON, PROCEDURE, or BEGIN, the scope of the prefix is that of the statement itself. The scope includes any expressions appearing in the statement; it does not include a procedure explicitly activated by the statement.

Consider the following example:

```
(SIZE): A: PROCEDURE
      .
      .
      .
      ON SIZE GO TO A_ERROR;
      .
      .
      .
      CALL B;
      .
      .
      END A ;
(SIZE, NOOVERFLOW): B: PROCEDURE;
      .
      .
      ON SIZE GO TO B_ERROR;
      .
      .
      .
      RETURN;
      END B;
```

In this example, the prefix (SIZE) enables the SIZE condition for procedure A and specifies that if a size

error occurs during any calculation in procedure A, an interruption is to take place.

The prefix (SIZE, NOOVERFLOW) for procedure B specifies the same requirement with respect to a SIZE error for procedure B; in addition, it specifies for procedure B that any interruption that might be caused by an OVERFLOW condition is to be suppressed.

After the beginning of execution of procedure A, and before the execution of the first ON statement, any size error will result in an interruption with standard system action. After execution of the first ON statement, and before execution of the ON statement in the activated procedure B, any SIZE error will result in an interruption with execution of the statement GO TO A_ERROR;. After execution of the ON statement in procedure B, the statement GO TO B_ERROR; becomes established for the SIZE condition, but the effect of the previous ON statement is suspended only temporarily. After the RETURN statement in procedure B is executed, the effect of the previous ON statement is reinstated, so that SIZE errors occurring after this point result again in the execution of the statement GO TO A_ERROR;.

If any overflow condition occurs during the execution of procedure A, an interruption will result with the standard system action for the OVERFLOW condition. However, for any occurrence of an overflow condition during the execution of procedure B, the interrupt will be suppressed.

In the following example:

```
(NOOVERFLOW): A: PROCEDURE;
                .
                .
                .
(OVERFLOW):    B: BEGIN;
                .
                .
                .
                END B;
                .
                .
                .
                END A;
```

interruptions will be suppressed for overflow conditions occurring during execution of that part of procedure A that is not included in block B. Overflow conditions occurring during execution of block B will result in an interruption.

ON Conditions

The following discussion presents some of the condition names used to identify ON-conditions and explains the circumstances under which the conditions occur, the standard system action specified by PL/I that would be taken in the absence of a programmer-specified action and, where applicable, the result of any calculation affected by a condition.

- **CONVERSION**

This condition occurs when conversion (either internal or during input/output) from one data type to another causes erroneous results; no assumption should be made about the result of the conversion; the standard system action is to list a comment and cause the ERROR condition to occur.

- **ENDFILE (filename)**

This condition occurs during a GET or READ operation when an attempt is made to read past the file delimiter of the specified file; the standard system action is to list a comment and cause the ERROR condition. If the EVENT option has been specified, the interrupt awaits execution of the associated WAIT statement.

- **ENDPAGE(filename)**

This condition occurs during a PUT operation when an attempt is made to start a new line beyond the limit specified by the PAGESIZE option in the OPEN statement; the condition occurs only once per page so that additional lines can be printed on the page as a result of a programmer-specified action; the standard system action is to start a new page and continue processing.

- **ERROR**

This condition occurs when an error situation forces processing to terminate; the standard system action is to cause the FINISH condition to occur.

- **FINISH**

This condition occurs immediately before processing reaches its normal termination; the standard system action is to terminate processing.

- **FIXEDOVERFLOW**

This condition occurs when the result of a fixed-point arithmetic operation exceeds the maximum size defined by the implementation; the result is undefined; the standard system action is to list a comment and cause the ERROR condition to occur.

- **OVERFLOW**

This condition occurs when the exponent of a floating-point number exceeds the maximum size defined by the implementation; the result is undefined; the

standard system action is to list a comment and cause the ERROR condition to occur.

- **SIZE**

This condition occurs when there is a loss of high-order bits or digits caused by assigning a fixed-point value to a data name declared with the FIXED attribute; the SIZE condition depends upon the declared size of the data name and not upon the maximum size for fixed-point numbers defined by the implementation; no assumption should be made about the result of the assignment; the standard system action is to list a comment and cause the ERROR condition to occur.

- **SUBSCRIPTRANGE**

This condition occurs when a subscript is evaluated and found to lie outside its specified bounds; no assumption should be made about the result of the evaluation; the standard system action is to list a comment and cause the ERROR condition to occur.

- **TRANSMIT(filename)**

This condition occurs when a permanent transmission error is detected on the specified file; the standard system action is to list a comment and cause the ERROR condition to occur. If the EVENT option has been specified, the interrupt awaits execution of the associated WAIT statement.

- **UNDERFLOW**

This condition occurs when the exponent of a floating-point number is smaller than the permitted minimum defined by the implementation; the result is undefined. The standard system action is to list a comment and cause the ERROR condition to occur.

- **ZERODIVIDE**

This condition occurs when division by zero is attempted; no assumption should be made about the result of the division; the standard system action is to list a comment and cause the ERROR condition to occur.

Storage Allocation

Because the internal storage of a computer is limited in size, the efficient use of this storage during the execution of a program is frequently a crucial consideration. PL/I provides several methods for controlling the allocation of storage to the values of a particular data name in a program.

Allocation of storage to the values of a data name may occur statically, that is, before execution of the program, or dynamically, that is, during execution. Storage may be allocated dynamically to the values of a data name and subsequently be released.

Every data name in a program is described with a storage class, which specifies the manner of storage allocation for the data name. There are four storage classes; each is specified by declaring a data name with one of the four storage class attributes: STATIC, AUTOMATIC, CONTROLLED, or BASED. The storage class attributes can be declared for arrays and major structures also. The storage class of a data name may be described either contextually or by default.

Static Storage

Storage for a data name with the STATIC attribute is allocated before execution of the program and is never released during execution. A data name declared with the STATIC attribute may be declared with either the EXTERNAL or INTERNAL scope attribute (discussed earlier in this chapter). An EXTERNAL data name with an unspecified storage class has, by default, the STATIC attribute.

Automatic Storage

If a data name has the attribute AUTOMATIC, the block in which this data name is declared determines the dynamic allocation for the data name. When this block is activated during execution of a program, storage is allocated to the data name, and remains allocated until termination of the block.

Note that termination of a block by means of a GO TO statement may cause simultaneous termination of other blocks and, consequently, simultaneous release of storage for all data names declared in these blocks with the AUTOMATIC attribute.

The scope attribute of a data name declared with the AUTOMATIC attribute must be INTERNAL. The INTERNAL attribute may be explicitly declared or may apply by default. A data name declared with the INTERNAL attribute but with an unspecified

storage class has, by default, the **AUTOMATIC** attribute. If both the storage and scope attributes are not specified for a data name, **AUTOMATIC** storage is assumed to apply to it.

Controlled Storage

When a data name has the attribute **CONTROLLED**, storage allocation must be specified for the data name with the **ALLOCATE** and **FREE** statements. **CONTROLLED** data names may be declared with either the **EXTERNAL** or **INTERNAL** scope attribute.

Based Storage

The **BASED** attribute relates to list processing and **RECORD** transmission. The **BASED** attribute can establish a pointer variable that automatically maintains the storage address of the data associated with the based variable.

A data name having the **BASED** attribute may identify and describe data having any storage class, but the **EXTERNAL** attribute cannot be associated with it. The default storage class for the pointer variable is **AUTOMATIC**.

ALLOCATE Statement

The **ALLOCATE** statement, when executed, causes storage to be allocated to a specified data name declared with either the **CONTROLLED** or **BASED** attribute. An **ALLOCATE** statement may have the following form:

```
ALLOCATE data name attribute 1 . . . attribute
n;
```

The data name in an **ALLOCATE** statement may be the name of an array or of a major structure. If storage is to be allocated for a major structure, then the entire structure, including all level numbers and identifiers, must be included in the **ALLOCATE** statement in the same manner as they appear in the **DECLARE** statement. The **ALLOCATE** statement may allocate to a structure an amount of storage that differs from the amount of storage specified for the structure in a **DECLARE** statement. When this occurs, attributes are required in the **ALLOCATE** statement only for those elementary items in the structure that require a new size.

The only attribute names permitted in an **ALLOCATE** statement are **BIT**, **CHARACTER**, and **INITIAL**. These attributes are required only to indicate a change in the length of a string or to specify a new initial value to be assigned when allocation occurs;

otherwise, no attributes are needed. Because the attributes **FIXED**, **FLOAT**, **DECIMAL**, and **BINARY** are not permitted in an **ALLOCATE** statement, it is not possible to alter the storage size allocated to data items described with these attributes.

When the length of a string or the number of items in a dimension of an array is specified in an **ALLOCATE** statement, it overrides similar information given in a **DECLARE** statement. If the length of a string or the number of items in a dimension of an array is not specified in an **ALLOCATE** statement, it must be specified in a **DECLARE** statement.

Asterisks may be used in place of the string length or the number of items in a dimension for a data name specified in an **ALLOCATE** statement. The asterisks indicate that the string length or dimension limits are the same as those of the most recent allocation for the data name.

