

## Data Processing Techniques

## Introduction to the List Processing Facilities of PL/I

This manual discusses and illustrates usage of PL/I facilities for organizing, processing, and relocating data in list form. A data list is a chain of based variable structures that contain data plus pointers that link the structures. List-processing techniques are useful for handling data that has logical complexities not conveniently represented by conventional PL/I array and structure representation.

Illustrative programs were processed by the PL/I (F) Compiler (Version 4) under control of the IBM System/360 Operating System (Release 16).

The list-processing facilities of PL/I are an advanced topic in programming; this manual is intended for the experienced programmer. Additional information is presented in *Techniques for Processing Data Lists in PL/I* (GF20-0018), *Techniques for Processing Pointer Lists and Lists of Lists in PL/I* (GF20-0019), and *Techniques for Processing Relocatable Lists in PL/I* (GF20-0020).

The IBM Program Product PL/I Optimizing Compiler for DOS, and the IBM Program Product PL/I Optimizing and PL/I Checkout Compilers for OS provide list processing function additional to the facilities described in this manual; for example, more than one REFER option can be specified for a based structure.

The IBM Program Product SIMPL/I (Simulation Language Based on PL/I) has a subset of list processing keywords that can be used with regular PL/I keywords to handle list processing applications. SIMPL/I supports convenient creation of circular lists, lists of lists, and arrays of lists. SIMPL/I has built-in functions to manipulate (insert, delete, etc.) list components. Output from SIMPL/I is used as input to the IBM Program Product OS PL/I Optimizing Compiler or the IBM Program Product OS PL/I Checkout Compiler.

Manuals that discuss the above-mentioned IBM Program Products are available from your IBM marketing representative.

# IBM

**Reprint (October 1975)**

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, Technical Publications—Systems Department, Dept. 824, 1133 Westchester Avenue, White Plains, New York 10604.

© Copyright International Business Machines Corporation 1973

# Table of Contents

	<i>Page</i>		<i>Page</i>
Chapter 1. Introduction . . . . .	1	3E. Arrays and Structures of Based Variables . . . . .	28
1A. Addressing Storage . . . . .	1	3E1. Qualifying Based Variables with Pointer Variables . . . . .	28
1B. Data Movement . . . . .	1	3E2. Pointer Qualification in a Subroutine . . . . .	30
1C. Storage Allocation . . . . .	3	3E3. Restrictions on Based Variables . . . . .	32
1D. The Uses of Lists . . . . .	4	3E4. Contextual Declarations of Pointer Variables . . . . .	32
1D1. Information Storage and Retrieval . . . . .	5	3F. Advantages of Absolute Addressing . . . . .	32
1D2. System Simulation . . . . .	5	3F1. Reducing Data Movement . . . . .	32
1D3. Engineering Design . . . . .	5	3F2. Sorting with Pointer Variables . . . . .	32
1D4. Computer-Software Production . . . . .	5	3F3. Sorting with an Array of Pointer Variables . . . . .	32
1D5. Text Editing . . . . .	6	3G. Associating Data Items in Scattered Locations . . . . .	35
1D6. Artificial Intelligence . . . . .	6	3G1. Associating Data Items through an Array of Pointer Variables . . . . .	35
1E. Purpose of this Manual . . . . .	6	3G2. Linking Data Items through Pointer Variables . . . . .	37
1F. Summary of Chapter 1 . . . . .	6	3H. Review of Techniques For Addressing Data Items . . . . .	43
Chapter 2. The Organization of Data in PL/I . . . . .	7	3I. Summary of Chapter 3 . . . . .	44
2A. Element Items . . . . .	7	Chapter 4. Lists and the Dynamic Allocation of Storage . . . . .	45
2B. Arrays . . . . .	7	4A. Based Storage . . . . .	45
2C. Structures . . . . .	9	4A1. Allocating Based Storage . . . . .	45
2D. Arrays of Structures . . . . .	10	4A2. The SET Option in an Allocate Statement . . . . .	46
2E. Advantages of Arrays and Structures . . . . .	10	4A3. Freeing Based Storage . . . . .	46
2E1. Reducing Program Size . . . . .	10	4A3A. Multiple References in a FREE Statement . . . . .	47
2E2. Avoiding Data Scanning . . . . .	15	4A3B. Implicit Freeing of Storage . . . . .	47
2F. Summary of Chapter 2 . . . . .	15	4A4. An Example of Based Storage Used in a Sort Procedure . . . . .	47
Chapter 3. Techniques for Addressing Data . . . . .	16	4A5. Allocating Based Storage for a Self-defining Structure . . . . .	48
3A. Types of Addresses . . . . .	16	4A5A. The REFER Option . . . . .	49
3B. Symbolic Addresses . . . . .	16	4A5B. An Example of the REFER Option in a Sort Procedure . . . . .	50
3C. Relative Addresses . . . . .	16	4A6. Allocating and Freeing Based Storage Within an Area . . . . .	51
3C1. The Use of Subscript Values in Auxiliary Arrays . . . . .	16	4A6A. The AREA Attribute . . . . .	51
3C2. Using Subscript Values to Link Elements . . . . .	18	4A6B. The IN Option . . . . .	52
3D. Absolute Addresses . . . . .	24	4A6C. The Area On-condition . . . . .	53
3D1. The Relationship of Relative and Absolute Addresses . . . . .	24	4A6D. An Example of an Area Variable in a Sort Procedure . . . . .	53
3D2. Pointer Variables . . . . .	24	4B. Organization of Data in List Form . . . . .	53
3D2A. How to Obtain a Value For a Pointer Variable . . . . .	25	4B1. The Main Parts of a List . . . . .	53
3D2B. Using Pointer Variables in Assignment Statements . . . . .	25	4B2. Advantages of Lists . . . . .	58
3D2C. Using Pointer Variables in Operational Expressions . . . . .	26		
3D3. Based Variables . . . . .	26		
3D3A. Assigning Pointer Values with the SET Option . . . . .	26		
3D3B. Using a Based Variable in a Function Procedure . . . . .	27		

	<i>Page</i>		<i>Page</i>
4B3. Types of Lists . . . . .	58	5K4. Converting Data Lists to and from Relocatable Form . . . . .	75
4C. Review of Techniques for Organizing Based Storage in List Form . . . . .	58	5K5. Sorting Relocatable Data Lists . . . . .	77
4D. Summary of Chapter 4 . . . . .	61	5K6. External Relocation . . . . .	78
Chapter 5. Facilities for Relocating Data Lists . . . . .	62	5K6A. Writing Relocatable Data Lists . . . . .	78
5A. Treating Lists as Units Within Areas . . . . .	62	5K6B. An Example that Creates Relocatable Data Lists in a Work Area and Writes Them into a File . . . . .	79
5B. Assigning Areas to Other Areas . . . . .	62	5K6C. An Example that Creates Relocatable Data Lists in an Output Buffer and Writes Them into a File . . . . .	80
5C. The Extent of an Area . . . . .	63	5K6D. Reading Relocatable Data Lists . . . . .	81
5D. The Effect of Extent on Area Assignment . . . . .	64	5K6E. An Example that Sorts and Prints Relocatable Data Lists Contained in a File . . . . .	81
5E. The AREA ON-condition For Area Assignment . . . . .	64	5L. Review of Techniques for Creating Relocatable Data Lists . . . . .	82
5F. Computing the Extent of an Area . . . . .	65	5M. Summary of Chapter 5 . . . . .	85
5G. The Length of an Area . . . . .	65	Appendix. Summary of List-Processing Facilities . . . . .	86
5H. The Effect of Area Assignment on Pointer Values . . . . .	66	Index . . . . .	87
5I. Addressing the Contents of an Assigned Area . . . . .	66		
5J. Offset Variables . . . . .	66		
5J1. Assigning Values to Offset Variables . . . . .	67		
5J2. The NULLO Built-in Function . . . . .	68		
5J3. Restrictions on Offset Variables . . . . .	69		
5K. Relocation of Data Lists . . . . .	69		
5K1. Internal Relocation . . . . .	69		
5K2. Relocatable Data Lists . . . . .	71		
5K3. A Subroutine that Assigns a Relocatable Data List to Another Area . . . . .	74		

## Preface

The list-processing facilities of PL/I provide more general methods of allocating and organizing internal computer storage than are available with array and structure organizations. PL/I supplies these facilities through special data attributes, built-in functions, interrupt conditions, and executable statements, which allow the programmer to manipulate storage addresses and to link scattered storage areas during the course of program execution.

Linking scattered storage through address manipulation produces a general type of data organization called a *list*, and that aspect of programming concerned with organizing and managing lists is referred to as *list processing*. PL/I, however, does not define a specific type of list organization; instead, it allows storage to be linked in an arbitrary manner, so that a list can contain any combination of data elements, arrays, and structures. This type of organization even permits a list to contain other lists as its components.

The major advantage of lists over conventional array and structure organizations is the efficiency they permit in the use of storage. A list need reserve only the storage it is actually using at any given moment during program execution; storage need not lie dormant in anticipation of maximum requirements. As additional list components are required, their storage is linked to the list and, when list components are no longer needed, their storage is linked to other lists or reserved in a storage pool for further use. Such flexibility in storage management also reduces data movement and frees the programmer from having to know exactly how much storage a list will require.

But list processing supplies more than an efficient technique for using storage. It also furnishes a method for organizing and manipulating data whose structure is not

conveniently represented with PL/I arrays and structures. Structured data of this type occurs in many non-numeric applications, such as information storage and retrieval, system simulation, engineering design, computer-software production, text editing, and artificial intelligence. List processing preserves the natural structure of the data involved in such applications and, as a result, avoids unnecessary programming complexity.

This manual discusses the basic facilities for list processing in PL/I and uses a simple sort application throughout the text to show how lists are created and processed; printouts of program compilations appear with the discussions. The numbering scheme for illustrations associates them with pertinent paragraphs. For example, the paragraph numbered 1C references figures 1C-1, 1C-2, and 1C-3.

Because the list-processing facilities of PL/I form an advanced topic in programming, this manual assumes that the reader is an experienced programmer with a knowledge of PL/I equivalent at least to that presented in *A PL/I Primer*, Form C28-6808. Familiarity is assumed with array and structure organization and with methods for creating and invoking subroutines and functions. The programming examples in this manual are concerned mainly with illustrating the use of the list processing facilities and, as a result, frequently sacrifice efficient programming techniques for the sake of clarity. No references are made to particular implementations of PL/I, but information on the F-level list-processing facilities appears in *IBM System/360: PL/I (F) Language Reference Manual* (GC28-8201), and in *IBM System/360 Operating System: PL/I (F) Programmer's Guide* (GC28-6594).



.

.



.

.



## Chapter 1. Introduction

### 1A. ADDRESSING STORAGE

A computer generally uses a serial addressing scheme, which permits a series of instructions to be placed in successive storage locations. Similarly, because the data processed by a program is also stored in core, the serial structure of core storage is usually imposed upon collections of data items that are organized in tabular form. Successive data items are then obtained through simple address incrementation.

But the simplicity of incremental addressing, as applied to data collections, becomes a disadvantage when the arrangement of data items within a collection must vary during the course of program execution. Such variation in data organization frequently occurs in non-numeric applications, and produces two major problems: the data movement problem, which is concerned with inefficiencies in program running time caused by extensive rearrangements of data, and the storage allocation problem, which is caused by excessive allocation of storage in anticipation of maximum data requirements. The following discussions show how both problems have led to the development of list-processing facilities.

### 1B. DATA MOVEMENT

Many computer applications require a collection of data items to be arranged in a specific order before processing may proceed. Such ordering permits faster retrieval of the items than is normally possible with a serial search through the unordered collection. The ordering process itself,

however, frequently becomes time consuming, particularly when conventional sorting methods are used. An exorbitant inefficiency in processing time may result when several different orderings of the same items occur during the running of a program.

The amount of data that is moved to obtain the desired reorderings determines the degree of inefficiency. As an example, consider the representation of an array (Figure 1B-1) which contains four data items arranged in ascending sequence. Each item is a string of three characters, and a blank item specifies an available position in the array. Figure 1B-2 shows the array after the character string 'B72' has been inserted in proper sequence. Three items have been moved to provide room for the string inserted in the second array position. If the array were larger and the items longer, inserting an item could require considerable movement of data.

One way of avoiding excessive data movement is to permit collected data items to occupy non-contiguous storage locations. A method for such organization is suggested by branch instructions, which allow computer control to transfer to non-consecutive locations (as required by sub-routines and program loops). The location to which control is transferred is specified within the branch instruction itself. A similar use of storage addresses is possible with data items. Each item in an ordered collection can have attached to it the address of the next item in sequence; then logically successive items need not occupy physically contiguous storage locations.

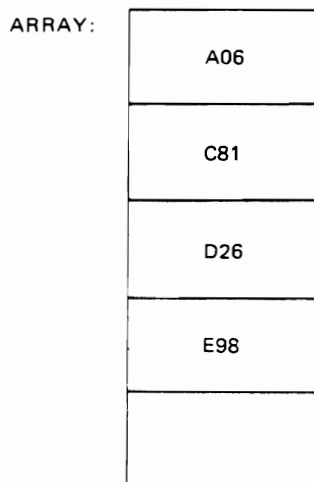


Figure 1B-1. Array before insertion of data item

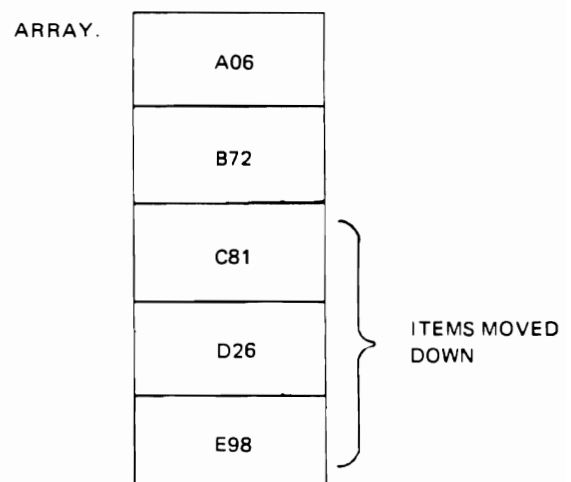


Figure 1B-2. Array after insertion of data item

Schematic representations of storage addresses are used in this manual. A rectangle with a protruding arrow represents an address element (Figure 1B-3). The arrow indicates that the rectangle contains an address that "points to" another location. When the actual address is not important to the discussion, it does not appear within the rectangle. An address element that specifies no address (that is, a null address) uses no arrow but contains the word NULL within the rectangle (Figure 1B-4). Figure 1B-5 shows an address element attached to a data item, and similar use of a null address element appears in Figure 1B-6. These diagrams provide a means for representing a collection of non-contiguous data items.

Consider Figure 1B-7, which uses an address element at the top to point to the first data item in a non-contiguous collection. The first data item, in turn, uses an attached address to point to the second data item in the collection. This process continues until all items are linked. The null address attached to the last item indicates the end of the collection.

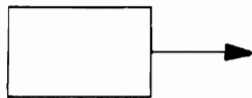


Figure 1B-3. An address element



Figure 1B-5. An address element attached to a data item



Figure 1B-4. A NULL address element

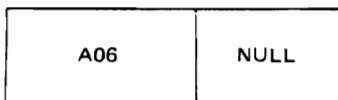


Figure 1B-6. A NULL address element attached to a data item

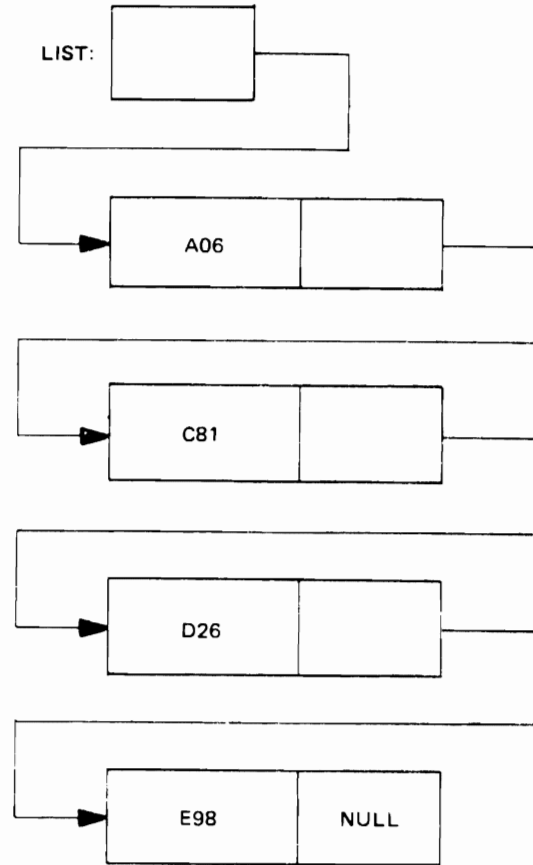


Figure 1B-7. List before insertion of data item

The resulting organization shown in Figure 1B-7 forms a *list*, which, for the purposes of this manual, is defined to be a collection of non-contiguous data items linked in any desired order by means of attached address elements. Note that since each data item in a list has an attached address element, two data items cannot be physically contiguous. An address element, however, can be contiguous to the data item toward which it points. In general, however, an address element is located remotely from the data item toward which it points.

Figure 1B-8 demonstrates how address manipulation permits a data item to be inserted into a list. Before insertion, the list contains four data items which are linked in ascending sequence (Figure 1B-7). After insertion, the items remain in sequence, but the inserted item 'B72' occupies the second position in the list. Insertion occurs by assigning the address of 'B72' to the address element attached to 'A06' and, in turn, by assigning the address of 'C81' to the address element attached to 'B72'. Manipulating addresses in this manner permits list items to remain at their original locations and eliminates the processing time that would normally be spent in moving the items.



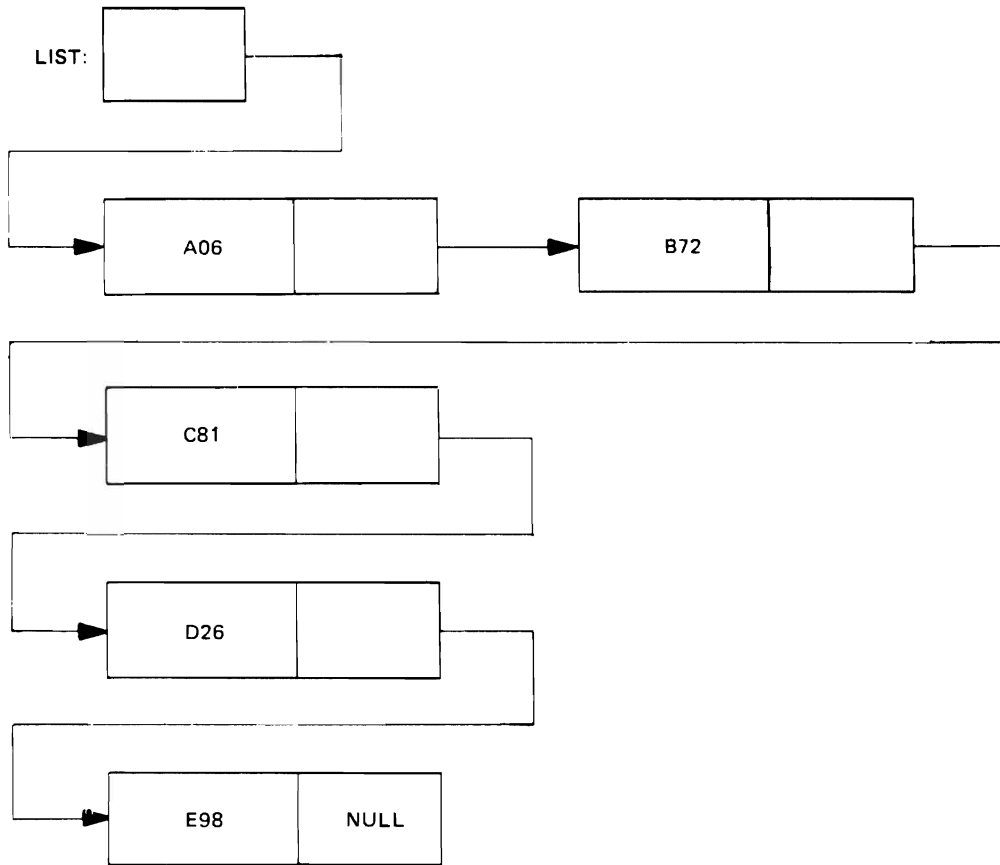


Figure 1B-8. List after insertion of data item

### 1C. STORAGE ALLOCATION

In many applications, the amount of storage allocated for an array is based on the maximum amount of data that might be assigned to the array. Should only a portion of the array be needed during a particular run of the program, the excess storage reserved for the array will remain unused. Permitting storage to lie dormant in this manner frequently reduces the effective storage utilization of the entire program.

A major advantage of the list organization shown in Figure 1B-8 is that a list requires only the storage it is actually using at any given moment and does not have to reserve additional storage in anticipation of maximum requirements. When a data item is inserted into a list, only then is the storage for the item linked to the list; otherwise, the storage is free to be used elsewhere in the program. Similarly, when an item is deleted from a list, the storage occupied by the item need not remain linked to the list; it can be linked to another list or held in a storage pool for further use.

Sharing of storage among lists is conveniently controlled through a special list that links all storage not currently in use. Consider Figure 1C-1, which contains two lists. The first list links four data items, and the second

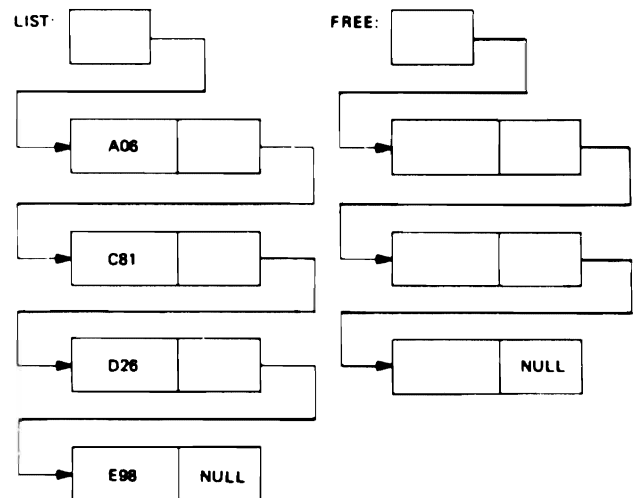


Figure 1C-1 Data list and free-storage list before insertion of new data item

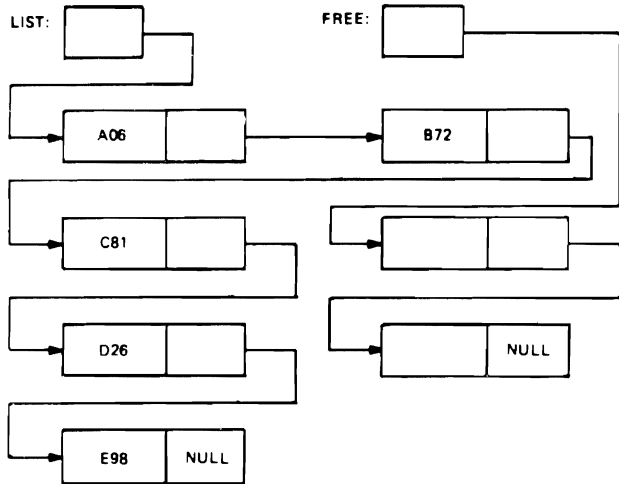


Figure 1C-2 Data list and free-storage list after insertion of new data item

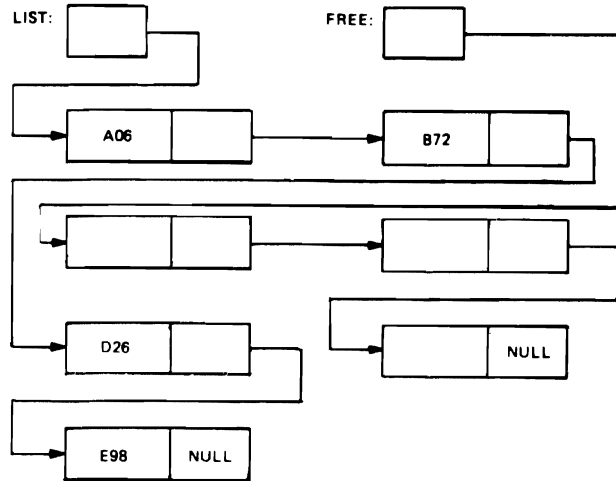


Figure 1C-3 Data list and free-storage list after deletion of third data item

list contains free storage for three list components. Insertion of a data item into the first list requires storage to be linked to the list. This storage is obtained from the free-storage list, as shown in Figure 1C-2 where the item 'B72' is inserted in proper sequence into the first list. After insertion, the data list links five items instead of four, and the free-storage list contains storage for two list components instead of three.

Deletion of an item from the data list causes the associated storage component to be linked to the free-storage list. Figure 1C-3, for example, shows how the third item is deleted from the data list and how the storage for the item is linked to the free-storage list.

This use of a list as a storage pool also frees the programmer from having to know precisely how long each list in a program may become. Such freedom frequently simplifies the design of many computer applications that have unpredictable or varying storage requirements.

## 1D. THE USES OF LISTS

The type of list organization presented in the previous discussions possesses a simple linear ordering: each data item except the first has one predecessor, and each data item except the last has one successor. But PL/I does not restrict list organizations to linear orderings; it also allows multidimensional orderings, which are obtained by attaching an arbitrary number of address elements (instead of one) to each data item in a list. Additional address ele-

ments permit a list item to be linked to other list items in many different directions, so that the ordering within the list can be made to reflect the multidimensional ordering associated with spatial configurations. With additional addresses, it is also possible to link lists in a backward as well as a forward direction, and to form circular and intertwined lists. It is even possible for lists themselves to be linked together to form higher-level lists.

The term *list processing* applies to those programming activities concerned with the construction, management, and application of list organizations. The computer applications that benefit most from list-processing techniques generally process large collections of data items that are interrelated in a logically complex manner. With lists, it is possible to model the logical interrelationships found in structured data and to avoid the organizational distortions and storage inflexibilities frequently associated with arrays.

List processing simplifies the design of computer applications in many areas, particularly those areas concerned with the non-numeric aspects of computer programming, such as information storage and retrieval, system simulation, engineering design, computer-software production, text editing, and artificial intelligence. These application areas are concerned more with the logical structure and organization of data and less with numeric computation. The following discussions briefly describe the use of list-processing techniques in each area.

### 1D1. Information Storage and Retrieval

This application area is concerned with computer techniques for storing and searching large quantities of information related to specific fields of interest, such as business records, medical histories, government reports, statistical indexes, and library catalogs. List processing provides efficient methods for inserting, deleting, updating, and retrieving data in such collections, and allows many different orderings and cross references of the stored items to be maintained without unnecessary data duplication. For example, with list processing, three separate copies of a library card catalog, arranged by title, author, and subject, can be replaced by a single copy within the computer. Attaching three address elements to the data content of each card permits simultaneous linking of each card to separate lists for title, author, and subject.

### 1D2. System Simulation

A system, in general terms, is any collection of elements (animate or inanimate) that are united to accomplish a specific objective. Examples include traffic control systems, manufacturing plants, work routing systems, biological organisms, air defense systems, communication networks, business enterprises, postal systems, and electronic computers. Many large-scale systems cannot be studied analytically because their structural complexities defy adequate mathematical description. But such systems can be modeled or simulated through computer programs that operate in a time-sequential manner similar to the systems themselves.

Simulation programs maintain large collections of data items that describe the dynamic state of the system at particular intervals of time. For example, simulation of a traffic control system might require current information about the state of all traffic lights, the block-by-block traffic on each avenue, and the locations of major traffic jams caused by accidents or road repair. List processing permits such information to be organized in list form and allows the lists to expand and contract dynamically during the running of the simulation.

### 1D3. Engineering Design

Engineering structures, such as automobiles, airplanes, missiles, machines, bridges, highways, radios, television sets, space satellites, and electronic computers, progress through many design stages before an acceptable prototype is achieved. Blueprints generally record the design criteria at each stage and, therefore, must be redrawn each time the design changes. The number of blueprints involved in many engineering projects can easily run into the thousands and, as a result, produce a serious problem in information storage and retrieval.

List processing provides a convenient way of storing

blueprint information in machine-processable form so that it is easily modified and kept up to date. But the computing flexibilities associated with list processing also allow graphic display devices and automatic drafting boards to be incorporated into the design process itself. The current design of an engineering structure, for example, can be shown on the scope of a graphic display device where it is quickly modified by the design engineer who uses a light pen to indicate desired changes. When reflected in the current design, such changes can then be used to generate a completely new blueprint on an automatic drafting board.

Graphic display devices also permit functional illustrations of engineering structures to be projected on a scope, so that the design engineer can see the structures in operation. As an example, the engineer can display a set of machine gears on a scope and then make the gears rotate under program control. In this way, he can observe how the gear teeth mesh and, if necessary, specify changes with a light pen.

Projecting and modifying engineering designs through graphic display devices requires flexible methods for representing the design within the computer. Such methods are available through list-processing techniques, which not only assist in organizing the logical structure of the designs but also maintain efficient control over the varying storage requirements generated by design changes.

### 1D4. Computer-Software Production

Software systems, such as operating systems, compilers, assemblers, generators, subroutine libraries, and industrial application packages, frequently exist in multiple versions which are designed to run on different machine configurations of varying sizes and compositions. Many experimental techniques are continually being developed to automate the production of such software and to eliminate much of the redundant effort associated with separate implementations.

One technique that has made significant contributions to the automation of software production uses specification languages (meta-syntactic languages) to describe the organizational structure of software systems and the hardware on which they function. Software and hardware descriptions made in such languages serve as input to special compilers (meta-compilers), which generate the desired version of a software system for a specific machine configuration.

A specification language designed for software production generally possesses a free format, is open ended so that it can incorporate additional language features, and usually employs recursive techniques which permit extremely compact descriptions of software and hardware characteristics. The descriptions produced in such a language reflect the organizational complexities of the software and hardware being described. The compilers that

process these descriptions perform intricate analyses to generate the desired version of a software system.

These meta-compilers often use special list organizations, called push-down lists or stacks, to simplify the scanning and analysis of source descriptions made in specification languages. The compilers also rely on list-processing methods to maintain efficient control over storage requirements, which vary widely, depending upon the complexity of the source description under analysis.

#### 1D5. Text Editing

The preparation of printed material in the publishing industry frequently involves many alterations to the original text before final copy is obtained. These changes affect copy preparation at all stages; source text undergoes repeated typing, and even after type has been set, modifications are still performed, often at great expense. Cheaper, quicker, and more flexible methods of producing printed text, however, are becoming available with the increased use of computers in the printing industry.

Once source copy is recorded in machine processable form (on magnetic tape, for example), it can be modified under program control. Text-editing programs can be used to insert, delete, and reformat copy, and an output tape can also be generated to control the operation of automatic typesetting devices, such as linotype and photo-composition machines.

The functions performed by text-editing programs generally require extensive movement of data within the computer. Characters, words, sentences, lines, paragraphs, pages, chapters, and sections are inserted, deleted, modified, or shifted to produce the desired text, and the variable lengths of these textual groupings also affect the efficient use of storage. By employing list-processing techniques, text-editing programs can minimize data movement and maintain control over storage efficiency.

#### 1D6. Artificial Intelligence

This application area deals with computer-related techniques for supplementing human intelligence. It attempts to simulate the cognitive processes involved in such activities as game playing (chess, checkers), theorem proving (geometry, algebra, symbolic logic), and pattern recognition (perception, discrimination, induction, hypothesis formulation).

These activities produce extremely complex problems which are usually solved in stages. Each stage, however, often generates a huge set of subproblems, all of which cannot be examined (even on a computer) in a reasonably short period of time. In a game of chess, for example, the possible lines of play from a given chess position can easily involve trillions of moves. But the combinations of winning moves continually change as new positions arise

at each stage of play. As a result, exhaustive analysis of all possible moves in a game of chess is not generally possible for a human or a computer.

Selecting proper subproblems without analyzing all possibilities at each stage of problem solution is a major concern of artificial intelligence. Short cuts involving classification techniques or heuristic methods are needed to discriminate among subproblems, and the discovery of such short cuts forms the main research activity of artificial intelligence.

Most computer programs designed to mechanize problem-solving behavior must contend with varying numbers of subproblems and unpredictable variations in problem structure. This type of variability calls for flexible methods of managing computer storage and organizing the structure of a problem. Such methods are available through list-processing techniques.

#### 1E. PURPOSE OF THIS MANUAL

Detailed development of the application areas described in the previous discussions lies beyond the scope of this manual. As indicated, however, the various applications possess common characteristics that permit efficient use of list-processing methods. The purpose of this manual, then, is to serve as an introduction to the basic facilities for list-processing in PL/I, and to show how these facilities form the basis of list-processing methods. The following chapters employ variations of an elementary sort program to show how lists are created and processed.

#### 1F. SUMMARY OF CHAPTER 1

- A. A list is a collection of non-contiguous data items linked in any desired order by means of attached address elements.
- B. The term *list processing* refers to that area of programming concerned with the construction, management, and application of lists.
- C. The use of lists can produce improvements in program execution time by avoiding unnecessary data movement.
- D. The use of lists can also improve overall use of storage by maintaining a common storage pool, from which storage is obtained when needed and to which storage is returned when not needed.
- E. The computer applications that can benefit most from list-processing techniques generally process large collections of data items that are interrelated in a logically complex manner.

## Chapter 2. The Organization of Data in PL/I

Proper understanding of the list-processing facilities in PL/I requires a basic knowledge of the internal data organizations provided by PL/I, namely, element items, arrays, structures, and arrays of structures. This chapter presents brief descriptions and simple examples of these organizations. The discussion is restricted to those aspects of internal data organization that are relevant to the list-processing facilities of PL/I.

List-processing deals primarily with the organization and manipulation of data and data addresses within the internal storage facilities of a computer. Although it provides more flexible methods of handling data than are available solely with element items, arrays, structures, and arrays of structures, list-processing does not replace these data organizations; it builds from them.

### 2A. ELEMENT ITEMS

PL/I statements refer to data items either individually as single data elements, or collectively as arrays and structures. Single data elements, called *element items*, have arithmetic, string, or label attributes and appear in PL/I expressions either as constants or as variables. The following examples show some types of element constants that may appear in PL/I programs:

Element Constant	Data Type
751.62	Arithmetic: FIXED DECIMAL (5,2)
11011.101B	Arithmetic: FIXED BINARY (8,3)
43.8E6	Arithmetic: FLOAT DECIMAL (3)
111.101E10B	Arithmetic: FLOAT BINARY (6)
'A1B2C3'	String: CHARACTER (6)
'10110111'B	String: BIT (8)
#3 INPUT:	Label: (the identifier to the left of a colon)

The DECLARE statement associates attributes with element variables as shown in the following examples:

```
DECLARE COST FIXED DECIMAL (5,2);
```

```
DECLARE FREQUENCY FLOAT BINARY (21);  
DECLARE TITLE CHARACTER (25);  
DECLARE MASK BIT (6);  
DECLARE SWITCH LABEL;
```

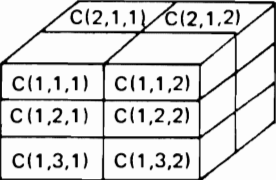
In many applications, data items occur in collections such as tables and records. To simplify the processing of such collections, PL/I provides facilities that allow data items to be organized and manipulated in array and structure form.

### 2B. ARRAYS

When all the element items in a collection have the same attributes, the items can be arranged in a multidimensional, tabular form, called an *array*. Examples of PL/I array declarations, with associated serial and spatial representations, appear in Figure 2B-1. The serial representations indicate how arrays are stored within the computer; note that the rightmost subscript values vary most quickly when array elements are referred to in succession.

For serial representation, array dimensions may be interpreted as grouping levels (indicated by the braces in Figure 2B-1). With this interpretation, a one-dimensional array is an array of element items; a two-dimensional array is an array of subarrays of element items; a three-dimensional array is an array of subarrays of subarrays of element items; and so on. The number of element items or subarrays at a given level is determined by the bounds of the dimension for that level. The second declaration in Figure 2B-1, for example, defines B as an array of three subarrays, each of which contains two fixed-point binary values as element items. This array has two dimensions. The lower bound of the first dimension is 1; the upper bound is 3. The second dimension has 1 as the lower bound and 2 as the upper bound.

Spatial representations of arrays occur in geometrical applications and apply to arrays with one, two, or three dimensions. These representations frequently simplify the definition and analysis of physical problems. Except for one-dimensional arrays, however, spatial representations have no direct representation within the serial storage of a computer.

ARRAY DECLARATION	SERIAL REPRESENTATION	SPATIAL REPRESENTATION						
DECLARE A(2) FIXED DECIMAL(4);	$A \left\{ \begin{array}{l} 1 \\ 2 \end{array} \right. \begin{array}{ c } \hline A(1) \\ \hline A(2) \\ \hline \end{array}$	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="padding: 5px;">A(1)</td> <td style="padding: 5px;">A(2)</td> </tr> </table> <p style="text-align: center;">(ONE-DIMENSIONAL)</p>	A(1)	A(2)				
A(1)	A(2)							
DECLARE B(3,2) FIXED BINARY(8);	$B \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \end{array} \right. \left\{ \begin{array}{l} 1 \\ 2 \\ 1 \\ 2 \\ 1 \\ 2 \end{array} \right. \begin{array}{ c } \hline B(1,1) \\ \hline B(1,2) \\ \hline B(2,1) \\ \hline B(2,2) \\ \hline B(3,1) \\ \hline B(3,2) \\ \hline \end{array}$	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="padding: 5px;">B(1,1)</td> <td style="padding: 5px;">B(1,2)</td> </tr> <tr> <td style="padding: 5px;">B(2,1)</td> <td style="padding: 5px;">B(2,2)</td> </tr> <tr> <td style="padding: 5px;">B(3,1)</td> <td style="padding: 5px;">B(3,2)</td> </tr> </table> <p style="text-align: center;">(TWO-DIMENSIONAL)</p>	B(1,1)	B(1,2)	B(2,1)	B(2,2)	B(3,1)	B(3,2)
B(1,1)	B(1,2)							
B(2,1)	B(2,2)							
B(3,1)	B(3,2)							
DECLARE C(2,3,2) CHARACTER(1);	$C \left\{ \begin{array}{l} 1 \\ 2 \end{array} \right. \left\{ \begin{array}{l} 1 \\ 2 \\ 3 \\ 1 \\ 2 \\ 3 \end{array} \right. \left\{ \begin{array}{l} 1 \\ 2 \\ 1 \\ 2 \\ 1 \\ 2 \\ 1 \\ 2 \\ 1 \\ 2 \end{array} \right. \begin{array}{ c } \hline C(1,1,1) \\ \hline C(1,1,2) \\ \hline C(1,2,1) \\ \hline C(1,2,2) \\ \hline C(1,3,1) \\ \hline C(1,3,2) \\ \hline C(2,1,1) \\ \hline C(2,1,2) \\ \hline C(2,2,1) \\ \hline C(2,2,2) \\ \hline C(2,3,1) \\ \hline C(2,3,2) \\ \hline \end{array}$	 <p style="text-align: center;">(THREE-DIMENSIONAL)</p>						
DECLARE D(2,2,2,2) BIT(8);	$D \left\{ \begin{array}{l} 1 \\ 2 \end{array} \right. \left\{ \begin{array}{l} 1 \\ 2 \end{array} \right. \left\{ \begin{array}{l} 1 \\ 2 \end{array} \right. \left\{ \begin{array}{l} 1 \\ 2 \\ 1 \\ 2 \\ 1 \\ 2 \\ 1 \\ 2 \\ 1 \\ 2 \\ 1 \\ 2 \\ 1 \\ 2 \\ 1 \\ 2 \end{array} \right. \begin{array}{ c } \hline D(1,1,1,1) \\ \hline D(1,1,1,2) \\ \hline D(1,1,2,1) \\ \hline D(1,1,2,2) \\ \hline D(1,2,1,1) \\ \hline D(1,2,1,2) \\ \hline D(1,2,2,1) \\ \hline D(1,2,2,2) \\ \hline D(2,1,1,1) \\ \hline D(2,1,1,2) \\ \hline D(2,1,2,1) \\ \hline D(2,1,2,2) \\ \hline D(2,2,1,1) \\ \hline D(2,2,1,2) \\ \hline D(2,2,2,1) \\ \hline D(2,2,2,2) \\ \hline \end{array}$							

8 Figure 2B-1. Array declarations with corresponding serial and spatial representations

## 2C. STRUCTURES

Structures use a system of level numbers to arrange element items in hierarchical fashion. Unlike array elements, however, the elements of a structure need not have the same attributes.

An example of a PL/I structure declaration appears in Figure 2C-1. The diagram in Figure 2C-2 represents the

relative storage requirements for the 13 element items of the structure. A structure may contain other structures as well as element items. In Figure 2C-2, the major structure TIME\_CARD contains the minor structures NAME and DATE. The appearance of the array HOURS in this example also shows that a structure may contain arrays. A representation of the grouping within this structure is given in Figure 2C-3.

```

DECLARE
1 TIME_CARD,
  2 NAME,
    3 LAST CHARACTER(15),
    3 FIRST_INITIAL CHARACTER(1),
    3 MIDDLE_INITIAL CHARACTER(1),
  2 MAN# CHARACTER(6),
  2 DEPT# CHARACTER(4),
  2 DATE,
    3 MONTH FIXED DECIMAL(2),
    3 DAY FIXED DECIMAL(2),
    3 YEAR FIXED DECIMAL(2),
  2 HOURS(5) FIXED DECIMAL(3,1);
  
```

Figure 2C-1. Structure declaration for TIME\_CARD

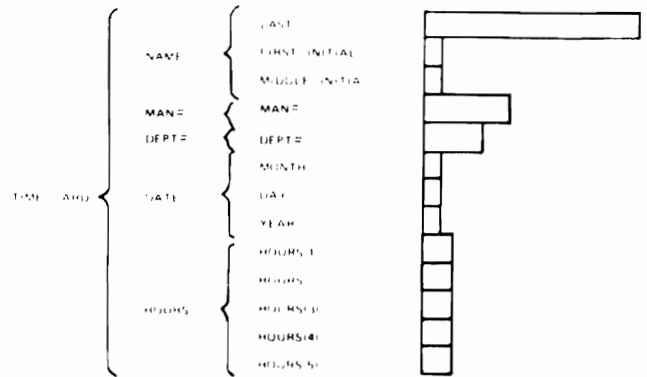


Figure 2C-2. Relative storage requirements for TIME\_CARD

NAME			MAN NO	DEPT NO	DATE			HOURS				
Last	First Initial	Middle Initial			Month	Day	Year	Hours(1)	Hours(2)	Hours(3)	Hours(4)	Hours(5)

Figure 2C-3. A time card

## 2D. ARRAYS OF STRUCTURES

PL/I allows a dimension attribute to appear with a structure name in a DECLARE statement. This type of declaration defines an array of structures, which is an array that contains structures with identical names and levels.

Consider the DECLARE statement in Figure 2D-1. This declaration defines an array of structures that contains three array members (see Figure 2D-2). Each array member is a structure that contains four element items (the high, low, and mean temperature readings for a specified date).

```

DECLARE
1 TEMP(3),
  2 DATE    CHARACTER(6),
  2 HIGH    FIXED DECIMAL(4,1),
  2 LOW     FIXED DECIMAL(4,1),
  2 MEAN    FIXED DECIMAL(4,1),

```

Figure 2D-1. Declaration of an array of structures

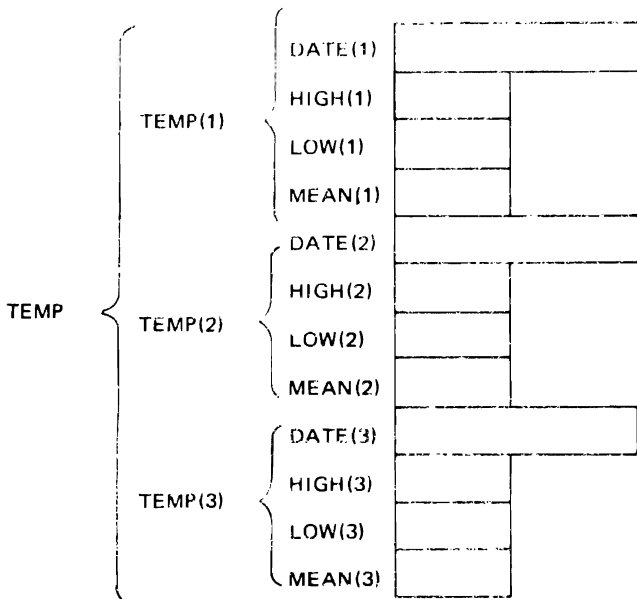


Figure 2D-2. Relative storage requirements for an array of structures

## 2E. ADVANTAGES OF ARRAYS AND STRUCTURES

The array and structure facilities of PL/I allow the data collections that occur in many different types of applications to be represented in a direct, natural way. For example, tax rates, wage scales, time schedules, inventory counts, physical measurements, statistical observations, frequency distributions, and function values are often recorded and used (even independently of computer applications) in formats that closely resemble the array and structure organizations of PL/I. By permitting the natural organizations of data items to be retained in computer applications, the array and structure facilities of PL/I tend

to emphasize the application-oriented, rather than the machine-oriented, aspects of computer programming. This emphasis on the application generally produces greater ease and clarity of programming.

