AIDS TO PL/I LIST PROCESSING

USING AREA VARIABLES

abstract- several subprograms written in Assembler to be called from PL/I are described. They aid in storage recycling, input, and output of linked based variables within PL/I AREA variables.
AIDS TO PL/I LIST PROCESSING USING AREA VARIABLES

Generalized list-processing (with more than one link field per node), also known as plex-processing, is gaining in popularity as a programming technique. It is particularly useful in programs dealing with information retrieval, graphic data processing, and simulation of complex systems.

We often wish to use list processing techniques in conjunction with an algorithmic language such as FORTRAN or PL/I. Unfortunately as these languages were not designed primarily to handle lists, we must augment their capabilities with special subroutine packages. FORTRAN, which has no real list processing primitives other than arrays, has been augmented with the SLIP package. PL/I has more built-in list processing capabilities. However, even so, it is limited in storage recycling and input-output of plex structures.

The PL/I programmer has most control over list-processing if he allocates the nodes he will use within AREA variables. An AREA variable is simply a contiguous block of storage, of programmer-determined size in which nodes (based structures) may be allocated. Several subprograms, written in Assembler to be called from PL/I, have been created to aid in the use of PL/I AREA variables. Input and Output are aided by use of PUTAREA and GETAREA. Storage recycling is aided by use of MOVAREA, GARBCOL, GETAREA, PUTAREA, ISIZE, IEXTENT, and ILARGE.
INPUT AND OUTPUT OF LINKED STRUCTURES.

OS/360 PL/I gives the user only very limited input and output capabilities for handling linked structures. The Language Reference Manual tells us that we may use record input and output for area variables and gives examples of how this may be done. If we wished to have a linked structure written out to secondary storage to be read in by a subsequent program, we would have to do something equivalent to the following.

1. Declare a structure containing the area in which the linked structure would be allocated and also containing offset variables which would enable us to get a handle on the linked structure. For example we would declare a structure like the following:

   DECLARE 1 AREASTRUCT,
      2 OFFSET1 OFFSET (DUMMYAREA),
      2 OFFSET2 OFFSET (DUMMYAREA),
      ...
      ...
      2 OFFSETN OFFSET (DUMMYAREA),
      2 AREAVAR AREA,
      DUMMYAREA AREA BASED (AREAP);

   The pointer 'AREAP' would be assigned to the value of ADDR (AREASTRUCT.AREAVAR) and thus the offsets would really refer to 'AREASTRUCT.AREAVAR'. This indirectness in referring to AREASTRUCT.AREAVAR within the offset declarations is required because of the PL/I implementation restriction that offsets refer to level 1 based area variables.

2. All based structures that are a part of the linked structure to be output would have to be allocated within the area variable (AREAVAR). No pointer variables would be allowed within the based structures making up the linked structure. Therefore all links between based structures in the linked structure would have to be in the form of offsets from the area variable. For example typical LISP-type nodes (based structures) might look like the following.

   DECLARE 1 LISPNODE BASED (NODEPOINT),
      2 TYPE FIXED BINARY, /*ATOM OR NODE*/
      2 CAROFFSET OFFSET (DUMMYAREA),
      2 CDROFFSET OFFSET (DUMMYAREA),
      1 ATOM BASED (ATOMPOINT),
      2 TYPE FIXED BINARY, /*ATM OR NODE*/
      2 PRINTNAME CHAR (8),
      2 VALUE FIXED BINARY;
3. When we wished to write out the linked structure we would execute a statement of the form:

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

Similarly to read the linked structure from the file for processing by the later program we would execute a statement of the form:

```
READ FILE(OUTPUT) TO (AREASTRUCT);
```

The disadvantages of this technique include the following:

1. The structures for input and output containing the offsets and the area variable must be identical in number of offsets and size of area.