Consider the following example:

```
A: PROCEDURE;
    DECLARE PRICE (100) FIXED (4, 2)
    CONTROLLED, COUNT FIXED (2)
    INITIAL (50);
    .
    .
    .
    ALLOCATE PRICE;
    .
    .
    .
    ALLOCATE PRICE (75);
    .
    .
    .
    ALLOCATE PRICE (*);
    .
    .
    .
    ALLOCATE PRICE (COUNT);
    .
    .
    .
END A;
```

The array called **PRICE** is declared to consist of 100 positions. The attribute **CONTROLLED**, however, indicates that storage for **PRICE** will be allocated only when an **ALLOCATE** statement is executed for **PRICE**. The first **ALLOCATE** statement in the example uses the number 100 specified in the **DECLARE** statement as the number of positions in **PRICE**. When the second **ALLOCATE** statement is executed, only 75 array positions are allocated to **PRICE**. The third **ALLOCATE** statement uses the number of positions established in the most recent allocation to **PRICE**.

The last **ALLOCATE** statement uses the value of **COUNT** to determine the number of positions to be allocated to **PRICE**.

The storage class for **COUNT** is **AUTOMATIC**. Each time procedure **A** is activated, storage is allocated to **COUNT**, and the integer **50** is assigned as the value of **COUNT**. When control leaves procedure **A** the storage allocated to **COUNT** is released; however, the storage allocated to **PRICE** remains allocated until released by a **FREE** statement.

An **ALLOCATE** statement may be used to allocate storage for more than one data name. Successive data names appearing in an **ALLOCATE** statement are separated by commas. In the following example:

```
ALLOCATE SCHEDULE(50), COST,
CODE CHARACTER(10), AMOUNT;
```

the storage required for the data names **SCHEDULE**, **COST**, **CODE**, and **AMOUNT** is allocated when the **ALLOCATE** statement is executed.

FREE Statement

The **FREE** statement causes the storage most recently allocated to specified data names to be released. The general form of a **FREE** statement is:

```
FREE data name 1, . . . , data name n;
```

Each data name must be of the **CONTROLLED** storage class and may also be the name of an array or a structure. If a specified data name has no allocated storage when the **FREE** statement is executed, no action occurs for that data name.

The following example illustrates the use of a **FREE** statement together with an **ALLOCATE** statement:

```
A: PROCEDURE;
  DECLARE TAX(100) FIXED(4,2)
  CONTROLLED;
  .
  .
  .
  ALLOCATE TAX;
  .
  .
  .
  GET LIST(TAX);
  .
  .
  .
  PUT EDIT(TAX)(F(4,2));
  .
  .
  .
  FREE TAX;
  .
  .
  .
END A;
```

TAX is declared as an array containing 100 positions and has the **CONTROLLED** storage class. When the **ALLOCATE** statement is executed storage is generated for **TAX**. Data is then read into and written from the **TAX** table and, when the **FREE** statement is executed, the storage for **TAX** is released.

Storage may be allocated to a data name in one block and freed in another block if the data name has been declared so that its scope includes both blocks.

PL/I and COBOL Comparison: Program Structure

The following discussion compares the ways in which programs are constructed in **PL/I** and in **COBOL**. Although both languages are problem-oriented and employ the statement as the basic program element for processing data and for altering the sequence of program execution, it is in the area of program construction that **PL/I** and **COBOL** differ the most. The following list points out some of the more significant differences in the program structure of the two languages.

1. In general, a **COBOL** program is equivalent to one external procedure in **PL/I**. The Data Division and the Environment Division of **COBOL** correspond for the most part to the **DECLARE** statement in **PL/I**. The **COBOL** Procedure Division is equivalent to the executable statements in a **PL/I** procedure. The function served by the **COBOL** Identification Division is provided in **PL/I** by comments.
2. The **ENTER** statement in **COBOL** is equivalent to the **CALL** statement in **PL/I** when the **CALL** statement is used to activate separately compiled external procedures. However, the concept of nested procedure blocks (internal procedure blocks) in **PL/I** has no counterpart in **COBOL**. Consequently, it is not possible within the same **COBOL** source program to define internal subprograms to which arguments may be assigned.
3. **COBOL** does not provide the equivalent **PL/I** facilities for automatic and controlled allocation of storage.
4. The **AT END**, **INVALID KEY**, and **SIZE ERROR** options in **COBOL** are provided in **PL/I** by the **ON** statement. However, **PL/I** provides a fuller range of interruption conditions than does **COBOL**.
5. The effect of the **ALTER** statement in **COBOL** is achieved in **PL/I** by assigning a label constant to the label name in a **GO TO** statement.
6. The **PERFORM** statement in **COBOL** and the **DO** statement in **PL/I** may be used for similar purposes. The **PERFORM** statement, however, is used for out-of-line loop control whereas the **DO** statement is used for in-line loop control.

Introduction

The discussions in previous chapters have explained how data is described, how it is transmitted into and out of a computer, and how a program is organized to control the sequence of statement execution. This chapter discusses the means provided by PL/I for manipulating and processing data.

In general, data processing is concerned with using available data to generate or modify other data. For example, the data contained in an employee's time card can be used to generate his paycheck and to update his year-to-date earnings stored in a master file. Such manipulations are indicated in PL/I by means of expressions that employ data names, constants, and operators. When an expression is evaluated, the value of the expression is assigned to a data name by means of the assignment statement. The following discussion shows how different types of expressions may be employed in an assignment statement to generate or modify string and arithmetic data items. This chapter includes a discussion of the picture specification and how it is used to modify data through editing.

Assignment Statement

The form of an assignment statement may be written:

data name = expression;

No keyword appears at the beginning of an assignment statement. The equal sign serves as a keyword and states that the value of the expression on the right is to be assigned as the value of the data name on the left. An expression in an assignment statement may consist of:

1. A constant
2. A data name
3. A sequence of constants, data names, and operators

The following assignment statement illustrates the use of a constant as an expression:

```
PRICE = 19.95;
```

When this statement is executed, a data item equivalent in value to the constant 19.95 is created and stored in the storage area associated with PRICE. Any previous data item stored in the storage area for PRICE is no longer available, because it is replaced by the

new data item. The representation of the new data item must conform to the attributes of PRICE; this may involve the creation of a data item that differs in its characteristics from the constant 19.95. For example, if PRICE has the attributes FIXED DECIMAL (5, 2), indicating that the data item named PRICE is a five-digit fixed-point decimal number with two decimal places, then the data item created for PRICE in the above example will be equivalent to the constant 019.95; the zero digit on the left is automatically provided to satisfy the 5-digit length as specified by the attributes of price.

Assume PRICE has the same attributes as described above and is employed in the following assignment statement:

```
PRICE = 5;
```

Execution of this statement creates for PRICE a data item equivalent to the constant 005.00; again, the zero digits are provided to satisfy the requirements for length and number of decimal places as specified by the attributes of PRICE.

It is also possible for the length of a numeric constant to exceed the length specified for PRICE. Consider the following example:

```
PRICE = 12345.678;
```

If PRICE has the attributes FIXED DECIMAL (5,2), the data item created for PRICE is equivalent to the constant 345.67; in this case two digits are truncated on the left and one on the right to satisfy the requirements for length and number of decimal places as specified by the attributes of PRICE.

In addition to automatically adjusting the size of a created data item, an assignment statement may perform more complicated types of data adjustment. For example, it could convert the representation of an arithmetic value from floating-point to fixed-point. Suppose PRICE had the attributes described above, and appeared in the following assignment statement:

```
PRICE = 1.23E2;
```

Execution of this statement would cause the data item created for PRICE to be equivalent to the constant 123.00. The floating-point decimal number 1.23E2 is converted to the equivalent fixed-point decimal number 123; this number is then extended on the right with zeros to provide two decimal places as required by the attributes of PRICE.

The following statement employs the same attributes for PRICE stated above, but involves an automatic conversion from fixed-point binary to fixed-point decimal.

```
PRICE = 100.1B;
```

In effect, the fixed-point binary constant 100.1B is converted to the fixed-point decimal constant 4.5; this number is extended with zeros to satisfy the number of digits and decimal places required by the attributes of PRICE. A data item equivalent to the constant 004.50 is then assigned to PRICE.

The statement above could also have been written using a floating-point binary constant.

```
PRICE = 1.001E2B;
```

Because the floating-point binary constant 1.001E2B is equivalent in value to the fixed-point binary constant 100.1B, the same data item assigned to PRICE by the previous statement is also assigned by this statement.

String data can be used in an assignment statement. Consider the following example where NAME has the attribute CHARACTER (5).

```
NAME = 'JONES';
```

When this statement is executed, a data item equivalent to the character-string constant 'JONES' is created and stored in the storage area associated with NAME.

If the length of the character-string constant exceeds the length specified by the attributes of NAME, characters are removed from the right end of the string created for NAME. Consider the statement:

```
NAME = 'SMITHSONIAN';
```

If NAME still has the attribute CHARACTER (5), then execution of this statement creates the character-string constant 'SMITH' and assigns it to NAME. Because the data item assigned to NAME cannot exceed a length of five characters, the six characters at the right-hand end of the character-string constant 'SMITHSONIAN' are deleted.

When the length of the character-string constant is less than the length specified for NAME, the character string created for NAME is extended on the right with blank characters until the specified length is attained.