But arrays and structures do more than simplify the writing of programs; they also affect the overall efficiency of programs. With these organizations, it is possible to reduce program storage and to improve program execution time. For instance, the subscripting and looping facilities of arrays often produce more compact programs by eliminating redundant statements. Similarly, structures avoid time-consuming data scans by providing direct access to the subfields of data strings.

### 2E1. Reducing Program Size

The array facilities of PL/I improve program efficiency by allowing compact program loops to replace repeated sequences of statements. The significance of the resulting reduction in program size becomes evident when attempts are made to program without arrays, that is, solely with element items.

As an illustration, consider the sort procedure SORT1 in Figure 2E1-1. This program obtains five successive records from the standard system input file (SYSIN), sorts the five records into ascending order on the first three characters of each record, and puts the five sorted records into the standard system output file (SYSPRINT). These steps are then repeated until all input records have been processed. (To simplify the discussion, the input file is assumed to contain a multiple of five records.)

A sample printout produced by SORT1 appears in Figure 2E1-2; a flowchart of the program is shown in Figure 2E1-3.

An important feature of SORT1 is that it does not use arrays. Therefore, each of the five records involved in the sort has a distinct name: CARD1, CARD2, CARD3, CARD4, and CARD5. Treating the records as individual elements and not as members of an array increases the lengths of the DECLARE, GET, and PUT statements in SORT1 and forces the sort to use four separate IF statements for successive comparisons of the five records.

Figure 2E1-4 illustrates successive passes of the sort technique used in SORT1. This technique is a type of transposition sort. (Note that the efficiency of this technique compared to other sort techniques is not under consideration; the present discussion is preliminary to demonstration of the advantages of arrays.)

Figure 2E1-5 shows how sorting with an array can result in a shorter program. SORT2 produces the same results as SORT1 of Figure 2E1-1. Instead of the five element items CARD1, CARD2, CARD3, CARD4, and CARD5, however, SORT2 uses a five-member array named CARD. The array name reduces the lengths of the DECLARE, GET, and PUT statements. (For example, the statement GET EDIT (CARD) (A(80)); causes five cards



```

SORT1:
F2E1_1:
PROCEDURE OPTIONS (MAIN);
DECLARF
  (CARD1, CARD2, CARD3, CARD4, CARD5,
  SAVE) CHAR (80);
ON ENDFILE (SYSIN) BEGIN;
CLOSE FILE (SYSPRINT);
GO TO OVER;
END;

INPUT:
GET
  EDIT (CARD1, CARD2, CARD3, CARD4,
  CARD5) (A(80));
PUT
  SKIP (2) LIST ('INPUT TO SORT1:');
PUT
  EDIT
  (CARD1, CARD2, CARD3, CARD4, CARD5)
  (SKIP, A(80));

SORT:
  K = 0;
  IF
    SUBSTR (CARD1,1,3) > SUBSTR (CARD2,1,3)
  THEN
    DO;
      K = 1; SAVE = CARD1; CARD1 = CARD2;
      CARD2 = SAVE;
    END;
  IF
    SUBSTR (CARD2,1,3) > SUBSTR (CARD3,1,3)
  THEN
    DO;
      K = 1; SAVE = CARD2; CARD2 = CARD3;
      CARD3 = SAVE;
    END;
  IF
    SUBSTR (CARD3,1,3) > SUBSTR (CARD4,1,3)
  THEN
    DO;
      K = 1; SAVE = CARD3; CARD3 = CARD4;
      CARD4 = SAVE;
    END;
  IF
    SUBSTR (CARD4,1,3) > SUBSTR (CARD5,1,3)
  THEN
    DO;
      K = 1; SAVE = CARD4; CARD4 = CARD5;
      CARD5 = SAVE;
    END;
  IF
    K = 1
  THEN
    GO TO
    SORT;
  PUT
    SKIP (2) LIST ('OUTPUT FROM SORT1:');
  PUT
    EDIT
    (CARD1, CARD2, CARD3, CARD4, CARD5)
    (SKIP, A(80));
  GO TO
  INPUT;

OVER:
END
  SORT1;

```

Figure 2E1-1. Sorting without an array

```

INPUT TO SORT1:
771_FIFTH-----5
A96_SECOND-----2
A14_FIRST-----1
W17_FUURTH-----4
FO2_THIRD-----3

OUTPUT FROM SORT1:
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FUURTH-----4
Z71_FIFTH-----5

INPUT TO SORT1:
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FUURTH-----4
Z71_FIFTH-----5

OUTPUT FROM SORT1:
A14_FIRST-----1
A96_SECOND-----2
FO2_THIRD-----3
W17_FUURTH-----4
Z71_FIFTH-----5

INPUT TO SORT1:
W17_FUURTH-----4
Z71_FIFTH-----5
A96_SECOND-----2
A14_FIRST-----1
FO2_THIRD-----3

OUTPUT FROM SORT1:
A14_FIRST-----1
A96_SECOND-----2
FO2_THIRD-----3
W17_FUURTH-----4
Z71_FIFTH-----5

```

Figure 2E1-2. Sample printout produced by SORT1

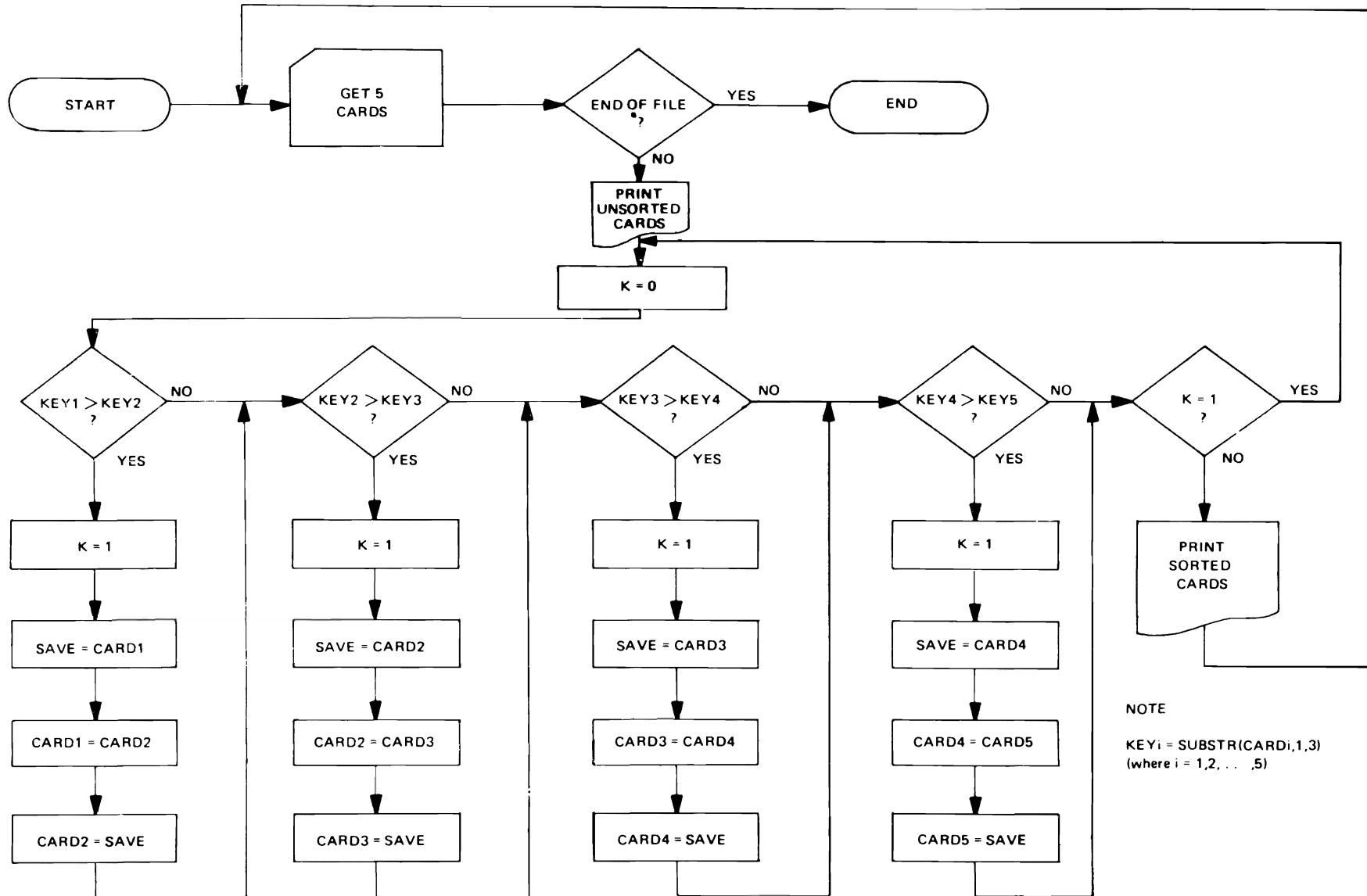


Figure 2E1-3. Flowchart for procedure SORT1

to be read). The array name also allows a single IF statement within a DO loop to replace the four IF statements used in SORT1 for comparing successive records.

Besides being a shorter program, SORT2 is more easily modified than SORT1. A change in the number of records to be sorted by SORT2 requires adjustments only in the size of the array CARD and in the upper limit of the DO statement. Corresponding modifications in SORT1 would require changes in the number of record names used by the DECLARE, GET, and PUT statements, as well as a change in the number of IF statements used for comparing successive records.

The flowchart for SORT2 appears in Figure 2E1-6.

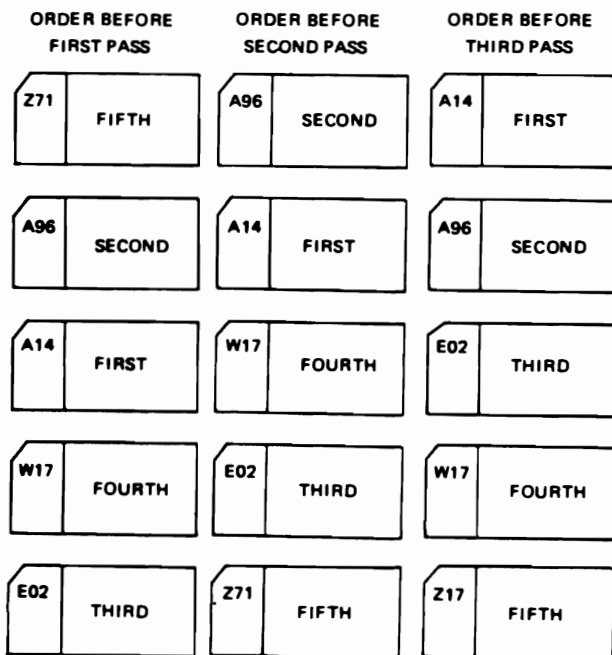


Figure 2E1-4. Example of sorting by successive interchanges

```

SORT2:
F2E1_5:
PROCEDURE (OPTIONS (MAIN));
  DECLARE
    (CARD(5), SAVE) CHARACTER(40);
  UN ENDFILE (SYSIN) BEGIN;
  CLOSE FILE (SYSPRINT);
  GO TO OVER;
  END;

INPUT:
  GET
  EDIT (CARD) (A(40));

SORT:
  K = 0;

COMPARE:
  DO I = 1 TO 4;
    IF
      SUBSTR(CARD(I),1,3) > SUBSTR(CARD
      (I + 1), 1, 3)
    THEN
      DU;
        K = 1; SAVE = CARD(I); CARD(I) =
        CARD(I + 1); CARD(I + 1) = SAVE;
      END;
    END
  COMPARE;
  IF
    K = 1
  THEN
    GO TO
    SORT;
  PUT
  SKIP (2) LIST ('OUTPUT FROM SORT2:');
  PUT
  SKIP (2) EDIT (CARD) (A(40), SKIP);
  PUT
  SKIP;
  GO TO
  INPUT;

OVER:
  END
  SORT2;

OUTPUT FROM SORT2:
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

OUTPUT FROM SORT2:
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

OUTPUT FROM SORT2:
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

```

Figure 2E1-5. Sorting with an array

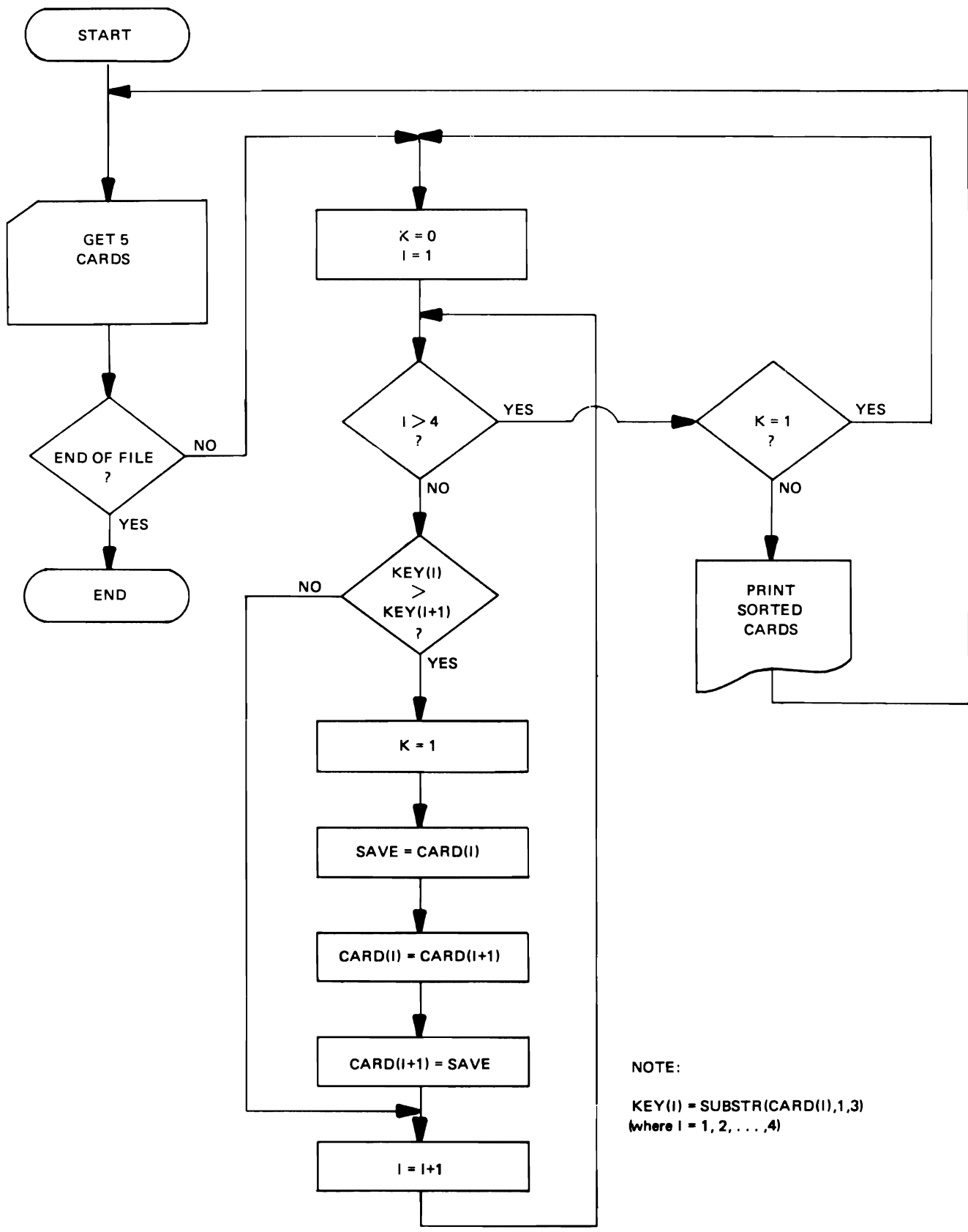


Figure 2E1-6. Flowchart for procedure SORT2

## 2E2. Avoiding Data Scanning

An improvement can be made to the SORT2 program of Figure 2E1-5 by not using the built-in function SUBSTR, which obtains the first three characters of each record. Although SUBSTR is a useful function for obtaining a portion of a string, the function is time-consuming and should be avoided when reduction in program execution time is desired.

One way of removing SUBSTR from the SORT2 program is to declare the explicit structure of each element in the array CARD. Then the first three characters of each element can be referred to directly by name rather than indirectly through SUBSTR.

The SORT3 program in Figure 2E2-1 contains explicit structure declarations for the members of CARD. The first three characters in each member of CARD are named KEY and the remaining 77 are named DATA. This declaration of an array of structures permits SORT3 to avoid references to the built-in function SUBSTR and, as a result, produces a shorter execution time for SORT3 than for SORT2. In general, explicit declarations improve data-access time and thereby provide faster execution time.

## 2F. SUMMARY OF CHAPTER 2

- A. PL/I statements refer to data items either individually as element items or collectively as arrays, structures, or arrays of structures.
- B. Array and structure facilities permit the natural organizations of data items to be retained, and thus tend to emphasize the application-oriented rather than the machine-oriented aspects of computer programming.
- C. The use of arrays can shorten programs by eliminating redundant statements.
- D. The use of structures can avoid time-consuming data scans by providing direct access to the subfields of data strings.

```

SORT3:
F2E2_1:
PROCEDURE OPTIONS (MAIN);
  DECLARE
    1 CARD(5),
    2 (KEY CHARACTER(3),
      DATA CHARACTER(77));
    1 SAVE,
    2 (KEY CHARACTER(3),
      DATA CHARACTER(77));
  ON ENDFILE (SYSIN) BEGIN;
  CLOSE FILE (SYSPRINT);
  GO TO OVER;
  END;

INPUT:
  GET
  EDIT (CARD)(A(3), A(77));
SORT:
  K = 0;
COMPARE:
  DO
    I = 1 TO 4;
  IF
    CARD.KEY(I) > CARD.KEY(I + 1)
  THEN
  DU:
    K = 1; SAVE = CARD(I);
    CARD(I) = CARD(I + 1);
    CARD(I + 1) = SAVE;
  END;
  END
  COMPARE;
  IF
    K = 1
  THEN
  GO TO
  SORT;
  PUT
  SKIP (2) LIST ('OUTPUT FROM SORT3:');
OUTPUT:
  DO
    I = 1 TO 5;
  PUT
  SKIP EDIT (CARD(I))(A(3), A(77));
  END
  OUTPUT;
  PUT
  SKIP(2):
  GO TO
  INPUT;
OVER:
  END
  SORT3:

OUTPUT FROM SORT3:
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

OUTPUT FROM SORT3:
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

OUTPUT FROM SORT3:
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

```

Figure 2E2-1. Sorting with an array of structures

## Chapter 3. Techniques for Addressing Data

The previous chapter showed how array and structure organizations permit reductions in program size and improvements in data-access time. These efficiencies, however, are diminished in many programs by the additional execution time that results from extensive movements of data within the computer. For example, the three sort programs SORT1, SORT2, and SORT3 of the previous chapter move entire records when interchanges occur. This practice causes a large amount of data to be moved and, as a result, produces a slow sort. Execution time is affected not only by the number of records being sorted but also by the size of each record. Generally, the less data that has to be moved, the faster the sort.

This chapter shows how data movement can be reduced by manipulating the addresses of data items rather than the items themselves, and also shows how scattered data items may be organized into collective units without moving or duplicating the items.

### 3A. TYPES OF ADDRESSES

A data address, as used in computer programming, specifies a particular storage location within which data can be stored and from which it can be retrieved. PL/I provides three major types of data addresses: symbolic, relative, and absolute. Brief explanations of each type appear in the following discussions.

### 3B. SYMBOLIC ADDRESSES

A symbolic address is an identifier that appears in a source program in place of an actual storage address (which, in machine language, is generally represented by a numeral). In PL/I, a symbolic address consists of a sequence of alphameric and break characters, the first of which must be alphabetic.

Symbolic addresses free the programmer from having to keep track of specific storage locations, simplify program organization and modification, and allow mnemonic names (such as TAX, PAYROLL, DATE1, and MASTER\_FILE) to be associated with data items. When a source program is translated into machine language, the compiler associates each symbolic address with a specific storage location. The ability to form such associations distinguishes symbolic programming from numeric coding.

### 3C. RELATIVE ADDRESSES

As demonstrated by the SORT1 program of Figure 2E1-1, programming solely with element items generally results in large programs, because distinct names are needed for each item. Organizing element items into arrays, however, reduces the number of distinct names in a program, permits address modification through changes in subscript values, and usually produces compact programs.

A reference to an item in an array consists of the array name followed by a subscript expression that specifies the position of the item in the array. In such a reference, the array name is a symbolic address, but the subscript expression is a relative address, because it specifies the location of the item relative to the beginning of the array.

Alteration of subscript values provides a restricted form of address modification that may be used to reduce data movement during execution of a program. When used for this purpose, subscript values may be stored in auxiliary arrays or employed as element links within arrays. Illustrations of these methods are given in paragraphs 3C1 and 3C2.

#### 3C1. The Use of Subscript Values in Auxiliary Arrays

The SORT4 program in Figure 3C1-1 shows how a second array N may be used to reduce data movement in a sort of an array of structures named CARD. SORT4 processes the same input and produces the same results as SORT1, SORT2, and SORT3 in the preceding chapter. After assigning input data to the five structures of CARD, SORT4 then assigns the integers 1 through 5 (in fixed-point binary form) to the five successive positions of N. Each element of N is treated as a subscript value that specifies (points to) one of the structures in CARD (see Figure 3C1-2).

As SORT4 compares the fields named KEY in successive structures of CARD, necessary interchanges are performed not on the structures of CARD but on the values of the corresponding elements of N. Moving the element values of N, rather than the 80-character structures of CARD, causes less data to be moved.

At the completion of the sort, the structures of CARD remain in their original (physical) order, but the element values of N are rearranged. Taken in succession, the rearranged values of N specify the sorted structures of CARD in ascending order on their KEY fields (see Figure 3C1-3).

As indicated in Figure 3C1-4, successive structures in the sorted array CARD may be referred to by the expressions CARD(N(1)), CARD(N(2)), CARD(N(3)),

```

SORT4:
F3C1_1:
PROCEDURE OPTIONS (MAIN);
DECLARE
    N(5) FIXED BINARY(3),
    1 CARD(5),
    2 (KEY CHARACTER(3),
    DATA CHARACTER(7));
ON ENDFILE (SYSIN) BEGIN;
CLOSE FILE (SYSPRINT);
GO TO OVER;
END;

INPUT:
GET
EDIT(CARD)(A(3), A(7));
INITIAL:
DO
    I = 1 TO 5; N(I) = I;
END
INITIAL;
SORT:
K = 0;
COMPARE:
DO
    I = 1 TO 4;
    IF
        KEY(N(I)) > KEY(N(I + 1))
    THEN
        DO;
            K = 1; J = N(I); N(I) = N(I + 1);
            N(I + 1) = J;
        END;
    END;
    COMPARE;
    IF
        K = .1
    THEN
        GO TO
        SORT;
    PUT
    SKIP (2) LIST ('OUTPUT FROM SORT4:');
OUTPUT:
DO
    I = 1 TO 5;
    PUT
    SKIP EDIT: (CARD(N(I)))(A(3),A(7));
END
OUTPUT;
PUT
SKIP(2);
GO TO
INPUT;
OVER:
END
SORT4;

```

```

OUTPUT FROM SORT4:
-----1
A14_FIRST-----2
A96_SECOND-----3
E02_THIRD-----4
W17_FOURTH-----5
Z71_FIFTH-----5

OUTPUT FROM SORT4:
-----1
A14_FIRST-----2
A96_SECOND-----3
E02_THIRD-----4
W17_FOURTH-----5
Z71_FIFTH-----5

OUTPUT FROM SORT4:
-----1
A14_FIRST-----2
A96_SECOND-----3
E02_THIRD-----4
W17_FOURTH-----5
Z71_FIFTH-----5

```

Figure 3C1-1. Using an auxiliary array of subscript values to reduce data movement in a sort

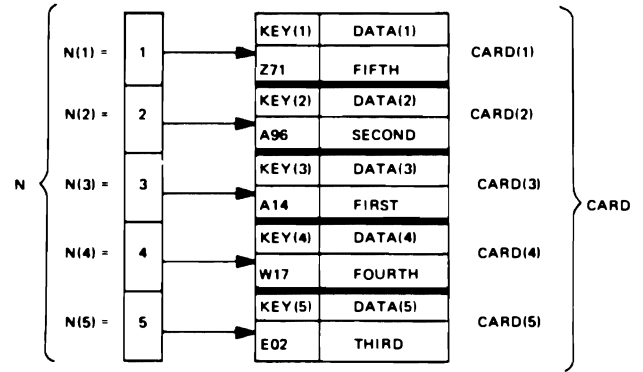


Figure 3C1-2. Auxiliary array and array of structures before sort

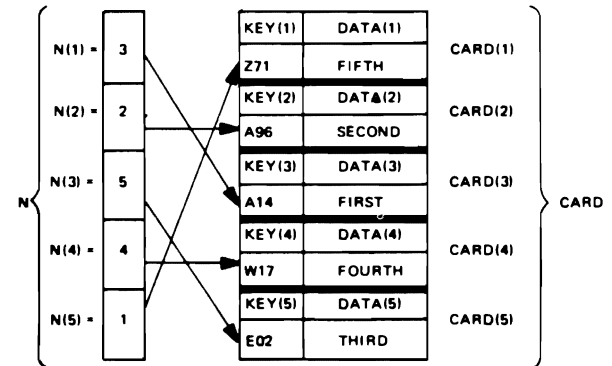


Figure 3C1-3. Auxiliary array and array of structures after sort

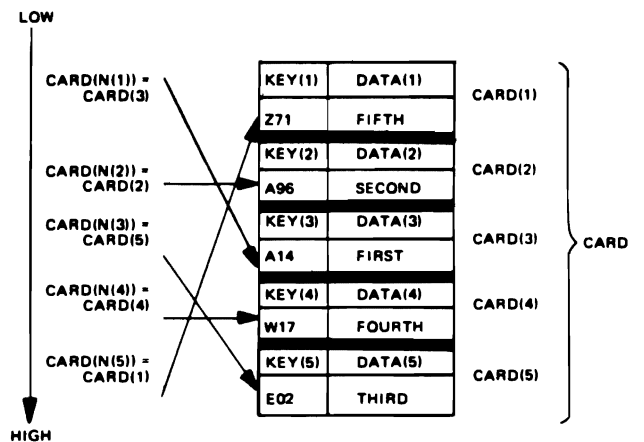


Figure 3C1-4. Sequential references to the sorted elements in an array of structures

CARD(N(4)), and CARD(N(5)). The compactness of these expressions shows the advantages of using subscripted subscripts when the elements of one array specify the elements of another array.

Although the structures in CARD appear only once, SORT4 specifies two orderings of CARD: the original (physical) order and the sorted (logical) order. With additional arrays of subscript values, it is possible to obtain further orderings from a single copy of CARD. This use of auxiliary arrays, then, not only reduces data movement but also avoids data duplication. Similar benefits may be obtained from auxiliary arrays when array elements are gathered, scattered, inserted, deleted, or merged.

### 3C2. Using Subscript Values to Link Array Elements

Subscript values can be used also to link (chain) the elements of an array into an ordered sequence. One way of doing this appears in the SORT5 program of Figure 3C2-1, which processes the same type of input data and produces the same type of output as SORT4.

```

SORT5:
F3C2_1:
PROCEDURE OPTIONS (MAIN);
  DECLARE
    1 CHAIN,
    2 H FIXED BINARY (3),
    2 E(5),
    3 CARD,
    4 KEY CHARACTER (3),
    4 DATA CHARACTER (77),
    3 L FIXED BINARY (3);
  ON ENDFILE (SYSIN) BEGIN;
  CLOSE FILE (SYSPRINT);
  GO TO OVER;
  END;

INPUT:
  GET
  EDIT (CARD) (A(3),A(77));
INITIAL:
  H = 1;
  L(1) = 2; L(2) = 3; L(3) = 4;
  L(4) = 5; L(5) = 0;
SORT:
  K_TEST = 0;
  IF
  KEY(H) > KEY(L(H))
  THEN
  DO;
  K_TEST = 1; J_SAVE = L(H);
  N_SAVE = L(J_SAVE);
  L(J_SAVE) = H; L(H) = N_SAVE;
  H = J_SAVE;
  END;
  I = H;
  DO
  WHILE (L(L(I)) ≠ 0);
  IF
  KEY (L(I)) > KEY(L(L(I)))
  THEN
  DO;
  K_TEST = 1; J_SAVE = L(L(I));
  N_SAVE = L(J_SAVE);
  L(J_SAVE) = L(I);
  L(L(I)) = N_SAVE;
  L(I) = J_SAVE;
  END;
  END;

```

Because the methods used in SORT5 are more complex than those used in the previous program (SORT4), the need for SORT5 might be questioned, particularly since both programs achieve the same results with equivalent efficiency. The main purpose for developing SORT5, however, is not to improve SORT4 but to introduce (through the already familiar methods of subscripting) the linkage techniques that underlie all list-processing methods. As later discussions show, the chained sequence of CARD structures used by SORT5 forms a type of primitive list organization.

SORT5 uses an array of structures called CHAIN, which contains an element item H and an array of structures E. Each of the five structures in E consists of a minor structure CARD and an element item L.

The program begins by assigning input data to the five structures named CARD (which are members of the array named E), and then assigns the integers 1 through 5 (in fixed-point binary form) to the element H and the first four L elements; the fifth element L receives a value of zero.

```

      I = L(I);
END;
  IF
  K_TEST = 1
  THEN
  GO TO
  SORT;
  PUT
  SKIP (2) LIST (*OUTPUT FROM SORT5*);
  PUT
  SKIP;
  I = H;
  DO
  WHILE (I ≠ 0);
  PUT
  SKIP EDIT (CARD (I)) (A(3), A(77));
  I = L(I);
  END;
  PUT
  SKIP (2);
  GO TO
  INPUT;
OVER:
  END
  SORT5;

```

```

OUTPUT FROM SORT5:
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

```

```

OUTPUT FROM SORT5:
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

```

```

OUTPUT FROM SORT5:
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

```

Figure 3C2-1. Using subscript values as element links to reduce data movement in a sort of an array of structure



As illustrated in Figure 3C2-2, element H acts as a header for CHAIN; that is, its value (in this case, 1) serves as a subscript item that specifies (points to) the first CARD structure (namely, CARD (1)). The value of element L(1), which is associated with CARD(1), specifies the subscript of the second CARD structure (namely, CARD (2)), and so on, until all CARD structures in CHAIN have been linked. The zero value of the last L element indicates the end of CHAIN.

SORT5 compares the KEY fields of successive CARD structures. Required interchanges are performed on the values of the H and L elements rather than on the CARD structures themselves; as a result, little data is moved during the sort. At the completion of the sort, as shown in Figure 3C2-3, the CARD structures remain in their original (physical) order, but the values of the H and L elements (underlined in the figure) link the structures in ascending order on their KEY fields.

The steps in Figure 3C2-4 show how SORT5 uses the following statements to transpose the order of the first two CARD structures:

```
J_SAVE = L(H);
N_SAVE = L(J_SAVE);
L(J_SAVE) = H;
L(H) = N_SAVE;
H = J_SAVE;
```

The variables J\_SAVE and N\_SAVE (both declared by default) serve as work variables for saving intermediate results. Also note that the interchange of the two CARD structures CARD(1) and CARD(2) requires modification of three link values: H = 2; L(1) = 3; and L(2) = 1;.

Step 4 of Figure 3C2-4 shows how the CARD structures are linked after the first interchange has been completed. At that point in the sort, the CARD structures occur in the following order:

CARD(2), CARD(1), CARD(3), CARD(4), CARD(5)

This is the logical order in which the CARD structures are linked; their original physical order remains unchanged.

When an interchange does not involve the first (logically first) CARD structure in the sorted sequence, header H is not modified; and, as the steps of Figure 3C2-5 show, SORT5 uses a different set of instructions for the interchange:

```
I = H;
J_SAVE = L(L(I));
N_SAVE = L(J_SAVE);
L(J_SAVE) = L(I);
L(L(I)) = N_SAVE;
L(I) = J_SAVE;
```

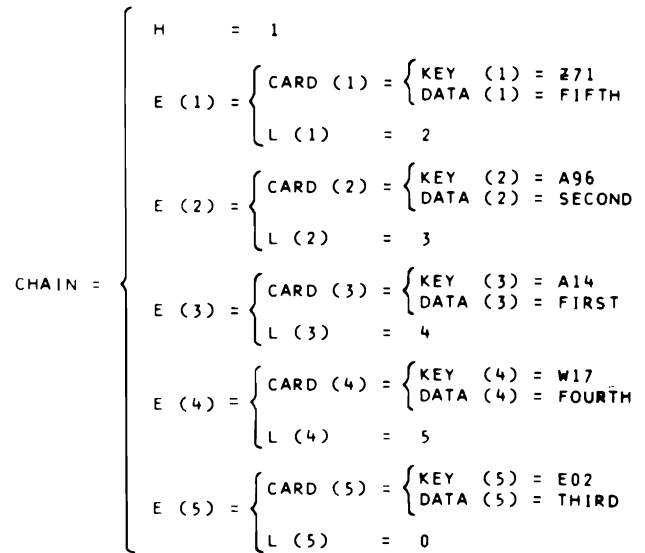


Figure 3C2-2. Array of linked structures before sort

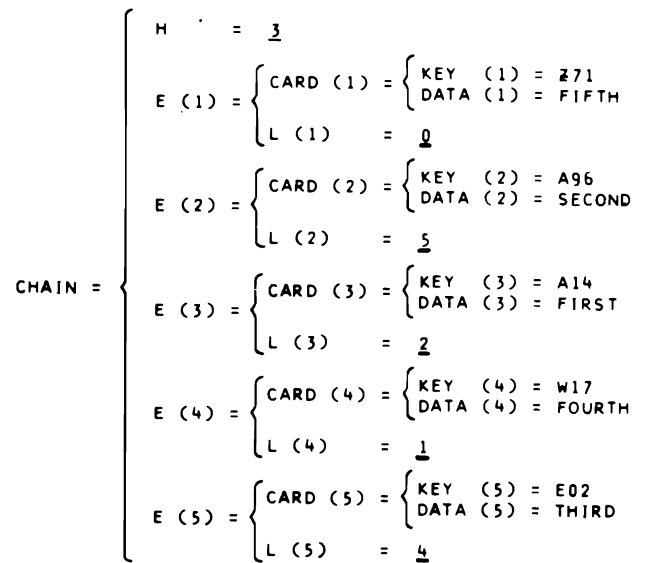
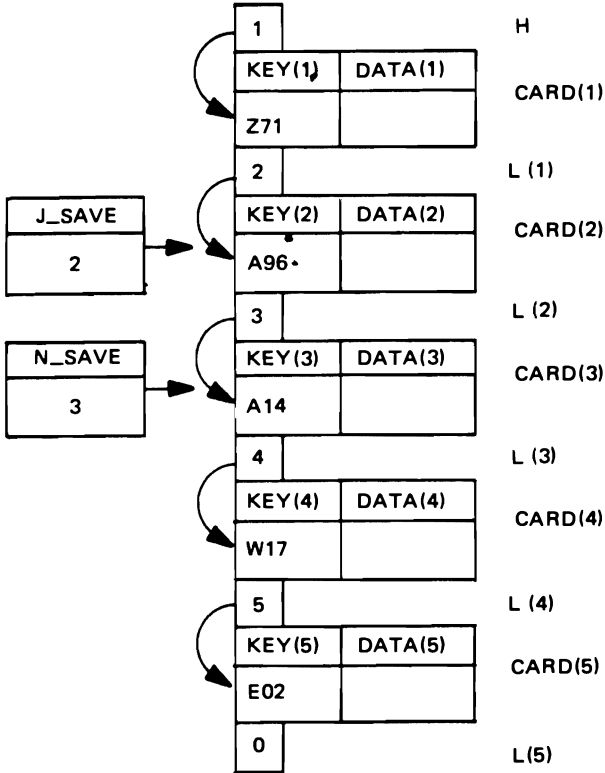
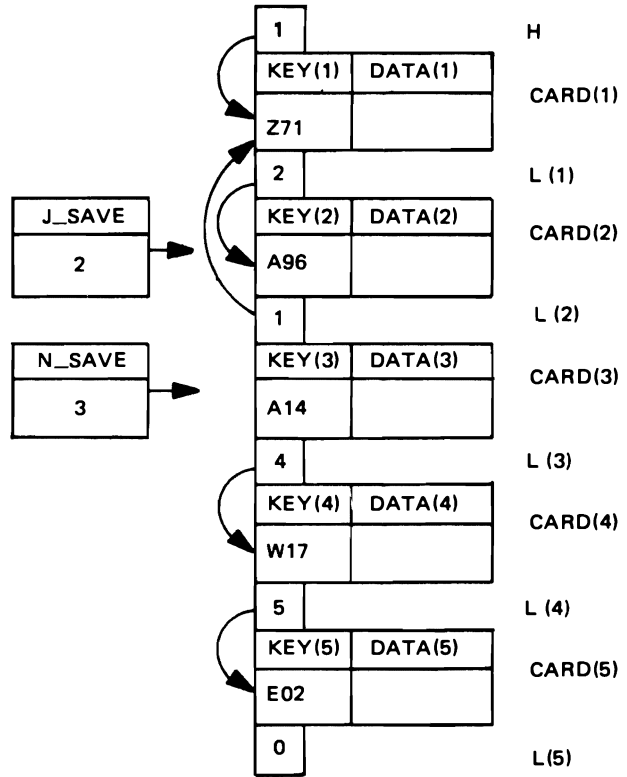


Figure 3C2-3. Array of linked structures after sort

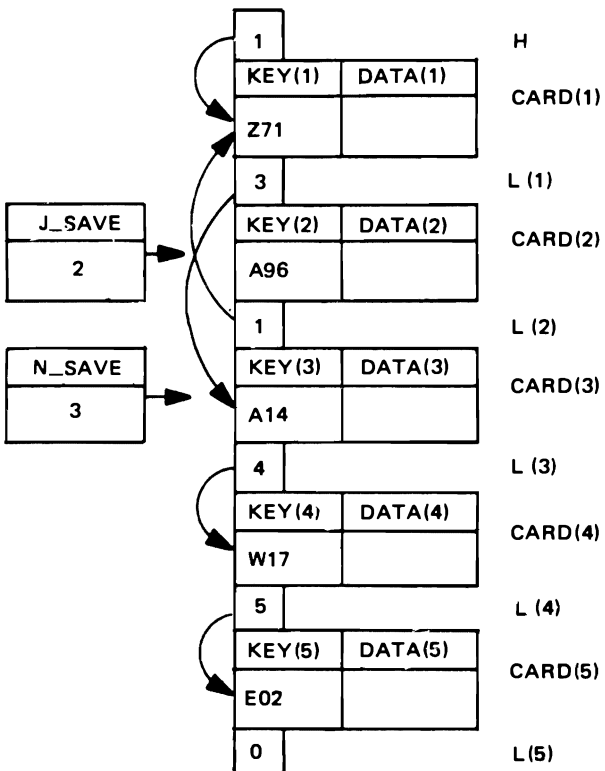
Step 1  $J\_SAVE = L(H); N\_SAVE = L(J\_SAVE);$



Step 2  $L(J\_SAVE) = H;$



Step 3  $L(H) = N\_SAVE;$



Step 4  $H = J\_SAVE;$

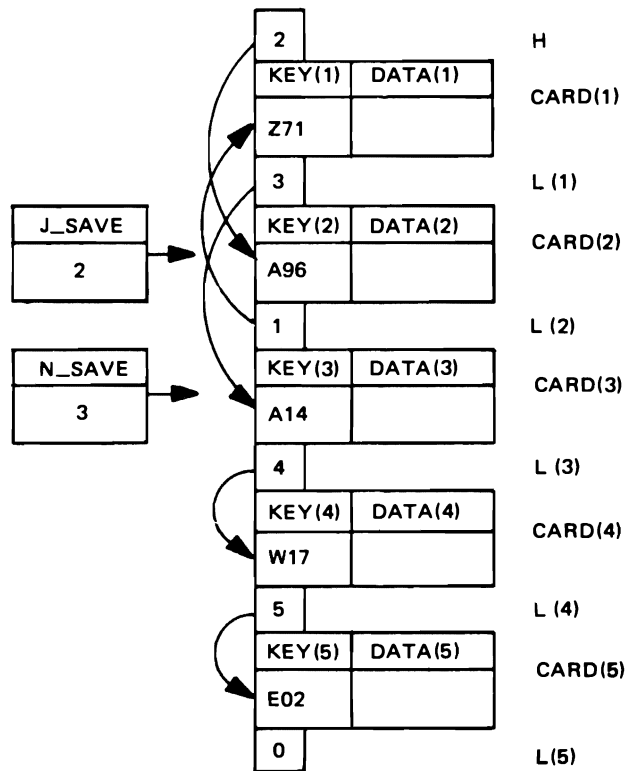
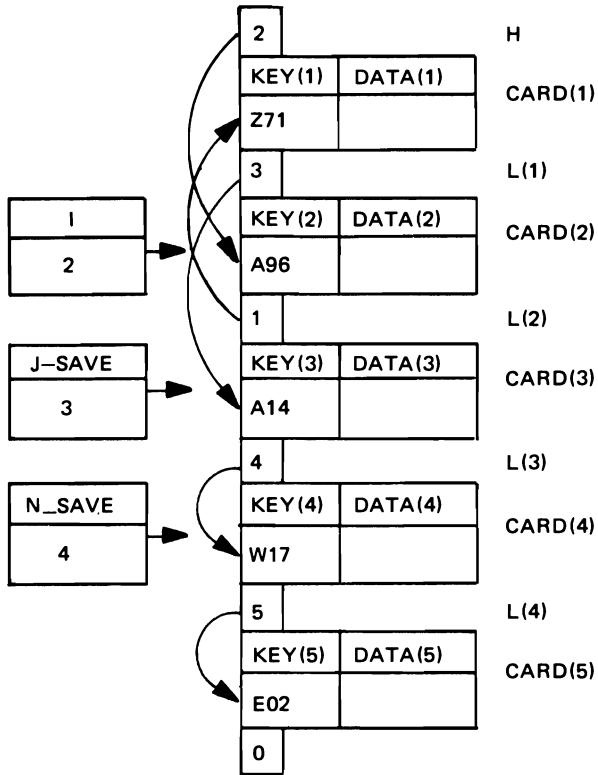
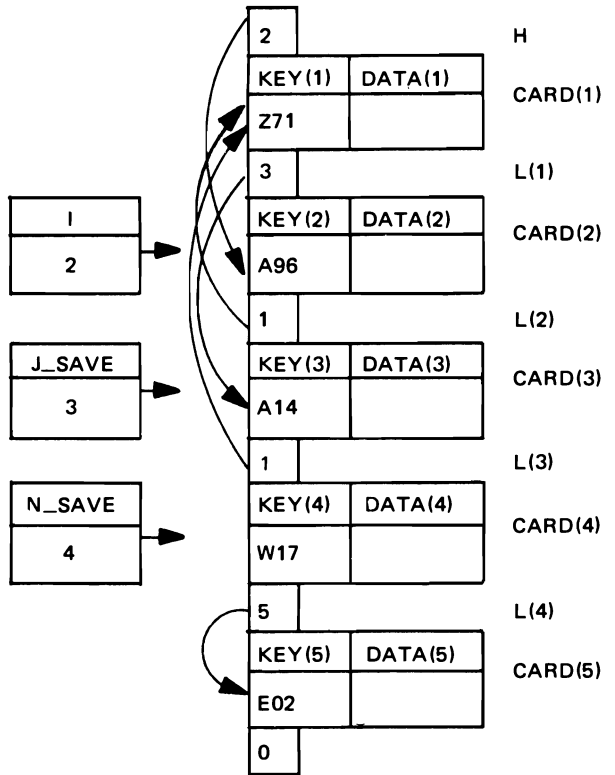


Figure 3C2-4. Performing the first interchange, which involves CARD(1) and CARD(2)