2. Offsets rather than pointers must be used for all links. Since the current version of the PL/I compiler (version 5.0) does not allow offset variables to directly locate based variables, extra assignment statements OFFSET to POINTER are needed when the "handle" OFFSETS are to be used to locate a based variable. Each time an offset variable is used as a link to a based variable, the value of the offset must be added to the address of the base of the area variable in order to determine the pointer value which will be used to locate the linked-to based variable.

3. The entire area must be written out and read-in even though much or most of it does not contain active based variables. In the extreme, we can see that an empty area of size 32,000 bytes would take as much I/O time and secondary storage space as a completely full area.

4. Unless spanned records are used for input-output, the blocksize of the input-output file must be larger than the areasize. This can cause a large amount of storage to be needed for input-output buffers if the area is large.

To overcome these disadvantages, two subprograms have been written called GETAREA and PUTAREA. In contrast with the technique mentioned above, only currently active based structures will be transmitted for I/O; the number of locator variables used as "handles" to the linked structure need not be the same for input and output; the blocksize of the output and input file need be no larger than a small minimum size (80 bytes); the sizes of the area variables used in input and output need not be the same (the only restriction being that the input area have a size large enough to contain the active based variables that had been written previously); and pointer variables rather than offsets are used as locator variables.
STORAGE RECYCLING

An important feature of any generalized list-processing system is the ability to dynamically create nodes. The other side of the coin, returning nodes that are no longer needed so that their storage space may be reused, is similarly very important. The general problem of returning storage for reuse is sometimes called "garbage collection". However, this term seems more appropriate to describe one particular method, and in the sequel we shall refer to the general problem as "storage recycling".

Several different techniques have been suggested and implemented for handling the storage recycling problem. The most obvious method is to have an explicit "release node" function. This would have the main program, upon finding that a particular node is no longer required, call a subroutine which would return the storage occupied by the node back to the pool of unallocated storage. This is the technique available to the PL/I programmer when he uses "FREE" statements. This is also the technique employed by OS/360 in its "FREEMAIN" macro. This "release" function can be implemented fairly easily and can be extremely efficient particularly when nodes of a constant fixed size are used. Its major drawback is that in many applications of list-processing it is difficult, if not impossible, for the program to determine if a node is no longer required, particularly if nodes are "shared" among several lists. The use of a "release" function also suffers from the problem of storage fragmentation or "checker-boarding", when nodes of different sizes are used.

Another suggested solution to the storage recycling problem entails the keeping of "reference counts". This is the technique used in the SLIP system. With this technique, each node contains a field which will indicate how many pointers point to the node. Every time a new pointer or link is assigned the address of the node, the reference count field of the node is incremented. Similarly, every time a pointer is assigned a new value, the reference count of the node it previously pointed to is decremented. If the reference count of a node ever goes to zero the node is returned to the storage pool and the reference counts of all nodes pointed to by links within the returned node are decremented. A slight refinement would release a node when its reference count became zero, but not decrement the reference counts of nodes pointed to by the released node until the node was reallocated for a new use. In this manner, if there is no shortage of storage, overhead from storage management can be kept to a minimum. The reference count technique is able to handle most "shared node" situations and does not require the programmer to determine if a node will not be used again, for if the reference count becomes zero, that means the node is no longer accessible so it is impossible for it to be referenced.
Another advantage of reference counts is that like the explicit release function, collection of reusable space is spread out over time, thus making it a good candidate for real-time applications. Its major disadvantage is that it demands overhead on every pointer assignment for incrementing and decrementing the appropriate reference counts. It also suffers from the "checker-boarding" problem when different size nodes are used. Nodes, which indirectly point to themselves as in circular lists or rings will never be recycled as their reference counts cannot go to zero.

The method of "garbage collection" has been used quite successfully. This technique does nothing until the available storage for nodes has been exhausted. At this point, the "marking subroutine" is called. This subroutine recursively "marks" all accessible nodes by the following algorithm.

1. Mark all nodes pointed to by the pointer variables not contained within nodes (global pointers).
2. Mark all nodes pointed to by link fields within marked nodes.