In the case of varying-length character string, blank characters are not appended on the right. For example, assume TITLE has been declared to be the name of a varying-length character string with maximum length 30, that is, it has the attributes VARYING CHARACTER (30). The following statement:

```
TITLE = 'ACCOUNTING PRACTICE';
```

creates a character string data item equivalent to the string constant 'ACCOUNTING PRACTICE' and assigns it to TITLE. No blank characters are appended on the right because the character length specified for TITLE may vary. When this statement is executed the current length for TITLE becomes 19 characters. This length may be increased or decreased by subsequent assignment statements. However, when the length of the created character string exceeds the maximum length of 30, the excess characters are deleted from the right end of the created string.

Bit-string constants are treated like character-string constants, except that short bit strings are extended on the right with zero bits. Truncation of bit strings also occurs on the right. The following example illustrates the use of a bit-string constant in an assignment statement.

```
CODE = '10101' B;
```

Assume that CODE has the attribute BIT(10); then the bit string constructed for CODE is equivalent to the bit-string constant '1010100000'B.

PL/I also allows a label constant to appear as an expression in an assignment statement. When this occurs, the name on the left-hand side of the equal sign in the assignment statement must be a label name. Assume that the name POINTER has been declared with the attribute LABEL and consider the use of POINTER in the following assignment statement:

```
POINTER = DEDUCTION;
```

Because POINTER is a label name, DEDUCTION must be a statement label. When this statement is executed the data item created for POINTER is the statement label DEDUCTION. There are two restrictions on the use of a label constant in an assignment statement. The label constant must not be an entry name, that is, it must not be the label of a PROCEDURE statement. This restriction prevents the erroneous transfer of control to a PROCEDURE statement by means of a GO TO statement (PROCEDURE statements must be activated only by a CALL statement). The second restriction is that the scope of the label constant must be the same as the scope of the label name on the left side of the assignment statement. Without this restriction a GO TO statement might erroneously attempt to send control to a block outside the scope of the label name.

The expressions considered so far have consisted solely of constants. An expression to be considered in an assignment statement may be a data name rather than a constant. The following example illustrates the use of such an expression:

```
PRICE = COST;
```

When this statement is executed, a data item equivalent in value to the data item currently assigned to `COST` is created and stored in the storage area associated with `PRICE`. The data item associated with `COST` remains unmodified. The conversion techniques discussed for constants also apply to this statement. For example, if the attributes of `PRICE` specify a fixed-point decimal number, and the attributes of `COST` specify a floating-point binary number, the creation of the data item for `PRICE` will involve a conversion from floating-point binary representation to fixed-point decimal representation.

Label names are also permitted as expressions in an assignment statement. If the names `SWITCH` and `POINTER` are declared with the `LABEL` attribute, then execution of the following assignment statement:

```
POINTER = SWITCH;
```

assigns to `POINTER` the current location identified by `SWITCH`.

Conversion Between Data Types

Data names and expressions of unlike data type can be used in assignment statements. For example, if `CODE` is declared with the attribute `CHARACTER (10)` and `MASK` with an attribute `BIT (10)`, the following assignment statement:

```
CODE = MASK;
```

will create, as a data item for `CODE`, a string of ten characters that contains only the characters 1 and 0. The 1 bits in `MASK` become the 1 characters in `CODE`, and the 0 bits in `MASK` become the 0 characters in `CODE`. If the character length specified for `CODE` exceeds the bit length specified for `MASK`, the data item created for `CODE` is extended on the right-hand side with blank characters. Truncation of excess characters on the right of the data item created for `CODE` occurs when the bit length specified for `MASK` exceeds the character length specified for `CODE`.

The example above illustrates how an assignment statement may convert the bit-string representation of a value to the equivalent character-string representation. In an assignment statement, there are six ways in which data values can be converted from one representation to another.

1. Bit-string Data to Character-String Data
2. Character-String Data to Bit-String Data
3. Bit-String Data to Arithmetic Data
4. Character-String Data to Arithmetic Data
5. Arithmetic Data to Character-String Data
6. Arithmetic Data to Bit-String Data

The following discussion states the rules employed by PL/I for each of these conversions.

Bit-String Data to Character-String Data

A character string is created that contains 1 characters and 0 characters. The 1 characters in the character string correspond to the 1 bits in the bit string. The 0 characters correspond to the 0 bits.

Character-String Data to Bit-String Data

A bit string is created that contains 1 bits and 0 bits. The 1 and 0 bits in the bit string correspond, respectively, to the 1 and 0 characters in the character string. No characters other than 1 and 0 characters are permitted in the character string; otherwise, the `CONVERSION` error-condition will occur (see "ON-Conditions" in Chapter 4).

Bit-String Data to Arithmetic Data

The value of the bit string is interpreted as an unsigned fixed-point binary integer and is represented as an arithmetic data item. This data item has the attributes of the data name for which it is created (for example, floating-point decimal, fixed-point binary, etc.). When the length of a varying-length bit string is zero, the value of the bit string (and of the arithmetic data item) is zero.

Character-String Data to Arithmetic Data

The character string must have the form of a PL/I arithmetic constant, otherwise the `CONVERSION` error-condition will occur (see the `ON` statement in Chapter 4). The value of this arithmetic constant is used to create an arithmetic data item. The attributes of the data name for which the arithmetic data item is created determine the representation of the data item (for example, fixed-point decimal, floating-point binary, etc.). When the length of a varying-length character string is zero, the value of the character string (and of the arithmetic data item) is zero.

Arithmetic Data to Character-String Data

The value of the arithmetic data item is represented as a character string. The format of this character string is determined by the rules for list-directed output discussed in Chapter 3.

Arithmetic Data to Bit-String Data

The value of the arithmetic data item is represented as a fixed-point binary number. The integral portion of this number is then used to create the bit-string data item.

Expressions Containing Operators

PL/I provides four classes of operators: arithmetic, comparison, bit, and concatenation. These operators may be combined with constants and data names to form more intricate expressions than previously considered in this chapter. Such expressions may appear not only in assignment statements, but also in other statements. For example, the IF and DO statements employ expressions for control purposes. In the DECLARE statement, expressions may appear with the INITIAL, BIT, and CHARACTER attributes. Expressions are used in input/output statements with some of the control format items and as optional repetition factors before format items. Expressions are also used to specify subscript values for subscripted names. An expression may also be used in the DISPLAY statement to create a message.

Arithmetic Expressions

Five arithmetic operators are available in PL/I for arithmetic expressions:

| | |
|----|------------------|
| + | (addition) |
| - | (subtraction) |
| * | (multiplication) |
| / | (division) |
| ** | (exponentiation) |

The exponentiation operator consists of two adjacent asterisks and is used to raise a value to a power. For example, the expression:

COUNT**2

indicates that the value of COUNT is to be squared.

Arithmetic expressions may employ parentheses in order to group the constants and data names appearing in the arithmetic expression and to alter the order in which the operations are executed. A priority system (discussed later) is employed to control the sequence of operations when parentheses are not present in an expression.

The following examples illustrate how data names and constants may be combined with arithmetic operators to form arithmetic expressions:

| | |
|--------------|----------------------|
| GROSS-NET | (SUM/COUNT)**0.5 |
| 139.99+TAX | +50 |
| NUMBER*PRICE | -(2*INDEX) |
| 144*11.50 | A+((B-C)*(D+E)) |
| 256/LENGTH | ((B**2)-(4*A*C))**.5 |

Blank characters are not required on either side of operators or parentheses. Only + and - are permitted as prefix operators. A prefix operator applies to constants and data names that appear to the right of the

prefix operator; an infix operator applies to the constants and data names on either side of the infix operator. In these examples, the + and - operators are used as prefix operators in +50 and in -(2*INDEX), and as infix operators in 139.99+TAX and GROSS-NET.

When parentheses are not employed, the priority of arithmetic operators is:

** , prefix + , prefix - (highest priority)
* , /
infix + , infix - (lowest priority)

Consider the evaluation of the following expression:

A = COMMISSION+BASE*ADJUSTMENT

Because multiplication has a higher priority than addition, the value of BASE is first multiplied by the value of ADJUSTMENT and the result is then added to the value of COMMISSION. In order to specify the execution of addition before the execution of multiplication, the same expression may employ parentheses as follows:

B = (COMMISSION+BASE) * ADJUSTMENT

In this expression, the values of COMMISSION and BASE are added, and the resulting value is then multiplied by the value of ADJUSTMENT. Note that the values given by the two expressions are not equal.

Successive operators of equal priority are executed from left to right; for example, in the expression:

BASE+COMMISSION+OVERTIME

BASE and COMMISSION are added first and the result is then added to OVERTIME. However, the operators **, prefix +, and prefix - are an exception to this rule; when used successively, these operators are executed from right to left. The following expression demonstrates this usage:

AVERAGE** -0.5

This expression is evaluated as though it were written as:

AVERAGE**(-0.5)

The following examples are additional illustrations of the priority rules discussed above:

| | |
|----------|------------------------------|
| A*B/C | is equivalent to (A*B)/C |
| A/B*C | is equivalent to (A/B)*C |
| A*B-C*D | is equivalent to (A*B)-(C*D) |
| A-B*C-D | is equivalent to (A-(B*C))-D |
| A*B** -C | is equivalent to A*(B**(-C)) |
| -A-B | is equivalent to (-A)-B |

In the last expression, the first minus sign is a prefix - and is applied to the A before the infix - is executed.