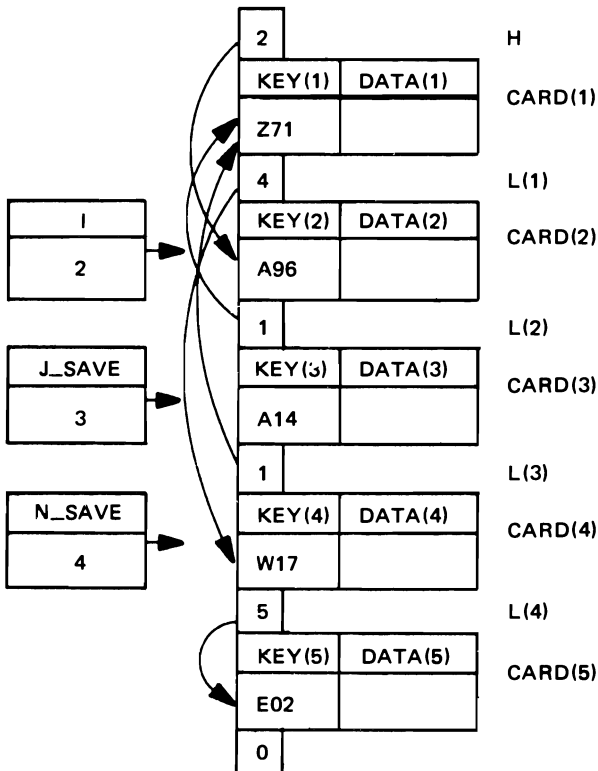
Step 1  $I = H$ ;  $J\_SAVE = L(L(I))$ ;  $N\_SAVE = L(J\_SAVE)$ ;



Step 2  $L(J\_SAVE) = L(1)$ ;



Step 3  $L(L(I)) = N\_SAVE$ ;



Step 4  $L(I) = J\_SAVE$ ;

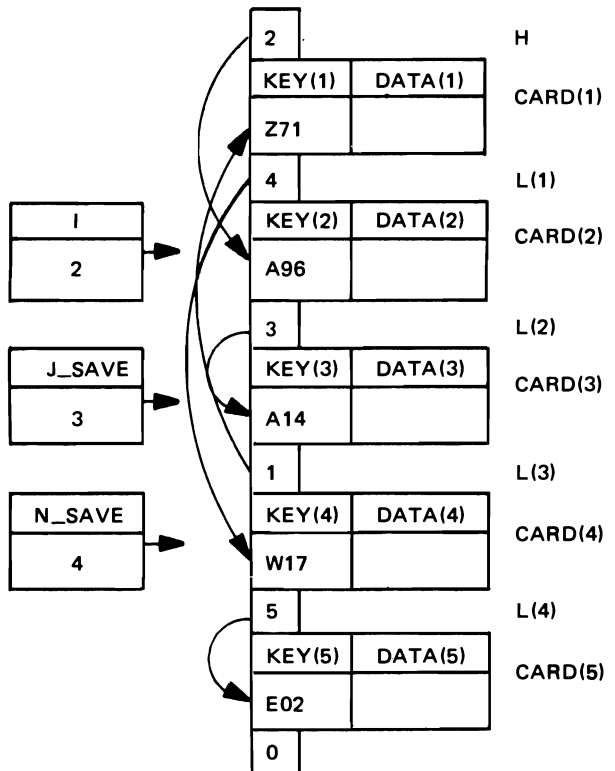


Figure 3C2-5. Performing the second interchange, which involves CARD(1) and CARD(3)

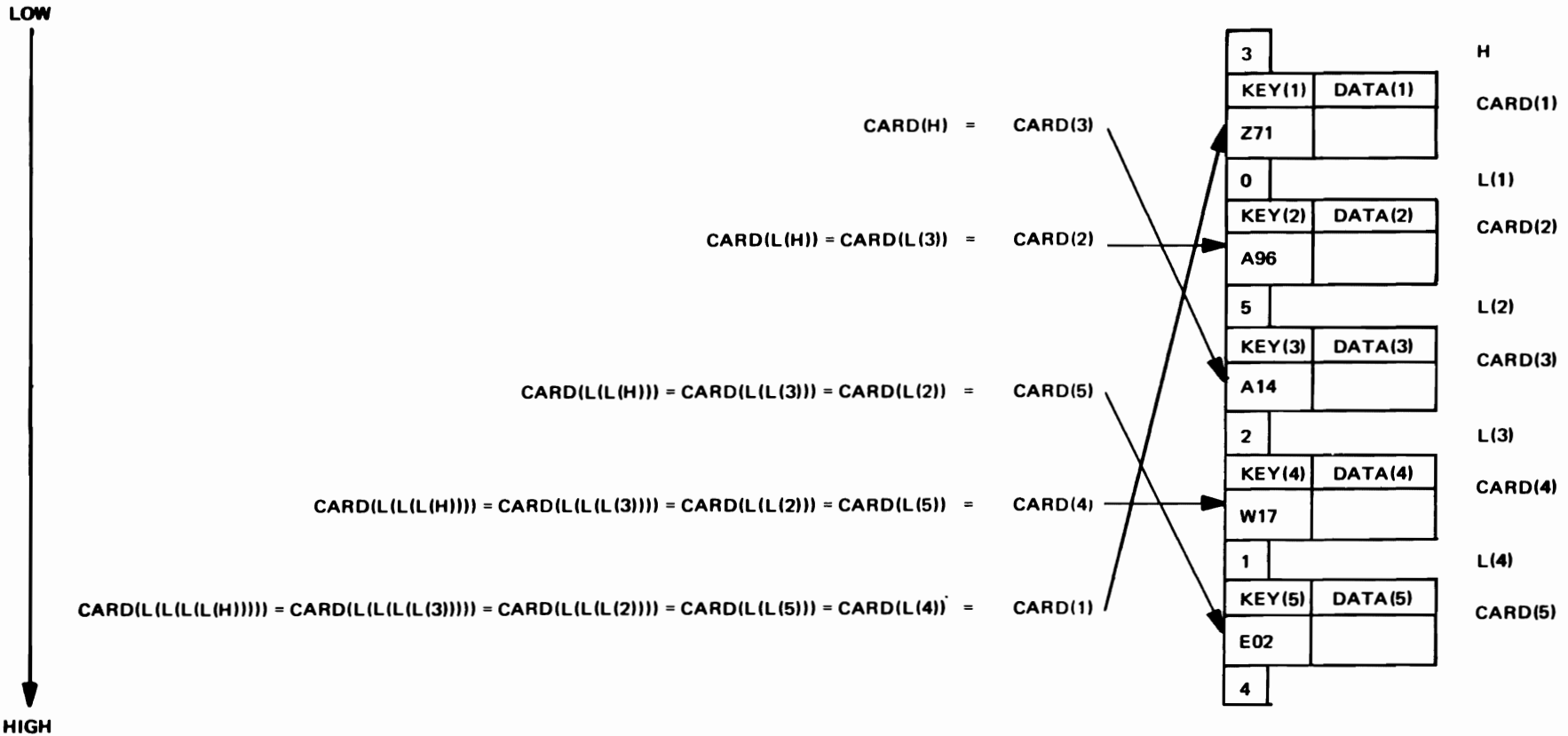


Figure 3C2-6. Sequential references to the sorted elements in a linked array of structures

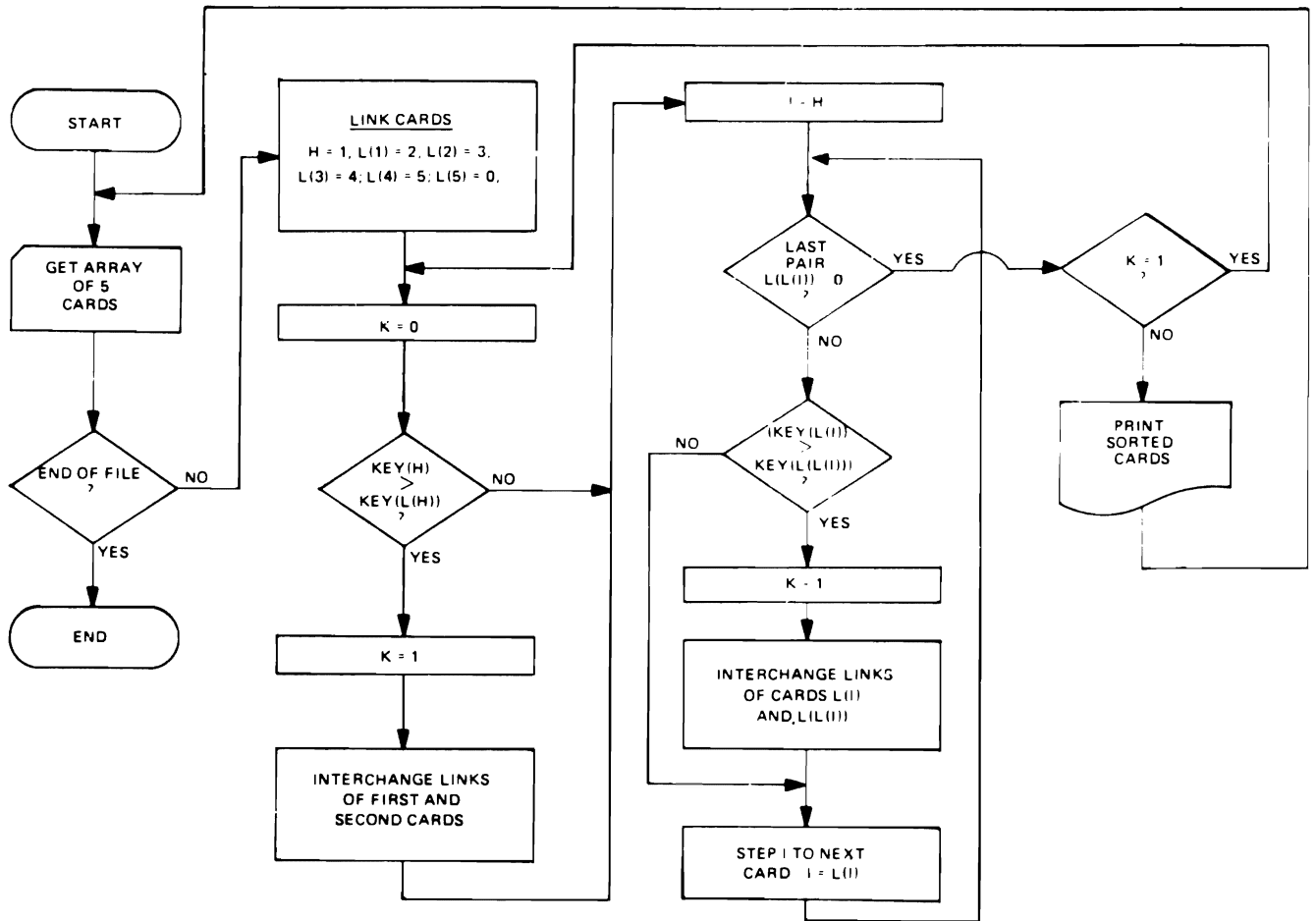


Figure 3C2-7. Flowchart for procedure SORT5

Since SORT5 executes these instructions under control of a DO-loop, variable I is used to specify successive CARD structures in the chained sequence; header H is not used for this purpose, because its value must be retained to determine which structure occurs first in the sequence.

Note the use of subscripted subscripts in Figure 3C2-5. Expression L(L(I)) in Step 1, for example, refers to the link value of CARD(1), since I = 2 and L(I) = 1. The figure illustrates the effect of the second interchange, which involves CARD(1) and CARD(3). Step 4 shows the CARD structures linked in the following order after the interchange has been completed:

CARD(2), CARD(3), CARD(1), CARD(4), CARD(5)

A complete sort of the CARD structures links them in the following order:

CARD(3), CARD(2), CARD(5), CARD(4), CARD(1)

Figure 3C2-6 shows how the sorted CARD structures may be referred to in succession by the expressions: CARD(H), CARD(L(H)), CARD(L(L(H))), CARD(L(L(L(H)))) and CARD(L(L(L(L(H))))). The compactness of these expressions illustrates the advantages of using successive levels of subscripted subscripts.

A flowchart for SORT5 appears in Figure 3C2-7.

### 3D. ABSOLUTE ADDRESSES

The addressing techniques used in the two previous examples (SORT4 and SORT5) depend upon relative addresses (subscripted array names). This dependency upon subscripts prevents similar applications of address manipulation to data items that are not contained in arrays. Furthermore, subscripting increases the time it takes the machine-language version of the program to refer to an element of an array. At execution time, the program must calculate the actual address of the array element in core storage and use this address in place of the relative address.

#### 3D1. The Relationship of Relative and Absolute Addresses

As an illustration of the relationship between relative addresses and actual addresses, consider the array of structures CARD shown in Figure 3D1-1. Each structure in the array is identified by a subscripted array name: CARD(1), CARD(2), CARD(3), CARD(4), and CARD(5). The same array appears in Figure 3D1-2, but the subscripted array name for each structure has been replaced by a possible storage address. It is assumed that each structure occupies 80 addressable storage positions and that the first structure begins at location 1000. Then the structures have the successive addresses 1000, 1080, 1160, 1240, and 1320. For convenience, the addresses are specified in decimal notation and not in binary, as would be required in the machine code for IBM System/360.

These addresses are called *absolute addresses*, because they specify the actual locations of the structures in storage. An absolute address is never relative to any location other than the first position of storage. By contrast, a relative address can be relative to any location in storage and, consequently, must be converted to an absolute address before it can be used by the program at execution time. Although the program automatically performs the conversion from relative to absolute form, such conversions may add to the running time of the program.

PL/I does not provide a direct means for controlling the absolute address of a data item. It is the operating system that determines which storage areas are available to a program at execution time. However, once a specific storage location has been allocated for a data item, PL/I makes it possible to obtain the absolute address of the location and to assign the absolute address to a special type of variable called a *pointer variable*.

Pointer variables can be organized into arrays and used in a manner similar to the way subscripts were used in SORT4 and SORT5. But pointer variables provide advantages over subscripts: they eliminate the need for storing data items in arrays and thereby avoid the time spent in converting relative addresses to absolute form. The following discussions present the rules for manipulating pointer variables and illustrate their use with examples.

CARD(1)	KEY(1)	DATA(1)
	Z71	FIFTH
CARD(2)	KEY(2)	DATA(2)
	A96	SECOND
CARD(3)	KEY(3)	DATA(3)
	A14	FIRST
CARD(4)	KEY(4)	DATA(4)
	W17	FOURTH
CARD(5)	KEY(5)	DATA(5)
	E02	THIRD

Figure 3D1-1. An array of structures showing a **relative address** (subscripted array name) for each structure

1000	KEY(1)	DATA(1)
	Z71	FIFTH
1080	KEY(2)	DATA(2)
	A96	SECOND
1160	KEY(3)	DATA(3)
	A14	FIRST
1240	KEY(4)	DATA(4)
	W17	FOURTH
1320	KEY(5)	DATA(5)
	E02	THIRD

Figure 3D1-2. An array of structures showing a possible **absolute address** for each structure

#### 3D2. Pointer Variables

Pointer variables have absolute addresses as their values. Declaration of an identifier with the POINTER attribute establishes the identifier as a pointer variable.

EXAMPLES:

```

DECLARE P POINTER;
DECLARE (Q, R) POINTER EXTERNAL STATIC;
DECLARE T(5) POINTER INTERNAL, V(-2:2,
-3:3)POINTER;
DECLARE 1A,
        2X CHARACTER(15),
        2Y POINTER;
DECLARE 1 TABLES,
        2 I(5) POINTER,
        2 J(0:4) POINTER;
    
```

As shown in these examples, PL/I allows pointer variables to be individual element variables or elements of arrays and structures. A pointer variable can have any storage class and scope, and the usual default rules for these attribute types also hold for a pointer variable.

### 3D2A. How to Obtain a Value for a Pointer Variable

Before a pointer variable can be manipulated, it must be assigned an absolute address. One way of obtaining an absolute address is to use the built-in functions ADDR and NULL. A reference to the built-in function ADDR has the following form:

ADDR(argument-variable)

The value returned by ADDR is the absolute address of the specified argument variable. As an example, consider the following assignment statement:

P = ADDR(X);

Assume P is a pointer variable and X is a data variable. Then the reference ADDR(X) obtains the absolute address of the storage location allocated for X, and the statement assigns this absolute address as the value of pointer P.

The argument variable in a reference to ADDR must be an identifier that specifies one of the following types of variables:

- A. an element variable
- B. an array
- C. an element of an array
- D. a major or minor structure
- E. an element of a structure

The argument can be of any data type and storage class, but special results occur when the following conditions apply to the argument of ADDR:

- AA.** When the argument is a controlled variable, the value of the ADDR reference specifies the absolute address of the current generation of the argument. If no storage has been allocated for the controlled argument, ADDR specifies a null address (which indicates unallocated storage).
- BB.** When the argument is a parameter of a containing procedure, the value of the reference to ADDR represents the absolute address of the argument associated with the parameter (including the absolute address of a dummy associated argument).
- CC.** When the argument is an array expression that specifies two or more noncontiguous elements, a reference to ADDR is not valid. For example, the elements of an array cross-section whose reference

includes asterisks not in the rightmost position or positions do not occupy contiguous storage locations. Since ADDR obtains a single absolute address, it has no way of relating noncontiguous storage positions with that address.

A reference to the built-in function NULL uses no arguments and has the following form:

NULL

This function returns a null address value that does not identify any location in storage. As later discussions illustrate, this special address value is used to clear pointer variables and to test for unallocated storage.

### 3D2B. Using Pointer Variables in Assignment Statements

PL/I permits pointer variables to appear in the following forms of the assignment statement:

1. element-pointer-variable = element-pointer-expression;
2. pointer-array = element-pointer-expression;
3. pointer-array = pointer-array;

Two or more variables separated by commas may appear on the left side of these statements for multiple assignment. An element-pointer expression on the right side of an assignment statement must be either an element-pointer variable or a function reference (built-in or programmer-defined) that specifies an element pointer value, that is, a single absolute address.

Assignment of an element-pointer expression to a pointer array causes the value of the expression to be assigned to every element of the pointer array. When a pointer array appears on the right side of an assignment statement, the number of dimensions and the bounds for each dimension of the array on the right must be identical to those of the receiving pointer array on the left.

#### EXAMPLES:

Assume the following pointer declarations:

```
DECLARE(P,Q,R,T(5),V(-2:2,-3:3)) POINTER;  
DECLARE 1 A, 2 X CHARACTER(15), 2Y POINTER;  
DECLARE 1 TABLES, 2 I(5) POINTER, 2 J(0:4)  
POINTER;
```

The following statements illustrate possible pointer assignments:

1. P = ADDR(A);
2. Q, R = NULL;
3. T(4), V(-2, -3) = P;
4. A. Y = T(4);
5. TABLES. I, TABLES. J = NULL;
6. T = TABLES. I;

Statements 2, 3, and 5 perform multiple assignments. The last three statements show name qualification applied to pointer variables.

### 3D2C. Using Pointer Variables in Operational Expressions

PL/I allows only two operators to use pointer variables as operands: the comparison operators equal (=) and not equal ( $\neq$ ). A major consequence of this restriction is that arithmetic operations cannot be performed on absolute addresses.

#### EXAMPLES:

Assume that A and B are arithmetic variables and that P, Q, R, S, and T are pointer variables; then the following statements contain permissible uses of pointer variables as operands:

1. IF P = NULL THEN GO TO L;
2. A = (Q  $\neq$  R);
3. IF (T=NULL) | ((T=ADDR(B)) & (S  $\neq$  NULL)) THEN P=Q;

The last example shows that simple comparisons of pointer variables may be compounded.

### 3D3. Based Variables

Although the value of a pointer variable represents the absolute address of a specified data item, the pointer itself provides no information about the attributes of the data item. Such descriptive information is needed, however, for proper manipulation of the data item at execution time. For example, a pointer variable can specify the absolute address of an array A, but neither the pointer name nor the pointer value indicates the dimensions of A or the characteristics of its elements.

To associate descriptive information with a pointer variable, PL/I provides a special type of variable called the based variable, which is declared with the following attribute:

BASED(element-pointer-variable)

The element-pointer variable appearing in the BASED attribute cannot be a based variable itself nor can it be subscripted.

Declaration of a based variable does not assign an address to the pointer variable specified in the associated BASED attribute. Before reference can be made to a based variable, an address must be assigned to its associated pointer variable.

Any reference to a based variable applies the attributes of the based variable to the storage location specified by the associated pointer variable. Consider the following declaration:

```
DECLARE NAME CHARACTER(15) BASED(P);
```

This statement declares the 15-position character string called NAME to be a based variable and associates the pointer variable P with NAME. A subsequent reference to NAME will then cause the location given by P to be treated as a storage area for a 15-position character string. For example, let NAME appear in the following assignment statement:

```
NAME = 'JOHN';
```

This statement assigns the character-string constant 'JOHN' to a 15-position storage area at the location given by P. The four characters of the constant are positioned to the left in the area and are followed by eleven blank characters.

The attribute BASED(element-pointer-variable) is defined to be a storage-class attribute along with STATIC, AUTOMATIC, and CONTROLLED. The appearance of BASED in a DECLARE statement, however, does not produce an allocation of storage. Only when an absolute address is assigned to the pointer variable related to the based variable does storage become associated with the based variable. For example, consider the previous declaration of the based variable NAME. Not until an absolute address is assigned to pointer P does storage become associated with NAME. When this association occurs, NAME is said to be "based on" P.

The value of the pointer variable in a BASED attribute can specify a location of any data type and storage class, including POINTER data and BASED storage. Care must be taken, though, when changing the pointer value related to a based variable to assure compatibility between the attributes of the based variable and the data at its newly assigned location.

#### 3D3A. Assigning Pointer Values with the Set Option

Executing an assignment statement is not the only way of assigning an address to a pointer variable. Another way is to use a SET option with one of the following statements:

```
READ FILE (file-name) SET (element-pointer-variable);  
LOCATE based-variable FILE (file-name) SET (element-  
pointer-variable);  
ALLOCATE based-variable SET (element-pointer-  
variable);
```

The READ and LOCATE statements perform record-oriented transmission; they process sequential buffered-files and allow logical records to be retrieved from and stored in file buffers. The ALLOCATE statement allocates storage for a based variable and assigns the location of the allocated storage to the pointer variable specified in the



SET option. Further discussion of the ALLOCATE statement appears in paragraph 4B1. The following discussion presents brief explanations of the SET option in READ and LOCATE statements.

The READ statement obtains the location of the next logical record in a buffer associated with the specified file, and assigns the location to the element-pointer variable given in the SET option. A based variable associated with the same pointer will then relate to the fields of the logical record. The based variable is effectively overlaid on the logical record in the buffer.

The LOCATE statement allocates the next available storage area for the specified based variable within a buffer associated with the file. The location of the allocated storage is assigned to the element-pointer variable given in the SET option. The LOCATE statement need not contain a SET option; when it does not, an implied SET is assumed, which uses the pointer variable in the BASED attribute of the specified based variable.

Record-oriented transmission statements may read and write the values of pointer variables. A pointer value that has been written, however, cannot be assumed to locate the same data if it is read back into storage (further discussion of this appears in Chapter 5). Under no circumstances may pointer values be read or written with stream-oriented transmission statements.

### 3D3B. Using a Based Variable in a Function Procedure

Function procedure MEAN in Figure 3D3B-1 shows how a based variable can be used with an array of pointers to simulate a variable-length parameter list. This function procedure uses one parameter P, which is an array of pointers. The single asterisk in the dimension attribute of P indicates that P is one dimensional and has the same dimension bounds as its associated argument in an invocation of MEAN. This use of the asterisk notation allows the function to process pointer-array arguments of different sizes.

Since the elements of P are associated with pointer values and the number of its elements can vary, P acts as a variable-length parameter list. Variability can also be achieved by associating a null pointer value with one of the P elements.

MEAN assumes that the non-null pointer values in P specify the locations of fixed point decimal values. The function computes the average of these arithmetic values and, as indicated by the attributes in the PROCEDURE statement, returns the average as a fixed-point decimal value.

Assignment of each non-null pointer value in P to pointer S causes the attributes of based variable VALUE to be applied to the locations specified in P. When all elements in P are null, the function returns a zero value.

```
T_MEAN:
F3D3B-1:
PROCEDURE OPTIONS (MAIN);
DECLARE
    MEAN RETURNS (FIXED),
    (T(10), X) FIXED (5),
    (P(10), S) POINTER;
PUT
    PAGE LIST ('INPUT ARRAY T IS: ');
#100:
DO
    I = LBOUND (T,1) TO HBOUND (T,1);
GET
    EDIT (T(I)) (F(5));
PUT SKIP DATA (
    T(I));
P(I) = ADDR(T(I));
END
#100:
PUT SKIP (2);
X = MEAN (P);
PUT
    SKIP (3) LIST
    ('RESULT RETURNED BY PROC **MEAN** =', X);
MEAN:
PROCEDURE
    (P) FIXED (5);
DECLARE
    (P(*), S) POINTER,
    VALUE BASED (S) FIXED (5),
    (M, N) FIXED (5);
M, N = 0;
LOOP:
DO
    I = LBOUND (P,1) TO HBOUND (P,1);
IF
    P(I) = NULL
    THEN
    DO:
        S = P(I);
        M = M + VALUE;
        N = N + 1;
END;
END
LOOP;
IF
    N = 0
    THEN
    RETURN (0);
ELSE
    RETURN (M/N);
END
MEAN;
END
T_MEAN;
```

```
INPUT ARRAY T IS:
T(1) = 11;
T(2) = 22;
T(3) = 33;
T(4) = 44;
T(5) = 55;
T(6) = 66;
T(7) = 77;
T(8) = 88;
T(9) = 99;
T(10) = 11;
```

RESULT RETURNED BY PROC 'MEAN' =

50

Figure 3D3B-1. Using a based variable and an array of pointers to simulate a variable-length parameter list

The lower and upper bounds of the array argument associated with parameter P are determined by the built-in functions LBOUND and HBOUND, reference to which occurs in the DO statement of MEAN.

### 3E. ARRAYS AND STRUCTURES OF BASED VARIABLES

Besides being used in the declarations of element items, the BASED attribute can also appear in the declaration of an array, a structure, or an array of structures. When applied to a structure or an array of structures, BASED must appear at level 1 and, consequently, applies to all members of the structure or array of structures.

Care is also required when assigning an absolute address to the pointer variable of a based array or structure to assure that the address actually specifies the location of an array or a structure. For example, it is generally incorrect to assume that the address of an array or a structure is the same as the address of its first element. Compilers frequently place descriptive information, applicable to the entire array or structure, before the first element of the array or structure in the object program. This descriptive information can cause the address of the complete array or structure to differ from that of the first element when the program is executed.

As an example, consider the following sequence of statements:

```

DECLARE
TABLE(5) FIXED DECIMAL(2),
1 CARD,
  2 KEY CHARACTER(3),
  2 DATA CHARACTER(77),
(P,Q,R,S) POINTER;
.
.
.
P = ADDR(TABLE);
Q = ADDR(TABLE(1));
.
.
.
R = ADDR(CARD);
S = ADDR(CARD.KEY);
.
.
.

```

It cannot be assumed in this example that pointers P and Q have the same address value; the address of TABLE is not necessarily equal to the address of its first element TABLE(1). The same distinction applies to pointers R and S; the address of structure CARD generally differs from the address of its first element CARD.KEY.

When a component of a based structure is referred to, the address of the component, relative to the beginning of the structure, is automatically accounted for and need not be adjusted (offset) by the programmer. Consider the following statement sequence:

```

DECLARE P POINTER
  1 I BASED(P), 2 J CHARACTER(10), 2 K CHARACTER(20),
  1 L, 2 M CHARACTER(10), 2 N CHARACTER(20);
.
.
.
GET EDIT (M,N) (A(10), A(20));
P = ADDR(L);
PUT LIST(K);
.
.
.

```

In this example, values for element variables M and N in structure L are obtained from the standard system-input file (SYSIN) by the edit-directed GET statement. Assignment of the address of L to pointer P associates the description of the based structure I with the storage of structure L. The reference to K in the list-directed PUT statement causes the value of N to be printed. Note that no adjustment in the address of structure L assigned to pointer P is necessary to obtain the value of N, even though N is the last element in the structure.

#### 3E1. Qualifying Based Variables with Pointer Variables

PL/I allows a based variable to be associated with more than one storage area at the same time. This multiple association is possible because a based variable by itself does not specify a data item, but only a description of storage. A based variable specifies the attributes and extent of the storage with which it is associated. But a change in the value of the pointer variable specified in the declaration of the based variable causes the based variable to become associated with different storage, and consequently, with new data. It is the combination of pointer variable and based variable, therefore, that determines the location and description of a data item.

So far, the BASED attribute is the only facility that has been presented for associating a pointer variable with a based variable. Since only one pointer variable can appear in a BASED attribute, some other facility is required for simultaneously associating two or more pointers with the same based variable. The PL/I facility that permits this multiple association is called pointer qualification. It is used to distinguish among two or more storage areas associated with the same based variable, and allows other pointers to override the pointer that was specified in the declaration of the based variable.

The pointer qualification symbol is a composite symbol that resembles an arrow. It consists of a minus sign immediately followed by a greater-than symbol ( $\rightarrow$ ). This composite symbol, however, does not signify an operation; its function is similar to that of the period symbol used in the qualified name of a structure element. When used, the pointer qualification symbol must always appear between two references. The reference on the left must be either an element-pointer variable or a reference to the built-in function ADDR. When it is an element-pointer variable, it cannot be subscripted nor can it be of the based storage class. The reference on the right of the pointer qualification symbol must be a based variable.

*A pointer qualification symbol applies the storage description of the based variable on its right to the storage location specified by the pointer value on its left; the pointer originally declared with the based variable is overridden. As an example, consider the following assignment statement:*

$$A \rightarrow B = C \rightarrow B;$$

Assume B is a based variable, and A and C are nonbased, element-pointer variables. The expression  $C \rightarrow B$  refers to a data item that has the attributes declared for B and the location specified by C. Similarly, the expression  $A \rightarrow B$  determines the location and attributes of the area to which the data item is assigned. Thus, the pointer qualification symbols used in this assignment statement associate the attributes declared for the based variable B with the two distinct storage areas addressed by pointers A and C. This association constitutes *pointer qualification*.

The examples of pointer qualification given below use the variables described in the following DECLARE statement:

```
DECLARE
(P,Q) POINTER,
  1 W BASED(P),
  2 X FIXED,
  2 Y FLOAT,
  2 Z POINTER,
(A, B BASED(Q)) FIXED;
```

P and Q are pointer variables. W is a based structure, and X, Y, Z are based element-variables within W. Observe that Z is a based element-pointer variable. A is a non-based fixed-point variable, and B is a based fixed-point variable.

Assume that P and Q have been assigned absolute addresses; then the following statements contain valid references to the elements of the based structure W:

- A.  $Y = X;$
- B.  $P \rightarrow Y = P \rightarrow X;$

In statement A, the references to the based elements X and Y do not involve the pointer qualification symbol; therefore, the pointer variable P given in the BASED attribute of W implicitly specifies the location of the structure addressed by P within which the attributes of X and Y are to be applied. If desired, *explicit qualification* of X and Y by P can be specified, as shown in the second statement. Statements A and B are effectively equivalent.

- C.  $Q \rightarrow Y = Q \rightarrow X;$

Statement C shows that a different pointer, in this case Q, can be used to override pointer P, which appears in the based declaration of W.

- D.  $P \rightarrow Y = Q \rightarrow Y;$
- E.  $Q \rightarrow X = P \rightarrow X;$

The references in statements D and E show that two different pointers may qualify the same based variable at the same time. This is the multiple association of storage areas with a based variable, as discussed previously. In statement D, the attributes of element Y are applied simultaneously to the two different structure locations given by P and Q. Similarly, statement E applies the attributes of element X to the structure locations given by P and Q.

- F.  $Y = (Q \rightarrow X) + (Q \rightarrow Y);$
- G.  $P \rightarrow Y = (Q \rightarrow X) + (Q \rightarrow Y);$

Statements F and G are equivalent. They add the values of elements X and Y within the structure location addressed by Q, and assign the sum to element Y within the structure given by P. Statement G contains explicit pointer qualification of the receiving based variable; statement F does not.

- H.  $Q \rightarrow W = W;$

Assignment of a based structure occurs in statement H. Structure W at location P (implicit) is assigned to structure W at the location specified by Q (explicit).

- I.  $ADDR(A) \rightarrow B = X;$
- J.  $Y = ADDR(A) \rightarrow B;$

Statements I and J use references to the built-in function ADDR to qualify references to the based element B. In each case, ADDR(A) returns a pointer value that qualifies B. Note that statement J, unlike statement I,

involves conversion of the assigned value from FIXED type to FLOAT type.

- K. Q->Z = P;
- L. Z = Q;

Pointer assignment occurs in statements K and L. P and Q are each pointers, and Z is a based pointer. Pointer Q in statement K explicitly qualifies based pointer Z, which receives the value of pointer P. In statement L, pointer Z receives the value of pointer Q, but Z is a based pointer, which is implicitly qualified by pointer P. Statement L is equivalent to P->Z = Q;

- M. Q->W.Y = Q->W.X;

Statement M shows pointer qualification used with name qualification. Q overrides pointer P, which was declared with based structure W. This statement also involves conversion of the assigned value from FIXED type to FLOAT type.

### 3E2. Pointer Qualification in a Subroutine

Subroutine procedure SWAP in Figure 3E2-1 contains an example of pointer qualification used to interchange the values of pairs of variables. When invoked, the subroutine associates parameter P with a one-dimensional pointer array of even, but arbitrary, length. Each pointer value in P specifies the location of a fixed-point integer of length 8, and SWAP interchanges the numbers at successive pairs of locations. The interchange of individual pairs of values resembles that done in the previous sort routines, except that SWAP does not perform sorting.

The subroutine associates the based variable CARD with each location specified in P. However, since each interchange involves two locations, the subroutine must associate CARD with two different locations at the same time. This double association is achieved by qualifying CARD with separate pointers, S and R, as shown in the following statements:

```
SAVE = S->CARD;
S->CARD = R->CARD;
R->CARD = SAVE;
```

The first location of each pair of locations is assigned to pointer S, the second to pointer R.

The number of locations specified in array parameter P is determined by the built-in functions LBOUND and HBOUND.

The illustrations in Figure 3E2-2 show how SWAP processes four character strings.

```
TEST_SWAP:
F3E2_1:
PROCEDURE OPTIONS (MAIN);
DECLARE
    P(10) POINTER,
    T(10) FIXED (8);
    J = LBOUND (T,1);
    K = HBOUND (T,1);
    PUT
        PAGE LIST ('INPUT ARRAY T IS: ');
#100:
DO
    I = J TO K;
    GET
        EDIT (T(I)) (F(8));
    PUT
        SKIP DATA (T(I));
        P(I) = ADDR(T(I));
END
#100:
CALL SWAP (P);
PUT
    SKIP (3) LIST
    ('RESULT OF PROCEDURE **SWAP**');
#200:
DO
    I = J TO K;
    PUT
        SKIP DATA (T(I));
END
#200:

SWAP:
PROCEDURE (P);
DECLARE
    (P(*), R, S) POINTER,
    (CARD BASED (S), SAVE) FIXED (8);
    L = LBOUND (P,1);
    M = HBOUND (P,1);

LOOP:
DO
    I = L TO M -1 BY 2;
    R = P(I+1);
    S = P(I);
    SAVE = S->CARD;
    S->CARD = R->CARD;
    R->CARD = SAVE;
END
LOOP;
RETURN;

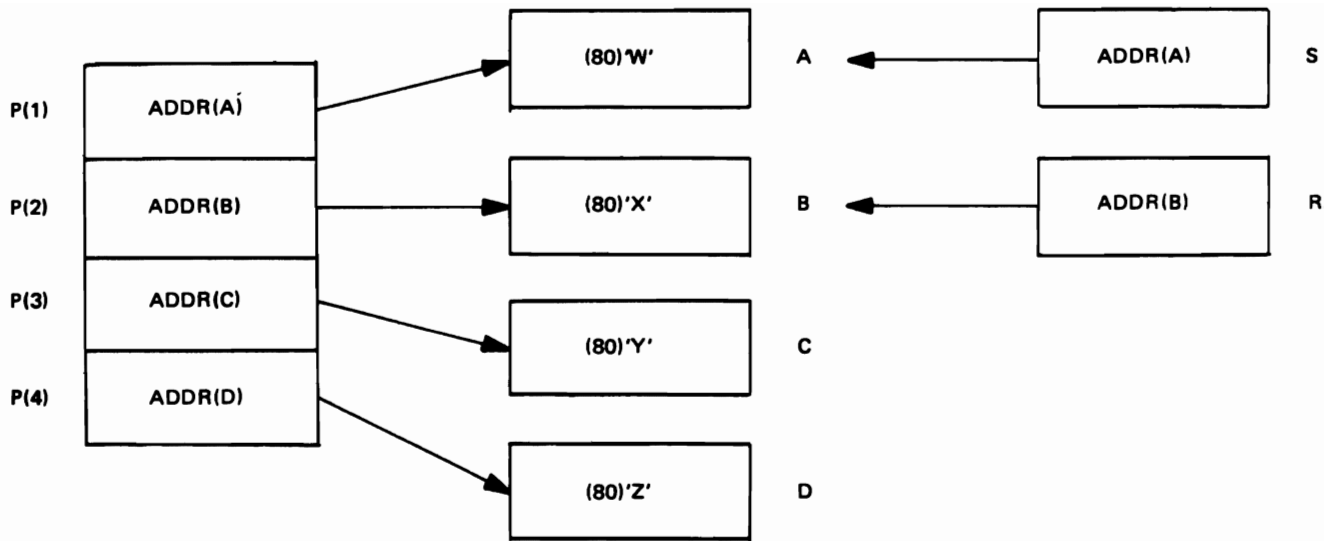
END
SWAP;

END
TEST_SWAP;

INPUT ARRAY T IS:
T(1)= 1234567;
T(2)= 89012345;
T(3)= 67890123;
T(4)= 45678901;
T(5)= 23456789;
T(6)= 1234567;
T(7)= 89012345;
T(8)= 67890123;
T(9)= 45678901;
T(10)= 23456789;

RESULT OF PROCEDURE **SWAP**:
T(1)= 89012345;
T(2)= 1234567;
T(3)= 45678901;
T(4)= 67890123;
T(5)= 1234567;
T(6)= 23456789;
T(7)= 67890123;
T(8)= 89012345;
T(9)= 23456789;
T(10)= 45678901;
```

Figure 3E2-1. Using pointer qualification in a subroutine procedure

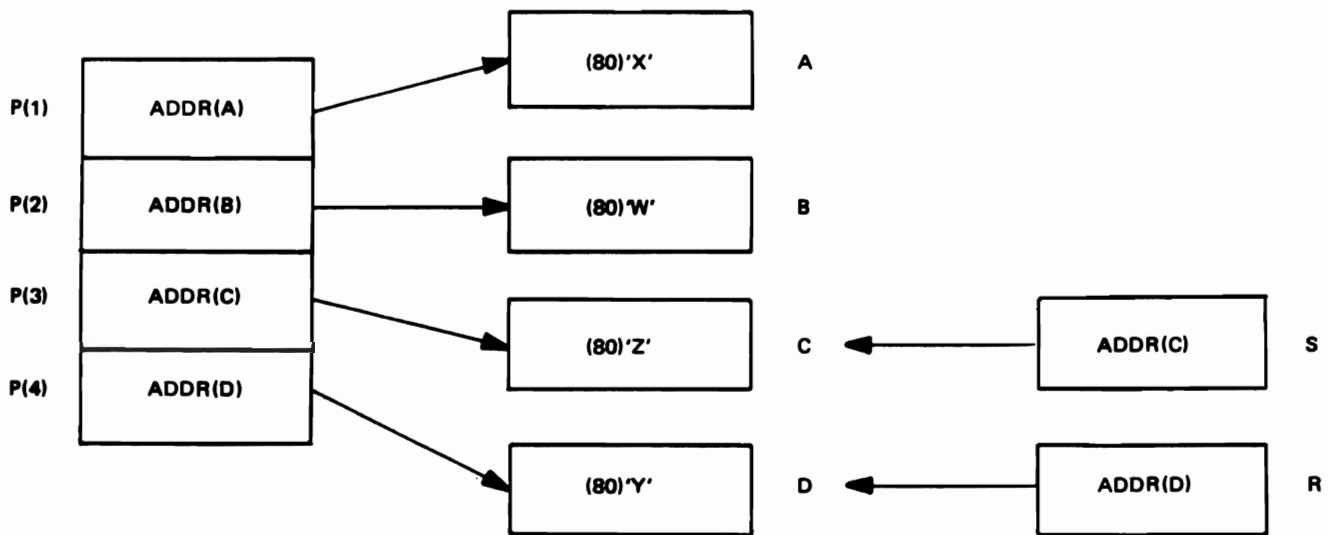


(BEFORE SWAPPING)

```

LOOP: DO = L BOUND (P,1) TO H BOUND (P,1) - 1 BY 2;
      R = P(I + 1); S = P(I);
      SAVE = S -> CARD;
      S -> CARD = R -> CARD;
      R -> CARD = SAVE;
END LOOP;

```



(AFTER SWAPPING)

Figure 3E2-2. How pointer qualification is used by subroutine procedure SWAP in Figure 3E2-2.

### 3E3. Restrictions on Based Variables

The following restrictions apply to based variables:

- A. The EXTERNAL attribute cannot appear in the declaration of a based variable, but a based variable can be qualified by an external pointer variable.
- B. Based variables cannot have the INITIAL attribute, nor can arrays of based labels be initialized by subscripted label prefixes.
- C. Data-directed input and output cannot transmit the value of a based variable.
- D. The BASED attribute cannot be specified for the parameters of subroutines or functions.
- E. The CHECK ON-condition cannot be applied to a based variable.
- F. The VARYING attribute cannot be applied to a based variable.

### 3E4. Contextual Declarations of Pointer Variables

The appearance of an identifier in one of the following contexts serves as a contextual declaration of the identifier as a pointer variable:

- A. in a BASED attribute
- B. in the SET option of READ, LOCATE, and ALLOCATE statements
- C. on the left of a pointer qualification symbol (->).

A contextually declared pointer variable receives the AUTOMATIC storage class and INTERNAL scope by default. If different attributes are desired, they must appear in an explicit declaration of the pointer variable, that is, along with the POINTER attribute. The pointer variable contextually declared with a based variable does not receive the null pointer value as a result of the based declaration.

Only the INITIAL CALL form of the INITIAL attribute is allowed in explicit declarations of pointers.

### 3F. ADVANTAGES OF ABSOLUTE ADDRESSING

The absolute addressing facilities provided by pointer and based variables permit the same reductions in data movement that were obtained earlier in this chapter by using subscript values as the relative addresses of array elements. The following discussions present examples that show how pointer variables and based variables, in addition to reducing data movement among array elements, also permit efficient organization and manipulation of scattered data items.

### 3F1. Reducing Data Movement

Procedure SORT4 showed how an auxiliary array of subscript values may be used to reduce data movement in a sort. The same sorting efficiency can be obtained by using pointer values in place of subscript values. Unlike the subscript values in SORT4 however, pointer values need not be members of an auxiliary array. The same sort technique can use individual pointer elements as well as pointer arrays. Examples of both methods appear in the following discussions.

### 3F2. Sorting with Pointer Variables

Procedure SORT6 in Figure 3F2-1 sorts the same input data processed by the previous sort examples. After assigning input data to the five structures in the array of structures called CARD, SORT6 assigns the absolute addresses of the structures CARD(1), CARD(2), CARD(3), CARD(4), and CARD(5) to the element pointers P1, P2, P3, P4, and P5. Since the based structure IMAGE has the same structuring and attributes as the structures in CARD, IMAGE may be used with appropriate pointer qualification to refer to the components of CARD.

SORT6 compares the KEY subfields in the successive structures of CARD; necessary interchanges are performed not on the structures of CARD but on the address values of the corresponding pointers P1 through P5. Moving pointer values rather than 80-character structures causes less data to be moved and produces a faster sort.

At the completion of the sort, the structures of CARD remain in their original (physical) order, but the values of pointers P1 through P5 are generally changed. Taken in succession, the pointers specify the sorted structures of CARD in ascending order on their KEY fields (see Figures 3F2-2 and 3F2-3). As indicated in Figure 3F2-4, successive structures in the sorted array CARD may be referred to by the expressions P1->IMAGE, P2->IMAGE, P3->IMAGE, P4->IMAGE, and P5->IMAGE.

### 3F3. Sorting with an Array of Pointer Variables

A more compact version of the previous example, SORT6, can be obtained by replacing the five element pointers P1 through P5 with a five-element array of pointers. Procedure SORT7 in Figure 3F3-1 shows how such an array, P, can be used to obtain the same results as SORT6.