This is efficiently implemented only as a recursive routine and as such requires a stack. Some very clever techniques for simply structured nodes, such as those found in LISP, allow the stack to be stored within the nodes being marked, thus not requiring additional storage for the stack. However in general, extra storage is necessary for keeping the stack used in finding accessible nodes. After the marking phase, either a "compaction" or "collection of garbage" is done. With "compaction" all marked nodes are moved to one end of the available memory and pointers are suitably changed, thus leaving one contiguous block of memory available for future allocations. The "collection" strategy does not move nodes, but instead goes through the node memory and puts all unmarked nodes onto a list of available space.

Garbage collection or garbage collection with compaction has as advantages: no work for recycling is done unless all available space is exhausted; only accessible nodes are retained, thus the programmer need not worry about keeping track of when he may release a node; and there is no overhead for each pointer assignment as is necessary for the reference count technique. The major disadvantage is that since all useful processing must halt while the garbage collector is active, real-time programs cannot generally use this technique. Another disadvantage is that a potentially large stack is generally needed at a time when storage is scarce. Also with garbage collection without compaction we again have the problem of "checker-boarding".
Several times we have mentioned the problem of "checker-boarding". What exactly is this? The problem is illustrated by the following. Consider five nodes of size 3 words, all stored contiguously in memory.

<table>
<thead>
<tr>
<th>NODE1</th>
<th>NODE2</th>
<th>NODE3</th>
<th>NODE4</th>
<th>NODE5</th>
</tr>
</thead>
</table>

Suppose now that nodes 2 and 4 are "freed". We now have 6 words available, but if a single request were made for a node of size 6 words, we could not fill it because the 6 available words are not contiguous. Clearly if we would restrict ourselves to nodes of a single size the problem would not exist.

The last technique we will consider can be called "recycling by copying". With this method we must have another area of storage large enough to contain all accessible nodes. We set two pointers to the address of the beginning of this storage. Let us call these pointers LOC1 and LOC2. For each non-null pointer variable that is not contained within a node we:

1. Copy the node pointed to by the pointer, to the location indicated by LOC2.
2. Replace the value of the pointer variable with the value of LOC2.
3. Mark the old copy of the node as having been moved and store the address of the new copy within the old node.
4. Advance LOC2 to point to the next available location following the new node.

In the cases where the pointer variable pointed to an old copy of a node already moved, we would have simply obtained the address of the new copy and would have stored that address value in the pointer variable.

After completing this we would continue as follows.

1. Look at each link field within the node pointed to by LOC1. If the link is null we do nothing with this link. If the node pointed to by the link is marked, indicating that it has already been copied, we get the address of the copy and store that in the link. Otherwise, we copy the node to the location pointed to by LOC2; store the value of LOC2 in the link; mark the old node as having been copied; store the address of the copy within the old node and advance LOC2 to the next available place following the new copy.
2. Advance LOC1 to the next node.
3. If LOC1 = LOC2 we are through, otherwise we repeat steps 1 to 3.

This technique has all the advantages and disadvantages of garbage collection with compaction, except that it does not require a stack of arbitrary size as the method is non-recursive. It is, however, much faster than garbage collection. Its major disadvantage is that a second storage area is needed. Normally, this would cut the available space for nodes in half. However, the second storage area could be on a disk or drum, though of course this would slow execution. It is much better than garbage collection when used on a virtual memory machine because of its much greater locality, and the procedure can be modified to put adjacent list elements on the same memory page.

As mentioned earlier OS/360 PL/I only provides the "explicit free" type of storage recycling. To overcome this, several subprograms have been written to perform "recycling by copying" (with and without the use of secondary storage) and "garbage collection without compaction". These routines written in Assembler, are called "MOVAREA" (for recycling by copying not using secondary storage), "PUTAREA" (recycling by copying to secondary storage), "GETAREA" (the second phase of recycling to secondary storage, reading the structures from the disk or drum), and "GARBCOL" (garbage collection without compaction). These routines act on nodes which have been allocated within PL/I AREA variables.
TEMPLATES

These subprograms (GARBCOL, MOVAREA, PUTAREA, GETAREA) give
the programmer the freedom to use many differently structured
nodes (based structures). In order for the subprograms to be
able to determine to which type of structure a pointer variable
points, a template number is required within every based
structure that will be processed. This template number is stored
in the first halfword of the structure. The template number is
really an index into a "template" array of halfword integers
(FIXED BINARY (15,0)). This index is the subscript of the first
entry in the "template" for the structure. Succeeding entries
for the template are located in the successive array elements.