Conversion of Arithmetic Data

Arithmetic operators in an expression may be used with data names and constants that possess different data characteristics. When the value of a decimal number and the value of a binary number are joined by an arithmetic operator, the value of the decimal number is converted to a binary value. Similarly, the value of a fixed-point data item is converted to a floating-point value, except when the fixed-point item is a positive exponent, that is, a positive integer appearing to the right of an exponentiation operator. In the latter case, the fixed-point value is not converted to floating-point.

When floating-point data appears on the left of an arithmetic operator, the result of the arithmetic operation is in floating-point format, and the accuracy of the result is the greater accuracy of the two numbers involved in the operation.

Fixed-point data on the left of an arithmetic operator causes the result to be in fixed-point format. However, if the operation is exponentiation and if the fixed-point item on the right is not a positive integer, both numbers involved in the exponentiation are converted to floating-point format and the result is in floating-point format.

Conversion from floating-point format to fixed-point format occurs only when explicitly required, as in the case of an assignment statement that has a data name described as fixed-point to the left of the equal sign. If this name cannot accommodate the converted floating-point value with full accuracy, truncation will occur as required on both sides of the decimal point, and the SIZE error-condition may be produced (see "ON Statement" in Chapter 4).

After conversions have occurred, the arithmetic operation is performed. The accuracy of intermediate arithmetic results occurring in the evaluation of an expression is automatically accounted for; however, each implementation of PL/I may restrict the accuracy of intermediate results to a specified maximum.

Arithmetic expressions are permitted as subscripts (see Chapter 2). When a subscript expression is evaluated, the result is converted, if necessary, to fixed-point binary and truncated, if necessary, to an integer. A maximum subscript value is specified for each PL/I implementation.

String data may also be employed in an arithmetic operation; when this occurs the string data is converted to arithmetic data. The conversion satisfies the rules discussed earlier in this chapter under "Conversion Between Data Types." The use of string data in arithmetic expressions may result in the inefficient execution of a compiled program; this is particularly true when the same expression must be evaluated many

times during the execution of a program. When efficiency is desired, string data should be converted to an arithmetic representation before it is employed in arithmetic expressions.

Comparison Expressions

A brief description of comparison expressions appears in Chapter 4. The following discussion presents a more detailed description of the various types of comparison expressions, and shows how comparison expressions may be used for purposes other than that of controlling the sequence of execution within an IF statement.

The eight comparison operators are:

- > (greater than)
- \neg > (not greater than)
- >= (greater than or equal to)
- = (equal to)
- \neg = (not equal to)
- < (less than)
- \neg < (not less than)
- <= (less than or equal to)

They may be used to form three types of comparison expressions: arithmetic, character, and bit. The values of arithmetic data items are compared algebraically; negative values compare lower than positive values. Character strings are compared character by character from left to right. When the character strings differ in length, both lengths are made equal by appending blank characters to the right of the shorter string. An implementation-defined collating sequence specifies the order of character comparison. Bit strings involve the left-to-right comparison of binary digits. When the bit strings differ in length, the shorter string is extended on the right with zero bits. A zero bit compares lower than a one bit.

The value of a comparison expression is represented by a bit string of length one. The bit constant '1'B represents the value of a true comparison; the bit constant '0'B represents the value of a false comparison. Consider the following comparison expression:

```
HOURS > 40
```

This expression is true when the numeric value of HOURS is greater than 40; when this occurs, the value of the comparison expression is the bit constant '1'B; otherwise, the value is '0'B.

The data names and constants used in a comparison expression may be of different data types. A priority on data types is used to convert the values involved in a comparison expression from one data type to another. Arithmetic data has the highest priority; character-string data, the next; and bit-string data, the lowest priority. Conversion of data types in a com-

parison expression is always from a lower priority to a higher priority. Conversion of data types follows the rules discussed earlier in this chapter.

A comparison expression may be used in IF statements. A comparison expression may also be used in an assignment statement. For example, the following assignment statement:

```
CODE = HOURS > 40;
```

contains the comparison expression `HOURS > 40`. When this statement is executed, the value of the comparison expression is represented as a bit string one bit long and is assigned to `CODE`. The assignment conforms to the attributes of `CODE` and may involve conversion from one data representation to another. Conversion, if required, follows the rules specified earlier in this chapter.

Comparison expressions may appear in arithmetic expressions. Consider the following assignment statement:

```
SALARY = BASE + (HOURS > 40)*BONUS;
```

When this statement is executed, the value of the comparison expression `HOURS > 40` is multiplied by the numeric value of `BONUS`. Because the value of the comparison is represented as a bit string, a conversion from bit-string representation to arithmetic representation is required before the multiplication is performed. If the comparison is true the value of `BONUS` is multiplied by the integer one; otherwise, the value of `BONUS` is multiplied by the integer zero. The result of the multiplication is then added to the value of `BASE`, and this result becomes the value of `SALARY`.

The comparison operators have a lower priority than the arithmetic operators. Hence, the following expression:

```
BASE + HOURS > 40 * BONUS
```

is equivalent to the expression:

```
(BASE + HOURS) > (40 * BONUS)
```

Bit-String Expressions

The three bit-string operators are:

```

 $\neg$       (not)
&       (and)
|       (or)

```

They are used with bit-string data to form bit-string expressions. The operator `&` and `|` are infix operators, that is, they operate upon a pair of bit strings. The operator `\neg` is a prefix operator and, consequently, operates on a single bit string.

When the `&` and `|` operators are applied to bit strings of unequal length, the lengths are made equal by appending zero bits to the right of the shorter bit

string. The result of a bit-string operation is a bit string equal in length to the bit strings involved in the operation.

Bit-string operations are performed bit by bit, from left to right. Each position in the resulting bit string has the value defined in the following table:

| If A is: | If B is: | Then \neg A is: | Then \neg B is: | Then A&B is: | Then A B is: |
|-------------|-------------|----------------------|----------------------|-----------------|-----------------|
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 |

Consider the three data names `MASK`, `CODE`, and `SWITCH`, and assume each has been declared with the attribute `BIT(6)`; also assume that:

```

MASK   has the value '010111'B
CODE   has the value '111111'B
SWITCH has the value '101'B

```

The following expressions illustrate the effect of applying bit-string operators to these data names.

```

 $\neg$ MASK           has the value '101000'B
SWITCH & CODE    has the value '101000'B
SWITCH | MASK    has the value '111111'B

```

Bit-string expressions that employ two or more bit-string operators are evaluated from left to right according to a priority on the bit-string operators. The operator `\neg` has the highest priority; the operator `|` has the lowest priority. As is the case with arithmetic expressions, parentheses may be used in bit-string expressions to alter the priority of bit-string operators. The following examples illustrate the effect of this priority.

```

A|B&C   is equivalent to A|(B&C)
A&B|C   is equivalent to (A&B)|C
 $\neg$ A|B&C  is equivalent to ( $\neg$ A)|(B&C)
 $\neg$ A&B|C  is equivalent to (( $\neg$ A)&B)|C
 $\neg$ A& $\neg$ B|C is equivalent to (( $\neg$ A)&( $\neg$ B))|C

```

Bit-string operators may be combined with arithmetic and comparison operators in the same expression. In general, bit-string operators have a lower priority than the comparison operators, and, as mentioned earlier, the comparison operators have a lower priority than the arithmetic operators. However, the operator `\neg` takes precedence over any operators immediately to its left or to its right. Parentheses may modify the priority of all operators. The following examples illustrate the use of arithmetic, comparison, and bit-string operators in the same expression.

$A > B \ \& \ C < D$
 is equivalent to
 $(A > B) \ \& \ (C < D)$
 $A + B < C - D \ | \ E > F$
 is equivalent to
 $((A + B) < (C - D)) \ | \ (E > F)$
 $\neg A < B \ \& \ C = D$
 is equivalent to
 $((\neg A) < B) \ \& \ (C = D)$
 $A \ \& \ \neg B \ | \ \neg C > D$
 is equivalent to
 $(A \ \& \ (\neg B)) \ | \ ((\neg C) > D)$
 $A + B \ \& \ C * D$
 is equivalent to
 $(A + B) \ \& \ (C * D)$

When expressions are evaluated, intermediate results are converted from one data representation to another as required by each operator. In the last example above, the arithmetic results of the expressions $A + B$ and $C * D$ are converted to bit strings before the operator $\&$ is carried out.

Bit-string expressions can be used as well as comparison expressions in an IF statement. In an IF statement, when a bit-string expression is evaluated and this resulting bit string contains one or more 1 bits, the expression is considered to be true; otherwise, it is considered to be false. In the following example:

```
IF  $\neg(A \ \& \ B)$  THEN GO TO REORDER;
ELSE GO TO INVENTORY;
```

when the bit string resulting from the evaluation of the bit-string expression $\neg(A \ \& \ B)$ contains one or more 1 bits, the statement GO TO REORDER is executed; otherwise, the statement GO TO INVENTORY is performed.