Because DO loops can be used to modify references to the pointer elements of array P, SORT7 requires fewer statements than SORT6. Care must be taken, however, when using the based structure IMAGE in references to the components of the array of structures CARD. A reference to IMAGE cannot be qualified by a subscripted pointer variable. Therefore, SORT7 uses the element pointers S and T to qualify usages of IMAGE. Then any element of array P that is to qualify IMAGE must have its

value assigned to pointer S or pointer T, which performs the actual qualification. Except for their use of a pointer array, the diagrams associated with SORT7 (Figures 3F3-2, -3, and -4) are identical to those associated with SORT6.

```

SORT6:
F3F2_1:
PROCEDURE OPTIONS (MAIN);
  DECLARE
    (P1, P2, P3, P4, P5) POINTER,
    1 IMAGE BASED(S),
    2 KEY CHARACTER(3),
    2 DATA CHARACTER(77),
    1 CARD(5),
    2 KEY CHARACTER(3),
    2 DATA CHARACTER(77);
  ON ENDFILE (SYSIN) BEGIN;
  CLOSE FILE (SYSPRINT);
  GO TO OVER;
  END;

INPUT:
  GET
  EDIT(CARD) (A(3), A(77));
  P1 = ADDR(CARD(1));
  P2 = ADDR(CARD(2));
  P3 = ADDR(CARD(3));
  P4 = ADDR(CARD(4));
  P5 = ADDR(CARD(5));

SORT:
  K = 0;
  IF
  THEN
  DU;
  K = 1; S = P1; P1 = P2; P2 = S;
  END;
  IF
  THEN
  DU;
  K = 1; S = P2; P2 = P3; P3 = S;
  END;
  IF
  THEN
  DU;
  K = 1; S = P3; P3 = P4; P4 = S;
  END;
  IF
  THEN
  DU;
  K = 1; S = P4; P4 = P5; P5 = S;
  END;
  IF
  THEN
  K = 1
  GO TO
  SORT;
  PUT
  EDIT (P1->IMAGE,P2->IMAGE,
  P3->IMAGE, P4->IMAGE,
  P5->IMAGE) (SKIP, A(3), A(77));
  PUT
  SKIP;
  GO TO
  INPUT;

OVER:
  END
  SORT6;

```

Figure 3F2-1. Using pointers to sort an array of structures

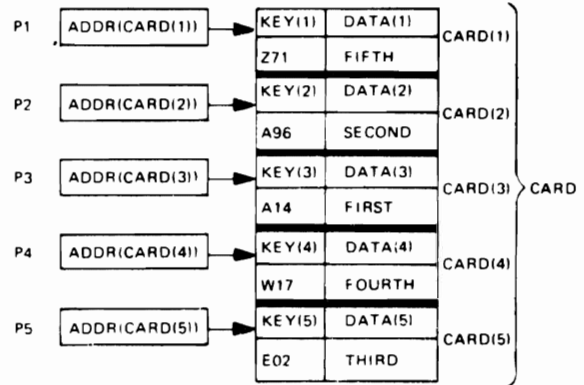


Figure 3F2-2. Pointer variables and array of structures before sort

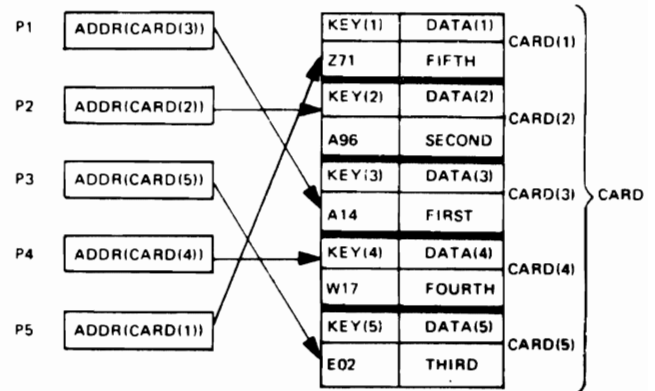


Figure 3F2-3. Pointer variables and array of structures after sort

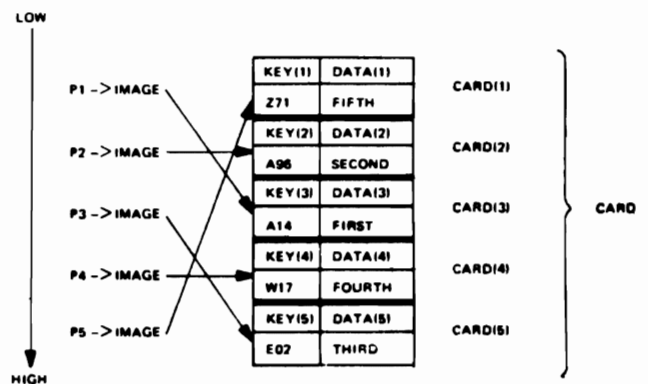


Figure 3F2-4. Sequential references to the sorted elements in an array of structures

```

SORT7:
F3F3_1:
PROCEDURE OPTIONS (MAIN);
  DECLARE
    (P(5), T) POINTER,
    1 IMAGE BASED(S),
    2 KEY CHARACTER(3),
    2 DATA CHARACTER(77),
    1 CARD(5),
    2 KEY CHARACTER(3),
    2 DATA CHARACTER(77);
  ON ENDFILE (SYSIN) BEGIN;
  CLOSE FILE (SYSPRINT);
  GO TO OVER;
  END;

INPUT:
  GET
  EDIT(CARD) (A(3), A(77));
  DO
    I = 1 TO 5; P(I) = ADDR(CARD(I));
  END;
SORT:
  K = 0;
LOOP:
  DO
    I = 1 TO 4;
    S = P(I); T = P(I + 1);
    IF
      (S->IMAGE.KEY) > (T->IMAGE.KEY)
    THEN
      DO;
        K = 1; P(I) = T; P(I + 1) = S;
      END;
    END
  LOOP;
  IF
    K = 1
  THEN
    GO TO
    SORT;
OUTPUT:
  DO
    I = 1 TO 5;
    S = P(I);
    PUT
    EDIT (S->IMAGE) (SKIP, A(3), A(77));
  END
  OUTPUT;
  PUT
  SKIP;
  GO TO
  INPUT;
OVER:
  END
  SORT7;

```

Figure 3F3-1. Using an array of pointers to sort an array of structures

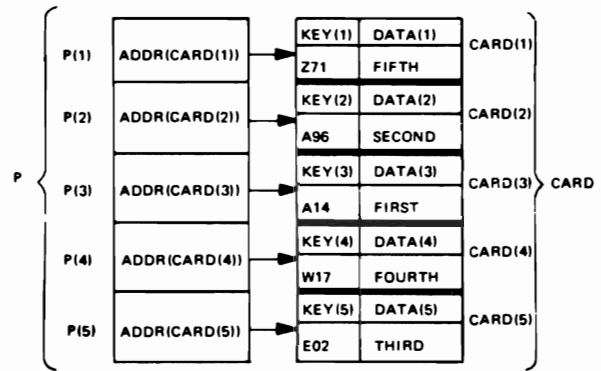


Figure 3F3-2. Array of pointers and array of structures before sort

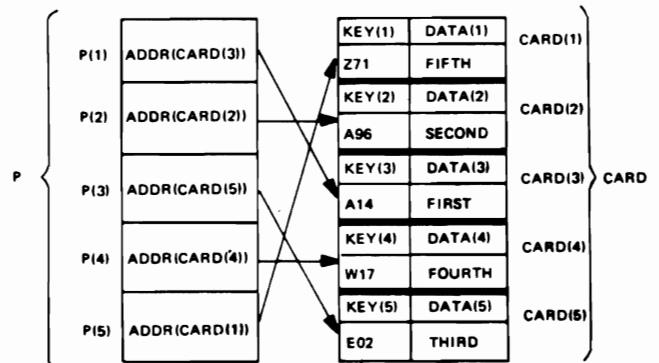


Figure 3F3-3. Array of pointers and array of structures after sort



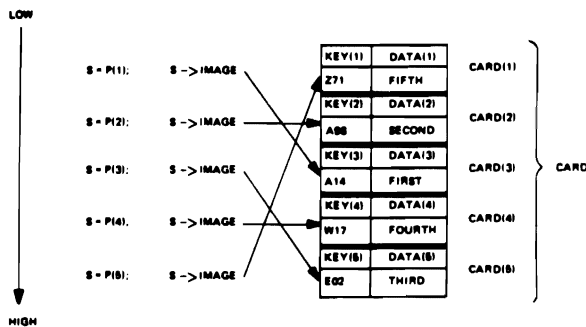


Figure 3F3-4. Sequential references to the sorted elements in an array of structures

### 3G. ASSOCIATING DATA ITEMS IN SCATTERED LOCATIONS

The two preceding procedures, SORT6 and SORT7, require the data items involved in a sort to be placed in an array of structures before sorting actually occurs. There is no need, however, for the items to be placed in such an array. Once the absolute address of each item is assigned to a position in the pointer array, the location of one data item relative to another has no effect upon the efficiency of the sort; scattered (noncontiguous) data items are sorted with the same efficiency as those contained in arrays, because only addresses are moved.

Either an array of pointers or a pointer variable attached to each item can be used to associate scattered data items with each other. Examples of both methods appear in the following discussions.

#### 3G1. Associating Data Items through an Array of Pointer Variables

Procedure SORT8 in Figure 3G1-1 uses an array of pointer variables to sort sets of structures that are not contained in an array. The procedure assigns the absolute addresses of the individual structures to the element positions in the array of pointers. At this stage, SORT8 becomes identical to SORT7 and produces the same results. This equivalence is possible because the relative locations of the structures (whether members of an array or not) have no effect upon the actual sorting operations.

The major difference between the two procedures is that SORT8 requires more statements than SORT7. These additional statements result from having to declare each

```

SORT8:
F3G1_1:
PROCEDURE OPTIONS (MAIN);
DECLARE
(P(5), T) POINTER,
1 IMAGE BASED(S),
2 KEY CHARACTER(3),
2 DATA CHARACTER(77),
1 CARD1,
2 KEY CHARACTER(3),
2 DATA CHARACTER(77),
1 CARD2,
2 KEY CHARACTER(3),
2 DATA CHARACTER(77),
1 CARD3,
2 KEY CHARACTER(3),
2 DATA CHARACTER(77),
1 CARD4,
2 KEY CHARACTER(3),
2 DATA CHARACTER(77),
1 CARD5,
2 KEY CHARACTER(3),
2 DATA CHARACTER(77);
ON ENDFILE (SYSIN) BEGIN;
CLOSE FILE (SYSPRINT);
GO TO OVER;
END:

INPUT:
GET
EDIT(CARD1, CARD2, CARD3, CARD4,
CARD5)
(A(3), A(77));
P(1) = ADDR(CARD1);
P(2) = ADDR(CARD2);
P(3) = ADDR(CARD3);
P(4) = ADDR(CARD4);
P(5) = ADDR(CARD5);

SORT:
K = 0;
LOOP:
DO
I = 1 TO 4;
S = P(I); T = P(I + 1);
IF
(S->IMAGE.KEY) > (T->IMAGE.KEY)
THEN
DO;
K = 1; P(I) = T; P(I + 1) = S;
END;
END
LOOP;
IF
K = 1
THEN
GO TO
SORT;

OUTPUT:
DO
I = 1 TO 5;
S = P(I);
PUT
EDIT (S->IMAGE) (A(3), A(77));
PUT
SKIP;
END
OUTPUT;
PUT
SKIP;
GO TO
INPUT;

OVER:
END
SORT8;

```

Figure 3G1-1. Using an array of pointers to sort structures not in an array

structure separately and also from having to use separate assignment statements to assign the absolute addresses of the structures to the array of pointers. The increase in program size is usually offset by a decrease in execution time, since data items are referred to more directly when not in an array.

Except for their use of scattered structures, the diagrams associated with SORT8 (Figures 3G1-2, -3, and -4) are identical to the diagrams associated with SORT7.

It is also possible to create a subroutine procedure from the common portions of SORT8 and SORT7. Such a procedure, SORT9, appears in Figure 3G1-5. This routine uses a pointer array P as a parameter to sort structures that are not necessarily members of an array.

Parameter P is a one-dimensional array of pointers that can be associated with argument arrays of arbitrary size (as indicated by the asterisk in the dimension attribute of P). The actual size of the array passed to P during an invocation of SORT9 is ascertained by the built-in function DIM. The contents of the argument array are the absolute addresses of the structures to be sorted. Each structure must have the same attributes as the based variable IMAGE.

Upon completion of SORT9, successive addresses in the argument array specify the order (from low to high) of the sorted structures.

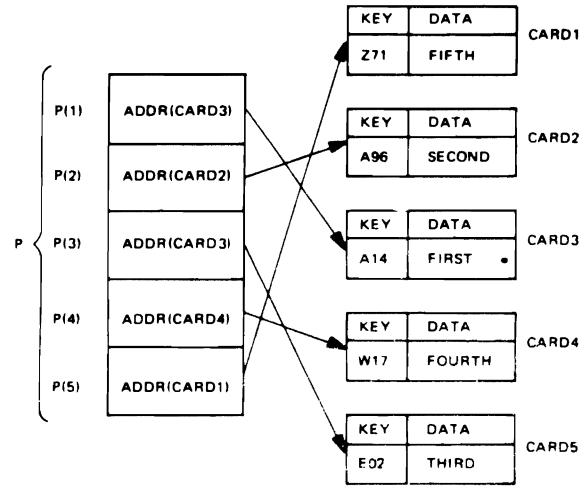


Figure 3G1-3. Array of pointers and individual structures after sort

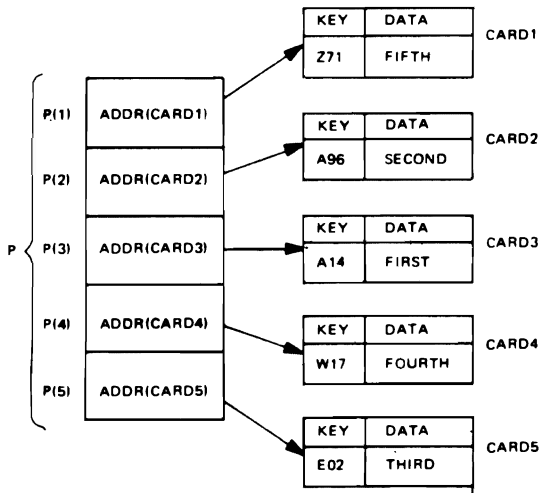


Figure 3G1-2. Array of pointers and individual structures before sort

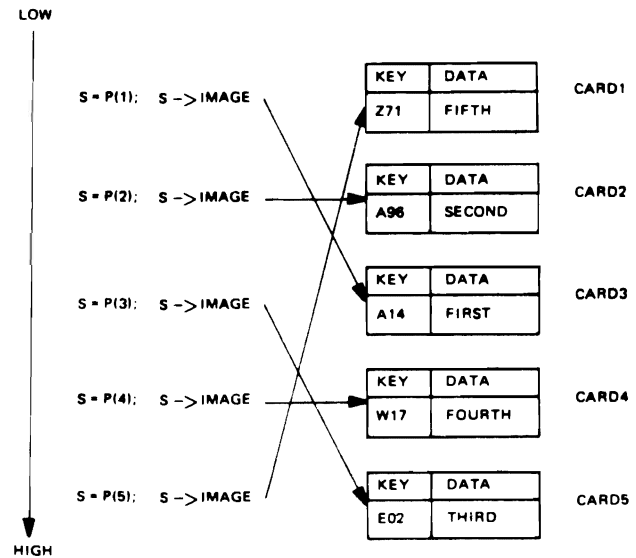


Figure 3G1-4. Sequential references to the sorted individual structures

```

SORT9:
PROCEDURE (P);
DECLARE
  P(*) POINTER,
  T POINTER,
  1 IMAGE BASED (S),
  2 KEY CHARACTER(3),
  2 DATA CHARACTER(77);
SORT:
  TFST = 0;
LOOP:
  DO
    I = 1 TO (DIM(P,1)-1);
    S = P (I); T = P (I + 1);
    IF
      (S->IMAGE.KEY) > (T->IMAGE.KEY)
    THEN
      DO;
        TEST = 1; P (I) = T; P (I + 1) = S;
      ;
    ;
  ;
END;
END
LOOP;
IF
  TEST = 1
  THEN
  GO TO
  SORT;
RETURN;
END
SORT9;

```

Figure 3G1-5. A subroutine procedure that sorts structures not in an array

### 3G2. Linking Data Items through Pointer Variables

An alternative method for organizing scattered data items into a collective unit appears in procedure SORT10 of Figure 3G2-1, which links the scattered items through a pointer variable attached to each item. This technique is a generalization of the method employed earlier by SORT5, in which subscript values were used to link the members of an array. Procedure SORT10, however, uses pointer values to link scattered items, which may or may not be members of an array.

SORT10 uses five individual but identical structures: CARD1, CARD2, CARD3, CARD4, and CARD5. Each structure contains a three-character element, KEY; a 77-character element, DATA; and a pointer element, L.

The program begins by assigning input data to each structure. It then assigns the absolute address of each structure to pointer L in the previous structure. The address of CARD1 is assigned to a pointer called HEAD, and pointer L in CARD5 receives a null address value (see Figure 3G2-2).

At this point in the program, the structures form a chain. Pointer HEAD specifies the location of the first structure in the chain, and pointer L of each structure contains the location of the next structure. A null address for a pointer L indicates the end of the chain.

As SORT10 compares the KEY fields of successive pairs of structures, required interchanges are performed on the address values of the HEAD and L pointers, rather

than on the data values of the structures themselves; as a result, less data is moved during the sort.

References to the chained structures use the based variable IMAGE qualified by an appropriate pointer. Since several structures in the chain may be referred to at the same time during an interchange, intermediate pointer variables, such as S, T, U, and V, are used to qualify simultaneous usages of IMAGE.

At the completion of the sort, as illustrated by Figure 3G2-3, the structures remain in their original (physical) positions, but the HEAD and L pointers link the structures in ascending order on their KEY elements.

The steps in Figure 3G2-4 show how SORT10 uses the following statements to transpose the order of the first two structures, CARD1 and CARD2:

```

S = HEAD->IMAGE.L;
T = S->IMAGE.L;
S->IMAGE.L = HEAD;
HEAD->IMAGE.L = T;
HEAD = S;

```

Observe that the interchange of the two structures CARD1 and CARD2 requires modification of three pointer variables: HEAD, CARD1.L, and CARD2.L.

Figure 3G2-4 (step 4) shows how the structures are linked after the first interchange has been completed. At that point in the sort the structures are linked in the following order:

CARD2, CARD1, CARD3, CARD4, CARD5

This is the logical order in which the structures are linked; their original physical order remains unchanged.

When an interchange does not involve the logically first structure in the chained sequence, pointer HEAD is not modified; and, as the steps in Figure 3G2-5 show, SORT10 uses a different set of instructions for the interchange.

```

U = HEAD;
S = U->IMAGE.L;
T = S->IMAGE.L;
V = T->IMAGE.L;
T->IMAGE.L = U->IMAGE.L;
U->IMAGE.L = S->IMAGE.L;
S->IMAGE.L = V;

```

These instructions are executed under control of a DO-loop, each cycle of which advances the pointers S, T, U, and V to successive structures. Assignment of a null value to T terminates the loop.

```

SORT10:
F3G2_1:
PROCEDURE OPTIONS (MAIN);
DECLARE
  (HEAD, S, T, U, V) POINTER,
  1 IMAGE BASED(S),
  2 KEY CHARACTER(3),
  2 DATA CHARACTER(77),
  2 L POINTER,
  1 CARD1,
  2 KEY CHARACTER(3),
  2 DATA CHARACTER(77),
  2 L POINTER,
  1 CARD2,
  2 KEY CHARACTER(3),
  2 DATA CHARACTER(77),
  2 L POINTER,
  1 CARD3,
  2 KEY CHARACTER(3),
  2 DATA CHARACTER(77),
  2 L POINTER,
  1 CARD4,
  2 KEY CHARACTER(3),
  2 DATA CHARACTER(77),
  2 L POINTER,
  1 CARD5,
  2 KEY CHARACTER(3),
  2 DATA CHARACTER(77),
  2 L POINTER;
ON ENDFILE (SYSIN)
BEGIN;
  CLOSE FILE (SYSPRINT);
  GO TO
  PROC_END;
END;
INPUT:
  GET
  EDIT(CARD1.KEY, CARD1.DATA,
  CARD2.KEY, CARD2.DATA, CARD3.KEY,
  CARD3.DATA, CARD4.KEY, CARD4.DATA,
  CARD5.KEY, CARD5.DATA) (A(3),
  A(77));
  HEAD = ADDR(CARD1);
  CARD1.L = ADDR(CARD2);
  CARD2.L = ADDR(CARD3);
  CARD3.L = ADDR(CARD4);
  CARD4.L = ADDR(CARD5);
  CARD5.L = NULL;

```

```

SORT:
  K = 0; S = HEAD->IMAGE.L;
  IF
  (HEAD->IMAGE.KEY) > (S->IMAGE.KEY)
  THEN
  DO;
    K = 1; T = S->IMAGE.L;
    S->IMAGE.L = HEAD;
    HEAD->IMAGE.L = T; HEAD = S;
  END;
  U = HEAD; S = U->IMAGE.L;
  T = S->IMAGE.L;
  DO
  WHILE (T->NULL);
  IF
  (S->IMAGE.KEY) > (T->IMAGE.KEY)
  THEN
  DO;
    K = 1; V = T->IMAGE.L;
    T->IMAGE.L = U->IMAGE.L;
    U->IMAGE.L = S->IMAGE.L;
    S->IMAGE.L = V;
  END;
  U = U->IMAGE.L; S = U->IMAGE.L;
  T = S->IMAGE.L;
END;
IF
  K = 1
  THEN
  GO TO
  SORT; S = HEAD;
OUTPUT:
  DO
  WHILE (S->NULL);
  PUT
  EDIT(S->IMAGE.KEY, S->IMAGE.DATA)
  (A(3), A(77));
  PUT
  SKIP;
  S = S->IMAGE.L;
END
  OUTPUT;
  PUT
  SKIP;
  GO TO
  INPUT;
PROC_END:
END
  SORT10;

```

Figure 3G2-1. Sorting structures that are linked by pointers

The steps in Figure 3G2-5 illustrate the effect of the second interchange, which involves CARD1 and CARD3. And Figure 3G2-5 (step 4) shows the structures are linked in the following order after this second interchange has been completed:

CARD2, CARD3, CARD1, CARD4, CARD5

A complete sort of the structures produces the following order:

CARD3, CARD2, CARD5, CARD4, CARD1

Figure 3G2-6 shows how the sorted structures may be referred to in succession. Observe how a DO statement

can be used to move through the chain to a particular structure. Also note the similarities between qualified pointers in Figure 3G2-6 and subscripted subscripts in Figure 3C2-6 (associated with SORT5).

Structures linked by pointer variables may also be sorted with a subroutine, as demonstrated by procedure SORT11 in Figure 3G2-7. This routine uses element-pointer variable HEAD as a parameter. The associated element-pointer argument in an invocation of SORT11 specifies the first structure in the chain of structures being sorted.

The structures are linked as in SORT10, and a null link indicates the end of the chain. The chain must contain at least two structures; otherwise, the number of structures is arbitrary.

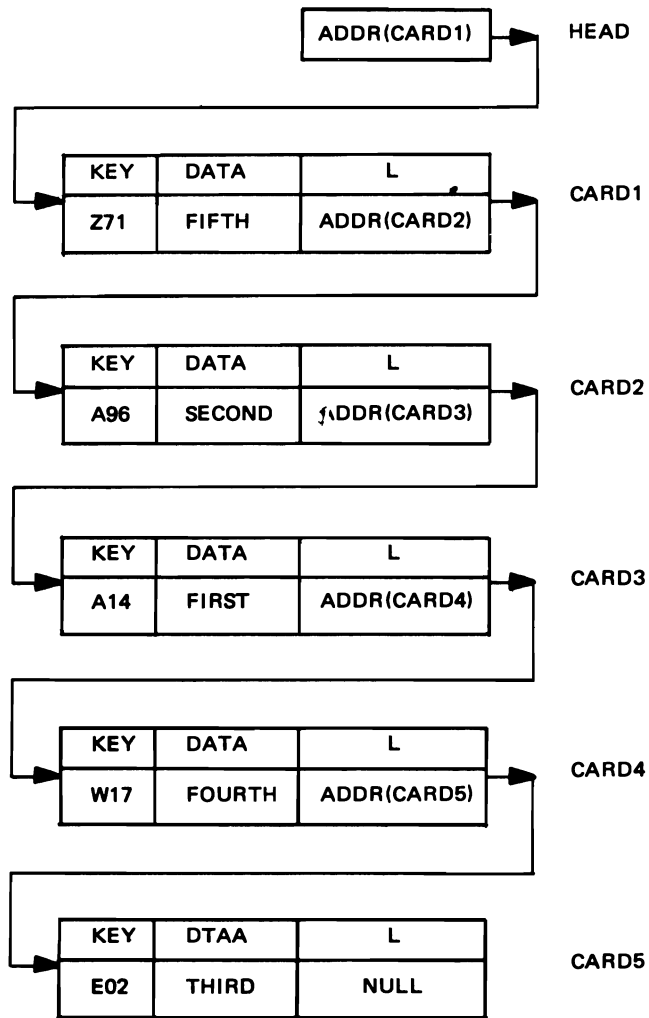


Figure 3G2-2. Linked structures before sort

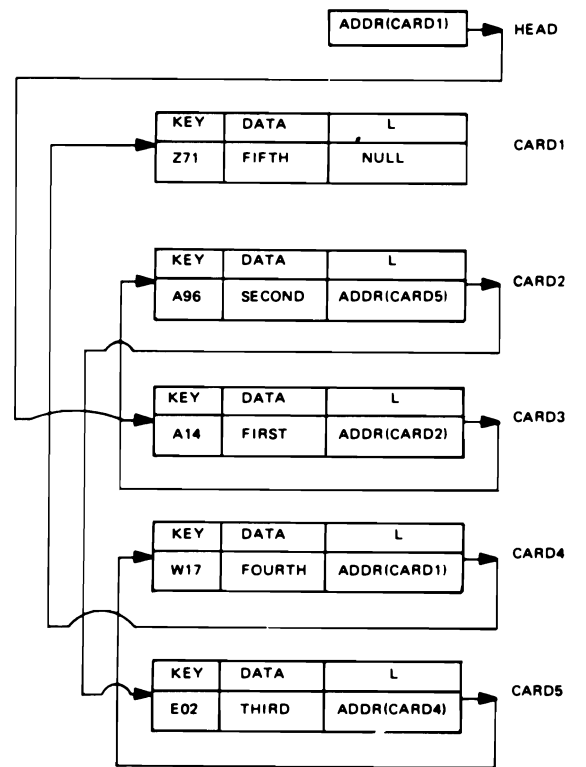
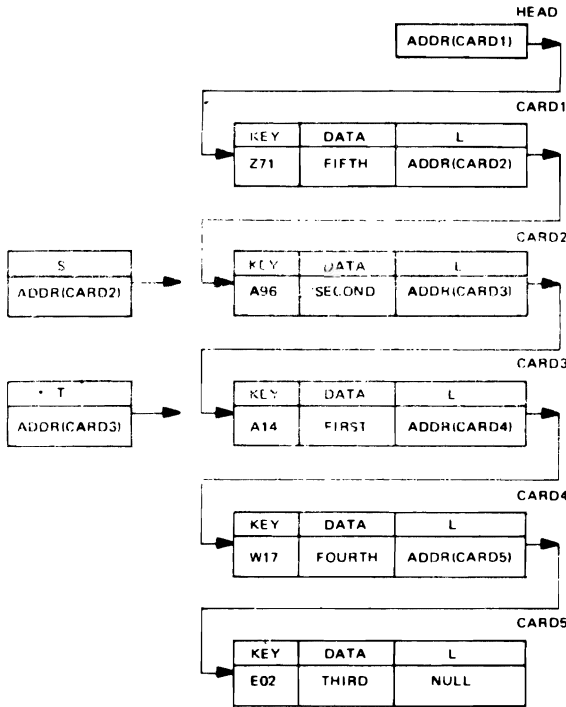
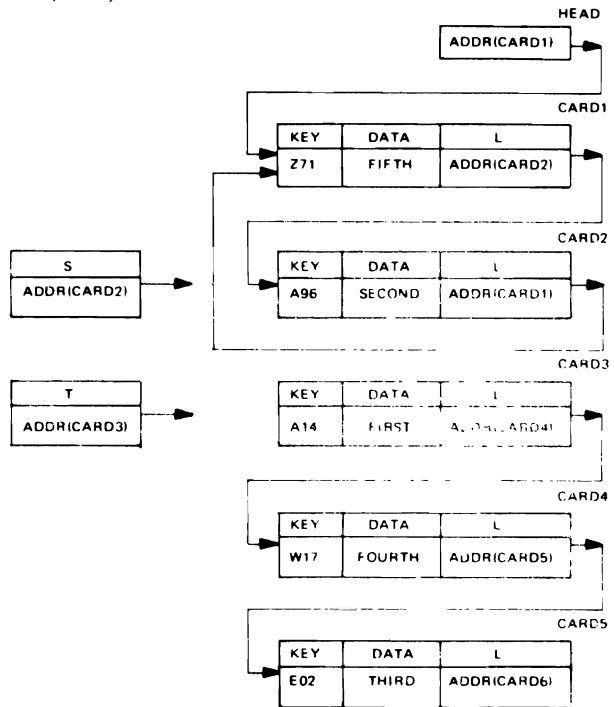


Figure 3G2-3. Linked structures after sort

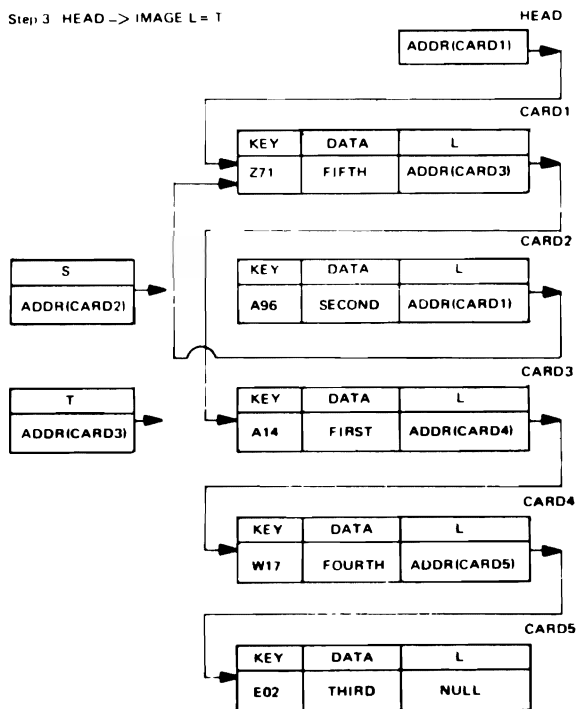
Step 1 S - HEAD -> IMAGE L; T - S -> IMAGE L;



Step 2 S -> IMAGE L = HEAD;



Step 3 HEAD -> IMAGE L = T



Step 4 HEAD = S.

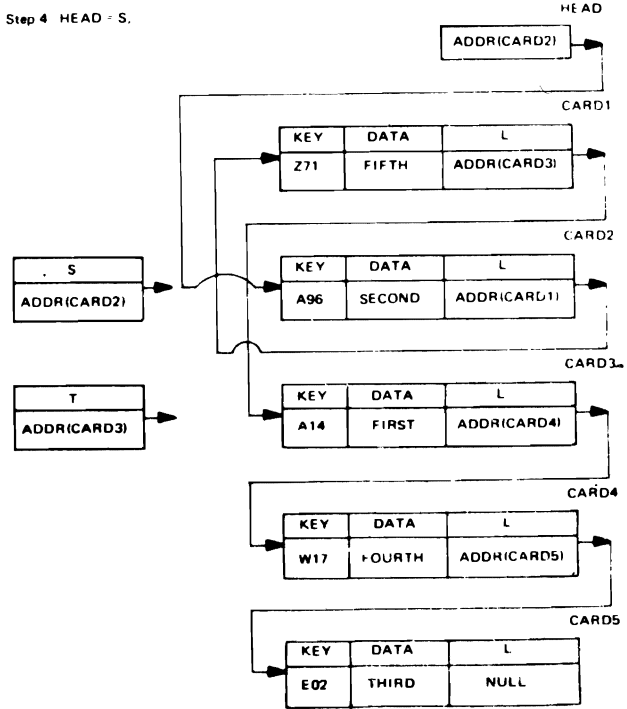
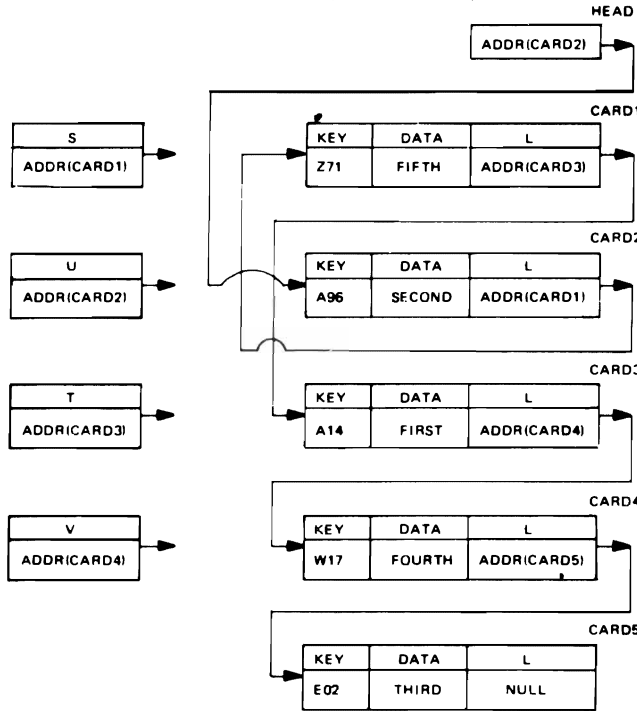
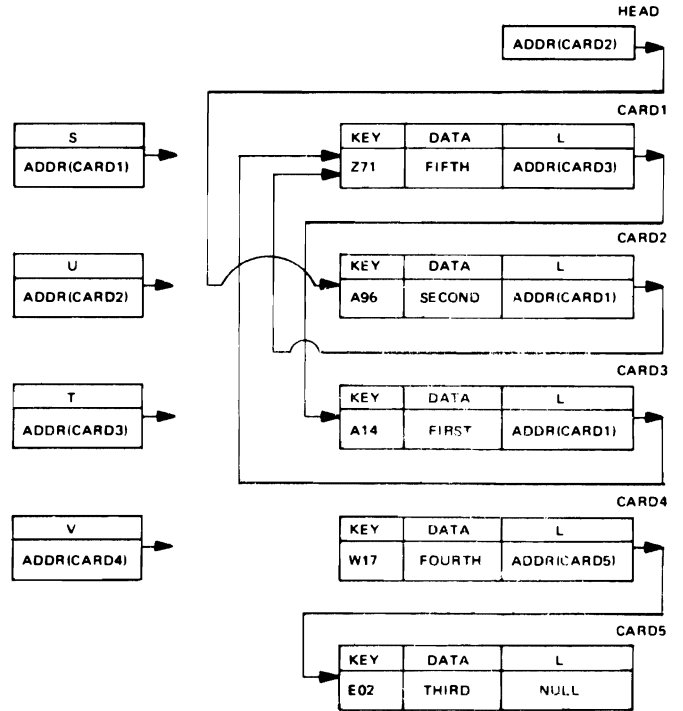


Figure 3G2-4. Performing the first interchange, which involves CARD1 and CARD2

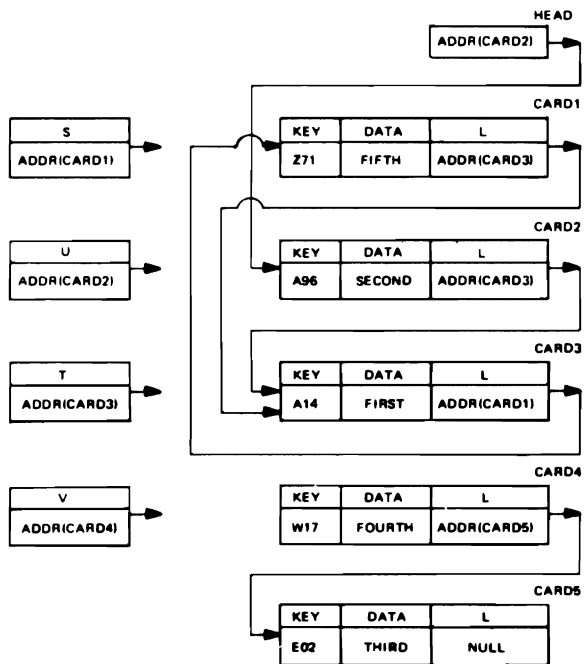
Step 1 U HEAD, S → U → IMAGE L; T → S → IMAGE L; V → T → IMAGE L.



Step 2 T → IMAGE L = U → IMAGE L;



Step 3 U → IMAGE L = S → IMAGE L;



Step 4 S → IMAGE L = V

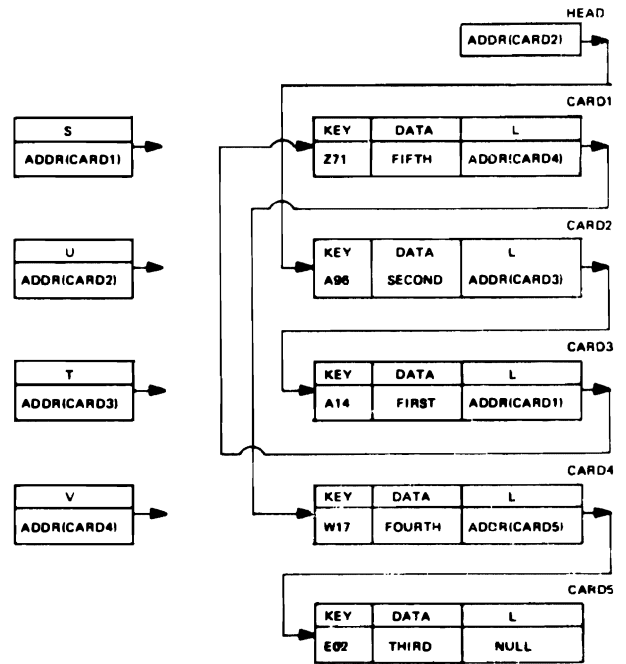


Figure 3G2-5. Performing the Second Interchange, which involves CARD 1 and CARD 3

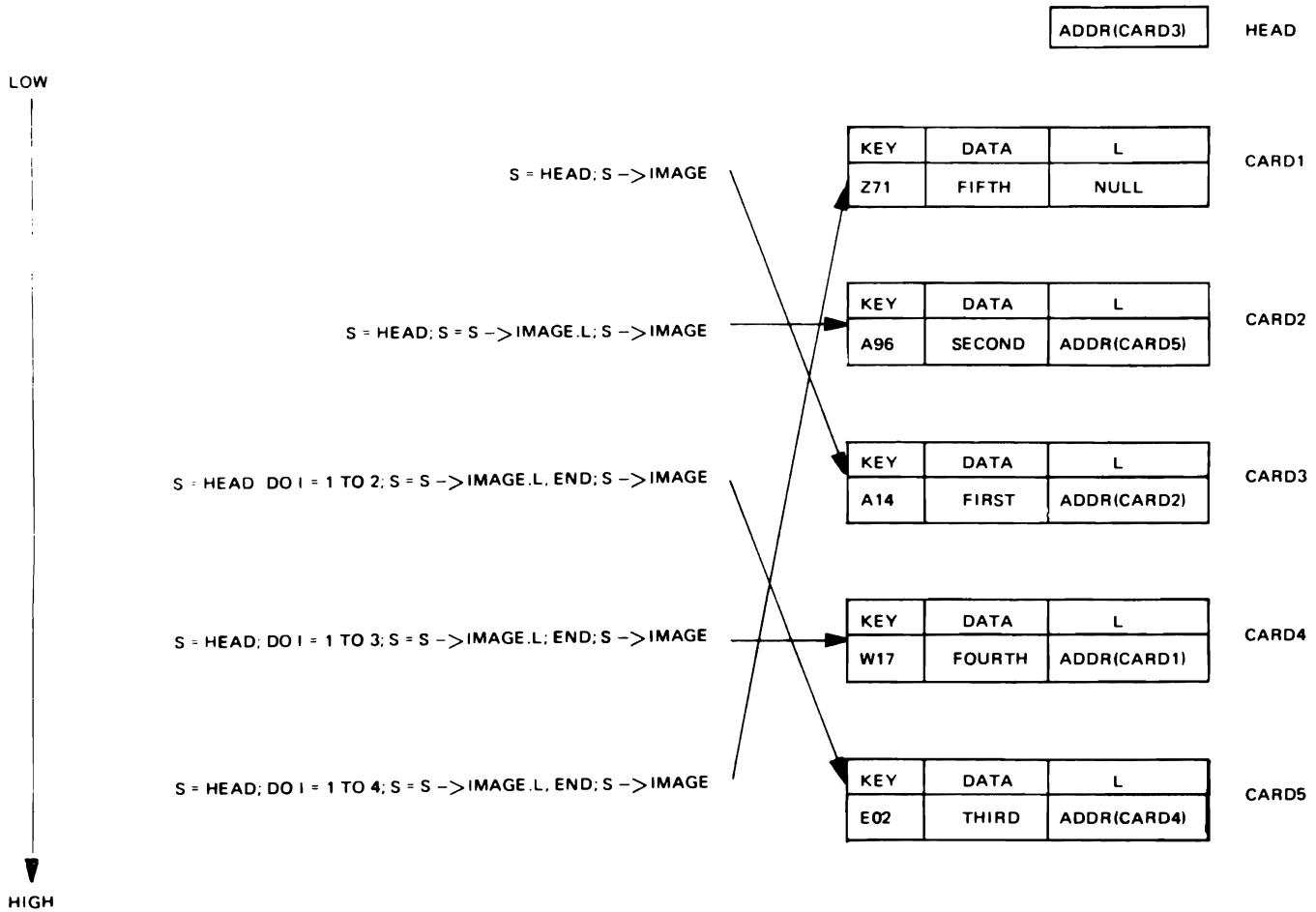


Figure 3G2-6. Sequential references to the sorted sequence of linked structures

```

SORT11:
F3G2_7:
PROCEDURE (HEAD);
  DECLARE
    (HEAD, S, T, U, V) POINTER,
    1 IMAGE BASED(S),
    2 (KEY CHARACTER (3),
    DATA CHARACTER(77),
    L POINTER);
SORT:
  K = 0; S = HEAD->IMAGE.L;
  IF
    (HEAD->IMAGE.KEY) > (S->IMAGE.KEY)
  THEN
  DO;
    K = 1; T = S->IMAGE.L;
    S->IMAGE.L = HEAD;
    HEAD->IMAGE.L = T; HEAD = S;
  END;
  U = HEAD; S = U->IMAGE.L;
  T = S->IMAGE.L;
  DO
    WHILE (T->IMAGE.L);
  IF
    (S->IMAGE.KEY) > (T->IMAGE.KEY)
  THEN
  DO;
    K = 1; V = T->IMAGE.L;
    T->IMAGE.L = U->IMAGE.L;
    U->IMAGE.L = S->IMAGE.L;
    S->IMAGE.L = V;
  END;
  U = U->IMAGE.L; S = U->IMAGE.L;
  T = S->IMAGE.L;
  END;
  IF
    K = 1
  THEN
  GO TO
  SORT;
  RETURN;
  END
  SORT11;

```

Figure 3G2-7. A subroutine procedure that sorts structures linked by pointer variables



### 3H. REVIEW OF TECHNIQUES FOR ADDRESSING DATA ITEMS

This chapter has shown how the movement of data may be reduced by manipulating the address of a data item rather than the item itself. The discussion has been concerned mainly with two types of addresses: relative addresses (subscript values) and absolute addresses (pointer values). Since programmers are generally more familiar with subscripts than with pointers, addressing techniques were developed first in terms of subscripts and then extended to include analogous usage of pointers.

The first technique for reducing the movement of data used an auxiliary array of address values, which permitted a set of cards to be sorted without actually moving the cards. An illustration of the subscript version of this technique appears in Figure 3H-1. An array of structures contains the cards being sorted, and an auxiliary array of subscript values specifies the order of the cards. Rearranging the subscript values rather than the cards reduces the movement of data. An illustration of the same technique in terms of pointer values appears in Figure 3H-2. Since pointer values are absolute addresses, there is no need for the cards being sorted to be contained in an array; the cards can be scattered throughout storage. Both versions (subscript and pointer) of this technique were implemented in procedures SORT4 and SORT8.