Fixed-Length Structures

Fixed length structures are based structures which do not
contain arrays or strings whose bounds or lengths are determined
by use of the REFER option. These structures are to be preferred
because the length of the structure does not have to be
dynamically calculated. This extra computation would slow the
execution of the subprograms significantly.

The first template entry (the entry indexed by the template
number) contains the length of the structure in bytes. This
length can be determined by the user by finding the length of
each element variable (as listed in the PL/I Language Reference
Manual) and determining the length of each array by multiplying
the length of its variables by the number of array elements.
Note that the PL/I compiler sometimes inserts bytes of padding
between variables to force alignment of certain variable types
such as pointers and arithmetic data. This padding must be
included in the length of the structure. For more information
about computing the length of structures, see the Data Mapping

An important concept when using these subprograms is that of
logical level number. Within PL/I declarations each new level of
a structure is preceded by an integer.

\texttt{e.g. DCL 1 PAYROLL,}
\begin{verbatim}
  2 NAME,
  3 LAST,
  2 HOURS;
\end{verbatim}

These numbers within the declaration need not be in successive
order for the substructures.

\texttt{e.g. DCL 1 PAYROLL,}
\begin{verbatim}
  3 NAME,
  5 LAST,
  2 HOURS;
\end{verbatim}
The logical level of a substructure is the number we would obtain if we did require the level numbers to be in successive order. An easy way of determining the logical level of an element of a structure is to write the fully qualified name of the element and count the number of identifiers separated by ".". For example, in the structure above, the fully qualified name of 'LAST' is 'PAYROLL.NAME.LAST'. The number of identifiers within the fully qualified name is 3 and thus the logical level of 'LAST' is 3. Similarly, the qualified name of 'HOURS' is 'PAYROLL.HOURS', thus the logical level of 'HOURS' is 2.

The entries of the rest of the template can be determined from the structure declaration as follows: For each logical level within the declaration we have the following entries.

1. The negative of the logical level number.
2. The number of substructure array elements at this level. If the substructure is undimensioned, this will simply be one.
3. If the structure element at this level is a substructure then we have the corresponding template entries for it. If the structure element at this level is not a substructure, the following entry or two entries indicate the number of non-pointer bytes or number of bytes containing pointers. If the element is not a pointer we simply have an entry containing the number of bytes contained by the element. However, if we have a pointer variable, the template will contain two entries for this element. The first contains the constant "32000" indicating that the second entry will contain the number of bytes in the pointer variable.

The template for the major structure is ended by an entry containing the constant zero.

Adjustable-length Structures

An adjustable-length structure, sometimes called a self-defining structure, is a based structure that contains an array or string whose upper bound or length is determined by the contents of some variable within the structure. For example we may have the following structure:

```
DCL 1 NAME1 BASED(P),
   2 TEMPLATE_NUMBER FIXED BINARY (15),
   2 M FIXED BINARY (15),
   2 NAMESTR CHAR (N REFER (M));
```

When an instance of this structure is allocated space by means of an ALLOCATE statement, the value of 'N' will be placed in the structure element 'M' and the string 'NAMESTR' will be given the storage space required for a string whose length is this same value of 'N'.

-9-
Another example of an adjustable length structure is:

DCL 1 NAME2 BASED(P),
    2 TEMPLATE_NUMBER FIXED BINARY(15),
    2 M FIXED BINARY (15),
    2 L (6: N REFER (M),7:15) DECIMAL;

When an instance of this structure is allocated, the value of
'N' will be placed in 'NAME2.M' and the array 'NAME2.L' will be
allocated enough storage for '(N-5)*9' array elements.