Concatenation Expressions

The concatenation operator $\|$ enables one bit or character string to be appended to the right end of another.

For example, the following concatenation expression:

```
'1234' \| 'ABCD'
```

is equivalent to the character-string constant:

```
'1234ABCD'
```

When bit strings are concatenated, the result is a bit string. The value of the following concatenation expression:

```
'1111'B \| '0000'B \| '1111'B
```

is equivalent to the bit-string constant:

```
'111100001111'B
```

When one of the data items in a concatenation operation is not a character string, the data item is

converted to character string, and the result is a character string. Consider the following expression:

```
'10101'B \| 'Z9X8Y7'
```

When evaluated, this expression is equivalent to the following character string:

```
'10101Z9X8Y7'
```

Execution of the following statement:

```
LABEL = NAME \| ADDRESS;
```

causes the values of NAME and ADDRESS to be converted, if necessary, to character strings. The character string representing the value of ADDRESS is then appended to the right end of the character string associated with NAME. The resulting character string is assigned to LABEL in conformity with the attributes of LABEL.

The concatenation operator may appear with arithmetic, comparison, and bit-string expressions; when so employed, this concatenation operator has the lowest priority of all operators.

Consider the following statement:

```
COMMENT = 'SALARY IS $' \| BASE + COMMISSION;
```

The arithmetic expression $BASE + COMMISSION$ is evaluated and the result is converted to a character string. This string is then appended to the character-string constant 'SALARY IS \$', and the result is assigned to COMMENT.

Array Expressions

The data names permitted in an assignment statement may be subscripted names. Consider the following declaration, where TAX, TAXABLE, and RATE are the names of three arrays:

```
DECLARE TAX (100) FIXED (4,2),
TAXABLE (100) FIXED (5,2), RATE (100)
FIXED (2,2);
```

The following assignment statement:

```
TAX (1) = TAXABLE (1) * RATE (1) ;
```

multiplies the first value in the TAXABLE array by the first value in the RATE array and stores the result in the first position of the TAX array. If similar calculations are to be performed for all positions in these arrays, a DO statement may be employed as follows:

```
DO I = 1 TO 100 BY 1 ;
TAX (I) = TAXABLE (I) * RATE (I) ;
END ;
```

When this group of statements is executed, the data name I is initially set to 1, and the assignment statement is executed. Control then returns to the DO statement; the value of I is increased to 2, and the assignment statement is executed again. This cycle of

operations is repeated until *i* has been incremented to 100.

PL/I provides a simpler way of obtaining the results described above; an array name may be used without subscripts to indicate that all positions in the array are to be operated upon. For example, the following assignment statement:

$$\text{TAX} = \text{TAXABLE} * \text{RATE} ;$$

is equivalent in effect to the DO group discussed above.

Operations performed on arrays are performed element by element; therefore, all arrays involved in an array expression must have the same number of elements and the same number of dimensions.

The result of an array expression is itself an array. A constant or a data name for an item that is not an array may be employed with an array name in an array expression; when so employed, the value of the constant or the data name is applied to each position in the array. For example, if a one-dimensional array called COUNT consisting of six elementary items had the following values:

$$5, -10, 11, -3, 2, 7$$

the array expression $-\text{COUNT}$ would have as its values the following:

$$-5, 10, -11, 3, -2, -7$$

The expression $3 * (-\text{COUNT})$ would have the following values:

$$-15, 30, -33, 9, -6, -21$$

If INDEX is another one-dimensional array consisting of six elementary items with the following values:

$$2, -10, -10, 1, 0, 40$$

then the value of the array expression $\text{COUNT} + \text{INDEX}$ would be:

$$7, -20, 1, -2, 2, 47$$

Similarly, the expression $\text{COUNT} * \text{INDEX}$ would have the value:

$$10, 100, -110, -3, 0, 280$$

Structure Expressions

The names of structures are also permitted in expressions. All structures in an assignment statement must have identical structuring, that is, each structure must contain the same number of data names; however, level numbers need not be identical. The data types written for items in structures need not be the same. Items in structures specified in an assignment statement may themselves be arrays or the structures themselves may be arrays. Should this be the case, arrays specified at equivalent levels in different structures must have the same number of dimensions and must have the same number of items in corresponding dimensions.

Consider the following structures:

$$\begin{aligned} &1 \text{ TOTAL}, 2 \text{ GROSS}, 2 \text{ NET}, 2 \text{ TAX}, \\ &1 \text{ OLD}, 2 \text{ OLD_GROSS}, 2 \text{ OLD_NET}, 2 \text{ OLD_TAX}, \\ &1 \text{ NEW}, 2 \text{ NEW_GROSS}, 2 \text{ NEW_NET}, 2 \text{ NEW_TAX}, \end{aligned}$$

The assignment statement:

$$\text{TOTAL} = \text{OLD} + \text{NEW};$$

is equivalent to the following sequence of statements:

$$\text{GROSS} = \text{OLD_GROSS} + \text{NEW_GROSS};$$

$$\text{NET} = \text{OLD_NET} + \text{NEW_NET};$$

$$\text{TAX} = \text{OLD_TAX} + \text{NEW_TAX};$$

This example shows that using the name of a structure in an assignment statement is a concise notation for using the data names of the elementary items within the structure. A structure expression causes elementary items in corresponding positions of the structures to be operated upon; therefore, the result of a structure expression is itself a structure.

Constants and data names that are not the names of structures or arrays may be used with structure names in a structure expression; when so used, the value of the constant or the data name applies to each elementary item in the structure. Using the structure TOTAL described above, consider the following statement.

$$\text{TOTAL} = \text{TOTAL} * 2;$$

This statement is equivalent to the following sequence of statements:

$$\text{GROSS} = \text{GROSS} * 2;$$

$$\text{NET} = \text{NET} * 2;$$

$$\text{TAX} = \text{TAX} * 2;$$

The names of structures and the names of arrays must not appear in the same expression. However, an expression may employ an array composed of structures. Consider the following structures:

$$1 \text{ TOTAL} (10), 2 \text{ GROSS}, 2 \text{ NET}, 2 \text{ TAX},$$

$$1 \text{ OLD} (10), 2 \text{ GROSS}, 2 \text{ NET}, 2 \text{ TAX},$$

$$1 \text{ NEW} (10), 2 \text{ GROSS}, 2 \text{ NET}, 2 \text{ TAX},$$

Each structure consists of 30 elementary items; the elementary items GROSS, NET, and TAX are repeated 10 times in each structure. The statement:

$$\text{TOTAL}(I) = \text{OLD}(J) + \text{NEW}(K);$$

is equivalent to the following sequence of statements:

$$\text{TOTAL}(I).\text{GROSS} = \text{OLD}(J).\text{GROSS} + \text{NEW}(K).\text{GROSS};$$

$$\text{TOTAL}(I).\text{NET} = \text{OLD}(J).\text{NET} + \text{NEW}(K).\text{NET};$$

$$\text{TOTAL}(I).\text{TAX} = \text{OLD}(J).\text{TAX} + \text{NEW}(K).\text{TAX};$$

Qualification in these statements is required to make GROSS, NET, and TAX unique. Each of the subscripts I, J and K specifies one of the ten possible sets of data names consisting of GROSS, NET, and TAX.

Because subscripts may be moved to the right end of a qualified name, the above statements may be written:

```
TOTAL.GROSS(I) = OLD.GROSS(J) +
    NEW.GROSS(K);
TOTAL.NET(I) = OLD.NET(J) + NEW.NET(K);
TOTAL.TAX(I) = OLD.TAX(J) + NEW.TAX(K);
```

Assignment BY NAME

An assignment statement involving structures may perform element-by-element assignment on the basis of corresponding names rather than on the basis of corresponding positions in the structures. This type of assignment is called assignment by name and is indicated by appending BY NAME (with a separating comma) to the right of the structure expression in the assignment statement, thus:

```
structure-name=structure-expression, BY
NAME;
```

When structures are used with the BY NAME option in an assignment statement, corresponding data names must be contained in their respective structures in the following manner: any names by which corresponding data names may be qualified must be identical, except for the name written in the assignment statement. Thus, the following structures could be used in an assignment statement with the BY NAME option:

```
1 MASTER,2NAME,3FIRST,3LAST,2ADDRESS,
1DETAIL,2ADDRESS,2NAME,3FIRST,3MIDDLE,
3LAST,
```

The corresponding names in these structures are NAME, FIRST, LAST, and ADDRESS. This example shows that the declared order of the items within structures used in an assignment statement with the BY NAME option need not be the same. This example also shows that it is not necessary for structures to have identical structuring when used in an assignment statement with the BY NAME option.

Using the structures described above, consider the following statement:

```
MASTER = DETAIL, BY NAME;
```

This statement is equivalent to the following sequence of statements:

```
MASTER.NAME.FIRST = DETAIL.NAME.FIRST;
MASTER.NAME.LAST = DETAIL.NAME.LAST;
MASTER.ADDRESS = DETAIL.ADDRESS;
```

Expressions are also evaluated on the basis of corresponding names when the expressions appear in an

assignment statement that uses the BY NAME option. Consider the following structures:

```
1 TOTAL, 2 GROSS, 2 NET, 2 TAX,
1 OLD, 2 DATE, 2 GROSS, 2 NET,
1 NEW, 2 DATE, 2 NET, 2 GROSS,
```

The following assignment statement:

```
TOTAL = OLD + NEW, BY NAME;
```

is equivalent to the following sequence of statements:

```
TOTAL.GROSS = OLD.GROSS + NEW.GROSS;
TOTAL.NET = OLD.NET + NEW.NET;
```

Although DATE appears in both the OLD and the NEW structures, no addition is carried out for DATE because this name is not part of the structure TOTAL.