Once it was seen how an auxiliary array of address values could be used to specify the order of a set of cards, a further generalization became possible. It was not necessary to store the address values in an auxiliary array; instead, they could be attached to the cards being sorted. These attached addresses then permitted the cards to be linked in a sequential manner without actually moving the cards. An illustration of this technique in terms of sub-

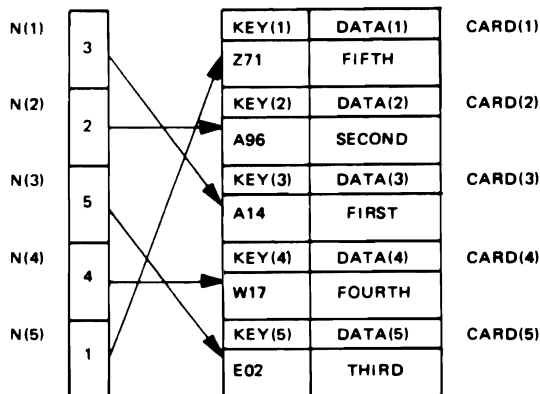


Figure 3H-1. Using an auxiliary array of subscript values to sort an array of structures

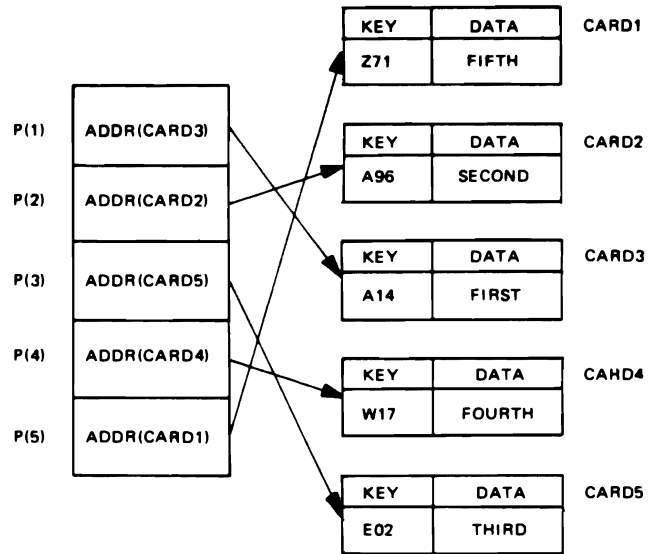


Figure 3H-2. Using an auxiliary array of pointer values to sort scattered structures

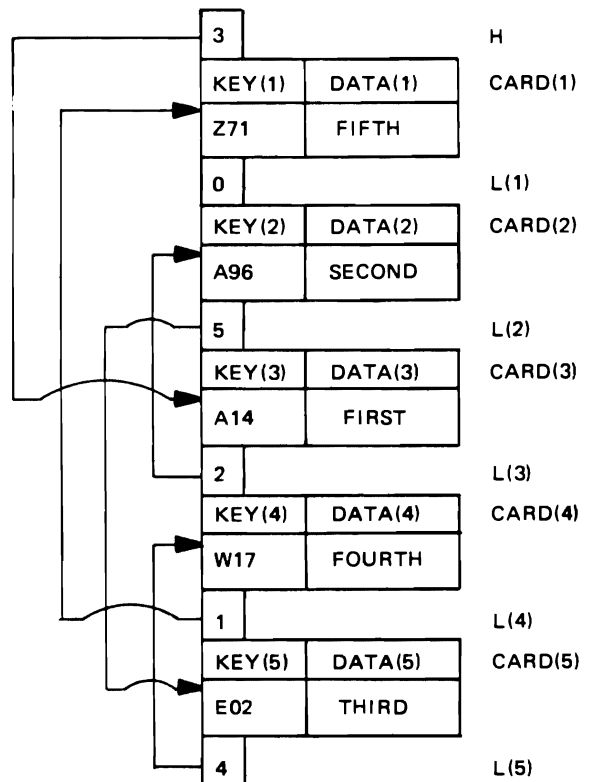


Figure 3H-3. Using subscript values to link an array of structures in sort order

script values appears in Figure 3H-3. Each card (along with an attached subscript value) is stored in an array of structures. The attached subscript specifies the relative position of the next sorted card within the array. The last of the sorted cards has an attached subscript value of zero, and the first card in the sorted sequence is specified by the subscript value of variable H. This linking technique contains the essential features of a list organization, with one exception. The cards cannot be scattered throughout storage; they must be contained in an array. However, the use of pointer values in place of subscript values removes this restriction, as shown in Figure 3H-4. It is this type of list organization that is discussed in the remaining chapters of this manual. The subscript version and pointer version of this linking technique were implemented in procedures SORT5 and SORT10.

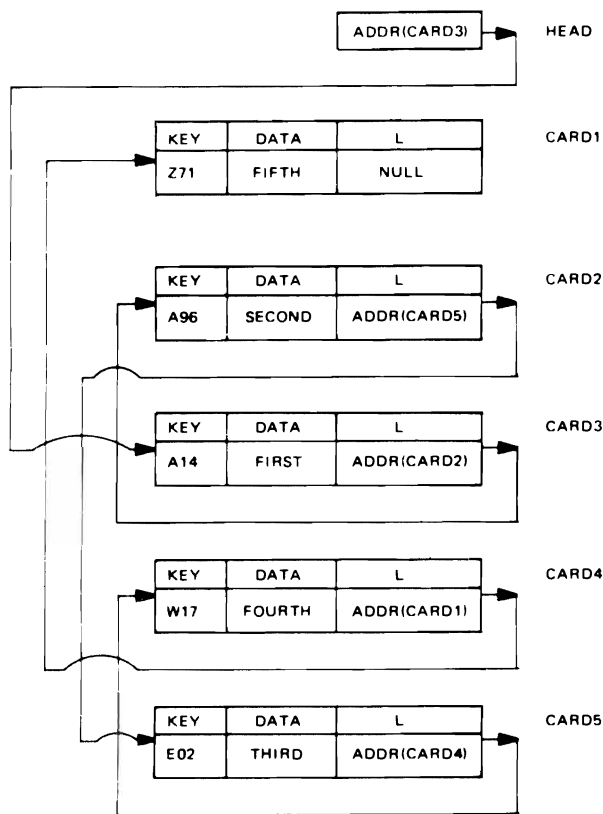


Figure 3H-4. Using pointer values to link scattered structures in sort order

### 3I. SUMMARY OF CHAPTER 3

- A. PL/I provides three major types of data addresses: symbolic, relative, and absolute.
- B. Symbolic addresses permit mnemonic names to be used in place of numeric addresses.
- C. Relative addresses are subscript expressions that specify the relative locations of items within arrays.
- D. Absolute addresses specify the actual locations of items within storage.
- E. The built-in function ADDR obtains the absolute address of a data item. The value of the built-in function NULL identifies no generation of storage; it is used to test for unallocated storage, and to indicate the end of a list.
- F. The attribute POINTER declares an identifier to be a pointer variable, which has an absolute address for its value. PL/I also permits arrays of pointers to be declared.
- G. The attribute BASED (element-pointer-variable) declares an identifier to be a based variable, which associates descriptive information with a pointer variable. The descriptive information determines the storage characteristics at the location given by the pointer variable. A based variable may be an element, an array, a structure, or an array of structures.
- H. The pointer qualification symbol ( $\rightarrow$ ) allows different pointer variables to be associated with the same based variable. The reference on the left of a pointer qualification symbol must be an unsubscripted, nonbased element-pointer variable, or a usage of the built-in function ADDR. The reference on the right of the pointer qualification symbol must be a based variable.
- I. When a reference to a based variable is not explicitly qualified by a pointer variable, the reference is assumed to be qualified by the pointer variable given in the BASED attribute associated with the based variable.
- J. An identifier is contextually declared to be a pointer variable by its appearance in one of the following contexts:
  - 1) in a BASED attribute
  - 2) in the SET option of READ, LOCATE, and ALLOCATE statements
  - 3) on the left of a pointer qualification symbol ( $\rightarrow$ ).
- K. A pointer variable receives a value through either the assignment statement or the SET option of READ, LOCATE, and ALLOCATE statements.
- L. Only the comparison operators equal (=) and not equal ( $\neq$ ) may use pointer variables as operands.
- M. Pointer and based variables permit reductions in data movement and allow associations to be formed among scattered data items.

## Chapter 4. Lists and the Dynamic Allocation of Storage

The preceding chapter showed how pointer variables and based variables may be used to reduce data duplication and data movement when scattered data items are organized into collective units within internal storage. A restriction underlying the addressing techniques of the preceding chapter, however, is that all storage locations associated with based variables must be declared and allocated independently of the based variables. As a result, the number of different storage locations that are to be associated with based variables must be known at the time a program is written. This requirement severely limits the applications of based variables to problems that have well defined storage needs and, for all practical purposes, eliminates those applications that determine and allocate needed storage during the course of program execution.

This chapter presents methods for eliminating these restrictions, so that storage may be allocated dynamically for based variables as the need arises during program execution. The discussion also shows how the data-addressing techniques of the previous chapter and the storage-allocation techniques of this chapter may be combined to form the basic facilities for list organization and manipulation.

### 4A. BASED STORAGE

When storage is allocated, an association occurs between a variable and a specified amount of storage. This association can occur before program execution begins and can remain in effect until the program is terminated; in this case, the allocation is said to be static. It is also possible for storage to be allocated after program execution begins and for the storage to be released (freed for possible reuse) before the program has finished; this type of allocation is called dynamic.

The storage-class attributes determine which type of allocation applies to a given variable. **STATIC** specifies static storage allocation. **AUTOMATIC**, **CONTROLLED**, and **BASED** each specify a type of dynamic storage allocation:

**A. Automatic storage** allocation for a variable occurs automatically when program control enters the block in which the variable is declared. Termination of a block automatically frees the automatic storage internal to the block. Freeing the automatic storage asso-

ciated with a variable causes the value of the variable to be lost.

- B. Controlled storage** allocation permits the programmer to have direct control over allocation by means of the **ALLOCATE** and **FREE** statements.
- C. Based storage** allocation also provides direct control over allocation by means of the **ALLOCATE** and **FREE** statements but, unlike controlled allocation, permits concurrent references to multiple storage areas allocated for the same based variable.

#### 4A1. Allocating Based Storage

Allocation of storage for a based variable is performed by the **ALLOCATE** statement, which has the following basic format:

```
ALLOCATE based-variable;
```

When executed, this statement allocates storage for the based variable and assigns the absolute address of the allocated storage to the pointer variable specified in the **BASED** attribute of the based variable. The attributes associated with the based variable determine the amount of storage that is allocated.

**EXAMPLE:**

```
DECLARE TABLE(5) BASED(P) FIXED DECIMAL(3);  
.  
.  
.  
ALLOCATE TABLE;
```

These statements declare **TABLE** to be a based array and allocate storage for an array of five elements, each of which is a three-digit fixed-point decimal integer. The location of the allocated storage is automatically assigned to pointer **P**.

Reallocation of storage for a based variable does not free previously allocated storage for the variable. Instead, both the old storage and the new storage are available, provided the old value of the associated pointer is saved before it is replaced by the location of the new storage. Several allocations of storage for the same based variable may be distinguished by appropriate pointer qualification.

**EXAMPLE:**

```
DECLARE T POINTER,  
        SWITCH BASED (P) BIT(2);  
ALLOCATE SWITCH;  
T = P;  
ALLOCATE SWITCH;  
T->SWITCH = '11'B;  
SWITCH = '10'B;
```

In this example, SWITCH is a based variable that represents a two-position bit string. T and P are pointer variables. After each allocation of storage for SWITCH, pointer P contains the address of the allocated storage. Pointer T is used to save the address of the first allocation before the second allocation is executed. The statement T->SWITCH = '11'B; assigns the bit-string constant '11'B to the first storage location allocated for SWITCH, and the statement SWITCH = '10'B; assigns the constant '10'B to the second storage location allocated for SWITCH.

Note how reallocation of based variables differs from that of controlled variables. Although reallocation of a controlled variable does not destroy its previously allocated storage, only the most recent allocation is available at any given time. Effectively, successive allocations of storage for a controlled variable are stacked. Execution of the FREE statement for a controlled variable releases the current storage allocated for the variable and makes the previous allocation available. Repeated execution of the FREE statement for a controlled variable will eventually release all storage that has been stacked for the variable. The FREE statement has no effect on a controlled variable that has no storage currently allocated for it.

**4A2. The SET Option in an ALLOCATE Statement**

As illustrated in the preceding example, each allocation of storage for a based variable assigns the address of the new allocation to the pointer variable specified in the BASED attribute associated with the based variable. When two or more allocations of storage are performed concurrently for the same based variable, the addresses of previous allocations must be saved in separate pointer variables; otherwise, the addresses will be lost. So far, the address of a previous allocation has been saved by means of the assignment statement. PL/I, however, provides the SET option in an ALLOCATE statement as an alternative method for assigning the address of an allocation to a pointer variable.

An ALLOCATE statement with a SET option has the following form:

```
ALLOCATE based-variable SET (element-pointer-  
variable);
```

This statement allocates storage for the based variable and assigns the address of the allocated storage to the pointer variable specified in the SET option. The pointer variable must represent a single pointer value; it cannot be the name of an array of pointers or a structure of pointers.

An ALLOCATE statement without a SET option is treated as having an implicit SET option that applies to the pointer variable in the BASED attribute of the allocated variable. An explicit SET option allows the programmer to specify a pointer variable different from the one given in the BASED attribute of the allocated variable. This other pointer receives the address of the allocated storage, and the pointer variable in the BASED attribute remains unchanged.

**EXAMPLE:**

```
DECLARE P POINTER,  
        VALUE BASED (Q) FLOAT;  
ALLOCATE VALUE;  
ALLOCATE VALUE SET (P);
```

The first ALLOCATE statement allocates storage for VALUE and assigns the location of the storage to pointer Q. The second ALLOCATE statement allocates additional storage for VALUE and assigns the location of this new storage to pointer P. The value of pointer Q and the storage allocated by the first ALLOCATE statement remain unchanged by the second allocation.

A single execution of an ALLOCATE statement may perform multiple allocations of storage.

**EXAMPLE:**

```
ALLOCATE VALUE, VALUE SET(P), SWITCH;
```

This statement is equivalent to the following set of statements:

```
ALLOCATE VALUE;  
ALLOCATE VALUE SET(P);  
ALLOCATE SWITCH;
```

Commas separate multiple references in an ALLOCATE statement. References to both based and controlled variables may appear in the same ALLOCATE statement.

**4A3. Freeing Based Storage**

Storage allocated for a based variable is freed for possible reuse by the FREE statement, which has the following basic format:

```
FREE based-variable;
```

This statement frees the storage currently associated with the based variable. The program obtains the address of this storage from the current value of the pointer variable declared in the **BASED** attribute of the based variable. The attributes of the based variable determine the amount of storage that is freed.

EXAMPLE:

```
DECLARE P POINTER,  
        ITEM BASED(Q) CHARACTER(10);  
ALLOCATE ITEM;  
ALLOCATE ITEM SET(P);
```

```
FREE ITEM;  
FREE P->ITEM;
```

In these statements, **P** and **Q** are pointer variables, and **ITEM** is a character-string based variable. Two allocations of storage occur for **ITEM**. Pointer **Q** contains the location of the first allocation; pointer **P**, the second. The first **FREE** statement frees the storage for **ITEM** at the location specified by **Q**. The second **FREE** statement frees the storage for **ITEM** at the location specified by **P**.

A based variable can be used to free storage only if that storage has been allocated for a based variable with the same attributes, including array bounds, string lengths, and arithmetic precisions. An attempt to free a based variable that has not been allocated produces unpredictable results.

#### 4A3A. Multiple References in a Free Statement

A single **FREE** statement may free storage for two or more based variables in one execution of the statement.

EXAMPLE:

```
FREE P->ITEM, ITEM;
```

This statement is equivalent to the following set of statements:

```
FREE P->ITEM;  
FREE ITEM;
```

Commas separate multiple references in a **FREE** statement. References to both based and controlled variables may appear in the same **FREE** statement.

#### 4A3B. Implicit Freeing of Storage

Under the following conditions, based storage is freed without the use of an explicit **FREE** statement:

- A. Based storage that has been allocated in an input/output buffer by the **LOCATE** statement is freed after the value of the associated variable is written into a file.
- B. Storage that has been effectively allocated by a **READ** statement with a **SET** option is freed by the next **READ** or **CLOSE** operation for the file.
- C. All storage is freed at the end of the task in which it was allocated, unless it was allocated within a storage area belonging to another task.

#### 4A4. An Example of Based Storage Used in a Sort Procedure

Procedure **SORT12** in Figure 4A4-1 obtains successive sets of input cards from the standard system input file (**SYSIN**). A control card precedes each set and contains a number that specifies the number of cards in the set. The number in the control card does not include the control card itself. The procedure sorts the cards of each set into ascending order on the first three characters of each card. It then puts the sorted cards (still preceded by their control count) into the standard system output file (**SYSPRINT**). These steps are repeated until all sets of input cards have been processed.

When the control card for a set of input cards is read and control enters the **BEGIN** block **PROCESS**, an array of pointers **P** is allocated automatically. The number of elements in **P** equals the number of cards in the set (that is, the number in the first column of the control card). **SORT12** then allocates based storage for the cards of the set as they are read and assigns the locations of the cards to the elements of pointer array **P**, which is used to sort the cards.

Actual sorting is performed by the subroutine-procedure **SORT9**, which appeared in Figure 3G1-5. The invoking reference to **SORT9** contains pointer array **P** as an argument. When control returns from **SORT9**, the successive pointer values in **P** specify the locations of the cards in sort sequence. The **DO**-group named **OUTPUT** successively assigns the card locations in the pointer array **P** to the pointer **S**. The reference **S->IMAGE** is used to print each card in sort sequence.

Before processing continues with the next set of input cards, the storage allocated for the present set of cards and for array **P** is freed. The automatic storage for array **P** is freed when control leaves the **BEGIN** block **PROCESS**.

```

SORT12:
F4A4_1:
PROCEDURE OPTIONS (MAIN);
  DECLARE
    1 IMAGE BASED (S),
    2 KEY CHARACTER (3),
    2 DATA CHARACTER (77),
    N FIXED DECIMAL (1);
  ON ENUFILE (SYSIN)
  BEGIN;
    CLOSE FILE (SYSPRINT);
    GO TO
      PROC_END;
  END;
INPUT:
  GET
    EDIT (N) (F(1));
  GET
    SKIP;
PROCESS:
  BEGIN;
  DECLARE
    P(N) POINTER;
CARDS:
  DO
    I = 1 TO N;
    ALLOCATE IMAGE;
    P(I) = S;
  GET
    EDIT(S->IMAGE)(A(3), A(77));
  END
  CARDS;
  CALL SORT9 (P);
COUNT:
  PUT
    DATA (N);
  PUT
    SKIP;
OUTPUT:
  DO
    I = 1 TO N;
    S = P(I);
  PUT
    EDIT(S->IMAGE)(A(3), A(77));
  PUT
    SKIP; FREE S->IMAGE;
  END
  OUTPUT;
  PUT
    SKIP;
NEXT:
  GO TO
    INPUT;
  END
  PROCESS;
SORT9:

```

```

F3G1_5:
PROCEDURE (P);
  DECLARE
    P(*) POINTER,
    T POINTER,
    1 IMAGE BASED (S),
    2 KEY CHARACTER(3),
    2 DATA CHARACTER(77);
  SORT:
    TEST = 0;
  LOOP:
    DO
      I = 1 TO (DIM(P,1)-1);
      S = P (I); T = P (I + 1);
      IF
        (S->IMAGE.KEY) > (T->IMAGE.KEY)
      THEN
        DO;
          TEST = 1; P(I) = T; P(I + 1) = S;
        END;
      END;
    LOOP;
    IF
      TEST = 1
    THEN
      O TO
        SORT;
      RETURN;
    END
  SORT9:
  PROC_END:
  END
  SORT12:

```

```

N= 5;
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

N= 5;
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

N= 5;
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

```

Figure 4A4-1. Allocating and freeing based storage for numbers of structures sorted with an array of pointers

#### 4A5. Allocating Based Storage for a Self-defining Structure

Self-defining data contains descriptive information about itself, such as its size, which can be used by the programmer for packing, unpacking, scanning, editing, and storage allocation purposes. A common application of self-defining data occurs with variable-length input/output records that contain a specification of their own size. PL/I permits a based structure to be self-defining by allowing it to contain either one adjustable string length or one adjust-

able array bound, the value of which is maintained by a variable within the structure itself.

However, such a variable cannot possess a value until storage has been allocated for the self-defining based array. And since the amount of storage to be allocated depends on the value of this variable, a facility is needed for associating a value outside a based structure with a variable within the structure prior to allocation. This facility in PL/I is called the REFER option.

#### 4A5A. The REFER Option

The declaration of a self-defining based structure contains a REFER option, which determines the amount of storage involved in an allocation of the structure. When storage is allocated for a self-defining based structure, the REFER option causes the value of a variable outside the structure to be assigned to a variable within the structure, and also causes this value to be used as the string length or array bound of a component involved in the same allocation of the structure.

The REFER option has the following general format:

element-variable REFER(element-variable)

The element variables must be unsubscripted fixed-point binary variables of default precision. The variable to the left of the keyword REFER must not be a component of the self-defining based structure. This variable can be qualified by both a pointer variable and the name of a structure component if the variable is part of a structure. The variable to the right of REFER must belong to the structure that contains the REFER option.

The REFER option can be used in the declaration of a based structure only as the length of a string, or as the bound of an array.

#### EXAMPLE:

```
DECLARE 1 STRING BASED (P),
        2 HEAD FIXED BINARY,
        2 BODY CHARACTER(SIZE REFER
        (HEAD)),
        SIZE FIXED BINARY;
```

This statement declares STRING to be a self-defining based structure, which contains the two elements HEAD and BODY. When storage is allocated for STRING, the length of the character string BODY is automatically set equal to the current value of SIZE, and this value, in turn, is automatically assigned to HEAD by the program. Any reference to STRING, other than a reference in an ALLOCATE statement, uses the value of HEAD to determine the length of BODY. For example, the statement FREE STRING; uses the value of HEAD to determine the amount of storage to be freed.

Note that the REFER option can appear only once in a structure declaration. When the option specifies a string length, the string must be an element variable and must be the last element variable in the structure declaration.

*If the REFER option appears as an array bound, the bound must be the upper bound of the leftmost dimension of the array variable with which it is used.* The REFER option must also belong to the last array variable in the structure declaration or to a minor structure containing the last element of the structure.

#### EXAMPLE:

```
DECLARE 1 TABLE BASED(P),
        2 UPPER FIXED BINARY,
        2 CONTENTS(0:N REFER(UPPER))
        CHARACTER(15);
```

In the declaration of TABLE, the REFER option specifies N as an adjustable upper bound for the one-dimensional array CONTENTS. Each element in array CONTENTS is a string of 15 characters.

#### EXAMPLE:

```
DECLARE
  1 PRESSURE BASED(P),
  2 VOLUME,
  3 INDEX FIXED BINARY,
  3 POUNDS FLOAT,
  2 TEMPERATURE,
  3 FLASH FIXED BINARY,
  3 RANGE FIXED BINARY,
  3 DEGREES(T REFER(RANGE),-32:0)
  FLOAT,
  T FIXED BINARY;
```

The declaration of PRESSURE uses the REFER option to specify T as the adjustable upper bound of the leading dimension of the two-dimensional array named DEGREES. As in the preceding example, the REFER option belongs to the last array in the structure.

It should be noted that since the adjustable bound in a self-defining based structure must appear as the leading dimension of the component with which it is declared, it is not possible for that component to inherit a dimension from a higher level component. Inherited dimensions automatically become the leading dimensions of the lower-level component.

#### EXAMPLE:

```
DECLARE
  1 SCHEDULE BASED (P),
  2 LIMIT FIXED BINARY,
  2 DISTANCE (10),
  3 RATE (25),
  3 TIME (25) FLOAT;
```

In this declaration, RATE and TIME inherit the bounds 1:10 of the leading dimension of DISTANCE. RATE and TIME both receive the bounds 1:10 and 1:25. Any attempt, therefore, to use the REFER option in the dimension attribute of TIME would be incorrect. However, the option could appear with DISTANCE, in place of 10.

EXAMPLE:

```

DECLARE
  1 STEAM BASED (P),
    2 VOLUME,
      3 INDEX FIXED BINARY,
      3 POUNDS FLOAT,
  2 TEMPERATURE FLOAT,
  2 LEADING FIXED BINARY,
  2 POOL (VECTOR REFER(LEADING)),
    3 GALLONS (5) FLOAT,
    3 DEGREES (5) FIXED DECIMAL,
    3 AREA (5) FLOAT,
  VECTOR FIXED BINARY;

```

The REFER option is used correctly in this example, because it appears with the minor array of structures named POOL, which contains the last component in the based structure STEAM. When storage is allocated for STEAM, the lower-level components of POOL, which are the arrays GALLONS, DEGREES, and AREA, inherit the value of VECTOR as their leading upper bound. The value of VECTOR is also assigned automatically to the element variable LEADING.

After storage has been allocated for a self-defining structure, an assignment statement can be used to change the value of the element variable on the right of the REFER option. When this change occurs, the following rules apply to the self-defining structure:

- A. The self-defining structure must not be freed until the element variable is restored to the value it had when allocated.
- B. The self-defining structure must not be written out while the element variable has a value greater than the value it received when allocated.
- C. The self-defining structure may be written out when the element variable has a value equal to or less than the value it received when allocated. The number of elements or the length of the string actually written is that specified by the current value of the element variable.

EXAMPLE:

```

DECLARE
  1 RECORD BASED(P),
    2 SIZE FIXED BINARY,
  2 ELEMENTS (COUNT REFER(SIZE)),
  COUNT FIXED BINARY INITIAL(100);
.
.
.
.
ALLOCATE RECORD;

```

```

SIZE = 90   SIZE = 90;
WRITE FILE(OUT) FROM(RECORD);
.
.
.
.

```

In this example, the first 90 values of array ELEMENTS are written along with the value of SIZE. An attempt to free RECORD at this point will create an error because SIZE must be restored to the value it had when allocated, namely, 100. Had SIZE been assigned a value greater than 100, the WRITE statement would produce an error.

*4A5B. An Example of the REFER Option in a Sort Procedure*

Procedure SORT13 in Figure 4A5B-1 contains an example of the REFER option. This procedure processes the same input and produces the same results as procedure SORT12 in Figure 4A4-1. The major difference between the two procedures occurs in the way based storage is allocated for each set of cards being sorted. SORT12 allocates storage individually for each card in a set and controls the allocations by means of a DO statement. SORT13, however, allocates storage collectively for all cards in a set by means of a self-defining array of structures called ARRAY, which contains the single element SIZE and a variable number of occurrences of the structure IMAGE.

The number in the first column of the control card before each set of cards specifies the number of cards in the set (excluding the control card itself). This number is assigned to variable N, which appears in the following expression associated with the declaration of IMAGE:

```
N REFER(SIZE)
```

The expression states that the number of occurrences of IMAGE within ARRAY is given by N and that after storage is allocated for ARRAY, the value of N is to be assigned to element SIZE within ARRAY.

This use of the REFER option permits a single execution of the ALLOCATE statement to allocate storage for all cards in a set and removes the need for individual allocations under control of a DO statement.

As in SORT12, actual sorting is performed by subroutine procedure SORT9, which appeared in Figure 3G1-5 of Chapter 3. The invoking reference to SORT9 contains pointer array P as an argument. When control returns from SORT9, the successive pointer values in P specify the locations of the cards in sort sequence.

Note that the card addresses in array P cannot be used to qualify the based structures in IMAGE when the cards are printed in sort sequence. Any pointer value that qualifies a based structure within IMAGE is assumed to qualify



```

SORT13:
F4A5B_1:
PROCEDURE OPTIONS (MAIN);
  DECLARE
    1 ARRAY BASED(R),
    2 SIZE FIXED BINARY,
    2 IMAGE (N REFER(SIZE)),
    3 KEY CHARACTER(3),
    3 DATA CHARACTER(77),
    1 MAP BASED (T),
    2 KEY CHARACTER (3),
    2 DATA CHARACTER (77),
    N FIXED BINARY,
    COUNT FIXED DECIMAL (1);
  ON ENDFILE (SYSIN)
  BEGIN;
    CLOSE FILE (SYSPRINT);
    GO TO
      PROC_FND;
  END;
  INPUT:
    GET
      EDIT (COUNT) (F(1));
    GET
      SKIP;
      N = COUNT;
      ALLUCATE ARRAY;
  PROCESS:
    BEGIN;
    DECLARE
      P(COUNT) POINTER;
  CARDS:
    DO I = 1 TO N;
      P(I) = ADDR(IMAGE(I));
    GET
      EDIT (IMAGE (I)) (A(3), A(77));
  END
    CARDS;
    CALL SORT9(P);
  OUTPUT:
    PUT
      DATA (COUNT);
    PUT
      SKIP;
    DO
      I = 1 TO N; T = P(I);
    PUT
      EDIT (T->MAP) (A(3), A(77));
    PUT
      SKIP;
  END;
    PUT
      SKIP;
      FREE ARRAY;
  NEXT:
    GO TO
      INPUT;
  END
    PROCESS;
  PROC_FND:
  FND
    SORT13;

COUNT= 5;
A14_FIRST-----1
A96_SECOND-----2
E02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5

```

Figure 4A5B-1. Allocating and freeing based storage for a self-defining array of structures which is sorted with an array of pointers

the entire containing structure ARRAY and will be automatically adjusted (offset) by the number of storage bytes separating the specified structure within IMAGE from the beginning of ARRAY. To avoid this erroneous address adjustment, an independent based structure named MAP is used to refer to the cards when they are printed in sort sequence.

Later examples in Chapter 5 show how the REFER option is used to create self-defining records in output files.

#### 4A6. Allocating and Freeing Based Storage within an Area

PL/I provides a type of variable called the *area variable*, which reserves storage for allocations of based variables. The area variable permits based allocations to be grouped as a unit for convenient input/output transmission or assignment to another area variable while maintaining the separate identity of each allocation. The following discussion describes how area variables are used to group based allocations. Later discussions in Chapter 5 present the use of area variables for relocating based allocations without invalidating the address values of associated pointer variables.

##### 4A6A. The AREA Attribute

An identifier becomes an area variable when it is declared with the AREA attribute, which has the following general format:

AREA(expression)

The integral portion of the value obtained from the expression represents the size of the area in bytes. The expression, however, is optional; when it is not used, an implementation-defined size is assumed by the PL/I compiler.

Although an area variable reserves storage for allocations of based variables, it can have any storage class. The size of an area with static storage class must appear in the AREA attribute as an unsigned fixed-point decimal integer constant. The AREA attribute is not restricted to element identifiers; it also applies to array and structure identifiers. PL/I also provides for area arguments and parameters; and the asterisk notation can be used to denote the size of an area parameter. The DEFINED attribute can be used to define an area on another area, through overlay or correspondence defining; both areas, however, must have the same size.

**EXAMPLE:**

```

DECLARE A STATIC AREA(32767),
        B AREA,
        C AREA(N),
        D AREA(S) CONTROLLED,
        E AREA(10000) BASED(P),
        F(5) AREA(400),
        I G,
        2 H AREA(100),
        2 I AREA(200),
        2 J AREA(300),
        K AREA DEFINED B,
        L AREA (*);

```

This statement specifies that:

- A. A is a static area variable that reserves 32767 bytes of storage.
- B. B is an automatic area variable that reserves an implementation-defined amount of storage.
- C. C is an automatic area variable whose size depends on the value of N current at the time the block to which it is internal is activated.
- D. D is a controlled area variable whose size depends either on the value of S at the time an ALLOCATE statement allocates storage for D, or on a size specification in the ALLOCATE statement which overrides S.
- E. E is a based area variable that reserves 10000 bytes of storage on each allocation.
- F. F is an area that contains five areas, each of which reserves 400 bytes of automatic storage.
- G. G is an area structure that contains the three areas H, I, and J. H reserves 100 bytes of automatic storage, I reserves 200 bytes, and J reserves 300 bytes.
- H. K is an area defined on area B.
- I. L is an area parameter that assumes the same size as its associated area argument in a subroutine or function invocation.

**4A6B. The IN Option**

The ALLOCATE statement uses the IN option for based allocations within an area. The statement has the following general format:

```

ALLOCATE based-variable SET(pointer-variable)
        IN(area-variable);

```

This statement allocates storage for the based variable within the specified area and assigns the location of the allocated storage to the pointer variable. The IN option is not required; when it is not used, the based variable is

allocated in a storage area provided by the operating system. When the SET option is used, it may appear either before or after the IN option. If the variable in an IN option is not explicitly declared, it is automatically assumed to be an area variable.

The FREE statement, as applied to based variables, has the following general format:

```

FREE based-variable IN (area-variable);

```

The IN option must appear in a FREE statement if the based allocation was made within an explicitly specified area; otherwise, the option is omitted.

**EXAMPLE:**

```

DECLARE STORE AREA(500) BASED (P),
        VALUE BASED (Q) FIXED DECIMAL (5,2),
        R POINTER;
ALLOCATE STORE:
/* P ADDRESSES AREA STORE. */
.
.
.
ALLOCATE VALUE IN(STORE);
/* Q ADDRESSES ALLOCATION OF VALUE IN
STORE. */
.
.
.
ALLOCATE VALUE IN(P->STORE) SET (R);
/* R ADDRESSES SECOND ALLOCATION OF VALUE
IN STORE. */
.
.
.
FREE VALUE IN(STORE);
/* FREES ALLOCATION OF VALUE ADDRESSED BY
Q. */
.
.
.
FREE R->VALUE IN(STORE);
/* FREES ALLOCATION OF VALUE ADDRESSED BY
R. */
.
.
.

```

The first ALLOCATE statement allocates 500 bytes of storage for area STORE and assigns the location of the allocated storage to pointer P.

The second ALLOCATE statement causes storage for based variable VALUE to be allocated within area

P->STORE and assigns the location of the allocated storage to pointer Q.

The third ALLOCATE statement causes another allocation of VALUE (different from Q->VALUE) within area P->STORE and sets pointer R equal to the location of the allocated storage.

The FREE statements employ the IN option because allocations of VALUE were explicitly made in STORE. Although the allocations of VALUE become free, the storage for area STORE remains allocated.

#### 4A6C. The AREA ON-Condition

An attempt to allocate based storage within an area that contains insufficient free storage for the allocation produces an AREA ON-condition. If no ON-unit appears for the AREA condition in an ON statement, the operating system issues a comment and raises the ERROR condition.

When an ON-unit is specified and a normal return occurs from the ON-unit, the ALLOCATE statement that raised the AREA condition is executed again. If the ON-unit has changed the value of a pointer qualifying (explicitly or implicitly) the reference to the inadequate area so that the pointer value specifies another area, the allocation is reattempted within the new area. Failure of the ON-unit to provide a larger area may place the program in an error loop.

Chapter 5 discusses other situations that may produce an AREA condition. The ONCODE built-in function can ascertain the type of situation that has raised the ON-condition.

#### 4A6D. An Example of an Area Variable in a Sort Procedure

Procedure SORT14 in Figure 4A6D-1 sorts successive sets of cards into ascending order on the first three characters of each card. The standard system-input file (SYSIN) contains the cards, and each set may contain a different number of cards. A trailer card with asterisks in the first three positions signals the end of each set of cards. After each set is sorted, it is written with its trailer card into the standard system-output file (SYSPRINT).

SORT14 allocates storage for based variable IMAGE throughout the empty area A (see Figure 4A6D-2). Pointer variable HEAD receives the address of the first allocation for IMAGE (see Figure 4A6D-3). The location of each subsequent allocation for IMAGE is assigned to pointer L of the previous allocation. Pointer L of the last allocation receives a null address value. Storage is allocated for IMAGE throughout area A until the AREA ON-condition occurs. At that point, all allocations for IMAGE form a linked chain to which input is assigned (see Figure 4A6D-3).

Input cards are assigned to successive positions in the chain until a trailer card is read. The pointer L associated

with the last card in storage then receives a null value, and the location of the first unused storage position in the chain is assigned to pointer UNUSED (see Figure 4A6D-4).

Actual sorting of the cards in storage is performed by subroutine procedure SORT11 of Figure 3G2-7, which was discussed in Chapter 3. Pointer HEAD serves as the argument in the invocation of SORT11. When control returns from SORT11, the cards are linked in ascending sort order (see Figure 4A6D-5).

After the sorted cards are written into the output file, the storage positions linked to pointer UNUSED are re-linked to the chain formed by HEAD (see Figure 4A6D-6). Processing then continues with the next set of cards.

Figure 4A6D-7 contains examples of sample input and output for SORT14.

## 4B. ORGANIZATION OF DATA IN LIST FORM

The storage allocation and pointer manipulation techniques used by procedure SORT14 (paragraph 4A6D) demonstrate basic methods for organizing and processing data items in list form. These methods involve the following general steps:

- A. linking successive allocations of based storage into a chained list (briefly called a list), which is identified by a pointer variable that contains the absolute address of the first storage component in the list
- B. inserting successive input items into the leading storage components of the list to form a list of data items
- C. retaining unused storage components in a separate list
- D. processing the data items in the list (this usually produces changes in the address values of the pointer links attached to the data items in the list)
- E. relinking used and unused storage components into a single list before processing the next set of input items.

Although these steps are general, they are not unique. Alternative methods can be developed for obtaining equivalent results. For example, based storage need not be allocated throughout an area before input items are read. Storage can be allocated as each item is read and freed when processing has been completed for a set of items. With this technique, no excess storage would be allocated for a set of input items. However, allocating and freeing storage repeatedly in this manner would be inefficient, since it is faster to allocate all necessary storage once, and to separate used from unused storage by pointer manipulation.

### 4B1. The Main Parts of a List

The list organization shown in Figure 4B1-1, while not the most general type of organization, illustrates the main parts contained in all lists. The list in this figure consists

```

SORT14:
F4A6D_1:
PROCEDURE OPTIONS (MAIN);
DECLARE
    (HEAD, UNUSED, S, T, U, V) POINTER,
    A AREA (1000),
    1 IMAGE BASED(S),
    2 (KEY CHARACTER(3),
    DATA CHARACTER(77)),
    L POINTER),
    1 CARD,
    2 KEY CHARACTER(3),
    2 DATA CHARACTER(77)),
    SORT11 ENTRY (POINTER);

    /* WHEN AREA CONDITION OCCURS, SET
    LAST ALLOCATED LINK TO NULL */
    ON AREA

BEGIN;
IF
    S = NULL
THEN
    EXIT;
    T->L = NULL;
    GO TO
    INPUT1;
END;

    /* WHEN END-OF-FILE CONDITION OCCURS
    FREE BASED STORAGE, CLOSE SYSPRINT
    FILE, AND END PROCEDURE */
    ON ENDFILE (SYSIN)

BEGIN;
    S = HEAD;
    DO
        WHILE(S->NULL);
            T = S->L; FREE S->IMAGE; S = T;
    END;
    CLOSE FILE (
    SYSPRINT);
    GO TO
    OVER;
END;

    /* THROUGHOUT AREA A, ALLOCATE AND
    LINK STORAGE FOR BASED VARIABLE
    IMAGE */
    S = NULL;
    ALLOCATE IMAGE IN(A) SET (S);
    HEAD, T = S;
    DO
        WHILE (1B);
            ALLOCATE IMAGE IN (A) SET (S);
            T->L = S; T = S;
    END;

    /*
    DO
        WHILE(1B) IS TERMINATED BY AREA
        CONDITION */

    /* ASSIGN CARDS TO BE SORTED TO
    LINKED STORAGE */
INPUT1:
    S = HEAD;
INPUT2:
    GET
    IF
        EDIT(CARD)(A (3), A(77));
        CARD.KEY = '***'
THEN
    /* SAVE LINK OF LAST CARD AND
    REPLACE IT BY NULL */
    DO;
        UNUSED = T->L; T->L = NULL;
        GO TO
        INPUT3;
    END;

    S->IMAGE = CARD, BY NAME;
    T = S; S = S->L;
    GO TO
    INPUT2;
INPUT3:

    /* PRINT INPUT SEQUENCE */
    PUT
    LIST ('SORT14 INPUT:');
    PUT
    SKIP;
    S = HEAD;
    DO
        WHILE (S->NULL);
        PUT
        EDIT(S->IMAGE.KEY, S->IMAGE.DATA)
        (A);
        PUT
        SKIP;
        S = S->L;
    END;
    PUT
    EDIT (CARD)(A(3), A(77));
    PUT
    SKIP(2);
SORTM:
    CALL
    SORT11 (HEAD);

    /* PRINT SORTED CARDS FOLLOWED BY
    TRAILER CARD */
    PUT
    LIST ('SORT14 OUTPUT:');
    PUT
    SKIP;
    S = HEAD;
    DO
        WHILE (S->NULL);
        PUT
        EDIT (S->IMAGE.KEY, S->IMAGE.DATA)
        (A(3), A(77));
        PUT
        SKIP;
        T = S; S = S->L;
    END;
    PUT
    EDIT (CARD)(A(3), A(77));
    PUT
    SKIP (2);

    /* RELINK UNUSED STORAGE AND
    CONTINUE PROCESSING INPUT */
    T->L = UNUSED;
    GO TO
    INPUT1;
OVER:
    END
    SORT14;

```

Figure 4A6D-1. Allocating and linking based storage in an area for varying numbers of structures that are to be sorted

```

SORT14 INPUT:
A02_SECOND-----2
F34_THIRD-----3
A23_FIRST-----1
...

SORT14 OUTPUT:
A23_FIRST-----1
A02_SECOND-----2
F34_THIRD-----3
...

SORT14 INPUT:
Z71_FIFTH-----5
A96_SECOND-----2
A14_FIRST-----1
W17_FOURTH-----4
F02_THIRD-----3
...

SORT14 OUTPUT:
A14_FIRST-----1
A96_SECOND-----2
F02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5
...

SORT14 INPUT:
A14_FIRST-----1
A96_SECOND-----2
F02_THIRD-----3
W17_FOURTH-----4
...

SORT14 OUTPUT:
A14_FIRST-----1
A96_SECOND-----2
F02_THIRD-----3
W17_FOURTH-----4
...

SORT14 INPUT:
W17_FOURTH-----4
Z71_FIFTH-----5
A96_SECOND-----2
Z71_SIXTH-----6
A14_FIRST-----1
F02_THIRD-----3
...

SORT14 OUTPUT:
A14_FIRST-----1
A96_SECOND-----2
F02_THIRD-----3
W17_FOURTH-----4
Z71_FIFTH-----5
Z71_SIXTH-----6
...

```

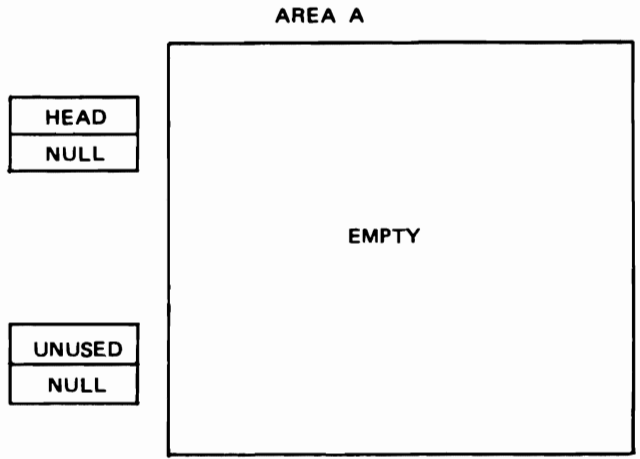


Figure 4A6D-2. Area A before based storage is allocated and linked for cards

Figure 4A6D-7. Printout of cards sorted by procedure SORT14

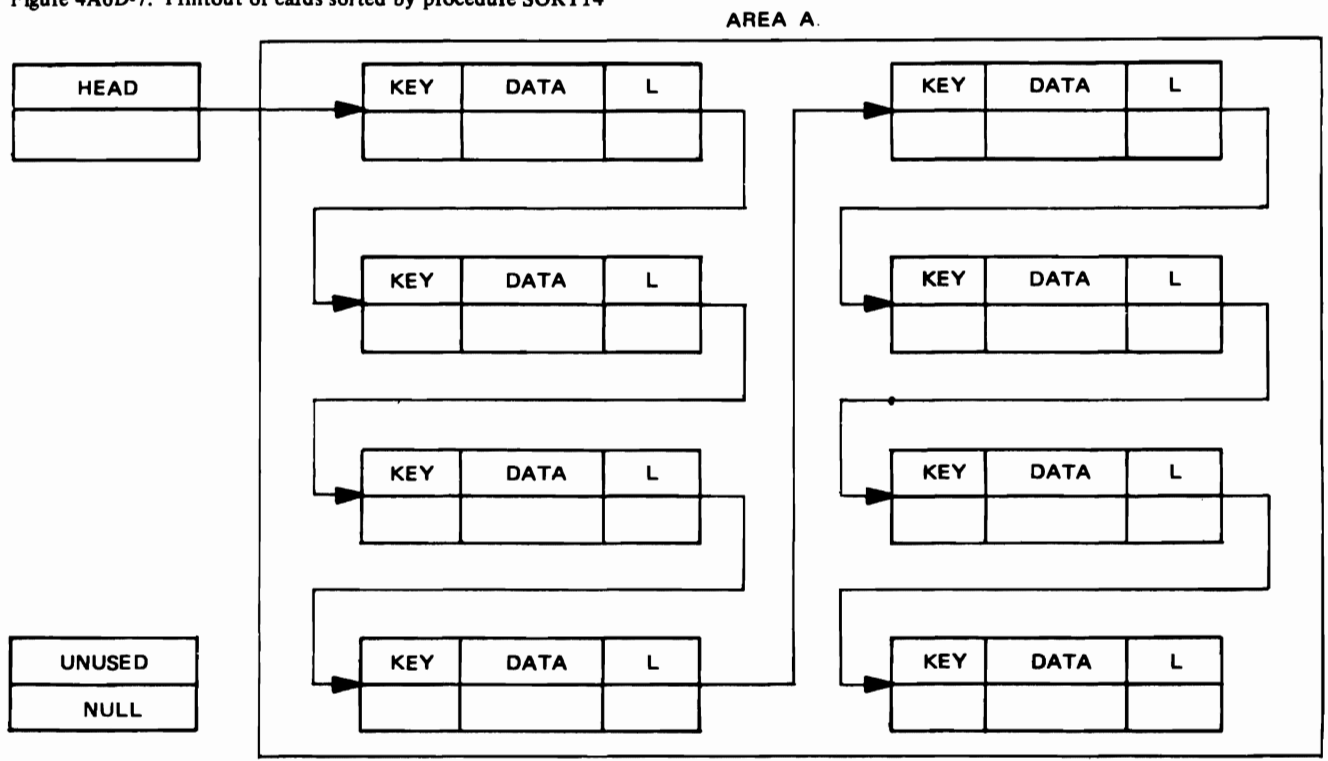


Figure 4A6D-3. Linked storage before cards are read

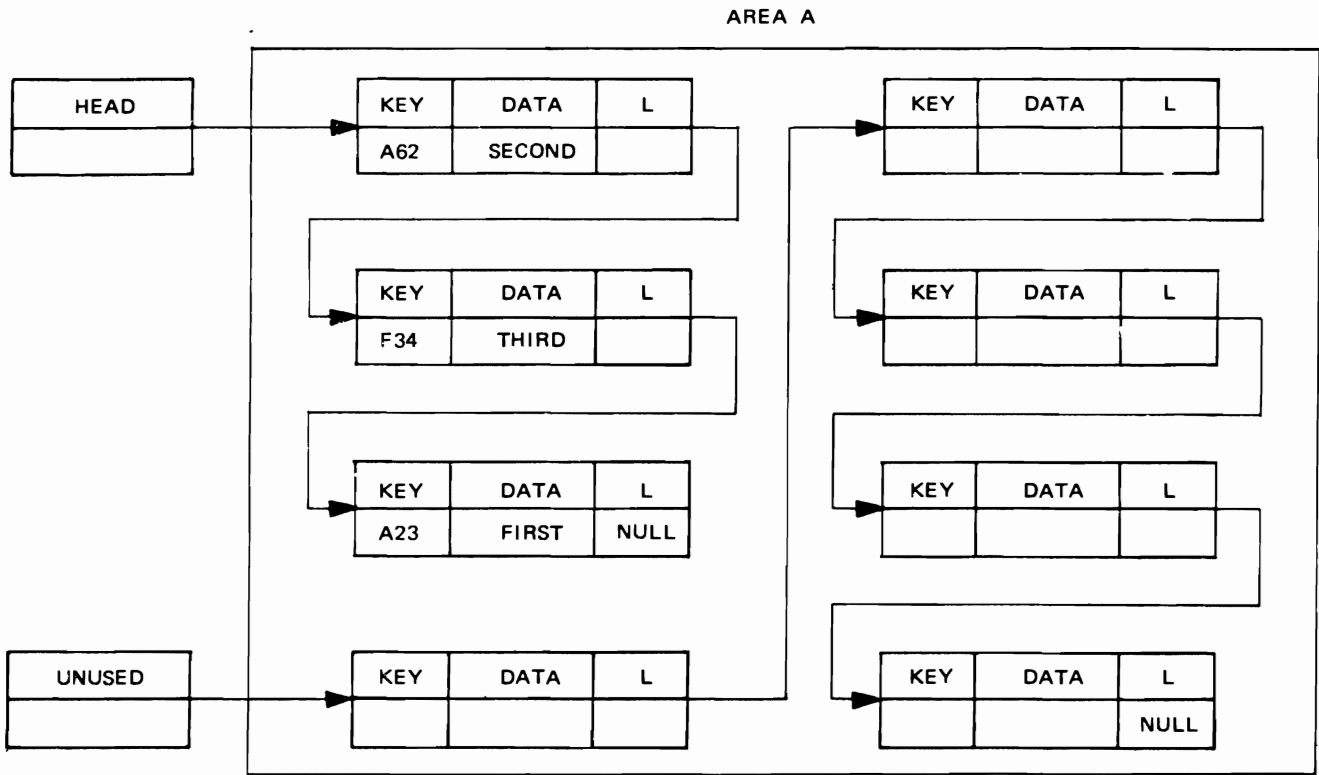


Figure 4A6D-4. Linked storage after cards are read

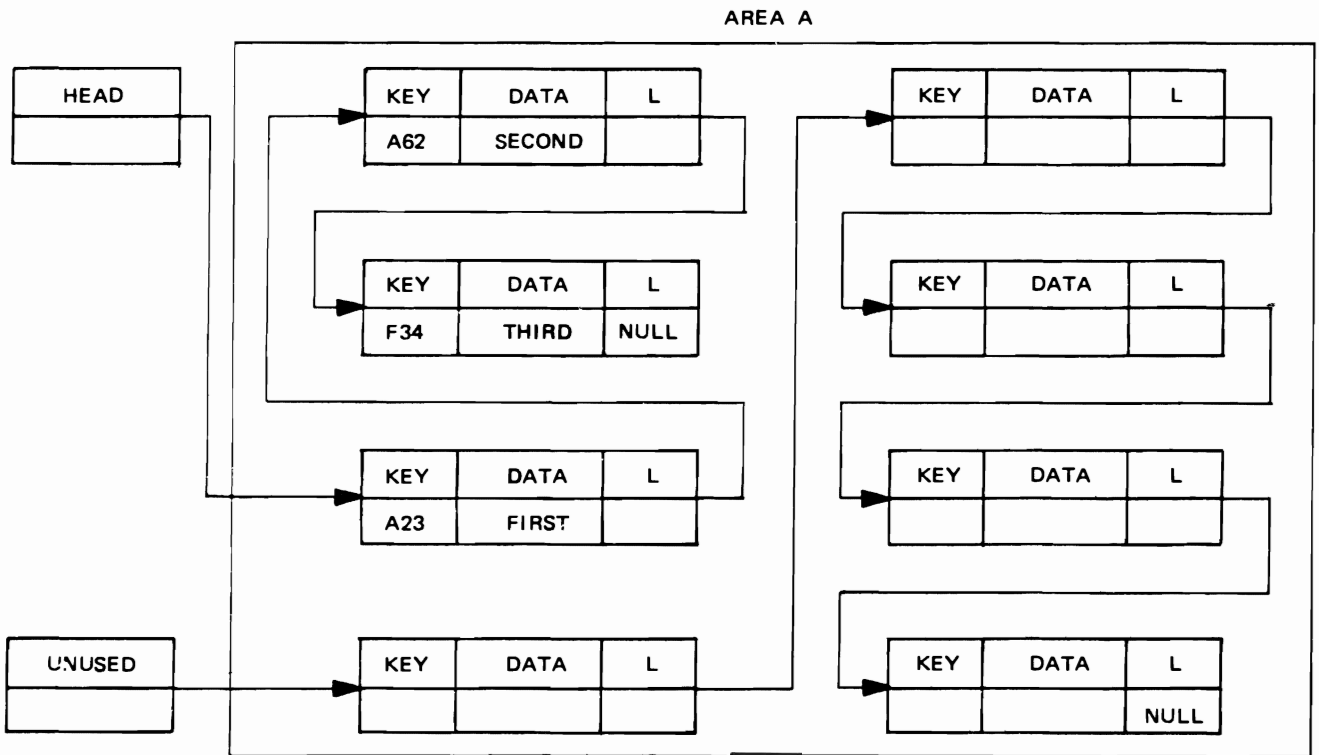


Figure 4A6D-5. Linked storage after cards are sorted

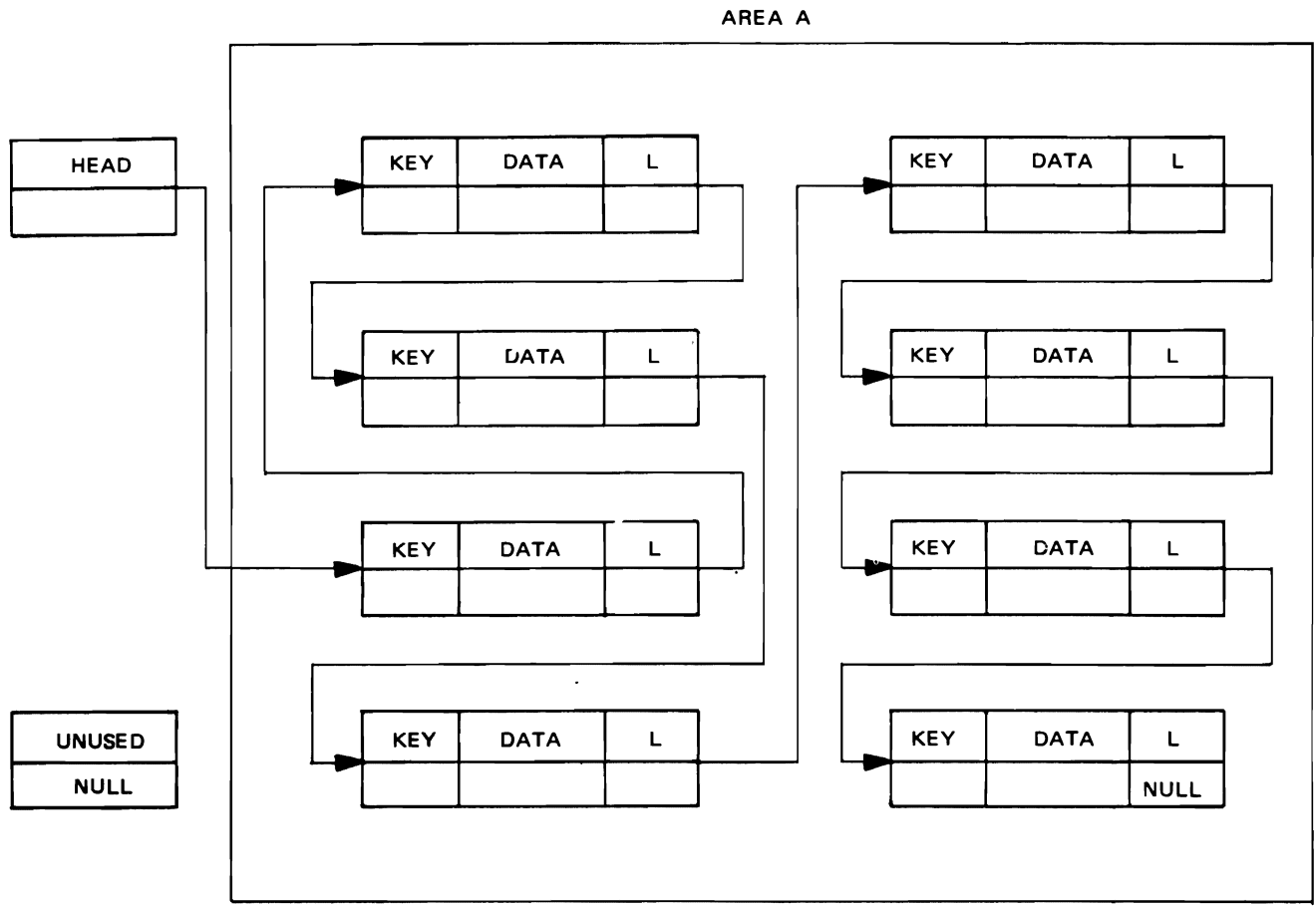
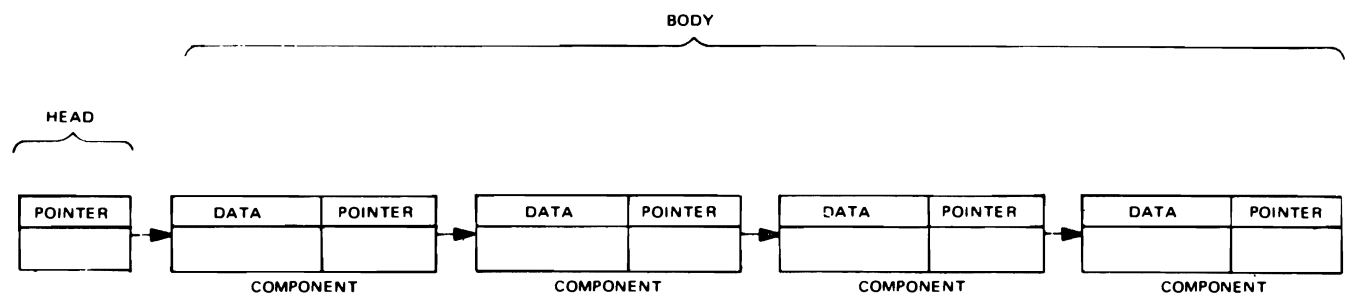


Figure 4A6D-6. Linked storage before next set of cards is read



(ARROWS REPRESENT POINTER VALUES THAT SPECIFY THE LOCATION OF THE NEXT COMPONENT)

Figure 4B1-1. A list with four components

of a head and a body. The head is a pointer variable that identifies the list and contains the address of the first component in the list. The body is a sequence of list components, each of which consists of a data item and a pointer variable. Except for the last pointer, which has a null address, the pointer in a component contains the address of the next component in the list.

This type of list organization always requires a head and permits the body of a list to contain an arbitrary number of components, limited only by available storage. It is even possible for the body of a list to contain no components; in this case, the list is said to be null (see Figure 4B1-2).

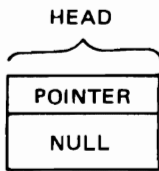


Figure 4B1-2. A null list

Figures 4B1-1 and 4B1-2 do not show the area within which storage has been allocated for list components. These illustrations emphasize the main parts of a list and deemphasize the environmental aspects of list organization. They also stress the close resemblance between a list and a one-dimensional array, the major difference being that successive components of a list need not occupy contiguous storage locations.

#### 4B2. Advantages of Lists

Lists retain the advantages obtained from absolute addressing:

- A. reducing data movement
- B. associating data items in scattered locations.

With these capabilities, items may be inserted into and deleted from lists without forcing other items in a list to be moved, as would be required with items in an array. Lists also give the programmer better control over storage allocation and permit the same storage to be used by several different variables at different times. Sharing storage in this manner also reduces the amount of storage that would ordinarily lie dormant in anticipation of maximum storage requirements for individual arrays and structures.

#### 4B3. Types of Lists

Figure 4B1-1 illustrates one of the simplest types of list organization. However, other organizations are possible, and the pointer and based variable facilities of PL/I allow three major types of lists to be created:

- A. Data lists (i.e. lists of data items)
- B. Pointer lists (i.e. lists of pointer values)
- C. Lists of lists

A data list, as shown in Figure 4B1-1, consists of linked data items. This type of list, however, possesses two important disadvantages: all data items in the list generally must have the same attributes, and the same data item cannot be shared by two or more lists at the same time; a distinct copy of the item must appear in each list, thus reducing conservation of storage.

These disadvantages can be removed by replacing the data items in a list with pointer variables that specify the locations of data items outside the list. Such lists are called pointer lists because they consist of linked pointers. They retain the advantages of list organization while allowing the same data item to be shared (pointed to) by different lists and permitting the data items associated with a list to have different attributes.

Note that it is possible to allow a data list to contain data items with different attributes. However, such a list must be processed on an individual basis. Pointer lists, on the other hand, permit general rather than specific processing techniques to be developed for all lists and still allow list items to possess a variety of attributes.

It is also possible for the items in a list to be other lists. In this case, the containing list is called a list of lists. This type of organization permits lists to possess advantages analogous to those associated with multidimensional arrays.

The techniques presented in SORT14 for inserting, deleting, and rearranging items in a data list also apply to pointer lists and lists of lists. These advanced types of lists, however, permit far more complicated arrangements of list items than is possible with the simple linear organization used in SORT14. Since detailed discussion of the methods for organizing and processing advanced list organizations lies beyond the scope of this manual, the remainder of the manual deals solely with data lists that possess the simple linear organization used in SORT14.

#### 4C. REVIEW OF TECHNIQUES FOR ORGANIZING BASED STORAGE IN LIST FORM

This chapter has shown how the ALLOCATE statement may be used to generate storage for a based variable as the need arises during program execution. Successive executions of an ALLOCATE statement for a based variable permit multiple generations of storage to be associated simultaneously with the same based variable. Each generation of storage is distinguished by a qualifying pointer, and should a particular generation of storage no longer be needed, it can be released by executing a FREE statement. Storage that has been freed becomes available for further



allocation. Allocating and freeing storage during program execution generally improves the efficiency of computer applications that have highly variable storage requirements (see the application areas described in Chapter 1).

Repeated allocation and release of storage, however, often increases the execution time of a program by a significant amount, particularly when the allocations are managed by the operating system. One way of reducing such inefficiency is to allocate required storage only once and to link the storage into a list. An illustration of such a list appears in Figure 4C-1 where 18 storage components have been allocated throughout an area and linked to form the list called UNUSED. As storage is required by other

lists, components can be unlinked from UNUSED and linked in turn to the new lists. Figure 4C-2 shows two lists (LIST1 and LIST2) that have obtained different amounts of storage from UNUSED. Similarly, when a storage component is no longer needed by a list, the component can be relinked to UNUSED where it becomes available for assignment to other lists as the need arises. These techniques for pooling storage were used in SORT14. They eliminate the inefficiency associated with repeated executions of ALLOCATE and FREE statements and at the same time maintain flexibility in storage assignment by means of address manipulation.

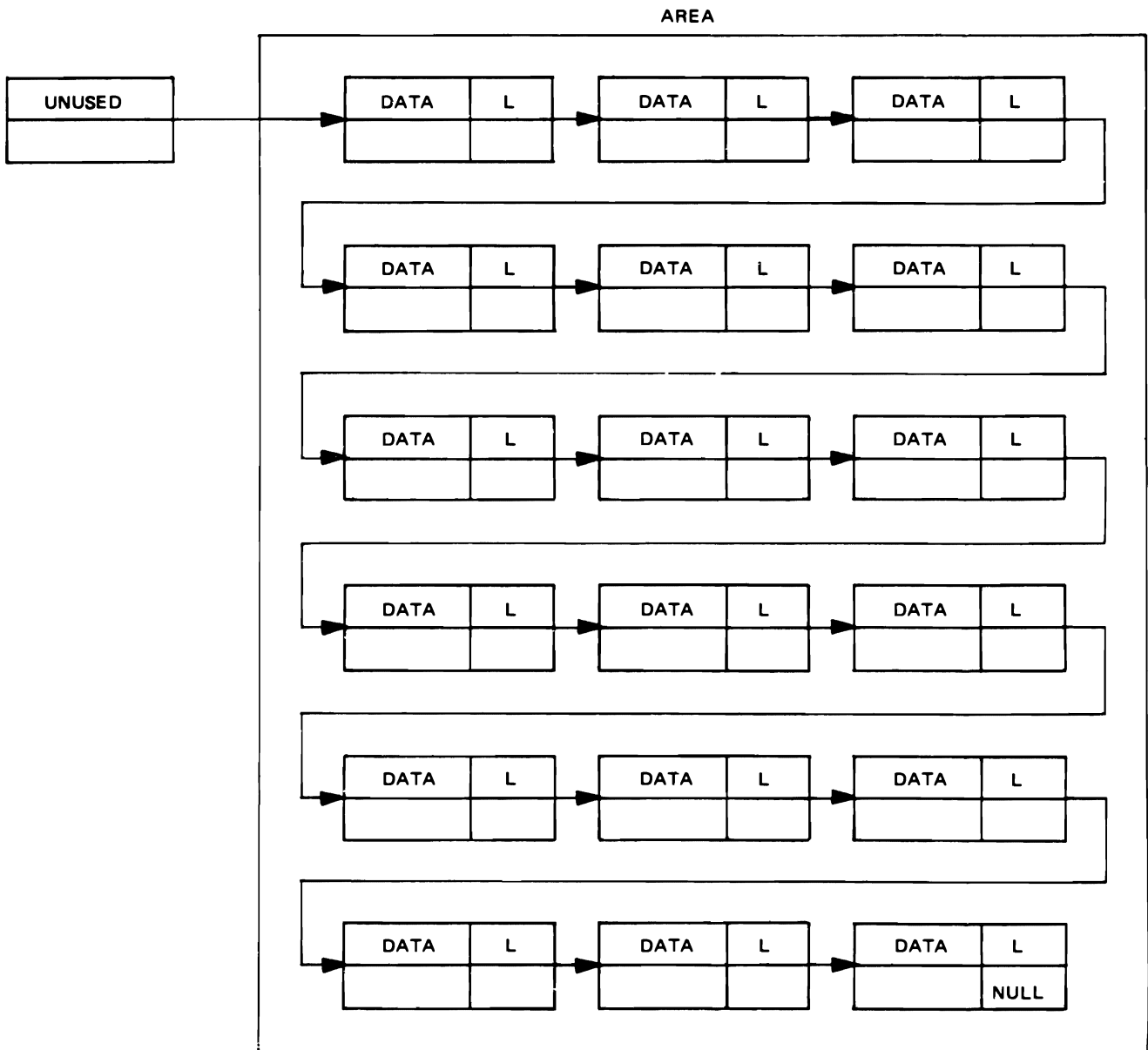


Figure 4C-1. Allocating storage components throughout an area and linking them into a list

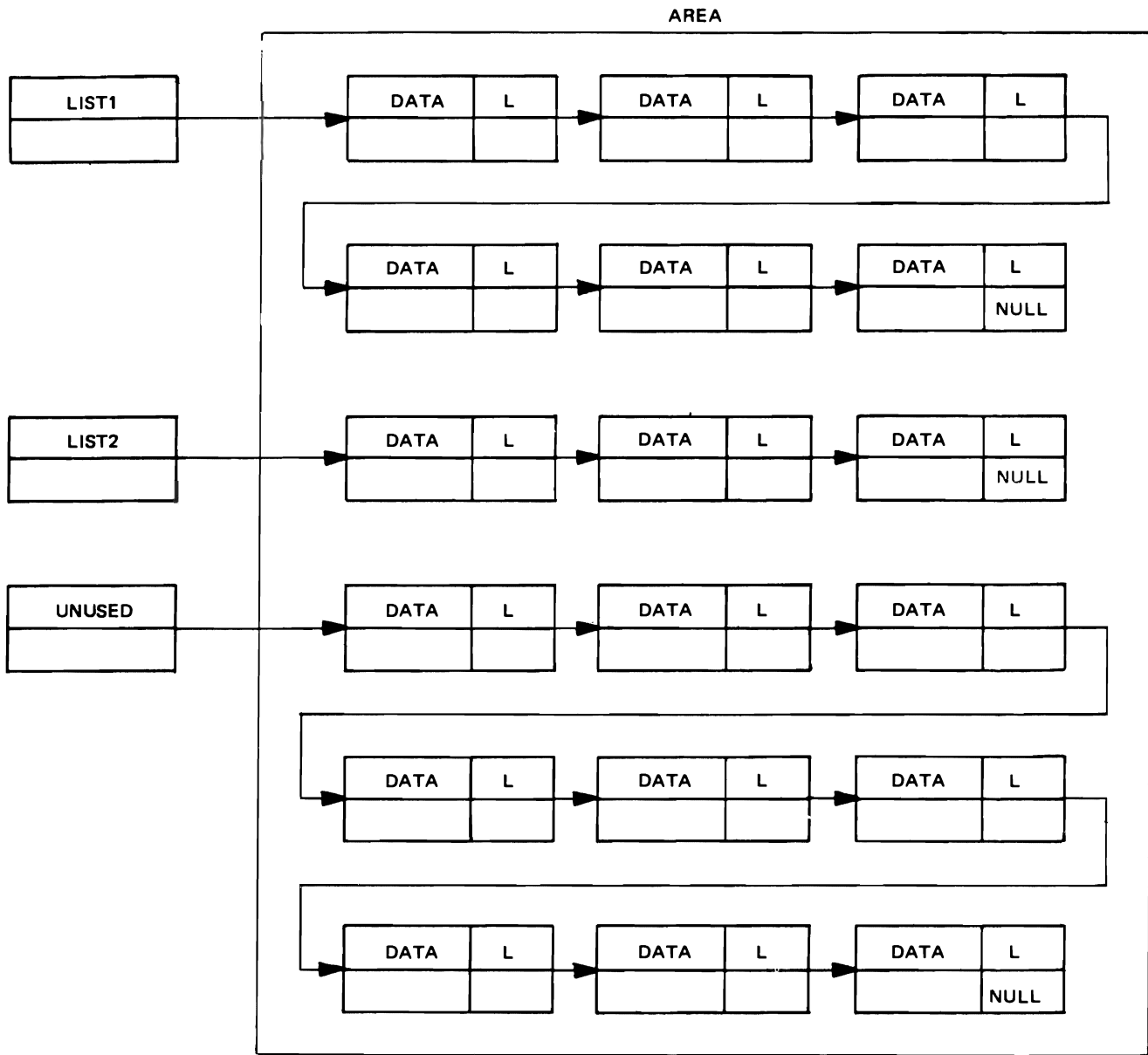


Figure 4C-2. Assigning unused storage components to other lists

#### 4D. SUMMARY OF CHAPTER 4

- A. The storage-class attribute **BASED** specifies a type of dynamic storage allocation.
- B. The **ALLOCATE** and **FREE** statements provide direct control over based storage and have the following general formats:

**ALLOCATE** based-variable SET(pointer-variable)  
IN(area-variable);  
**FREE** based-variable IN(area-variable);

- C. Successive allocations of storage for a based variable are not stacked as they are for a controlled variable. Two or more allocations for the same based variable may be referred to at the same time.
- D. The pointer variable specified in the **SET** option of an **ALLOCATE** statement receives the address of the based allocation.
- E. An **ALLOCATE** statement without a **SET** option causes the address of a based allocation to be assigned to the pointer variable in the **BASED** attribute of the allocated variable.
- F. The declaration of a self-defining based structure contains a **REFER** option, which has the following format:

element-variable **REFER**(element-variable)

This option determines the amount of storage involved in an allocation of the self-defining structure.

- G. The **REFER** option may appear only once in the declaration of a self-defining structure, and must occur either as a string length or as an array bound in the component at the end of the structure.
- H. When storage is allocated for a self-defining based structure, the **REFER** option causes the value of a var-

iable outside the structure to be assigned to a variable within the structure. This value also replaces the **REFER** option and determines the string length or array bound of the final component involved in the same allocation of the self-defining structure.

- I. The **AREA** attribute has the following form:

**AREA** (expression)

This attribute specifies an area variable, which may have any storage class and which reserves storage for allocations of based variables. The expression in the attribute determines the amount of storage (in bytes) that is reserved.

- J. The **IN** option of an **ALLOCATE** statement specifies the area within which storage is allocated for a based variable. When the **IN** option does not appear in an **ALLOCATE** statement, the operating system provides an area for the based allocation.
- K. The **IN** option must appear in a **FREE** statement if the based allocation was made within a specified area; otherwise, the option is omitted.
- L. An **AREA ON**-condition occurs when an attempt is made to allocate based storage in an area that does not contain enough free storage for the allocation.
- M. When an **ON**-unit appears in an **ON** statement for the **AREA** condition and a normal return occurs from the **ON**-unit, the **ALLOCATE** statement that raised the **AREA** condition is executed again. Looping will occur if the **ON**-unit does not increase the amount of free storage in the associated area.
- N. If no **ON**-unit appears in an **ON** statement for the **AREA** condition, the operating system issues a comment and raises the **ERROR** condition.

## Chapter 5. Facilities for Relocating Data Lists

The previous chapter showed how scattered data items may be linked with pointer variables to form list organizations that can be processed with a minimum of data movement. Many applications, however, require the components of a list to be moved collectively either between the internal and external storage devices of a computer, or solely within internal storage. But the list-processing techniques presented in the preceding chapter do not permit valid movement of a list as a unit, primarily because the values of pointer variables are absolute addresses which become invalid when the items they point to are moved to, or reallocated at, other storage locations. To overcome this limitation, PL/I provides special variables called offset variables, which allow list components to be linked by relative addresses that remain valid during list transmission and assignment. The linkage techniques developed in the preceding chapter still apply to relocatable lists, but offset variables are used in place of pointer variables.

This chapter shows how lists may be treated as collective units and how offset variables are used to form relocatable lists.

### 5A. TREATING LISTS AS UNITS WITHIN AREAS

PL/I allows the elements of an array or structure to be referenced collectively through an array or structure name. Similar reference, though, is not possible with a list, because a list is a data organization that is not explicitly known to PL/I. However, a list can be treated as a collective unit by referring to the area in which the list components have been allocated. Internal and external movement of a list then becomes possible by transmitting the containing area.

### 5B. ASSIGNING AREAS TO OTHER AREAS

The name of an area may be treated as an area variable, the value of which is the storage currently allocated for the area name. As a result, area variables can be used in assignment statements. When an area variable appears to the left of the equal sign in an assignment statement, the expression on the right must be another area variable or a function reference that returns an area. An area cannot be converted to any other type of data; therefore, an area can be assigned only to an area variable.

No operators, not even comparison operators, can be applied to area variables, and only the INITIAL CALL form of the INITIAL attribute may appear in an area declaration.

When an area is allocated, it automatically receives the empty state, which means that storage has not been allocated for any based variables within the area. An area that is not empty can be made empty by assigning it the value of an empty area, or the value of the built-in function `EMPTY`. The effect of such an assignment is to free all allocations of based variables within the receiving area. Note that the area itself does not become free but retains its storage in reserve for further allocations of based variables.

A reference to the built-in function `EMPTY` uses no arguments and has the following form:

`EMPTY`

An `EMPTY` reference cannot appear in an operational expression; its value is used solely to free storage allocated in a specified area.

### EXAMPLE

```
DECLARE
  (AREA1,AREA2) AREA(100),
  STRING BASED(P) CHARACTER(100);
.
.
.
ALLOCATE STRING IN(AREA1) SET(P);
P->STRING = (100)'X';
.
.
.
AREA2 = AREA1;
.
.
.
AREA1 = EMPTY;
.
.
.
```

In this example, both area variables `AREA1` and `AREA2` reserve 100 bytes of automatic storage. The based variable `STRING` is a 100-position character string, which is allocated in `AREA1` and to which a string of 100 X characters is assigned. When `AREA1` is assigned to `AREA2`, the contents of both areas become equal. Assignment of the `EMPTY` function to `AREA1` frees the storage allocated for `STRING` in `AREA1`.

Besides providing a convenient way of equating the contents of two or more area variables, area assignment also permits a list of based allocations to extend beyond the limits of a single area. For example, attempted allocation of a variable in a based area that contains insufficient free storage raises the AREA ON-condition. An associated ON-unit for the condition can then be used to assign the contents of the area to a reserve area, and to empty the original area for further allocations. Upon return from the ON-unit, the allocation can be attempted again within the original area.

**EXAMPLE**

```

DECLARE
(AREA1,AREA2) AREA(100),
STRING BASED(P) CHARACTER(100);
.
.
ON AREA
BEGIN;
  AREA2 = AREA1;
  AREA1 = EMPTY;
  GO TO A;
END;
.
.
A:  ALLOCATE STRING IN(AREA1) SET(P);
    P->STRING = (100)'X';
.
.
GO TO A;
.
.

```

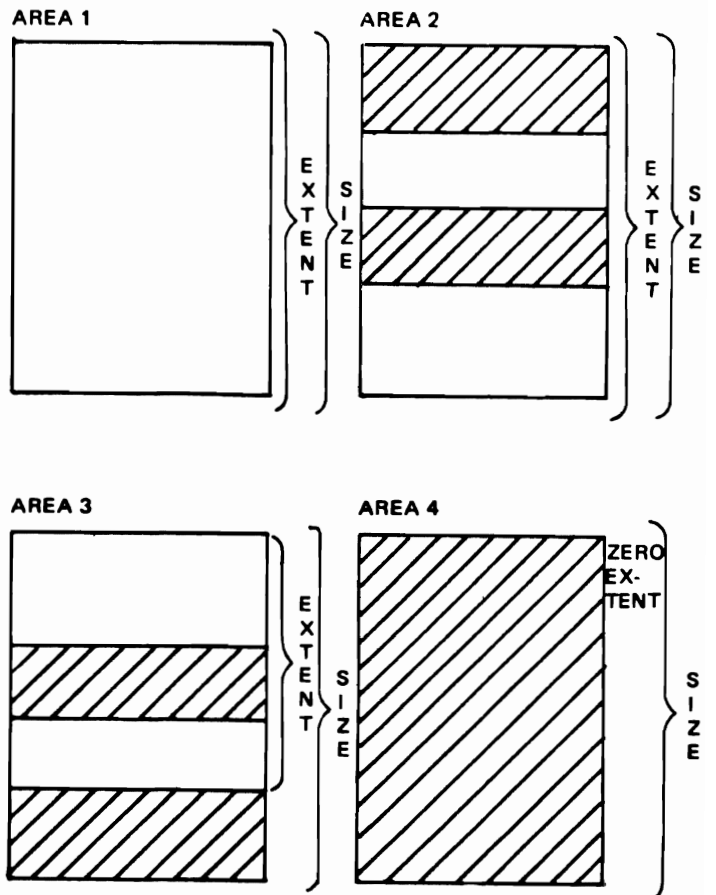
**EXAMPLE:**

In this example, both area variables AREA1 and AREA2 reserve 100 bytes of automatic storage. The based variable STRING is a 100-position character string, which is allocated in AREA1 and to which a string of 100 X characters is assigned. When more than one allocation of STRING is attempted in AREA1, the AREA ON-condition occurs, and control transfers to the specified ON-unit in the ON statement. The ON-unit saves the contents of AREA1 in AREA2, and empties AREA1 for further allocations. Control is then returned to the ALLOCATE statement, where another allocation of STRING is made in AREA1.

**5C. THE EXTENT OF AN AREA**

The amount of storage (in bytes) that is reserved when an area is allocated is called the *size* of the area and is specified by the integral value of the expression in the associated AREA attribute. The amount of storage that is currently allocated for based variables within an area determines the *extent* of the area, which is defined as the amount of storage between the start of the area and the end of the allocation most distant from the start. As a result, the extent of an area never exceeds the size of the area and even can be zero when the size is not zero; a reference to the built-in function EMPTY always produces an area of zero extent.

The diagrams in Figure 5C-1 illustrate the relationship between extent and size for several area configurations. Shaded portions of the diagrams represent free storage that is available for further allocation of based variables. In AREA1 of Figure 5C-1, all storage has been allocated; therefore the extent equals the size. In AREA2, the extent also equals the size, but the area contains free storage. In this case, the end of the allocation most distant from the start coincides with the end of the area. The extent of AREA3 is less than the size. AREA4 has zero extent, because no based storage is currently allocated within it.



(Shaded portions represent free storage)

Figure 5C-1. How area extent and area size are related

### 5D. THE EFFECT OF EXTENT ON AREA ASSIGNMENT

Assignment of an area effectively frees all allocations in the receiving area, and then assigns the extent of the area contents to the receiving area. All free-storage gaps are retained during area assignment, so that allocations within the assigned extent maintain their locations relative to each other. If the gaps were closed up, relative addressing techniques (which are discussed later) might become invalid.

Figure 5D-1 shows how area extents are assigned. The diagrams illustrate several area configurations both before and after assignment. Shaded portions of the diagrams again represent free storage. AREA1 and AREA2 have the same size but different extents. After assignment, both areas have the same size, extent, and contents. Note how the allocations in the assigned area retain their relative positions in the receiving area. AREA3 has a larger size than AREA4, but the extent of AREA3 is less than the size of AREA4; therefore, AREA3 can be assigned to AREA4 as indicated.

### 5E. THE AREA ON-CONDITION FOR AREA ASSIGNMENT

When the extent of an assigned area exceeds the size of the receiving area, an AREA ON-condition occurs, and the content of the receiving area becomes undefined. If the procedure does not contain an ON-unit for the AREA condition, the operating system issues a comment and raises the ERROR condition. When an ON-unit is specified and normal return occurs from the ON-unit, program execution continues from the point of interrupt. The same activity occurs when the AREA condition is raised by a SIGNAL statement. However, an AREA condition raised by an attempted allocation of storage produces different activity when a normal return occurs from the associated ON-unit (see paragraph 4B5D, "The AREA ON-Condition").

The ONCODE built-in function may be used to ascertain the type of situation that has raised an AREA ON-condition.

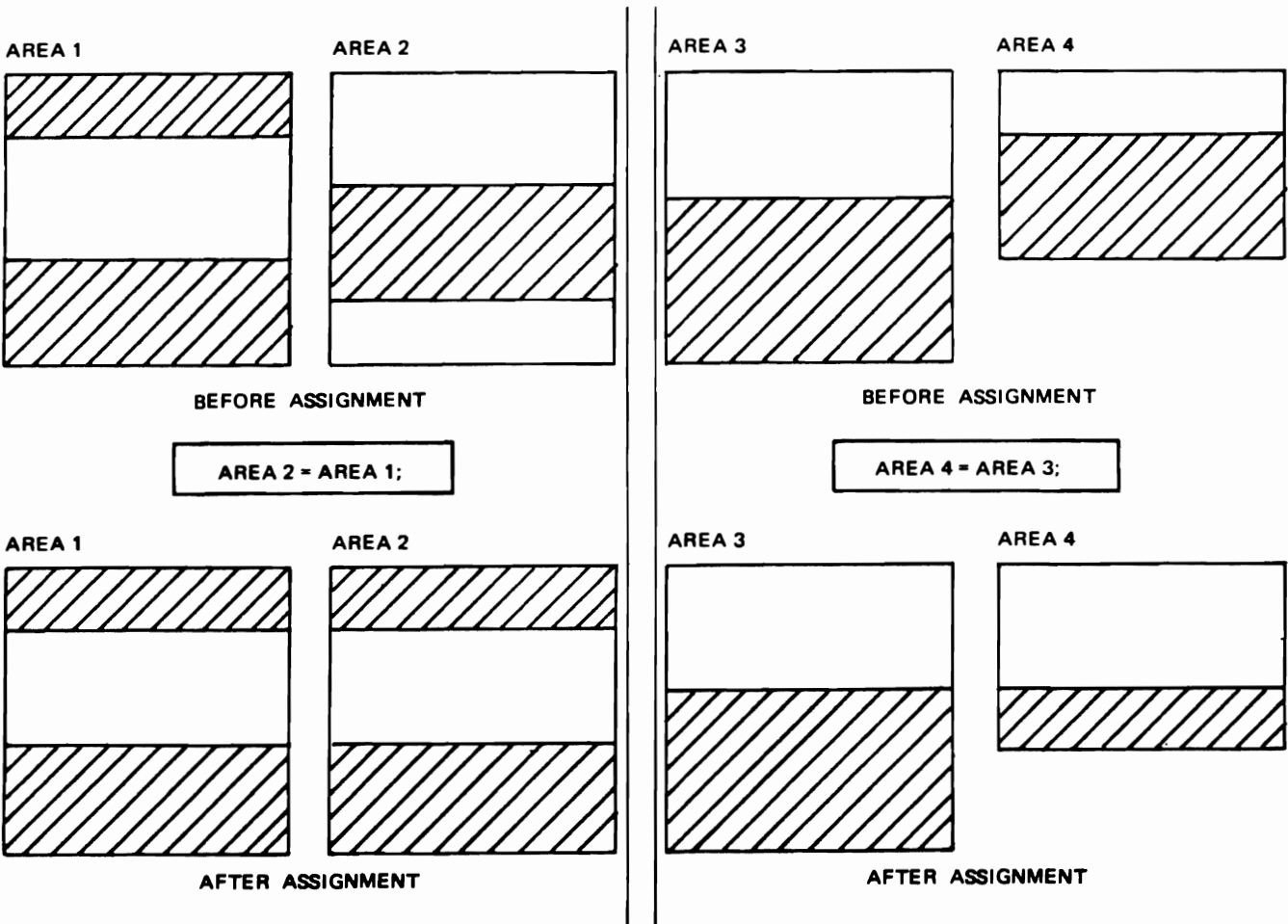


Figure 5D-1. How area extents are assigned

```

T_EXTENT:
F5F_1:
PROCEDURE OPTIONS (MAIN);
DECLARE
  A AREA (48),
  B BASED (P) CHARACTER (24),
  EXTENT RETURNS (FIXED (5)),
  COUNT FIXED (5);
  ALLOCATE B IN (A) SET (P);
  P -> B = (24) 'X';
  COUNT = EXTENT (A);
PUT LIST
  ('EXTENT OF A = '||COUNT||' BYTES');

EXTENT:
PROCEDURE
  (AREA) FIXED DECIMAL(5);
DECLARE
  AREA AREA (*),
  E FIXED DECIMAL (5),
  SPACE AREA(E) CONTROLLED;

  /* WHEN THE AREA CONDITION OCCURS,
  REALLOCATE SPACE WITH INCREASED SIZE,
  AND TRY ASSIGNMENT AGAIN. */
  (N AREA

BEGIN;
  FREE SPACE; E = E + 1;
  GO TO
  ALLOCATE_SPACE;
END;

  /* ALLOCATE SPACE WITH INITIAL SIZE
  OF ZERO. */
  E = 0;
ALLOCATE_SPACE:
  ALLOCATE SPACE;

  /* ASSIGN AREA TO SPACE. IF NOT
  POSSIBLE, AREA CONDITION OCCURS. */
  SPACE = AREA;

  /* WHEN THIS POINT IS REACHED, THE
  ASSIGNMENT IS SUCCESSFUL. FREE SPACE
  BEFORE RETURNING. */
  FREE SPACE;

  /* RETURN COMPUTED SIZE OF SPACE,
  WHICH EQUALS THE EXTENT OF AREA IN
  BYTES. */
  RETURN(E);
END
EXTENT;
END
T_EXTENT;

EXTENT OF A =      24 BYTES

```

Figure 5F-1. A function procedure that computes the extent of an area.

## 5F. COMPUTING THE EXTENT OF AN AREA

Area assignment can be used to compute the extent of an area. The function procedure EXTENT in Figure 5F-1 shows one way of performing such a computation and demonstrates how the AREA condition may be raised during area assignment. EXTENT contains the parameter AREA, which represents the area whose extent is to be computed. A reference to EXTENT produces a five-digit fixed-point decimal integer that specifies the extent in bytes.

The function uses a work area called SPACE, which initially has a zero size and to which AREA is assigned. When the size of AREA exceeds that of SPACE, the attempted assignment produces an AREA condition. The associated ON-unit reallocates SPACE with its size increased by one byte and reattempts the assignment. Repeated failure of the assignment causes the size (E) of SPACE to be increased until it is equal to the extent of AREA. At that point the function returns the computed extent. Faster computation of the extent is possible by increasing E with a large number of bytes, say 100, until E exceeds the extent of AREA. Rapid convergence to the extent is then produced by repeated halving of the two nearest values of E that straddle the extent.

## 5G. THE LENGTH OF AN AREA

The PL/I compiler associates each area variable with a control region that contains such information as the size of the area and the locations of free-storage gaps within the area. This control information is attached to the associated area contents when record-oriented input and output statements transmit an area variable. The length of the resulting record always exceeds the size of the associated area. The amount of additional storage required for area control information depends upon the implementation characteristics of the PL/I compiler and is obtained from the Programmer's Guide for the compiler.

To distinguish between area size and record length, PL/I uses the *length* of an area, which is the sum of the area size (specified in the AREA attribute) and the number of storage bytes required by the area control information. During record-oriented transmission of an area variable, it is the area length and not the area size that determines the record length. The programmer uses the record length in an ENVIRONMENT attribute for the associated file, or in a Data Definition statement under the Job Control Language for the Operating System.

## 5H. THE EFFECT OF AREA ASSIGNMENT ON POINTER VALUES

Pointer values contained in an area that is assigned to another area become invalid in the receiving area. This restriction also applies to null pointer values.

Correct transmission of a pointer value requires explicit assignment of the value to a receiving pointer variable. Later discussions present another method, which uses relocatable pointers called *offsets* to maintain the validity of pointers transmitted by area assignment.

## 5I. ADDRESSING THE CONTENTS OF AN ASSIGNED AREA

When a based variable is allocated in an area, the SET option in the ALLOCATE statement provides the address of the based allocation. Subsequent assignment of the area, however, does not provide a way of locating the based item in the new area. As an example, consider the following statements:

```
DECLARE
  ITEM BASED(P) CHARACTER(10),
  (AREA1,AREA2) AREA(100);
.
.
.
ALLOCATE ITEM IN(AREA1) SET(P);
AREA2 = AREA1;
.
.
.
```

Reference to ITEM in AREA1 may be made through the expression P->ITEM. Though AREA1 has been assigned to AREA2, a reference to ITEM in AREA2 is not possible, because the address of ITEM in AREA2 is not known and cannot be used to qualify ITEM. To overcome this difficulty, PL/I uses relative pointer variables called offset variables.