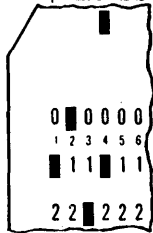
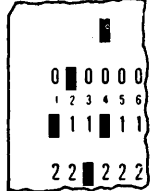
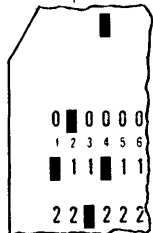
Data Editing

The previous discussions in this chapter have considered those aspects of data manipulation that involve the evaluation of expressions and the assignment of values to names. The following discussion presents an additional type of data manipulation that permits the detailed editing of data items for printing purposes. Editing may be described as an alteration of the format and/or punctuation of a character-string data item, usually for such purposes as improving readability or, as with paychecks, protecting the data item against unauthorized alteration. Editing involves the addition of characters to the data item and/or the replacement of specified characters with other characters. For example, in a payroll application, the digits representing an employee's salary might be 0015089. These digits would be much more meaningful on a paycheck in an edited form, such as \$**150.89; the asterisks would also hamper an attempt to alter the amount.

Editing is specified by a picture specification in either a PICTURE attribute (discussed in Chapter 2) or a P format description (discussed in Chapter 3). The PICTURE attribute appearing with a data name in a DECLARE statement determines how data items are edited when they are assigned to that data name. Data items may be assigned to data names by either the assignment statement, the GET statement, or the READ statement. The P format description is used to specify data editing during the execution of an edit-directed PUT statement.

The following discussion describes the picture characters that may be used to edit a character-string data item. Examples are used to illustrate the effect of each picture character and to show its relationship with the other picture characters.

| | Source Data | Picture Specification | Edited Data |
|----|--|---|--|
| . | 12345 | ZZZV.9 | 2345.0 |
| | The decimal point may be used only once in a picture specification of numeric data. It indicates that the corresponding position in the character string contains a decimal point. | | |
| + | 123 012 -12 000 | +999 ++99 +999 +999 | +123 b+12 bb12 b000 |
| | The + picture character is used the same way the \$ picture character is used, except that a + sign will appear when the numeric value of the character string is greater than or equal to zero; otherwise no sign will appear. | | |
| \$ | 123 123 012 012 123 012 000 | \$999 999\$ 999\$ \$999 \$\$99 \$\$99 \$\$\$. | \$123 123\$ 012\$ \$012 \$123 b\$12 bbbb |
| | The \$ picture character, when used once in a picture specification, must appear at either the left or the right end of the picture specification. It indicates that a \$ is present in the corresponding position of the character string. A sequence of \$ picture characters may appear at the left end of the picture specification to indicate a drifting \$; this specifies that leading zeros be changed to blanks and that the right-most leading zero be changed to a \$. When all numeric positions in the picture specification contain the \$ picture character and the numeric value of the character string is zero, all positions of the character string are changed to blank even if a decimal point is present. | | |
| * | 00100 00000 00123 | ***99 ***. \$*,999 | **100 ***. \$***123 |
| | The * picture character is similar to the Z picture character except that the * character is used instead of a blank character to replace a leading zero. The * picture character must not be used with a Z picture character in the same picture specification. | | |
| - | 123 -15 -001 000 | -999 -99 --99 -999 | b123 -15 bb-1 000 |
| | The - picture character is similar to the + picture character except that a - sign will appear when the numeric value of the character string is less than zero. | | |
| / | 12345 | 99/99/9 | 12/34/5 |
| | The / picture character is used the same way the comma picture character is used, except that a / is present in the corresponding position of the character string. | | |
| , | 1234 00123 | 9,999 \$\$\$,999 | 1,234 bbb\$123 |
| | The comma picture character specifies that a comma is present in the corresponding position of the character string. If zero suppression occurs, the comma will appear only if there is an unsuppressed digit to the left of the comma. If there is no unsuppressed digit to the left of the comma, then three possibilities arise: | | |
| | 00123 | *,999 | **123 |
| | a. The preceding character is * ; then the comma position will contain an asterisk. | | |
| | 00123 | \$\$,999 | bb\$123 |
| | b. The preceding character is a drifting \$, +, - or S; then the comma position is treated as though it too contained the drifting character. | | |
| | 00123 | ZZ,999 | bbb123 |
| | c. The preceding character is other than the above; then the corresponding position in the data contains a blank character. | | |
| B | 12345 | ZZB999 | 12b345 |
| | The B picture character specifies that a blank character be inserted in the corresponding position of a character string. | | |

| | Source Data | Picture Specification | Edited Data |
|----|---|---|--|
| CR | -123 1234 | \$Z.99CR \$ZZ.99CR | \$1.23CR \$12.34bb |
| DB | -1234 1234 | \$ZZ.99DB \$ZZ.99DB | \$12.34DB \$12.34bb |
| I | 1021 | Z99I |  |
| R | -1021 | Z99R |  |
| S | 12345 -1234 | \$99999 \$99999 | +\$1234 -\$1234 |
| T | 1021 | Z99T |  |
| V | 12345 123.4 123.4 | ZZZV99 ZV999 ZV.999 | 34500 3400 3.400 |
| Y | 01020 | Y9Y9Y | b1b2b |
| Z | 12345 01234 00000 0123 0123 0000 0000 | ZZZZZ ZZZZZ ZZZ. Z ZZZZ\$ \$ZZZZ \$ZZZZ ZZZ.9 | 12345 b1234 bbbb b123\$ \$b123 bbbb bbbbb |
| 9 | 12345 123 | 99999 \$99 | 12345 \$123 |

PL/I and COBOL Comparison: Data Manipulation

The following discussion compares the data manipulation features of PL/I and COBOL. Both languages use expressions to specify data calculations and the picture specification to edit data. However, PL/I uses a single statement for all types of data manipulation, whereas COBOL uses several different statements. The following list contains some of the more significant differences in the data manipulation features of both languages:

1. The effects of the COBOL statements MOVE, COMPUTE, ADD, SUBTRACT, MULTIPLY, and DIVIDE are achieved in PL/I with the assignment statement. However, when the MOVE statement in COBOL is used with groups (the equivalent of PL/I structures), data is moved

without regard to the level structure of the groups, and data conversion, if specified, is ignored. When the assignment statement in PL/I is applied to structures, the assignment is performed elementary item by elementary item and all data conversion is done as specified.

2. The BY NAME option in PL/I is similar to the CORRESPONDING option in COBOL.
3. PL/I permits expressions to use data items that contain edit characters. COBOL does not provide this feature.
4. The bit string operators in PL/I are similar to the logical operators in COBOL. However, COBOL has no data type that corresponds to the bit string data type of PL/I.
5. The concatenation operator of PL/I is not available in COBOL.

The following pages contain sample problems that demonstrate some of the ways in which PL/I may be used to solve data processing problems. The solution to each problem is by no means unique and makes no attempt at being optimum. Although these problems illustrate some concepts and uses of PL/I, they are not intended to teach programming or programming-systems design.

Problem 1 — A Book Pricing Problem

The catalog number, unit price, and title of each book stocked by a book distributor are kept on a master file. The file is arranged in sequence according to the catalog number of each book. The file is on tape and has a standard end-of-file mark. A transaction file, also on tape, contains orders for books that specify the catalog number, quantity of the book ordered, and the customer making the order. The orders are arranged according to catalog number in the same sequence as the master file. The last order in the transaction file contains the dummy catalog number 99999, and indicates the end of the file. The total price of each transaction is to be calculated; a 3% discount is given for orders over \$150.00. A report is to be printed showing the customer, catalog number, book title, quantity ordered, unit price, and total price.

In the master file the information for each book has the following format: the first five characters are alphameric and represent the catalog number; the next four are decimal digits and represent the unit price — a decimal point is assumed before the last two digits; the next 30 character positions contain alphameric data representing the book title.

The orders in the transaction file have the following format: the first five character positions contain an alphameric catalog number; the next five positions are blank; a numeric integer representing the quantity occupies the next four positions; then five blanks follow; the next 40 alphameric positions represent the customer.