## 5J. OFFSET VARIABLES

Declaration of an offset variable must be explicit and is made with the OFFSET attribute, which has the following form:

OFFSET (area-variable)

The area variable in parentheses must be based and unscripted, and must have an implied or explicit level number of one in its declaration.

EXAMPLES:

```
DECLARE
  AREA1 AREA BASED(P1),
  AREA2 AREA(500) BASED(P2),
  O OFFSET(AREA1),
  (M, N) OFFSET(AREA2) EXTERNAL STATIC,
  SWITCH CONTROLLED OFFSET(AREA1),
  T(5) OFFSET(AREA2) INTERNAL,
  V(-2:2, -3:3) OFFSET(AREA2),
  1 A, 2 X CHARACTER(15), 2 Y OFFSET(AREA1),
  1 TABLES, 2 I(5) OFFSET(AREA2), 2 J(0:4)
  OFFSET(AREA1);
```

As shown in these examples, PL/I allows offset variables to be individual element variables, or elements of arrays and structures. An offset variable can have any storage class and scope, and the usual default rules for these attribute types also hold for an offset variable. The area variable in an OFFSET attribute must be explicitly declared. If it is not, the variable is assumed to be an area variable by its appearance in the OFFSET attribute. However, an error will occur, because the area variable receives the automatic storage class by default. This type of storage violates the requirement that the area variable in an OFFSET attribute must have the based storage class.

It is possible, however, to associate an offset variable with an area that is not based. Consider the following statements:

```
DECLARE
  AREA1 AREA(2000),
  DUMMY_AREA AREA(2000) BASED(DUMMY_
  POINTER),
  O OFFSET(DUMMY_AREA);
.
.
.
DUMMY_POINTER = ADDR(AREA1);
.
.
.
```

AREA1 and DUMMY\_AREA are area variables. AREA1 reserves automatic storage, and DUMMY\_AREA reserves based storage. The OFFSET attribute for variable O uses DUMMY\_AREA and thus satisfies the requirement that the area specified in an OFFSET attribute must be based. When the address of AREA1 is assigned to DUMMY\_POINTER, DUMMY\_AREA becomes equivalent to AREA1. Subsequent references to offset variable O are then effectively associated with AREA1.



The size of the area declared for DUMMY\_AREA in the previous example does not have to be the same as the size of AREA1, and can even be zero. The only purpose of DUMMY\_AREA is to provide a level-one based area variable for the OFFSET attribute of variable O, so that variable O can be made relative to the starting address of AREA1. The size of DUMMY\_AREA is unimportant, because it does not affect the starting address assigned to DUMMY\_AREA through DUMMY\_POINTER.

### 5J1. Assigning Values to Offset Variables

The value of an offset variable is a relative address, that is, an address which is relative to (or "offset" with respect to) the beginning of the area associated with the offset variable. Addresses that are relative to an area permit data items within the area to be linked in list form, so that the address linkage of the list does not become invalid when the content of the area is assigned to another area (this point is discussed in more detail in paragraph 5K).

An assignment statement can be used to assign a value to an offset variable, and the offset variable can receive the values of pointer variables as well as the values of offset variables. When the value of a pointer variable is assigned to an offset variable, the assigned value is converted to an offset value. The conversion is performed automatically by subtracting from the value of the pointer variable the absolute address of the area specified in the OFFSET attribute for the offset variable. The address arithmetic performed by the program is equivalent to the following calculation:

$$\text{offset value} = (\text{pointer value}) - (\text{absolute address of area})$$

The offset value produced by this calculation is assigned to the offset variable, but the pointer variable retains its original value (which is an absolute address).

Similarly, when an offset variable is assigned to a pointer variable, the offset value is converted to a pointer value. The offset value is effectively added to the absolute address of the area specified in the associated OFFSET attribute:

$$\text{pointer value} = (\text{offset value}) + (\text{absolute address of area})$$

The resulting pointer value is assigned to the pointer variable, but the offset variable retains its original value (which is a relative address).

Note that the foregoing address computations are performed automatically by the program. PL/I does not allow the programmer to use arithmetic operations to specify address computations.

When an offset variable is assigned to another offset variable, the offset value is assigned without modification. The offset variables involved in the assignment do not have to be associated with the same area. Consequently, an offset address can be made relative to more than one area; it is this facility that permits the construction of relocatable data lists (discussed in paragraph 5K).

The following example shows how address values are assigned to offset variables; it also shows how to obtain the absolute address of an item located in an assigned area:

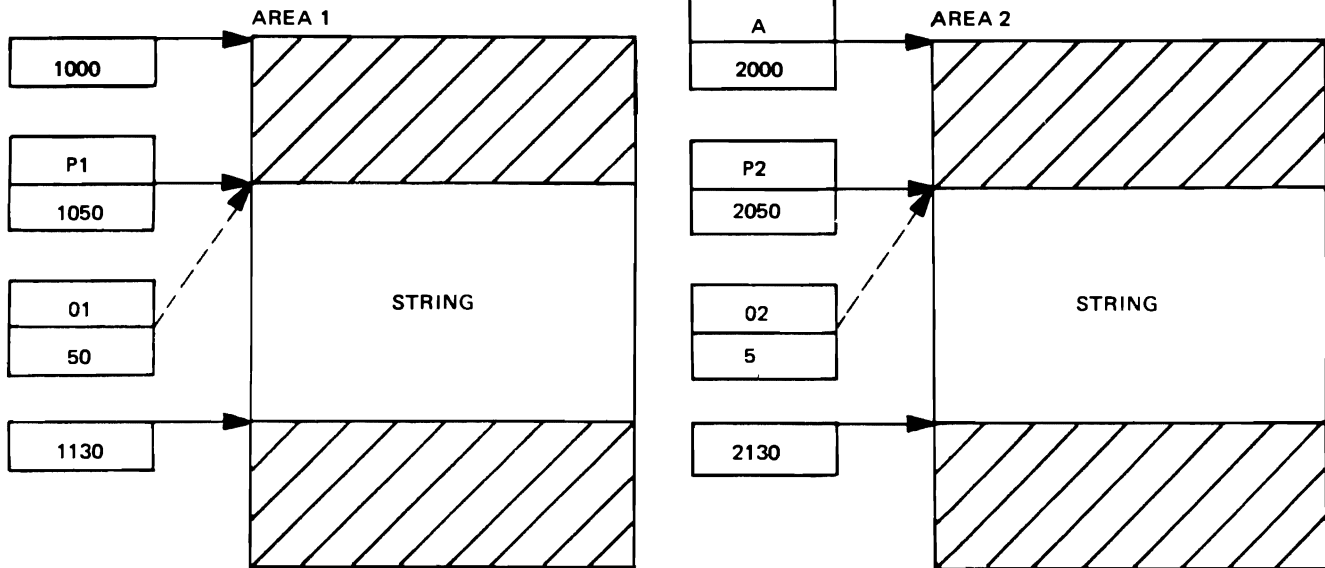
```

DECLARE
  AREA1 AREA(500),
  AREA2 BASED(A) AREA(500),
  DUMMY_AREA BASED(DUMMY_POINTER) AREA,
  (P1, P2) POINTER,
  O1 OFFSET(DUMMY_AREA),
  O2 OFFSET(AREA2),
  STRING BASED(P1) CHARACTER(80);
.
.
.
DUMMY_POINTER = ADDR(AREA1);
ALLOCATE STRING IN(AREA1) SET(P1);
ALLOCATE AREA2 SET(A);
.
.
.
A->AREA2 = AREA1;
O1 = P1;
O2 = O1;
P2 = O2;
.
.
.

```

AREA1 reserves 500 bytes of automatic storage, and AREA2 reserves 500 bytes of based storage. An implementation-defined amount of based storage is reserved by DUMMY\_AREA. AREA2 is itself allocated in an area provided by the operating system; pointer A specifies the location of AREA2. STRING is an 80-position based character-string allocated in AREA1, and pointer P1 specifies the address of STRING in AREA1. When the address of AREA1 is assigned to DUMMY\_POINTER, offset variable O1 becomes associated with AREA1.

After AREA1 is assigned to A->AREA2, both areas contain equivalent storage configurations. Assignment of pointer P1 to offset O1 produces the relative address of STRING in AREA1. This relative address remains unchanged when assigned to offset O2. Assignment of O2, in



```

ALLOCATE STRING IN(AREA1) SET(P1);          /*ALLOCATE STRING IN AREA1, and SET P1.*/
ALLOCATE AREA2 SET(A);                     /*ALLOCATE AREA2, AND SET A.*/

.
.
A -> AREA2 = AREA1;                         /*ASSIGN AREA1 TO A -> AREA2.*/
O1 = P1;                                    /*SET O1 TO RELATIVE ADDRESS OF STRING IN AREA1.*/
O2 = O1;                                    /*SET O2 EQUAL TO O1.*/
P2 = O2;                                    /*SET P2 TO ABSOLUTE ADDRESS OF STRING IN A -> AREA2.*/

```

Figure 5J1-1. Obtaining the address of a data item in an assigned area

turn, to P2 produces the absolute address of STRING in A->AREA2. Reference to STRING in A->AREA2 then becomes possible with the expression P2->STRING.

Figure 5J1-1 uses assumed addresses to illustrate the relationship between AREA1 and A->AREA2. The figure assumes AREA1 is located at address 1000, and A->AREA2 at address 2000. STRING is further assumed to be located in AREA1 at address 1050, which becomes the value of P1. Assignment of AREA1 to A->AREA2 places STRING in A->AREA2 at location 2050. Each occurrence of STRING occupies the same relative position (50th location) in both AREA1 and A->AREA2.

When P1 is assigned to O1, O1 receives the relative address value (50 = 1050 - 1000) of P1 with respect to AREA1 rather than the absolute value (1050) of P1. O2 receives the relative address 50 through assignment of O1 to O2. P2 then receives the absolute address value (2050 = 50 + 2000) of O2 when O2 is assigned to P2. Note that direct assignment of O1 to P2 would not be correct, because O1 is relative to AREA1 and not to A->AREA2; assignment of O1 to P2 produces an absolute address of 1050 rather than the desired 2050.

The broken lines in Figure 5J1-1 indicate offset variables to help distinguish them from pointer variables.

### 5J2. The NULLO Built-In Function

A null offset value can be assigned to an offset variable through the built-in function NULLO, which requires no arguments and has the following form:

NULLO

A reference to this function produces a null offset address, which specifies no relative storage location. A null offset address can be used to indicate the end of a list of components linked by offset addresses.

Although pointer values may be assigned to offset variables, and offset values, in turn, may be assigned to pointer variables, a null offset value cannot be assigned to a pointer variable. Similarly, a null pointer value cannot be assigned to an offset variable in place of a null offset value. These restrictions apply not only to explicit references to NULLO and NULL but also to assigned variables that currently have null offset and null pointer values.

As an illustration, if P is a pointer variable and O is an offset variable, P can be assigned to O provided P does not have a null value. When the value of P might be null, an IF statement may be used to ensure proper assignment:

```

IF P = NULL
  THEN O = NULLO;
  ELSE O = P;

```

A similar statement governs the correct assignment of O to P:

```

IF O = NULLO
  THEN P = NULL;
  ELSE P = O;

```

### 5J3. Restrictions on Offset Variables

The restrictions on pointer variables presented in Chapter 3 also apply to offset variables, with the addition of the following points:

- A. An offset variable cannot qualify a based variable. The offset value must first be assigned to a pointer variable, which is then used to qualify the based variable.
- B. Offset values and pointer values form a special type of program control data called the *locator* type. Locator data cannot be converted to any other type, nor can any other type of data be converted to locator type. Offset variables can receive offset and pointer values only; the same restriction applies to pointer variables.
- C. Locator variables can appear as arguments and parameters. An offset argument associated with an offset parameter, or a pointer argument associated with a pointer parameter requires no conversion and therefore produces no dummy argument. But an offset argument associated with a pointer parameter, or a pointer argument associated with an offset parameter does require conversion and will produce a dummy argument. Also, when an offset argument is associated with an offset parameter, both must be offset with respect to the same area for the argument-parameter association to be meaningful.
- D. The DEFINED attribute can be used for overlay or correspondence defining of an offset variable on another offset variable. The area variables named in the OFFSET attributes of the two offset variables need not be the same. The DEFINED attribute also applies to pointer variables. However, an offset cannot be defined on a pointer, nor can a pointer be defined on an offset.
- E. As with pointer variables, the comparison operators equal (=) and not equal ( $\neq$ ) are the only two operators that can use offset variables as operands.

## 5K. RELOCATION OF DATA LISTS

Once the components of a list have been allocated and linked within an area, references to the area name allow the list to be treated as a unit. The advantage of being able to refer to the list as a unit is that area assignment can be used to move the list to another area, such as a work area. The name of the area also permits collective transmission of the list to an external storage medium, such as magnetic tape or magnetic disk, from which the list can be retrieved for further processing. If area assignment were not available, the components of a list would have to be reallocated and linked individually within the new area. Moving the list in this manner would generally take more time than is required for direct assignment of the area.

A list that is linked by pointer values, however, does not remain properly linked when it has been moved. The components of the assigned list have new addresses that are not specified in the linking pointers of the list. Adjustment must be made to the values of the linking pointers to maintain proper linkage of the list. The following discussions show how these adjustments are made for both internal and external relocation.

### 5K1. Internal Relocation

Chapter 4 showed how based allocations may be linked in an area to form a data list. Relocation of such a list within internal storage is performed, in part, by assigning the area that contains the list to a receiving area. As discussed previously, however, the pointer values that link the components of the list become invalid in the receiving area when transmission is performed by area assignment.

Consider, for example, the incorrect area assignment illustrated in Figure 5K1-1. B1->BODY1 is a based area that contains a three-component data list, which is assigned to based area B2->BODY2. For continuity with previous examples, the list components in this figure have the same structure organization used in Chapter 4. In the figure, pointer HEAD1 specifies the location of the first list component in B1->BODY1. Offsets OHEAD1 and OHEAD2 are relative to BODY1 and BODY2 and are used to obtain the address value of pointer HEAD2, which specifies the location of the first list component in B2->BODY2. As before, broken lines indicate offset address values.

The following statements perform the assignment:

```

B2->BODY2 = B1->BODY1;
OHEAD1 = HEAD1;
OHEAD2 = OHEAD1;
HEAD2 = OHEAD2;

```

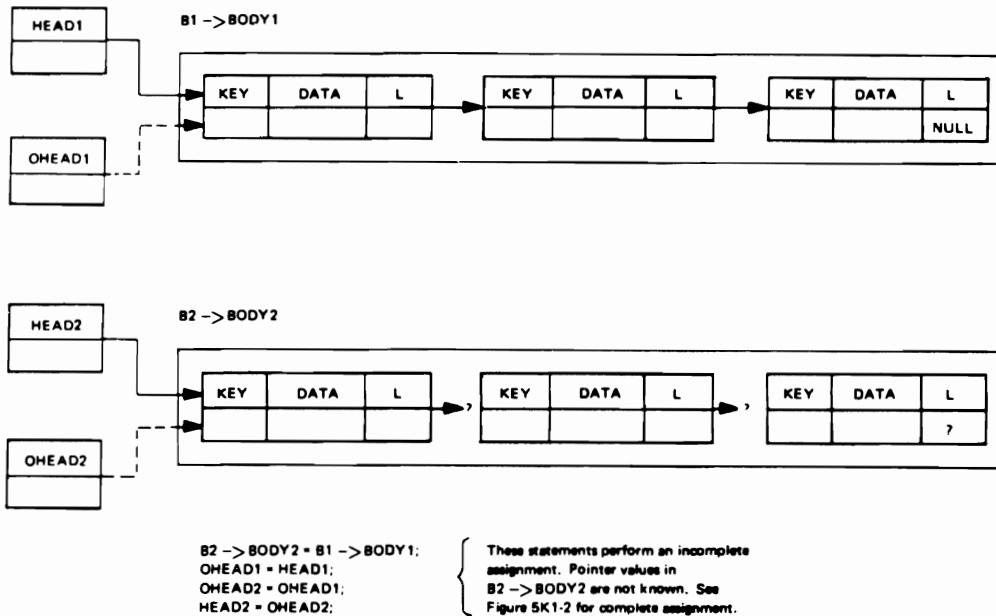


Figure 5K1-1. Incorrect assignment of a data list

These statements assign the data list in  $B1 \rightarrow BODY1$  to  $B2 \rightarrow BODY2$  and obtain the proper value for  $HEAD2$ , but they do not maintain the validity of the component pointers in the receiving area  $B2 \rightarrow BODY2$ . The pointers in  $B2 \rightarrow BODY2$  have unknown values (indicated by the question marks) and, therefore, do not link the list components. Note further that the null pointer in the last component also becomes invalid in the receiving area.

Even if the pointer values in the source area were transmitted to the receiving area without modification, the transmitted list would still be incorrectly linked, because the pointers would continue to specify the list components in the source area and not those in the receiving area.

Proper assignment of the data list appears in Figure 5K1-2, which illustrates the effect of the following statements:

```

B2->BODY2 = B1->BODY1;
OHEAD1 = HEAD1;
OHEAD2 = OHEAD1;
HEAD2 = OHEAD2;
TEMP1 = HEAD1;
TEMP2 = HEAD2;
DO WHILE(TEMP1->L = NULL);
  OHEAD1 = TEMP1->L;
  OHEAD2 = OHEAD1;
  TEMP2->L = OHEAD2;
  TEMP1 = TEMP1->L;
  SAVE = TEMP2;
  TEMP2 = TEMP2->L;
END;
SAVE->L = NULL;
  
```

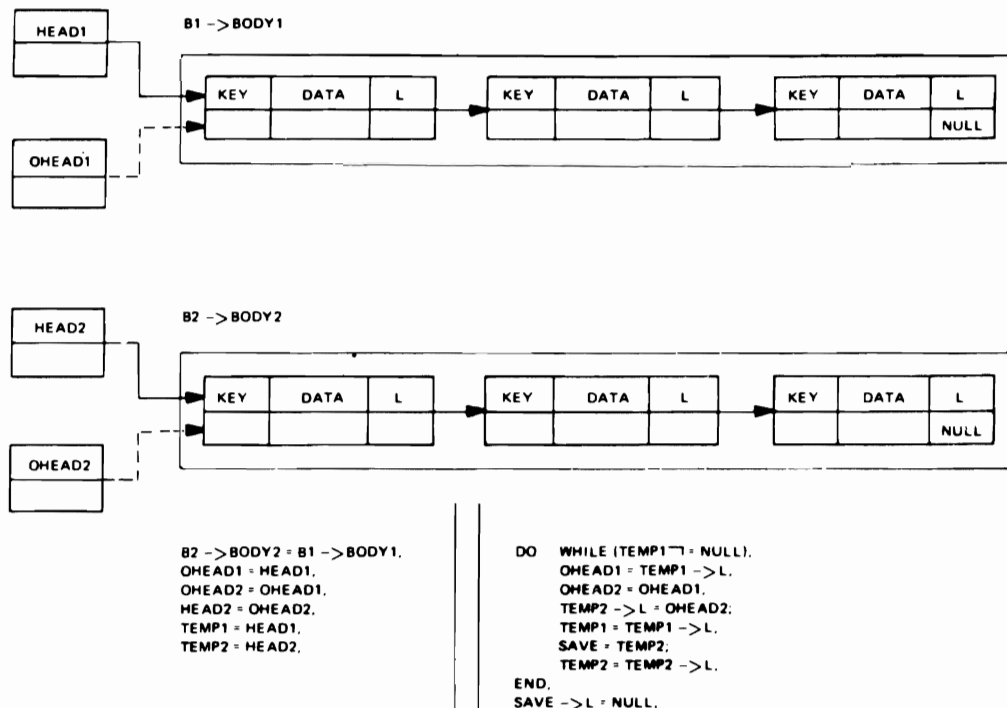


Figure 5K1-2. Correct assignment of a data list

These statements not only assign the data list in B1->BODY1 to B2->BODY2 but also adjust the values of the component pointers (the L's) in the receiving area. Pointers TEMP1, TEMP2, and SAVE serve as work pointers that specify the addresses of successive list components in B1->BODY1 and B2->BODY2.

The statements illustrated in Figure 5K1-2 can be incorporated into a subroutine, as shown by procedure MOVE\_L in Figure 5K1-3. MOVE\_L assigns the data list in one area to another area. The routine uses four parameters: BODY1, HEAD1, BODY2, and HEAD2. BODY1 represents the area that contains the data list to be assigned to BODY2. Pointer HEAD1 specifies the location of the first list component in BODY1. Similarly, pointer HEAD2 specifies the location of the first list component in BODY2 after the list has been assigned.

The area arguments associated with parameters BODY1 and BODY2 can be of any storage class, but MOVE\_L assumes that storage has been allocated for the arguments before invocation. If BODY2 is not large enough to receive the contents of BODY1, pointer HEAD2 receives a null value. The subroutine uses based areas DUMMY\_BODY1 and DUMMY\_BODY2 in the declarations of offsets OHEAD1 and OHEAD2 and overlays these areas on BODY1 and BODY2. As a result, BODY1 and BODY2

need not be of the based storage class, as would be required if they appeared in the OFFSET attributes of OHEAD1 and OHEAD2.

## 5K2. Relocatable Data Lists

The techniques used so far for assigning a data list from one area to another prevent the assignment from being direct. Not only must an area assignment be performed, but the pointer links of the assigned list must also be modified. A way of avoiding the additional programming needed for this pointer modification is to use offset variables, rather than pointer variables, as component links in the data list. Assignment of the list, then, requires no change in the values of the offset links, because the list components maintain their relative positions in the receiving area. Even the head of the list can be an offset variable, in which case the relative address of the first list components can also be assigned directly to the receiving head.

To distinguish between lists that are linked by pointer variables and lists that are linked by offset variables, the term *absolute list* is applied to a list linked by pointers (because pointer addresses are absolute addresses), and the term *relocatable list* is applied to a list linked by offsets (because offset addresses are relative addresses and hence relocatable).

```

T_MOVE_L:
PROCEDURE OPTIONS (MAIN);
DECLARE
  BODY_1 AREA (500),
  BODY_2 AREA (500),
  1 IMAGE BASED (S),
  2 KEY CHARACTER (3),
  2 DATA CHARACTER (7),
  2 L POINTER,
  1 CARD,
  2 KEY CHARACTER (3),
  2 DATA CHARACTER (7),
  MOVE_L ENTRY (AREA (500), PTR,
                AREA (500), PTR),
  (HEAD_1, HEAD_2, D,P,Q,H,T) POINTER;

  S = NULL;
  ALLOCATE IMAGE IN (BODY_1) SET (S);
  D,P,R,T = S;

DU
  I = 1 TO 4;
GET
  EDIT (CARD) (A (10));
  ALLOCATE IMAGE IN (BODY_1) SET (S);
  T -> IMAGE.L = S;
  T = S;
END;
  T -> IMAGE.L = NULL;
ASSIGN:
DU
  WHILE (R = NULL);
  R -> IMAGE = CARD, BY NAME;
  R = R -> IMAGE.L;
END
  ASSIGN:
PUT
  SKIP (2) LIST ('CONTENT OF BODY_1:');
DU
  WHILE (D = NULL);
PUT
  SKIP EDIT (D -> IMAGE.KEY,
            D -> IMAGE.DATA) (A, X(4), A);
  D = D -> IMAGE.L;
END;
  HEAD_1 = P;
  BODY_2 = EMPTY;
  HEAD_2 = NULL;

  CALL MOVE_L (BODY_1, HFAU_1,
              BODY_2, HFAU_2);

OUTPUT:
PUT
  SKIP (2) LIST
  ('KEY AND DATA ITEMS IN BODY_2:');

  Q = HEAD_2;
DU
  WHILE (Q = NULL);
PUT
  SKIP EDIT (Q -> IMAGE.KEY,
            Q -> IMAGE.DATA) (A, X(2), A);
  Q = Q -> IMAGE.L;
END;

MOVE_L:
PROCEDURE
  (BODY1, HEAD1, BODY2, HEAD2);
DECLARE
  BODY1 AREA(*),
  BODY2 AREA(*),
  DUMMY_BODY1 AREA BASED (D1),
  DUMMY_BODY2 AREA BASED (D2),
  OHEAD1 OFFSET(DUMMY_BODY1),
  OHEAD2 OFFSET(DUMMY_BODY2),
  1 IMAGE BASED (S),
  2 KEY CHARACTER (3),
  2 DATA CHARACTER (7),
  2 L POINTER,
  (HEAD1, HEAD2, TEMP1, TEMP2, SAVE)
  POINTER;
/*

```

```

IF
  THE AREA CONDITION OCCURS, BODY2 IS
  TOO SMALL TO RECEIVE THE CONTENTS OF
  BODY1. SET HEAD2 TO NULL, AND
  RETURN. */
  ON AREA
BEGIN;
  HEAD2 = NULL;
  GO TO
  END_MOVE_L;
END;
  /* ASSIGN BODY1 TO BODY2. IF HEAD1
  IS NULL, SET HEAD2 TO NULL, AND
  RETURN. */
  BODY2 = BODY1;
  IF
  HEAD1 = NULL
  THEN
  DU;
  HEAD2 = NULL; RETURN;
END;
  /* OVERLAY DUMMY_BODY1 ON BODY1 AND
  DUMMY_BODY2 ON BODY2 SO THAT THE
  OFFSETS OHEAD1 AND OHEAD2 ARE
  RELATIVE TO BASED AREAS. */
  D1 = ADDR(BODY1);
  D2 = ADDR(BODY2);

  /* ASSIGN ADDRESS OF FIRST LIST
  COMPONENT IN BODY2 TO HEAD2. */
  UHEAD1 = HEAD1;
  UHEAD2 = OHEAD1;
  HEAD2 = OHEAD2;
  /* ADJUST LINK POINTERS OF LIST IN
  BODY2. */
  TEMP1 = HEAD1;
  TEMP2 = HEAD2;
DU
  WHILE (TEMP1 = NULL);
  OHEAD1 = TEMP1->L;
  OHEAD2 = OHEAD1;
  TEMP2->L = OHEAD2;
  TEMP1 = TEMP1->L;
  SAVE = TEMP2;
  TEMP2 = TEMP2->L;
END;
  SAVE->L = NULL;
END_MOVE_L:
END
  MOVE_L;
END
  T_MOVE_L;

```

```

CONTENT OF BODY_1:
569 5690120
569 5690120
569 5690120
569 5690120
569 5690120

KEY AND DATA ITEMS IN BODY_2:
569 5690120
569 5690120
569 5690120
569 5690120
569 5690120

```

Figure 5K1-3. A subroutine procedure that assigns a data list to another list area

Figure 5K2-1 illustrates the organization of a relocatable data list; as usual, broken lines indicate offset variables. The figure also shows how list assignment may be performed with the following statements:

```
B2->BODY2 = B1->BODY1;
OHEAD2 = OHEAD1;
```

These statements assume that the offset links (the OL's) in based area B1->BODY1 and the offset head OHEAD1 are relative to B1->BODY1. Similarly, the offset links in B2->BODY2 and the offset head OHEAD2 are assumed to be relative to based area B2->BODY2. The following statement establishes the proper declaration for the variables used in the figure:

```
DECLARE
  BODY1 BASED(B1) AREA,
  BODY2 BASED(B2) AREA,
  DUMMY_BODY BASED(DUMMY_POINTER) AREA,
  OHEAD1 OFFSET(BODY1),
  OHEAD2 OFFSET(BODY2),
  1 COMPONENT BASED(COMPONENT_POINTER),
  2 KEY CHARACTER(3),
  2 DATA CHARACTER(77),
  2 OL OFFSET(DUMMY_BODY);
```

Declaration of offset variable OL with respect to DUMMY\_BODY permits OL to be made relative to any area by assigning the address of the desired area to DUMMY\_

POINTER. Arbitrarily, a default area size has been assumed for BODY1 and BODY2. If desired, a specific area size can be specified for each area.

By printing the KEY and DATA values of each list component in B1->BODY1 the following statements show how references are made to the elements of a relocatable list:

```
DUMMY_POINTER = ADDR(B1->BODY1);
COMPONENT_POINTER = OHEAD1;
DO WHILE(COMPONENT_POINTER ≠ NULL);
  PUT LIST(COMPONENT_POINTER->KEY,
           COMPONENT_POINTER->DATA);
  PUT SKIP;
  IF COMPONENT_POINTER->OL = NULLO
    THEN COMPONENT_POINTER = NULL;
  ELSE COMPONENT_POINTER =
    COMPONENT_POINTER->OL;
END;
```

Similar statements allow printing of the KEY and DATA values of each list component in B2->BODY2:

```
DUMMY_POINTER = ADDR(B2->BODY2);
COMPONENT_POINTER = OHEAD2;
DO WHILE(COMPONENT_POINTER ≠ NULL);
  PUT LIST(COMPONENT_POINTER->KEY,
           COMPONENT_POINTER->DATA);
  PUT SKIP;
  IF COMPONENT_POINTER->OL = NULLO
    THEN COMPONENT_POINTER = NULL;
  ELSE COMPONENT_POINTER =
    COMPONENT_POINTER->OL;
END;
```

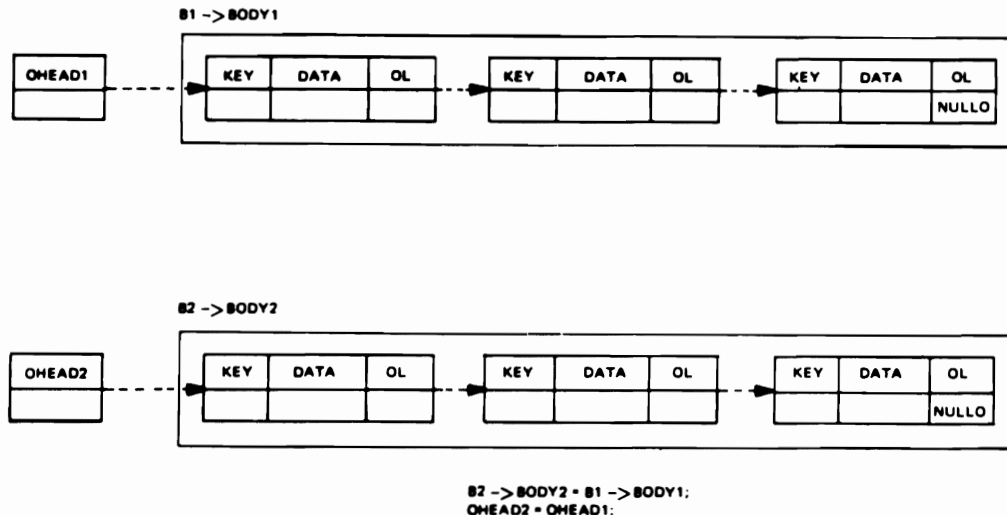


Figure 5K2-1. Assigning a relocatable data list

In each of the previous sets of statements, the address assigned to DUMMY\_POINTER determines the area to which the offset link OL becomes relative: area B1-> BODY1 in the first set of statements, area B2->BODY2 in the second set. Conversion of offset addresses to absolute addresses is performed by assigning the offsets to the pointer variable COMPONENT\_POINTER.

### 5K3. A Subroutine that Assigns a Relocatable Data List to Another Area

The techniques developed in the previous discussion for assigning a relocatable data list to another area may be summarized in subroutine form, as shown by subroutine procedure MOVE\_RL in Figure 5K3-1. MOVE\_RL uses four parameters: BODY1, OHEAD1, BODY2, and OHEAD2. BODY1 represents the area that contains the data list to be assigned to area BODY2. The relative location of the first list component in BODY1 is given by offset variable OHEAD1. Similarly, OHEAD2 specifies the relative location of the first list component in BODY2 after the assignment has been made. If BODY2 is not large enough to accept the contents of BODY1, OHEAD2 receives a null offset value. Under no circumstance does MOVE\_RL change the contents of BODY1.

BODY1 and BODY2 can be of any storage class, but storage is assumed to have been allocated for the corresponding area arguments upon entry to the subroutine. DUMMY\_BODY1 and DUMMY\_BODY2 are used to supply based areas for the OFFSET attributes of OHEAD1 and OHEAD2.

The following statements show how MOVE\_RL may be invoked:

```

.
.
.
DECLARE
LIST_BODY1 AREA(500),
LIST_BODY2 AREA(500),
DUMMY1 BASED(DUM1) AREA,
DUMMY2 BASED(DUM2) AREA,
LIST_OHEAD1 OFFSET(DUMMY1),
LIST_OHEAD2 OFFSET(DUMMY2);
.
.
DUM1 = ADDR(LIST_BODY1);
DUM2 = ADDR(LIST_BODY2);
.
.
CALL MOVE_RL(LIST_BODY1, LIST_OHEAD1,
LIST_BODY2, LIST_OHEAD2);
.
.

```

```

T_MOVE_RL:
PROCEDURE OPTIONS (MAIN);
DECLARE
L_BODY1 AREA (500),
L_BODY2 AREA (500),
DUM_BODY1 BASED (DUM1) AREA,
DUM_BODY2 BASED (DUM2) AREA,
L_OHEAD1 OFFSET (DUM_BODY1),
L_OHEAD2 OFFSET (DUM_BODY2),
I ITEM BASED (P),
Z CONSTANT CHARACTER (1),
Z OL OFFSET (DUM_BODY1),
Q POINTER,
MOVE_RL ENTRY (AREA (500),
OFFSET (DUM_BODY1), AREA (500),
OFFSET (DUM_BODY2));

DUM1 = ADDR (L_BODY1);
ALLOCATE ITEM IN (L_BODY1) SET (P);
L_OHEAD1, Q = P;
P -> CONSTANT = 'NEWMOVE';

DO
I = 2 TO 5;
ALLOCATE ITEM IN (L_BODY1) SET (P);
P -> CONSTANT = 'MOVE_RL';
Q -> OL = P; Q = P;
END;

Q -> OL = NULLO;
DUM2 = ADDR (L_BODY2);
L_BODY2 = EMPTY;
L_OHEAD2 = NULLO;

CALL MOVE_RL (L_BODY1, L_OHEAD1,
L_BODY2, L_OHEAD2);
DUM1, DUM2 = ADDR (L_BODY2);

OUTPUT:
PUT
SKIP LIST ('CONSTANTS IN L_BODY2');
IF
L_OHEAD2 = NULLO
THEN
Q = NULL;
ELSE
Q = L_OHEAD2;
DO
WHILE ( Q -> OL -> CONSTANT );
PUT
SKIP LIST (Q -> CONSTANT);
IF
Q -> OL = NULLO
THEN
Q = NULL;
ELSE
Q = Q -> OL;
END;
MOVE_RL:
PROCEDURE
(BODY1, OHEAD1, BODY2, OHEAD2);
DECLARE
BODY1 AREA(*), BODY2 AREA(*),
DUMMY_BODY1 BASED(D1) AREA,
DUMMY_BODY2 BASED(D2) AREA,
OHEAD1 OFFSET(DUMMY_BODY1),
OHEAD2 OFFSET(DUMMY_BODY2);
/*
IF
AREA CONDITION OCCURS, BODY2 IS TOO
SMALL TO RECEIVE CONTENTS OF BODY1.
SET OHEAD2 TO NULLO, AND GO TO END
OF SUBROUTINE. */
ON AREA
BEGIN;
OHEAD2 = NULLO;
GO TO
END_MOVE_RL;
END;
/* ASSOCIATE OHEAD1 AND OHEAD2 WITH
BODY1 AND BODY2. */
D1 = ADDR(BODY1);
D2 = ADDR(BODY2);
/* ASSIGN BODY1 AND OHEAD1 TO BODY2
AND OHEAD2. */
BODY2 = BODY1;
OHEAD2 = OHEAD1;
END_MOVE_RL:
END
CONSTANTS IN L_BODY2:
NEWMOVE
MOVE_RL
MOVE_RL
MOVE_RL
MOVE_RL
MOVE_RL
MOVE_RL

```

Figure 5K3-1. A subroutine procedure that assigns a relocatable data list to another area .



Because areas LIST\_BODY1 and LIST\_BODY2 have the automatic storage class and not the based storage class, based areas DUMMY1 and DUMMY2 are used to associate offset variables LIST\_OHEAD1 and LIST\_OHEAD2 with LIST\_BODY1 and LIST\_BODY2.

#### 5K4. Converting Data Lists to and from Relocatable Form

Although the previous discussions have shown how to assign relocatable data lists from one area to another, they have not shown how to create a relocatable list. The techniques developed in Chapter 4 for creating data lists dealt solely with lists in absolute form and did not consider the construction of relocatable lists. One way of creating a relocatable list is to convert an established absolute list to relocatable form. Subroutine procedure CONV\_AR in Figure 5K4-1 shows how this conversion can be made.

CONV\_AR uses four parameters: BODY1, HEAD, BODY2, and OHEAD. BODY1 represents the area that contains the absolute data list to be converted to relocatable form and assigned to area BODY2. Pointer HEAD specifies the address of the first list component in BODY1, and offset OHEAD receives the relative address of the first list component in BODY2 after the list has been converted. If the extent of BODY2 is not large enough to accept the contents of BODY1, OHEAD receives a null offset value. BODY1 and BODY2 can be of any storage class, but storage is assumed to have been allocated for the corresponding area arguments upon entry to the subroutine. The offset argument corresponding to OHEAD must be declared relative (either directly or through a dummy based area) to the area argument associated with BODY2. Execution of CONV\_AR does not change the contents of BODY1.

It is also possible to convert a relocatable data list to absolute form, as shown by subroutine procedure CONV\_RA in Figure 5K4-2. Again, this subroutine uses four parameters: BODY1, OHEAD, BODY2, and HEAD. BODY1 represents the area that contains the relocatable data list to be converted to absolute form and assigned to BODY2. Offset OHEAD specifies the relative address of the first list component in BODY1, and pointer HEAD receives the absolute address of the first list component in BODY2 after the list has been converted. If the extent of BODY2 is not large enough to accept the contents of BODY1, HEAD receives a null pointer value. Execution of CONV\_RA does not change the contents of BODY1, and the same restrictions apply to the parameters of CONV\_RA as they do to CONV\_AR in Figure 5K4-1. A later discussion in paragraph 5K6 under "External Relocation" shows how relocatable data lists can be constructed directly without being converted from absolute form.

```
T_CONV_RA:
T_CONV_AR:
PROCEDURE OPTIONS (MAIN);
DECLARE
  (BODY_1, BODY_2) AREA (500),
  DUMMY_BODY BASED (DUMMY_POINTER) AREA,
  (L_HEAD, Q) POINTER,
  L_OHEAD OFFSET (DUMMY_BODY),
  1 LIST BASED (P),
  2 KEY CHARACTER (3),
  2 DATA CHARACTER (7),
  2 L POINTER,
  CONV_AR ENTRY (AREA (500), POINTER,
  AREA (500), OFFSET (DUMMY_BODY)),
  CONV_RA ENTRY (AREA (500), OFFSET
  (DUMMY_BODY), AREA (500), POINTER);

  P = NULL;
  ALLOCATE LIST IN (BODY_1) SET (P);
  L_HEAD, Q = P;
  P->KEY = '444'; P->DATA = 'CONVERT';

DO
  I = 1 TO 4;
  ALLOCATE LIST IN (BODY_1) SET (P);
  P->KEY = '222'; P->DATA = 'TREVOC';
  Q -> L = P; Q = P;

END;

  Q -> L = NULL;
  DUMMY_POINTER = ADDR (BODY_2);
  BODY_2 = EMPTY; L_OHEAD = NULL;

  CALL CONV_AR
  (BODY_1, L_HEAD, BODY_2, L_OHEAD);

CONV_AR:
PROCEDURE
  (BODY1, HEAD, BODY2, OHEAD);
DECLARE
  (BODY1, BODY2) AREA (*),
  DUMMY_BODY BASED
  (DUMMY_POINTER) AREA,
  (HEAD, SAVE) POINTER,
  OHEAD OFFSET (DUMMY_BODY),
  1 COMPONENT1 BASED (C1),
  2 KEY CHARACTER (3),
  2 DATA CHARACTER (7),
  2 L POINTER,
  1 COMPONENT2 BASED (C2),
  2 KEY CHARACTER (3),
  2 DATA CHARACTER (7),
  2 OL OFFSET (DUMMY_BODY);

  /* IF AREA CONDITION OCCURS, BODY2 IS
  TOO SMALL TO RECEIVE CONTENT OF
  BODY1. GO TO NULL_LIST. */
  ON AREA
  GO TO
  NULL_LIST;
  /* IF LIST IN BODY1 CONTAINS NO
  COMPONENTS, GO TO NULL_LIST. */
IF
  HEAD = NULL
  THEN
  GO TO
  NULL_LIST;
  /* ASSOCIATE OFFSET VARIABLES OHEAD
  AND OL WITH BODY2. */
  DUMMY_POINTER = ADDR (BODY2);

  /* ALLOCATE COMPONENT2 IN BODY2,
  AND ASSIGN TO THE ALLOCATION IN
  THE KEY AND DATA VALUES OF THE FIRST
  LIST COMPONENT IN BODY1. */
  ALLOCATE COMPONENT2 IN (BODY2)
  SET (C2);
  OHEAD, SAVE = C2;
  C2->COMPONENT2 = HEAD->COMPONENT1,
  BY NAME;
```

Figures 5K4-1 (CONV\_AR) and 5K4-2 (CONV\_RA). Conversion of a data list from absolute to relocatable, and from relocatable to absolute

```

/* PERFORM SUCCESSIVE ALLOCATIONS
OF COMPONENT2 IN BODY2, AND ASSIGN
TO THE ALLOCATIONS THE KEY AND DATA
VALUES OF SUCCESSIVE LIST COMPONENTS
IN BODY1. */
C1 = HEAD->L;
DO
    WHILE (C1->=NULL);
    ALLOCATE COMPONENT2 IN (BODY2)
    SET (C2);
    SAVE->OL, SAVE = C2;
    C2->COMPONENT2 = C1->COMPONENT1,
    BY NAME;
    C1 = C1->L;
END;
/* ASSIGN A NULL OFFSET VALUE TO THE
OFFSET VARIABLE OL IN THE LAST
COMPONENT OF BODY2, THEN RETURN. */
SAVE->OL = NULL;
RETURN;
/* IF THIS POINT IS REACHED, ASSIGN
A NULL OFFSET TO OHEAD. */
NULL_LIST:
OHEAD = NULL;
END
CONV_RA:
IF
L_OHEAD = NULL
THEN
EXIT;
/* PREPARE TO INVOKE CONV_RA */
BODY_1 = BODY_2;
DUMMY_POINTER = ADDR (BODY1);
BODY_2 = EMPTY; L_HEAD = NULL;
CALL CONV_RA
(BODY_1, L_OHEAD, BODY_2, L_HEAD);
CONV_RA:
PROCEDURE
(BODY1, OHEAD, BODY2, HEAD);
DECLARE
(BODY1, BODY2) AREA (*),
DUMMY_BODY BASED (DUMMY_POINTER)
AREA,
OHEAD OFFSET (DUMMY_BODY),
(HEAD, SAVE, TEMP) POINTER,
1 COMPONENT1 BASED (C1),
2 KEY CHARACTER (3),
2 DATA CHARACTER (7),
2 OL OFFSET (DUMMY_BODY),
1 COMPONENT2 BASED (C2),
2 KEY CHARACTER (3),
2 DATA CHARACTER (7),
2 L POINTER;
/* IF AREA CONDITION OCCURS, BODY2
IS TOO SMALL TO RECEIVE CONTENTS OF
BODY1. GO TO NULL LIST. */
ON AREA
GO TO
NULL_LIST;

```

```

/* IF LIST IN BODY1 CONTAINS NO
COMPONENTS, GO TO NULL_LIST. */
IF
OHEAD = NULL
THEN
GO TO
NULL_LIST;
/* ASSOCIATE OFFSET VARIABLES OHEAD
AND OL WITH BODY1. */
DUMMY_POINTER = ADDR(BODY1);
/* ALLOCATE COMPONENT2 IN BODY2,
AND ASSIGN TO THE ALLOCATION THE KEY
AND DATA VALUES OF THE FIRST LIST
COMPONENT IN BODY1. */
ALLOCATE COMPONENT2 IN (BODY2)
SET (C2);
HEAD, SAVE = C2;
TEMP = OHEAD;
C2->COMPONENT2 = TEMP->COMPONENT1,
BY NAME;
/* PERFORM SUCCESSIVE ALLOCATIONS
OF COMPONENT2 IN BODY2, AND ASSIGN
TO THE ALLOCATIONS THE KEY AND DATA
VALUES OF SUCCESSIVE LIST COMPONENTS
IN BODY1. */
C1 = TEMP->OL;
DO
    WHILE (C1->=NULL);
    ALLOCATE COMPONENT2 IN (BODY2)
    SET (C2);
    SAVE->L, SAVE = C2;
    C2->COMPONENT2 = C1->COMPONENT1,
    BY NAME;
    IF
    C1->OL = NULL
    THEN
    C1 = NULL;
    ELSE
    C1 = C1 -> OL;
END;
/* ASSIGN A NULL VALUE TO THE
POINTER VARIABLE L IN THE LAST
COMPONENT OF BODY2, THEN RETURN. */
SAVE->L = NULL;
RETURN;
/* IF THIS POINT IS REACHED, ASSIGN
A NULL VALUE TO POINTER HEAD. */
NULL_LIST:
HEAD = NULL;
END
CONV_RA:
OUTPUT:
PUT
SKIP LIST ('CONTENT OF BODY_2:');
Q = L_HEAD;
DO
    WHILE (Q ->= NULL);
    PUT
SKIP EDIT (Q -> KEY, Q -> DATA)
(A, X(3), A);
Q = Q -> L;
END;
END
T_CONV_RA:
CONTENT OF BODY_2:
444 CONVERT
222 TREVNOG
222 TREVNOG
222 TREVNOG
222 TREVNOG

```

Figures 5K4-1 and 5K4-2. (Continued from preceding page)

### 5K5. Sorting Relocatable Data Lists

Once a list has been converted to relocatable form, the list need not be converted back to absolute form before it can be processed. Processing operations can be applied directly to relocatable lists. As an example, consider subroutine procedure SORT15 in Figure 5K5-1, which shows how a relocatable list of structures can be sorted. SORT15 uses two parameters: LIST\_OHEAD and LIST\_BODY.

LIST\_OHEAD is an offset variable that specifies the location of the first list component in area LIST\_BODY. As usual, LIST\_BODY can be of any storage class. The subroutine uses the same transposition technique employed in previous chapters and is similar to subroutine SORT11.

SORT15:

PROCEDURE

(LIST\_OHEAD, LIST\_BODY);

DECLARE

```
(HEAD, T, U, V) POINTER,
LIST_OHEAD OFFSET (DUMMY_BODY),
LIST_BODY AREA (*),
DUMMY_BODY AREA BASED
(DUMMY_POINTER),
1 COMPONENT BASED (C),
2 KEY CHARACTER (3),
2 DATA CHARACTER (77),
2 OL OFFSET (DUMMY_BODY);
/* ASSOCIATE OFFSET VARIABLES
LIST_OHEAD AND OL WITH LIST_BODY. */
DUMMY_POINTER = ADDR(LIST_BODY);
/* ASSIGN ADDRESS OF FIRST LIST
COMPONENT TO POINTER HEAD. */
```

IF

LIST\_OHEAD = NULL0

THEN

HEAD = NULL;

ELSE

HEAD = LIST\_OHEAD;

/\* IF LIST CONTAINS LESS THAN TWO  
COMPONENTS, NO SORT REQUIRED. \*/

IF

(HEAD = NULL) | (HEAD->OL = NULL0)

THEN

RETURN;

/\* SORT LIST. \*/

SORT:

K = 0;

/\* ASSIGN ADDRESS OF SECOND  
COMPONENT IN SEQUENCE TO POINTER C.\*/  
C = HEAD->OL;

/\* COMPARE KEY FIELDS OF SUCCESSIVE  
COMPONENTS AND PERFORM NECESSARY  
TRANSPOSITIONS. \*/

IF

(HEAD->KEY) > (C->KEY)

THEN

DO;

IF

C->OL = NULL0

THEN

T = NULL;

ELSE

T = C->OL;

C->OL = HEAD;

IF

T = NULL

Both subroutines use the same structure organization for the list components. The number of list components contained in LIST\_BODY is arbitrary and can even be zero.

Because an offset variable cannot qualify a based variable, the offset address must first be converted to a pointer address. As SORT15 demonstrates, the address conversion produces more instructions and generally adds to the sort time. The increased sort time, however, may not exceed the time required to convert the list back to absolute form before sorting is performed, and on this basis, a relocatable sort may be justified. Ordinarily, though, a list should be in absolute form when the list is to be processed extensively.

```
THEN
  HEAD->OL = NULL0;
ELSE
  HEAD->OL = T; HEAD = C; K = 1;
END;
  U = HEAD; C = U->OL;
  IF
    C->OL = NULL0
  THEN
    T = NULL;
  ELSE
    T = C->OL;
  DO
    WHILE (T->NULL);
    IF
      (C->KEY) > (T->KEY)
    THEN
      DO;
      IF
        T->OL = NULL0
      THEN
        V = NULL;
      ELSE
        V = T->OL;
        T->OL = U->OL;
        U->OL = C->OL;
      IF
        V = NULL
      THEN
        C->OL = NULL0;
      ELSE
        C->OL = V;
        K = 1;
      END;
      U = U->OL; C = U->OL;
      IF
        C->OL = NULL0
      THEN
        T = NULL;
      ELSE
        T = C->OL;
      END;
      IF
        K = 1
      THEN
        GO TO
        SORT;
        /* WHEN LIST IS SORTED, ASSIGN
        ADDRESS OF FIRST LIST COMPONENT
        TO LIST_OHEAD. */
        LIST_OHEAD = HEAD;
      END
    SORT15;
```

Figure 5K5-1. A subroutine procedure that sorts a relocatable list of structures

## 5K6. External Relocation

Although it is more convenient to assign a data list to another area when the list is in relocatable rather than absolute form, the major advantage of a relocatable list is that the list can be stored, without reorganization, on an external storage medium, such as magnetic tape or disk, from which it can be retrieved for later processing.

It is possible, however, to disassemble the components of a data list and to write them into a file in unlinked form. It is also possible to read the components back into internal storage and to reassemble the list by properly linking the components. But converting a list to and from unlinked form can be costly in execution time, particularly with list organizations that are far more intricately constructed than the simple linear lists discussed in this text. With relocatable lists, external transmission can be direct and, hence, generally more efficient in execution time.

The following discussions show how relocatable data lists are transmitted to and from a file and how such transmission permits a file of relocatable lists to be sorted and printed.

### 5K6A. Writing Relocatable Data Lists

Because stream-oriented input and output statements cannot transmit the address values of locator variables, record-oriented statements must be used to transmit relocatable lists. Output transmission of a list is performed by the LOCATE statement. This statement was discussed briefly in Chapter 3, and has the following form:

LOCATE based-variable FILE (file-name) SET (element-pointer-variable);

The LOCATE statement processes sequential, buffered files and allocates within an output buffer for the file the next available storage area for the specified based variable. The location of the allocated storage is assigned to the element-pointer variable given in the SET option. The LOCATE statement, however, need not contain a SET option; when it does not, an implied SET is assumed, which uses the pointer variable in the BASED attribute of the specified based variable. When the buffer is filled, its contents are automatically transmitted to the associated file. The buffer then becomes available again for further transmission.

Subroutine procedure LWRITE in Figure 5K6A-1 shows how the LOCATE statement can be used to transmit a relocatable data list to a file. The subroutine uses four parameters: OUTFILE, LIST\_OHEAD, LIST\_BODY, and BODY\_SIZE. OUTFILE is a sequential, buffered output-file used for record-oriented transmission. LIST\_OHEAD is an offset variable that specifies the relative address of the first list component in area LIST\_BODY, which can contain an arbitrary number of list components. BODY\_SIZE specifies the number of bytes in LIST\_BODY.

```
LWRITE:
PROCEDURE
  (OUTFILE, LIST_OHEAD,
   LIST_BODY, BODY_SIZE);
DECLARE
  DUMMY_BODY1 AREA BASED
  (DUMMY_POINTER1),
  DUMMY_BODY2 AREA BASED
  (DUMMY_POINTER2),
  LIST_OHEAD OFFSET (DUMMY_BODY1),
  LIST_BODY AREA (BODY_SIZE),
  BODY_SIZE FIXED DECIMAL (5),
  BIN_SIZE FIXED BINARY (15,0),
  OUTFILE FILE RECORD OUTPUT,
  1 OUTRECORD BASED (OUTPOINTER),
  2 B_SIZE FIXED BINARY (15,0),
  2 OHEAD OFFSET (DUMMY_BODY2),
  2 BODY AREA (BIN_SIZE REFER (B_SIZE));

  /* INITIALIZE SIZE OF BODY IN
  OUTRECORD. */
  BIN_SIZE = BODY_SIZE;

  /* LOCATE STORAGE IN OUTPUT BUFFER
  FOR OUTRECORD, AND ASSIGN LOCATION
  TO OUTPOINTER. */
  LOCATE OUTRECORD FILE (OUTFILE) SET
  (OUTPOINTER);

  /* ASSOCIATE OFFSET VARIABLE OHEAD
  WITH LOCATION OF OUTRECORD IN
  OUTPUT BUFFER. */
  DUMMY_POINTER2 = OUTPOINTER;

  /* ASSOCIATE OFFSET PARAMETER
  LIST_OHEAD WITH LIST_BODY. */
  DUMMY_POINTER1 = ADDR(LIST_BODY);

  /* ASSIGN LIST_OHEAD AND LIST_BODY
  TO OUTRECORD. */
  OUTPOINTER->OHEAD = LIST_OHEAD;
  OUTPOINTER->BODY = LIST_BODY;
END
LWRITE;
```

Figure 5K6A-1. A subroutine procedure that writes a relocatable data list into a file

Execution of LWRITE causes storage in the output buffer to be allocated for the self-defining based structure OUTRECORD, which contains the fixed binary variable B\_SIZE, the offset variable OHEAD, and the area variable BODY. The variable B\_SIZE represents the size of BODY, which is declared with the REFER option. The values of LIST\_OHEAD and LIST\_BODY are then assigned to OHEAD and BODY, which are eventually written along with B\_SIZE as a self-defining logical record in OUTFILE when the output buffer becomes full.

Because LIST\_BODY can have any storage class, the OFFSET attribute of LIST\_OHEAD, which requires a based area, uses based area DUMMY\_BODY1. Assignment of the address of LIST\_BODY to the pointer variable DUMMY\_POINTER1, which is associated with DUMMY\_BODY1, causes LIST\_OHEAD to become relative to LIST\_BODY.

When the LOCATE statement allocates storage for OUTRECORD, the location of the storage in the output buffer is assigned to the pointer variable OUTPOINTER. Assignment of OUTPOINTER, in turn, to DUMMY\_POINTER2, which is associated with DUMMY\_BODY2, causes OHEAD to become relative to the location of OUTRECORD in the output buffer. Note that assignment of LIST\_OHEAD to OHEAD produces no address modification, because both variables are offset variables. As usual, list components maintain their relative positions when assigned from LIST\_BODY to BODY.

Environmental information, such as input/output device type, unit number, recording density, block size, and record size, is not specified in an ENVIRONMENT attribute for OUTFILE. Data Definition (DD) statements are assumed to contain such information in the job step that calls for execution of the program under the operating system.

LOUT1:

```

PROCEDURE OPTIONS (MAIN);
  DECLARE
    1 CARD,
    2 KEY CHARACTER (3),
    2 DATA CHARACTER (77),
    1 COMPONENT BASED (COMPONENT_POINTER),
    2 KEY CHARACTER (3),
    2 DATA CHARACTER (77),
    2 OL OFFSET (DUMMY_BODY),
    1 OUTRECORD BASED (OUTPOINTER),
    2 B_SIZE FIXED BINARY (15,0),
    2 OHEAD OFFSET (DUMMY_BODY),
    2 BODY AREA (BIN_SIZE REFER(B_SIZE)),
    BIN_SIZE FIXED BINARY (15),
    DUMMY_BODY AREA BASED (DUMMY_POINTER),
    WORK_AREA AREA (550),
    LAST POINTER,
    OUTFILE FILE RECORD OUTPUT,
    LWRITE ENTRY (FILE, OFFSET
      (DUMMY_BODY), AREA (*), FIXED (5));

    /* ESTABLISH ENDFILE
      AND AREA ON-CONDITIONS. */
  ON ENDFILE (SYSIN)
  GO TO
  PROC_END;
  ON AREA
  BEGIN;
  PUT
  LIST (*INSUFFICIENT STORAGE *);
  CLOSE FILE (
  SYSPRINT);
  GO TO
  PROC_END;
END;

/*INITIALIZE SIZE OF BODY IN OUTRECORD*/
BIN_SIZE = 500;

/*FREE ALL ALLOCATIONS IN WORK_AREA,
AND ALLOCATE STORAGE FOR OUTRECORD
IN WORK_AREA. */
START:
  WORK_AREA = EMPTY;
  ALLOCATE OUTRECORD IN (WORK_AREA)
  SET (OUTPOINTER);

```

When the record size of OUTRECORD is specified in a DD statement, the record size must account for the storage associated with B\_SIZE, OHEAD, and BODY. The length, rather than the size, of BODY must be used (see the previous discussion in this chapter under "The Length of an Area", paragraph 5G). As stated earlier, the length of an area written as a record includes the number of bytes used for area control information as well as the size of the area.

#### 5K6B. An Example that Creates Relocatable Data Lists in a Work Area and Writes Them into a File

An application of the previously discussed subroutine LWRITE appears in program LOUT1 of Figure 5K6B-1, which shows how relocatable data lists can be created directly, rather than converted from absolute to relocatable form, and then written into a file.

```

/*ASSOCIATE OFFSET VARIABLE OL AND
OHEAD WITH BODY IN WORK_AREA.*/
DUMMY_POINTER = ADDR (OUTPOINTER
-> BODY);

/*INITIALIZE THE POINTER *LAST*/
LAST = NULL;

/* OBTAIN FIVE INPUT CARDS, AND
ASSIGN EACH TO COMPONENT STORAGE
ALLOCATED AND LINKED IN BODY WITHIN
WORK_AREA. */
DO
  I = 1 TO 5;
  ALLOCATE COMPONENT IN (OUTPOINTER
-> BODY) SET (COMPONENT_POINTER);
  GET
  EDIT (CARD) (A(3), A(77));
  COMPONENT_POINTER -> COMPONENT =
  CARD, BY NAME;
  IF
  LAST = NULL
  THEN
  OUTPOINTER -> OHEAD =
  COMPONENT_POINTER;
  ELSE
  LAST -> OL = COMPONENT_POINTER;
  LAST = COMPONENT_POINTER;
END;

/*ASSIGN NULL OFFSET TO OL IN LAST
COMPONENT. */
LAST -> OL = NULL;

/*WRITE OHEAD AND BODY INTO OUTFILE*/
CALL
  LWRITE (OUTFILE, OHEAD, BODY, 500);

/* PROC FSS NEXT FIVE INPUT CARDS,*/
GO TO
  START;
PROC_END:
  CLOSE FILE (
  OUTFILE);
END
  LOUT1;

```

Figure 5K6B-1. Creating relocatable data lists in a work area and writing them into a file

LOUT1 reads five cards from the standard system input file SYSIN, assigns each card to a relocatable list component allocated and linked in the area BODY. Offset OHEAD specifies the relative address of the first list component in BODY. Both OHEAD and BODY form elements of the self-defining based structure OUTRECORD, which is allocated in the work area WORK\_AREA. The variable B\_SIZE represents the size of BODY, which is declared with the REFER option. Subroutine LWRITE then writes OHEAD and BODY as a record into the output file OUTFILE, and processing continues with the next five input cards.

### 5K6C. An Example that Creates Relocatable Data Lists in an Output Buffer and Writes them into a File

The use of a work area in the previous program LOUT1 is not necessary, because no processing is performed on the relocatable lists created in the work area. Instead, the lists can be generated directly in an output buffer, as shown by program LOUT2 in Figure 5K6C-1. LOUT2 produces the same results as LOUT1 but replaces subroutine LWRITE with a LOCATE statement that allocates list storage directly in the output buffer associated with the file OUTFILE. Also, the records generated by LOUT2 are not self-defining, as they are in LOUT1. The data definition statement for LOUT1 records specify four more bytes than are specified for LOUT2. The extra four bytes are occupied by the offset variable contained in LOUT1 records. Such specifications relate to the implementation in use.

```
LOUT2:
PROCEDURE OPTIONS (MAIN);
DECLARE
    1 CARD,
    2 KEY CHARACTER (3),
    2 DATA CHARACTER (77),
    1 COMPONENT BASED (COMPONENT_POINTER),
    2 KEY CHARACTER (3),
    2 DATA CHARACTER (77),
    2 OL OFFSET (DUMMY_BODY),
    1 OUT_RECORD BASED (OUT_POINTER),
    2 OHEAD OFFSET (DUMMY_BODY),
    2 BODY AREA (500),
    DUMMY_BODY AREA BASED
    (DUMMY_POINTER),
    LAST_POINTER,
    OUTFILE FILE RECORD OUTPUT;
/* ESTABLISH ENDFILE AND AREA
UN-CONDITIONS. */
UN ENDFILE (SYSIN)
GO TO
END_LOUT2;
UN AREA
BEGIN;
PUT
LIST ('INSUFFICIENT STORAGE');
CLOSE FILE (
SYSPRINT);
GO TO
END_LOUT2;
END;
/* GET FIRST INPUT CARD IN NEXT SET
OF FIVE BEFORE LOCATING STORAGE
FOR OUT_RECORD IN OUTFILE BUFFER.
THIS STEP DETECTS END-OF-FILE
CONDITION BEFORE LOCATING
UNNECESSARY STORAGE IN BUFFER. */
START:
GET
EDIT (CARD) (A(3), A(77));
LOCATE OUT_RECORD FILE(OUTFILE)
SET (OUT_POINTER);
/* ASSOCIATE OFFSET VARIABLES OL
AND OHEAD WITH BODY IN OUTFILE
BUFFER. */
DUMMY_POINTER =
ADDR(OUT_POINTER->BODY);
```

```
/* ALLOCATE COMPONENT STORAGE FOR
FIRST CARD IN BODY WITHIN OUTFILE
BUFFER. */
ALLOCATE COMPONENT IN
(OUT_POINTER->BODY)
SET (COMPONENT_POINTER);
/* ASSIGN FIRST CARD TO COMPONENT
STORAGE. */
COMPONENT_POINTER->COMPONENT =
CARD, BY NAME;
/* ASSIGN ADDRESS OF FIRST COMPONENT
TO OUT_POINTER->OHEAD. */
(OUT_POINTER->OHEAD =
COMPONENT_POINTER;
/* SAVE ADDRESS OF COMPONENT IN WORK
POINTER LAST. */
LAST = COMPONENT_POINTER;
/* ASSIGN REMAINING FOUR CARDS OF
INPUT SET TO COMPONENT STORAGE
ALLOCATED AND LINKED IN BODY
WITHIN OUTFILE BUFFER. */
DO
I = 1 TO 4;
ALLOCATE COMPONENT IN
(OUT_POINTER->BODY)
SET (COMPONENT_POINTER);
GET
EDIT (CARD) (A(3), A(77));
COMPONENT_POINTER->COMPONENT =
CARD, BY NAME;
LAST->OL = COMPONENT_POINTER;
LAST = COMPONENT_POINTER;
END;
/* ASSIGN NULLO OFFSET TO OL IN
LAST COMPONENT. */
LAST->UL = NULLO;
/* PROCESS NEXT FIVE INPUT CARDS. */
GO TO
START;
END_LOUT2:
CLOSE FILE (
OUTFILE);
END
LOUT2;
```

Figure 5K6C-1. Creating relocatable data lists in an output buffer and writing them into a file

### 5K6D. Reading Relocatable Data Lists

Once relocatable lists are written into a file, they become available for further processing at a later time. Retrieval of a relocatable list from a file is achieved with the READ statement, which was discussed briefly in Chapter 3 and has the following form:

```
READ FILE (file-name) SET (element-pointer-variable);
```

The READ statement obtains the location of the next logical record in a buffer associated with the specified file and assigns the location to the element-pointer variable given in the SET option. A based variable associated with the same pointer will then relate to the fields of the logical record. The based variable is effectively overlaid on the logical record in the buffer.

Subroutine LREAD in Figure 5K6D-1 shows how a READ statement can be used to obtain the location of the next logical record in an input buffer. LREAD uses two parameters: INFILE and INPOINTER. INFILE is a sequential, buffered input-file used for record-oriented transmission. INPOINTER is a pointer variable that receives the address of the next logical record in the buffer associated with INFILE. When all records have been read from INFILE, execution of LREAD causes INPOINTER to receive a null pointer value.

### 5K6E. An Example that Sorts and Prints Relocatable Data Lists Contained in a File

An application of the previously discussed subroutine LREAD appears in program SORT16 of Figure 5K6E-1, which shows how relocatable data lists can be retrieved from a file and how the components of the retrieved list can be sorted and printed. Sorting is performed by subroutine SORT15, which was discussed previously.

Procedure SORT16 contains a declaration for B\_SIZE which is contained in the self-defining records written by LOUT1. The declaration for B\_SIZE is not used in SORT16 for reading records written by LOUT2.

```
LREAD:
  PROCEDURE (INFILE, INPOINTER);
  DECLARE
    INFILE FILE RECORD INPUT,
    INPOINTER POINTER;

    /* AT END OF INFILE,
       SET INPOINTER TO NULL,
       AND ALLOW NORMAL RETURN FROM
       ON-UNIT TO TERMINATE
       SUBROUTINE */
  ON ENDFILE (INFILE)
    INPOINTER = NULL;

    /* READ NEXT LOGICAL RECORD FROM INFILE,
       AND SET INPOINTER TO LOCATION
       OF RECORD IN INPUT BUFFER */
  READ FILE (INFILE) SET (INPOINTER);
END
LREAD;
```

Figure 5K6D-1. A subroutine that reads a relocatable data list from a file

```
SORT16:
PROCEDURE OPTIONS (MAIN);
DECLARE
  INFILE FILE RECORD INPUT,
  1 INRECORD BASED (INPOINTER),
  2 B_SIZE FIXED BINARY,
  2 OHEAD OFFSET (DUMMY_BODY),
  2 BODY AREA (500),
  1 COMPONENT BASED
  (COMPONENT_POINTER),
  2 KEY CHARACTER (3),
  2 DATA CHARACTER (77),
  2 OL OFFSET (DUMMY_BODY),
  DUMMY_BODY AREA BASED
  (DUMMY_POINTER),
  LREAD ENTRY (FILE, POINTER),
  SORT15 ENTRY (OFFSET(DUMMY_BODY),
  AREA(500));
/* READ NEXT RELOCATABLE LIST FROM
INFILE, AND ASSIGN LIST LOCATION TO
INPOINTER. */

START:
CALL
  LREAD (INFILE, INPOINTER);
/* IF INPOINTER IS NULL, END OF
INFILE REACHED. GO TO END OF
PROGRAM. */
IF
  INPOINTER = NULL
  THEN
    GO TO
    END_SORT16;
/* ASSOCIATE OFFSET VARIABLES
OHEAD AND OL WITH BODY OF LIST. */
  DUMMY_POINTER =
  ADDR(INPOINTER->BODY);
/* SORT LIST. */
CALL
  SORT15(INPOINTER->OHEAD,
  INPOINTER->BODY);
/* PRINT CONTENT OF SORTED LIST. */
IF
  INPOINTER->OHEAD = NULLO
  THEN
    COMPONENT_POINTER = NULL;
  ELSE
    COMPONENT_POINTER =
    INPOINTER->OHEAD;
DO
  WHILE (COMPONENT_POINTER->=NULL);
  PUT
    EDIT (COMPONENT_POINTER->KEY,
    COMPONENT_POINTER->DATA)
    (COLUMN(1), A(3), A(77));
  IF
    COMPONENT_POINTER->OL = NULLO
    THEN
      COMPONENT_POINTER = NULL;
    ELSE
      COMPONENT_POINTER =
      COMPONENT_POINTER->OL;
END;
  PUT
    SKIP (2);
/* PROCESS NEXT RELOCATABLE DATA
LIST. */
  GO TO
  START;
END_SORT16:
  CLOSE FILE (
  SYSPRINT);
END
  SORT16;
```

Figure 5K6E-1. Sorting and printing relocatable data lists contained in a file

## 5L. REVIEW OF TECHNIQUES FOR CREATING RELOCATABLE DATA LISTS

This chapter has shown how to organize a data list in relocatable form so that it can be moved conveniently from one location to another. The chief reason for considering the relocation of a data list occurs when the list is written into a file; subsequent processing of the list requires that it be retrieved from the file and assigned to an area in main storage. Since this storage area generally differs from the area that originally contained the list, the values of the pointer variables that link the components of the list become invalid in the new area.

One way of maintaining proper linkage is to reconstruct the list, component by component, in the new area. But this method is costly in execution time, particularly when frequent movement of the list is required. A more efficient way is to treat the list as a unit and assign its containing area to the new area. The list, however, will not be linked properly in the new area; the area assignment will have invalidated the pointer values within the transmitted list. Even if the pointers were to remain unchanged, they would still refer to the components of the list in the original area and not to the components in the new area. This difficulty is overcome by using offset variables rather than pointer variables to link the components of the list.

Figure 5L-1 contains illustrations of a data list linked by pointer variables (absolute form) and by offset variables (relocatable form). The offset head OHEAD and the offset links OL in the relocatable list contain the relative addresses of list components. These addresses are relative to the beginning of the containing area. They remain valid when the relocatable list is assigned to another area because the components of the list retain their same relative positions within the new area.

Since most lists are initially constructed in absolute form, relocatable lists are conveniently obtained by converting from absolute to relocatable form. Subroutine CONV\_AR showed how this conversion may be performed. Similarly, subroutine CONV\_RA demonstrated the reverse conversion (relocatable to absolute form). The essential techniques used by both of these subroutines are illustrated in Figure 5L-2, which shows how area assignment may be used to relocate a data item and how the location of the item may be obtained in the assigned area. The figure assumes that offset variable O1 has been declared with the attribute OFFSET(AREA1)

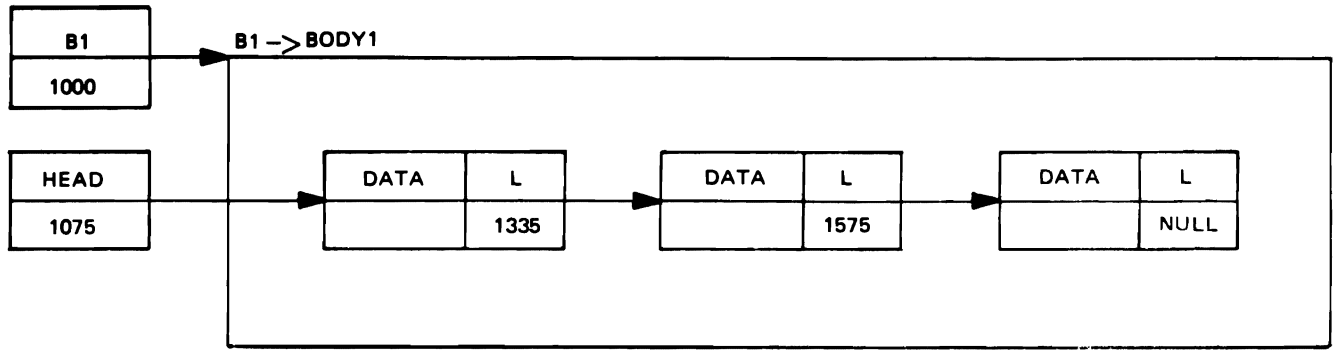
and that offset variable O2 has been declared with the attribute OFFSET(AREA2). These attributes cause all address values of O1 to be made relative to the beginning of AREA1 and those of O2 to be made relative to the beginning of AREA2. For illustration purposes, the figure assumes the arbitrary addresses 2000 and 4025 for AREA1 and AREA2.

It is also assumed that storage has been allocated for the based variable DATA\_ITEM at location 2075 within A1 → AREA1 and that this address has been assigned to pointer P1. The address of DATA\_ITEM relative to the beginning of A1 → AREA1 is obtained by assigning P1 to O1. The program automatically subtracts the address (2000) of A1 → AREA1 from the value (2075) of P1 and assigns the difference (75) to O1. When A1 → AREA1 is assigned to A2 → AREA2, DATA\_ITEM retains its relative location (75) within the new area. Note, however, that the absolute address of DATA\_ITEM in A2 → AREA2 is not immediately available. The relative location (75) of DATA\_ITEM in A2 → AREA2 must first be assigned to offset variable O2, which is associated with AREA2. Since the value of an offset variable is not changed when it is assigned to another offset variable, O2 receives the value 75 when O1 is assigned to O2. The absolute address of DATA\_ITEM in A2 → AREA2 is then obtained by assigning O2 to pointer P2; the address (4025) of A2 → AREA2 is automatically added to the value (75) of O2, and the sum (4100) is assigned to P1. When the subroutines CONV\_AR and CONV\_RA convert a data list to or from relocatable form, they apply these addressing techniques to each component in the list.

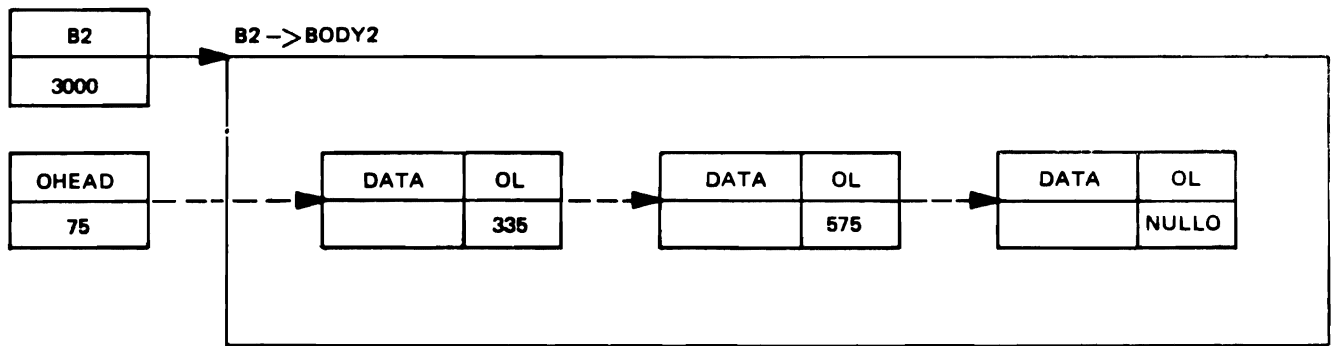
Figure 5L-3 shows how area assignment can be used to move two relocatable lists within a single area. The components of both lists retain their relative positions in the new area, as illustrated by the sample values of the offset links. (Note that the components of each list are arranged within the figure in logical sequence and are not scattered to show their physical positions in the area.)

When a relocatable list is written into a file, the LOCATE statement is first used to obtain storage for the list in an output buffer. Area assignment is then used to move the list into the buffer, and, when the buffer becomes full, it is automatically transmitted to the file. In turn, retrieval of a relocatable list from a file is achieved with a READ statement that contains a SET option. The SET option obtains the position of the list in an input buffer.



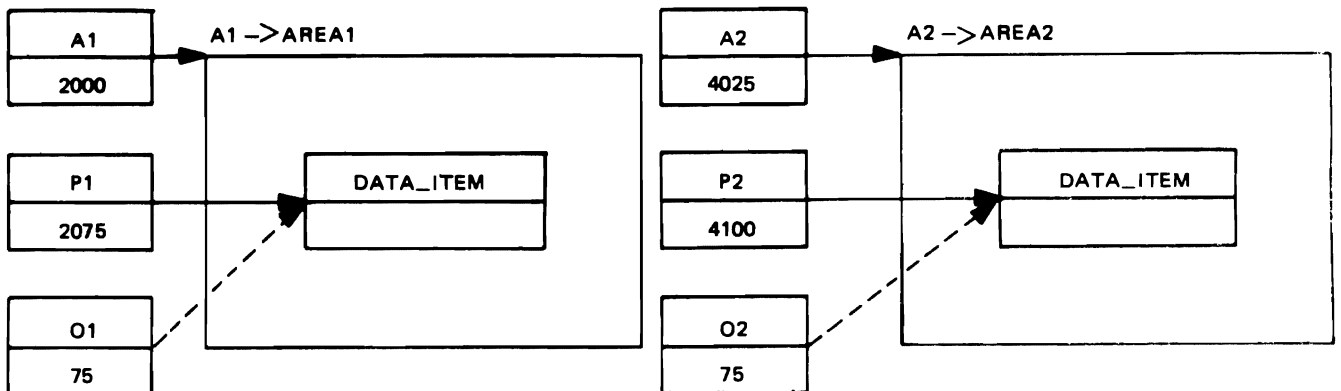


DATA LIST IN ABSOLUTE FORM



DATA LIST IN RELOCATABLE FORM

Figure 5L-1. A data list in absolute and relocatable form



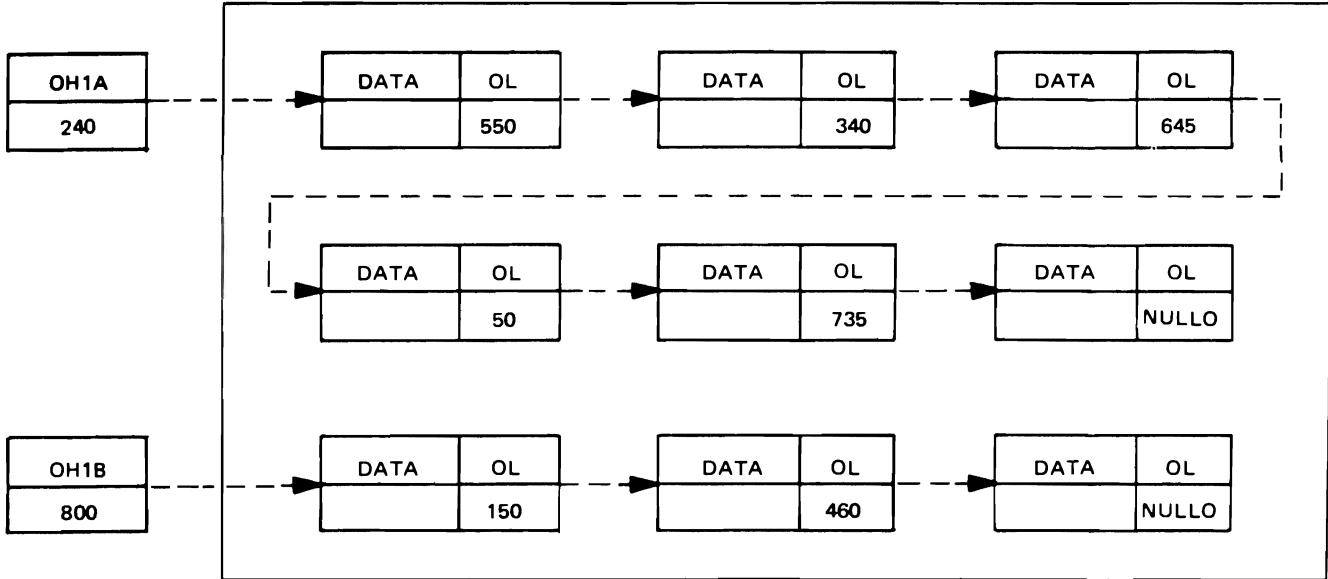
```

A2 -> AREA2 = A1 -> AREA1;      /* ASSIGN A1 -> AREA1 TO A2 -> AREA2.*/
O1 = P1;                          /* SET O1 TO RELATIVE ADDRESS OF DATA_ITEM IN A1 -> AREA1.*/
O2 = O1;                          /* SET O2 EQUAL TO O1.*/
P2 = O2;                          /* SET P2 TO ABSOLUTE ADDRESS OF DATA_ITEM IN A2 -> AREA2.*/

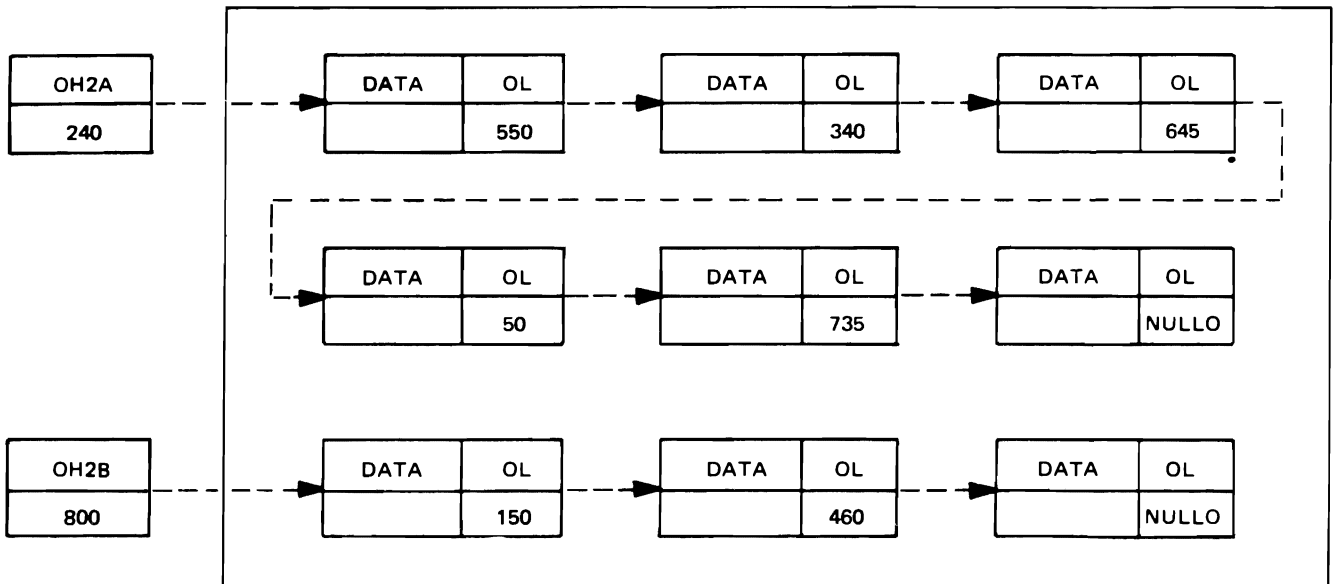
```

Figure 5L-2. Obtaining the absolute address of a data item in an assigned area

B1 -> BODY1



B2 -> BODY 2



B2 -> BODY2 = B1 -> BODY1;  
OH2A = OH1A;  
OH2B = OH1B;

Figure 5L-3. Assigning relocatable data lists to another area

## 5M. SUMMARY OF CHAPTER 5

- A. A list can be treated as a collective unit by referring to the area in which the list components have been allocated; internal and external movement of a list then becomes possible by transmitting the containing area.
- B. The assignment statement permits the contents of one area to be assigned to another area. However, pointer values in the assigned area become invalid in the receiving area.
- C. No operators can be applied to area variables.
- D. When an area is allocated, it receives the empty state. An area can also be made empty by assigning it the value of an empty area or the value of the built-in function `EMPTY`.
- E. The extent of an area is the number of storage bytes between the start of the area and the end of the allocation most distant from the start.
- F. Assignment of an area effectively frees all allocations in the receiving area, and then assigns the extent of the area to the receiving area.
- G. All free-storage gaps within an area are retained during area assignment, so that allocations within the assigned extent maintain their locations relative to each other.
- H. When the extent of an assigned area exceeds the size of the receiving area, an `AREA ON`-condition occurs, and the content of the receiving area becomes undefined.
- I. The length of an area is the sum of the area size (specified in the `AREA` attribute) and the number of storage bytes occupied by the area control information, which is supplied by the PL/I compiler.
- J. During record-oriented transmission of an area variable, the record length is determined by the length of the area, not the size of the area.
- K. An offset variable has a relative address as its value and is declared with the `OFFSET` attribute, which has the following form:

`OFFSET (area-variable)`

The area variable in parentheses must be based and unsubscripted, and have an implied or explicit level number of one.

- L. When the value of a pointer variable is assigned to an offset variable, the assigned value is made relative to the absolute address of the area specified in the `OFF-`

`SET` attribute for the variable. The address computation is equivalent to the following calculation:

$\text{offset value} = (\text{pointer value}) - (\text{absolute address of area})$

- M. When an offset variable is assigned to a pointer variable, the assigned offset value is converted to a pointer value. The offset value is effectively added to the absolute address of the area specified in the associated `OFFSET` attribute:

$\text{pointer value} = (\text{offset value}) + (\text{absolute address of area})$

- N. When an offset variable is assigned to another offset variable, the offset value is assigned without modification.
- O. A null offset value is assigned to an offset variable through the built-in function `NULLO`.
- P. A null offset value cannot be assigned to a pointer variable. Similarly, a null pointer value cannot be assigned to an offset variable.
- Q. The comparison operators equal (`=`) and not equal (`≠`) are the only two operators that can use offset variables as operands.
- R. An offset variable cannot qualify a based variable. The offset value must first be assigned to a pointer variable, which is then used to qualify the based variable.
- S. The values of locator variables (offsets and pointers) cannot be converted to any other type of data, nor can any other type of data be converted to locator type.
- T. Locator variables may be used as arguments and parameters. When an offset argument is associated with an offset parameter, both must be offset with respect to the same area.
- U. A relocatable data list is formed by using offset variables rather than pointer variables as component links in the list.
- V. Internal relocation of a relocatable data list is achieved with the assignment statement; the area that contains the list is assigned to another area.
- W. Only record-oriented transmission statements can be used for external relocation of lists. The `LOCATE` statement transmits a list to a file, and the `READ` statement retrieves a list from a file. The area that contains the list is transmitted to and from the file.

## Appendix. Summary of List-Processing Facilities

The following summary divides the list-processing facilities into five categories: attributes, built-in functions, ON-conditions, statements, and miscellaneous features. Facilities within each category appear in alphabetic order.

When used in the format of each facility, brackets [ ] denote optional items; braces { } indicate that a choice must be made from the enclosed items, which are separated by an "or" symbol |; and an ellipsis ( . . . ) specifies optional repetition of the preceding item.

### Attributes

AREA [{ (size-expression)|(\*) } ]  
BASED (element-pointer-variable)  
OFFSET (area-variable)  
POINTER

### Built-In Functions

ADDR (argument-variable)  
EMPTY  
NULL  
NULLO

### On-Condition

AREA

### Statements

ALLOCATE based-variable  
[IN (area-variable)]  
[SET (pointer-variable)]  
[, based-variable  
[IN (area-variable)]  
[SET (pointer-variable)]] . . . ;  
FREE based-variable  
[IN (area-variable)]  
[, based-variable  
[IN (area-variable)]] . . . ;  
LOCATE based-variable  
FILE (file-name)  
SET (pointer-variable);  
READ FILE (file-name)  
SET (pointer-variable);

### Miscellaneous Features

Pointer-qualification symbol:

->

REFER option:

element-variable REFER (element-variable)

## Index

	<i>Figure Number</i>	<i>Para- graph Number</i>	<i>Page Number</i>		<i>Figure Number</i>	<i>Para- graph Number</i>	<i>Page Number</i>
Absolute addresses		3D	24	SORT2	2E1-5		13
Absolute list		4B	53	SORT3	2E2-1		15
ADDR built-in function		3D2A	25	SORT4	3C1-1		17
Addressing storage		1A	1	SORT5	3C2-1		18
ALLOCATE statement		4A2	46	MEAN	3D3B-1		27
Area variables		5B	62	SWAP	3E2-1		30
Arrays		2B	7	SORT6	3F2-1		33
Arrays of structures		2D	10	SORT7	3F3-1		34
AREA attribute		4A6A	51	SORT8	3G1-1		35
AREA ON-condition		4A6C	53	SORT9	3G1-5		37
Assignment of areas		5B	62	SORT10	3G2-1		38
Assignment of offsets		5J1	67	SORT11	3G2-7		42
Assignment of pointers		3D2B	25	SORT12	4A4-1		48
Based storage		4A	45	SORT13	4A5B-1		51
Based variables		3D2	26	SORT14	4A6D-1		54
Buffer storage		5K6C	80	EXTENT	5F-1		65
Contextual pointers		3E4	32	MOVE_L	5K1-3		72
Data movement		1B	1	MOVE_RL	5K3-1		74
DEFINED attribute		4A6A	51	CONV_AR	5K4-1		75
Element items		2A	7	CONV_RA	5K4-2		76
EMPTY built-in function		5B	62	SORT15	5K5-1		77
Extent of an area		5C	63	LWRITE	5K6A-1		78
External relocation		5K6	78	LOUT1	5K6B-2		79
FREE statement		4A3	46	LOUT2	5K6C-1		80
IN option		4A6B	52	LREAD	5K6D-1		81
Length of an area		5G	65	SORT16	5K6E-1		81
List organization		4B	53	READ statement	5K6D		81
List usage		1D	4	REFER option	4A5A		49
LOCATE statement		5K6A	78	Relative addresses	3C		16
NULL built-in function		3D2A	25	Relocatable list	5K2		71
NULLO built-in function		5J2	68	Relocation of data lists	5K		69
Offset variables		5J	66	Restrictions on based variables	3E3		32
Pointer arrays		3G	35	SET option	4A2		46
Pointer qualification		3E1	28	Size of an area	5C		63
Pointer variables		3D2	24	Storage			
Program examples in figure number sequence:				automatic	4A		45
SORT1	2E1-1		11	based	4A		45
				controlled	4A		45
				Structures	2C		9
				Symbolic addresses	3B		16



**International Business Machines Corporation**  
**Data Processing Division**  
**1133 Westchester Avenue, White Plains, New York 10604**  
**(U.S.A. only)**

**IBM World Trade Corporation**  
**821 United Nations Plaza, New York, New York 10017**  
**(International)**