The report is to have 50 lines per page and is to be double-spaced. The first line of each page is to contain the heading BOOKS ORDERED, and each page is to be numbered in sequence, beginning with 1. The information printed on each line is to begin at character position 10 and is not to extend beyond character position 115. The customer, catalog number, book title,

quantity ordered, unit price, and total price appear in that order on each line, and are placed at the following character positions: 10, 52, 59, 91, 97, and 104. Actual decimal points, but no dollar signs, are to be printed in the unit price and total price. Leading zeros in these amounts are to be changed to blanks. The total price will always be less than \$10,000.00

Solution to Problem 1

```

BOOK-PRICING: PROCEDURE OPTIONS (MAIN);
DECLARE
  1 MASTER,
    2 CATALOG_NO CHARACTER (5),
    2 UNIT_PRICE  FIXED (4,2),
    2 TITLE       CHARACTER (30),
  1 TRANSACTION,
    2 CATALOG_NO CHARACTER (5),
    2 QUANTITY   FIXED (4),
    2 CUSTOMER   CHARACTER (40),
  1 REPORT,
    2 CUSTOMER   CHARACTER (40),
    2 CATALOG_NO CHARACTER (5),
    2 TITLE      CHARACTER (30),
    2 QUANTITY   FIXED (4),
    2 UNIT_PRICE FIXED (4,2),
    2 TOTAL_PRICE FIXED (6,2),
  PAGE_COUNT FIXED (2) INITIAL (0);
OPEN
  FILE (MASTER_FILE) INPUT,
  FILE (TRANSACTION_FILE) INPUT,
  FILE (REPORT_FILE) PRINT PAGESIZE (50)
  LINESIZE (115);
ON ENDPAGE (REPORT_FILE) GO TO HEADING;
ON ENDFILE (MASTER_FILE) GO TO ERROR;
HEADING: PAGE_COUNT = PAGE_COUNT + 1;
PUT FILE (REPORT_FILE) EDIT
  ('BOOKS ORDERED', 'PAGE', PAGE_COUNT)
  (PAGE, COLUMN(10), A(13),
  COLUMN(104),
  A(4), COLUMN(109), P'Z9');

IF PAGE_COUNT > 1
  THEN GO TO PRINT;
GET FILE (MASTER_FILE) EDIT (MASTER)
  (A(5), F(4,2), A(30));
TRANSACTION_READ: GET FILE
  (TRANSACTION_FILE) EDIT (TRANSACTION)
  (A(5), X(5), F(4), X(5), A(40));

```



```

IF TRANSACTION.CATALOG_NO = '99999'
  THEN GO TO FINISH;
COMPARE: IF TRANSACTION.CATALOG_NO =
  MASTER.CATALOG_NO
  THEN DO;
  GET FILE (MASTER_FILE) EDIT
    (MASTER) (A(5), F(4,2), A(30));
  GO TO COMPARE;
  END;
TOTAL_PRICE = TRANSACTION.QUANTITY *
  MASTER.UNIT_PRICE;
IF TOTAL_PRICE > 150.00 THEN TOTAL_PRICE =
  0.97 * TOTAL_PRICE;

REPORT = TRANSACTION, BY NAME;
REPORT = MASTER, BY NAME;
PRINT: PUT FILE (REPORT_FILE) EDIT (REPORT)
  (SKIP(2), COLUMN(10), A(40),
  COLUMN(52), A(5), COLUMN(59), A(30),
  COLUMN(91), F(4), COLUMN(97),
  P'ZZ.99', COLUMN(104), P'ZZZZ.99')
GO TO TRANSACTION_READ;
ERROR: DISPLAY ('END OF FILE ON MASTER
  BEFORE TRANSACTIONS FINISHED');
FINISH: CLOSE FILE (MASTER_FILE),
  FILE (TRANSACTION_FILE),
  FILE (REPORT_FILE);
END BOOK_PRICING;

```

Problem 2 — A Work Card Study

The computer is directed to read a series of work cards showing the hours worked by employees and then to compute the daily average for each employee. Each card contains several fields, one field for each data item; a comma separates fields. The first data item on each card is a name consisting of 15 alphameric characters. The next item is five alphameric characters representing a department number. The next five items are numeric and consist of five characters each with an actual decimal point in the third character position. These are daily time items, representing the hours worked on each of five workdays.

An output file is to be created according to the conventions of list-directed output. The data items on the input cards are to be duplicated in the output file. The data for each card is to be followed by the total time and the average daily time.

In the actual processing the five daily time items are treated as an array and a subscript N is used to obtain each time value in turn.

Solution to Problem 2

```

WORK_CARD: PROCEDURE OPTIONS(MAIN);
DECLARE N FIXED (1), WORK_FILE FILE,
  NEW_WORK_FILE FILE, NAME CHARACTER
  (15), DEPT CHARACTER (5),
  TIME (5) PICTURE '99.99',
  TOTAL_TIME PICTURE '99V99',
  AVERAGE_TIME PICTURE '99V99';
OPEN FILE (WORK_FILE) INPUT, FILE
  (NEW_WORK_FILE) OUTPUT;
ON ENDFILE (WORK_FILE) GO TO CLOSE;
READ: GET FILE (WORK_FILE) LIST (NAME,
  DEPT, TIME);
TOTAL_TIME = 0;
DO N = 1 TO 5 BY 1;
  TOTAL_TIME = TIME (N) + TOTAL_TIME;
  END;
AVERAGE_TIME = TOTAL_TIME / 5;
PUT FILE (NEW_WORK_FILE)
  LIST (NAME,DEPT,TIME,TOTAL_TIME,
  AVERAGE_TIME);
GO TO READ;
CLOSE: CLOSE FILE (WORK_FILE), FILE
  (NEW_WORK_FILE);
END WORK_CARD;

```

Problem 3 — A File Search

This example illustrates a type of analysis often required in marketing research and similar studies. A personnel file is searched to find all records containing certain specified data. Each record found is written in a separate file and, when finished, a count of the number of records found is displayed on a standard display device.

Each record consists of a 16-character name followed by a 24-bit code-string. Each bit position within the code-string represents specific personnel data.

Format of Code-String

| Items | Characteristics | Bit Positions |
|--------|---|---------------|
| Sex | Male | 1 |
| | Female | 2 |
| Age | Less than 20 | 3 |
| | At least 20 but not over 50 | 4 |
| | Over 50 | 5 |
| Height | Over six feet | 6 |
| | At least five feet, six inches, but not over six feet | 7 |
| | Less than five feet, six inches | 8 |

| | | |
|-----------|--|----|
| Weight | Over 185 pounds | 9 |
| | At least 120 pounds, but not over 185 pounds | 10 |
| | Less than 120 pounds | 11 |
| Eyes | Blue | 12 |
| | Brown | 13 |
| | Hazel | 14 |
| | Grey | 15 |
| Hair | Brown | 16 |
| | Black | 17 |
| | Grey | 18 |
| | Red | 19 |
| | Blond | 20 |
| | Bald | 21 |
| Education | College | 22 |
| | High School | 23 |
| | Grammar School | 24 |

The presence of a characteristic is indicated by a one bit in the corresponding bit position of the code-string. A zero bit indicates the absence of a characteristic.

A search is made to obtain the records of all persons having the following characteristics:

1. Females under 20 years of age, five feet six inches and over in height, from 120 to 185 pounds in weight, with either hazel or brown eyes, not bald, having a high school education.
2. Males over 50 years of age, over six feet in height, over 185 pounds in weight, and college-educated.

Solution to Problem 3

```
FILE-SEARCH : PROCEDURE OPTIONS (MAIN);
DECLARE
```

```
1 PERSONNEL-RECORD BASED (P),
2 NAME CHARACTER (16),
2 CODE-STRING,
3 SEX,
  (4 MALE,
  4 FEMALE) BIT (1),
3 AGE,
  (4 UNDER_20,
  4 TWENTY-TO-50,
  4 OVER-50) BIT (1),
```

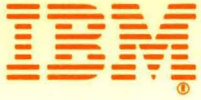
```
3 HEIGHT
  (4 OVER-6,
  4 FIVE-AND-A-HALF-TO-6,
  4 UNDER-FIVE-AND-A-HALF) BIT (1),
3 WEIGHT,
  (4 OVER-185,
  4 BETWEEN-185-AND-120,
  4 UNDER-120) BIT (1),
3 EYES,
  (4 BLUE,
  4 BROWN,
  4 HAZEL,
  4 GREY) BIT (1),
3 HAIR,
  (4 BROWN,
  4 BLACK,
  4 GREY,
  4 RED,
  4 BLOND,
  4 BALD) BIT (1),
3 EDUCATION,
  (4 COLLEGE,
  4 HIGH-SCHOOL,
  4 GRAMMAR-SCHOOL) BIT (1),
COUNT FIXED (5),
  (RESULT-FILE FILE OUTPUT,
  PERSONNEL-FILE FILE INPUT) RECORD;
OPEN FILE (PERSONNEL-FILE),
  FILE (RESULT-FILE);
ON ENDFILE (PERSONNEL-FILE) GO TO FINISH;
COUNT = 0;
INPUT: READ FILE (PERSONNEL-FILE) SET (P);
IF (FEMALE & AGE.UNDER_20
  & ¬HEIGHT.UNDER_FIVE-AND-A-HALF
  & WEIGHT.BETWEEN-185-AND-120
  & (EYES.HAZEL | EYES.BROWN)
  & ¬HAIR.BALD & EDUCATION.HIGH-SCHOOL)
  | (MALE & AGE.OVER_50 & HEIGHT.OVER_6
  & WEIGHT.OVER-185 & EDUCATION.COLLEGE)
THEN DO; COUNT = COUNT + 1;
  WRITE FILE (RESULT-FILE) FROM
    (PERSONNEL-RECORD);
  END ;
GO TO INPUT;
FINISH: DISPLAY ('COUNT IS ' || COUNT);
CLOSE FILE (PERSONNEL-FILE), FILE
  (RESULT-FILE);
END FILE-SEARCH;
```

Index

| | | | |
|-----------------------------------|------------|------------------------------------|------------|
| accuracy | 55 | concatenation operations | 57 |
| activation of blocks | 36, 40 | condition prefixes | 46 |
| addition | 54 | conditions | 47 |
| ALIGNED attribute | 21 | constants | 11 |
| ALLOCATE statement | 49 | bit-string | 11 |
| allocation | 49 | character-string | 11 |
| also see storage class attributes | | fixed-point binary | 11 |
| arguments | 38 | fixed-point decimal | 11 |
| arithmetic data | 10 | floating-point binary | 12 |
| attributes | 12 | floating-point decimal | 12 |
| arithmetic operations | 54 | statement-label | 12 |
| array | 18 | contextual declarations | 39 |
| allocation | 49 | also see declarations | |
| assignment | 57 | control | |
| bounds | 39 | format items | 31 |
| also see asterisks | | sequence of | 40 |
| dimensions | 19 | CONTROLLED attribute | 49 |
| expressions | 57 | also see storage | |
| manipulation | 58 | CONVERSION condition | 47 |
| of structures | 20, 58 | conversion | 53 |
| assignment | | arithmetic | 55 |
| array | 57 | in expressions | 51 |
| statement | 51 | type | 53 |
| string | 52 | data | |
| structure | 58 | aggregates | 16 |
| asterisks | 39, 49 | arithmetic | 10 |
| attributes | 12, 13, 14 | bit-string | 10 |
| defaults for | 15 | character set | 7 |
| factoring of | 22 | character-string | 13 |
| AUTOMATIC attribute | 48 | conversions | 53 |
| BASED attribute | 26 | description | 10 |
| based variable | 26 | editing | 59 |
| begin block | 22, 35 | format items | 28, 29, 30 |
| BEGIN statement | 22, 35 | manipulation | 51 |
| BINARY attribute | 12 | name | 8, 11, 53 |
| BIT attribute | 13 | specification | 30 |
| bit-string data | 10 | repetitive specification for | 30 |
| bit-string operation | 53, 56 | transmission | 26 |
| blanks | 7 | statements | 26, 27 |
| blocks | 35 | types | 10 |
| activation of | 36, 40 | defaults for | 15, 47 |
| begin | 35 | data-directed transmission | 33 |
| nested | 37 | data specification for | 33 |
| procedure | 35 | input | 33 |
| termination of | 40 | output | 33 |
| BUFFERED attribute | 24 | DECIMAL attribute | 12 |
| buffering attributes | 24 | declarations | |
| BY and TO | 44 | contextual | 39 |
| BY NAME option | 59 | external | 36 |
| CALL statement | 38 | scope of | 36 |
| CHARACTER attribute | 13 | DECLARE statement | 12, 40 |
| character string | | default attributes | 15 |
| data | 28 | DEFINED attribute | 15, 58 |
| pictures | 30, 61 | defined item | 15, 58 |
| also see string | | delimiters | 8 |
| characters | | descendance of blocks | 41 |
| alphabetic | 7 | DISPLAY statement | 34 |
| data character set | 7 | division | 54 |
| language character set | 7 | DO statement | 43 |
| special | 7 | EDIT | 27 |
| CLOSE statement | 25 | edit-directed transmission | 27 |
| collating sequence | 55 | format of | 28 |
| COLUMN format item | 31 | editing | |
| comment | 7 | symbols | 59, 60, 61 |
| comparison operations | 42, 55 | drifting | 61 |

| | | | |
|-------------------------|-----------|-------------------------|------------|
| ELSE clause | 43 | list-directed | |
| enable | 46 | data specification | 32 |
| END statement | 35 | input | 32 |
| use of | 35 | output | 32 |
| ENDFILE condition | 47 | transmission | 32 |
| ENDPAGE condition | 47 | LOCATE statement | 26, 27 |
| ENTRY attribute | 39 | multiplication | 54 |
| entry name | 39 | names | 8, 11 |
| ENVIRONMENT attribute | 24 | qualified | 17, 27 |
| ERROR condition | 47 | subscripted | 20, 27 |
| exponentiation | 54 | use of | 8, 45 |
| expressions | | nesting | 37 |
| arithmetic | 54 | NO prefix | 46 |
| array | 54, 57 | ON statement | 45 |
| evaluation of | 51 | use of | 45 |
| structure | 58 | ON-conditions | 45, 47 |
| subscripts | 20, 55 | nullification of | 46 |
| EXTERNAL attribute | 37 | prefixes used with | 46 |
| external declarations | 37 | programmer-defined | 45 |
| external procedure | 35 | OPEN statement | 25 |
| factoring | 22 | operations | |
| file | | arithmetic | 54 |
| attributes | 24 | array-array | 57 |
| closing | 25 | bit string | 56 |
| names | 25 | comparison | 55 |
| opening | 25 | concatenation | 57 |
| FILE attribute | 24 | priority of | 54 |
| FINISH condition | 47 | string | 57 |
| FIXED attribute | 12 | operators | |
| FIXEDOVERFLOW condition | 47 | arithmetic | 54 |
| fixed-point | 11 | bit string | 56 |
| FLOAT attribute | 12 | comparison | 55 |
| floating-point | 12 | infix | 54 |
| form, coding | 7 | prefix | 54 |
| format | | string | 57 |
| of data-directed output | 33 | output | 24 |
| of list-directed I/O | 32 | OUTPUT attribute | 24 |
| format items | | OVERFLOW condition | 47 |
| control | 31 | PACKED attribute | 21 |
| data | 28 | PAGE format item | 31 |
| format list | 28 | PAGESIZE option | 25 |
| FREE statement | 50 | parameters | 38 |
| FROM option | 26 | PICTURE attribute | 13, 30, 59 |
| GET statement | 27 | with numeric data | 14, 60 |
| GO TO statement | 41 | specification | 60 |
| IDENT option | 25 | with string data | 14, 61 |
| identifiers | 8 | picture format items | 60 |
| length of | 8 | pointer | 26 |
| keywords | 8 | POSITION attribute | 15 |
| IF statement | 42 | precision | 12 |
| infix operators | 54 | prefix | 46 |
| INITIAL attribute | 21 | operators | 54 |
| INPUT attribute | 24 | PRINT attribute | 24 |
| input/output | 24 | printing format items | 31 |
| INTERNAL attribute | 37 | priority of operations | 54 |
| internal procedure | 35 | procedure | 35 |
| internal to | 41 | external | 35 |
| keyword | 8 | internal | 35 |
| label | 8, 12, 53 | name | 35 |
| LABEL attribute | 21 | termination of | 35 |
| length | | PROCEDURE statement | 35 |
| data item | 51 | program structure | 35, 20 |
| identifiers | 8 | PUT statement | 27 |
| strings | 52 | qualified names | 17 |
| level numbers | 16 | quotation marks | 11 |
| also see structures | | READ statement | 26 |
| LIKE attribute | 21 | RECORD | |
| LINE format item | 31 | attribute | 24 |
| LINESIZE option | 25 | transmission statements | 26 |
| | | repetition | 11, 14, 30 |
| | | REPLY option | 34 |

| | | | |
|--------------------------------|----|-----------------------------------|---------|
| RETURN statement | 40 | data-directed | 33 |
| REWRITE statement | 27 | edit-directed | 27 |
| scope | | list-directed | 32 |
| of declarations | 36 | string | |
| of condition prefixes | 46 | assignment | 52 |
| scope attributes | 37 | attributes | 13 |
| sequence | | data | 10 |
| collating | 55 | STRING option | 33 |
| of control | 40 | structure | 16 |
| SET | 26 | assignment | 58 |
| sign picture characters | 60 | BY NAME | 59 |
| SIZE condition | 48 | declarations and attributes | 16 |
| SKIP format item | 31 | with LIKE attribute | 21 |
| SNAP option | 45 | level numbers | 17 |
| spacing, printing | 31 | SUBSCRIPTRANGE condition | 48 |
| specification | 30 | subscripts | 20 |
| statements | 8 | subtraction | 54 |
| STATIC attribute | 48 | SYSTEM option | 45 |
| storage | | TO and BY | 44 |
| also see allocation | | TRANSMIT condition | 48 |
| ALLOCATE statement | 49 | truncation on assignment | 51 |
| automatic | 48 | UNBUFFERED attribute | 24 |
| controlled | 49 | UNDEFINEDFILE condition | 48 |
| FREE statement | 50 | UNDERFLOW condition | 48 |
| static | 48 | UPDATE attribute | 24 |
| storage class attributes | 48 | variables, based | 26 |
| default for | 48 | VARYING | 13 |
| restrictions | 48 | WHILE clause | 43 |
| with structures | 49 | WRITE statement | 26 |
| STREAM | | ZERODIVIDE condition | 48 |
| attribute | 24 | zero suppression | 48, 60, |
| transmission modes | 27 | | |



International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York 10601
(USA only)

IBM World Trade Corporation
821 United Nations Plaza, New York, N.Y. 10017
(International)

SC20-1651-2

A Guide to PL/I for Commercial Programmers

Printed in USA SC20-1651-2