One item we must have within an adjustable length structure
to be processed by the subprograms is a FIXED BINARY (15)
variable containing the number of elements (refer-amount) in the
structure array dimensioned by REFER. If the REFER gives the
upper bound of an array whose lower bound is 1 or the length of
a character string, the refer-amount variable can be the same as
the variable in parentheses following REFER within the structure
declaration. However, if the lower bound of the array is not 1
there must be an additional FIXED BINARY(15) variable which will
contain as its value, the number of elements in the dimension of
the array whose upper bound is given by REFER. For example with
structure 'NAME2' we would need an additional FIXED BINARY
variable within the structure.

DCL 1 NAME2 BASED(P),
    2 TEMPLATE_NUMBER FIXED BINARY(15),
    2 REFERM FIXED BINARY(15),
    2 M FIXED BINARY (15),
    2 L (6: N REFER(M),7:15) DECIMAL;

This refer-amount variable ('REFERM' within the structure)
should be given the value '(M-5)' immediately after the
structure is allocated.

The next thing we must do to determine the template for an
adjustable length structure is to construct a declaration of the
structure which would be equivalent (in terms of data mapping)
to the desired structure, but which contains the REFER option
only as the upper bound of a 1 dimensional substructure. The
first structure ('NAME1') would be transformed (for purposes of
determination of the template) into:

DCL 1 NAME1 BASED (P),
    2 TEMPLATE_NUMBER FIXED BINARY (15),
    2 M FIXED BINARY (15),
    2 DUMMYYARR (1:N REFER(M)),
    3 NAMESTR CHAR (1);
'NAME2' would be transformed into:

DCL 1 NAME2 BASED(P),
  2 TEMPLATE NUMBER FIXED BINARY(15),
  2 REFANT FIXED BINARY (15),
  2 M FIXED BINARY (15,0),
  2 DUMMYC (6:N REFER (M)),
  3 L (7:15);

Note that this auxiliary structure need not actually be declared within the PL/I program, but is just an aid to constructing the template for the desired structure.

Now we are ready to actually construct the template from the auxiliary structure. The template is exactly the same as if the structure were fixed-length except for two entries. The first entry of the template, which for a fixed-length structure would contain the length of the structure, instead contains the negative of the offset from the beginning of the structure of the variable containing the refer-amount. The offset is measured in terms of bytes. The template entry which would contain the size of the array dimensioned by REFER also contains the negative of the offset in bytes of the refer-amount variable.

Reducing Template Size

To reduce the size of templates, element variables at a given logical level which precede all substructures at that level and are not dimensioned by REFER need not be preceded by the level numbers and dimension entries. When this facility is used the entries in the template simply alternate between number of non-pointer bytes and number of pointer bytes. If the first number of non-pointer bytes is zero the entry will be '32000' instead or zero.

Any template that maps to an equivalent storage, in terms of length of structure and positions of pointer variables, may be substituted.
EXAMPLE 1

Assume the following base structure declaration.

DCL 1 BASED1 BASED (P),
   2 TEMPLATE_NUMBER FIXED BINARY (15,0),
   2 SUBSTRUC (15),
      3 POINTER1(0:5) POINTER,
      3 POINTER2 POINTER,
      3 DATA1 CHAR (4),
      3 POINTER3 POINTER;

Let us assume that the template-index which will be stored in TEMPLATE_NUMBER is '15' and the array TEMPLATE will contain the template entries. For the above structure we will have the following:

\[
\begin{align*}
\text{TEMPLATE}(15) & = 542 \quad \text{Total length of structure.} \\
\text{TEMPLATE}(16) & = -1 \quad \text{Start level 1 structure} \\
\text{TEMPLATE}(17) & = 1 \quad \text{Size of array level 1 structure.} \\
\text{TEMPLATE}(18) & = -2 \quad \text{Start level 2 substructure.} \\
\text{TEMPLATE}(19) & = 1 \quad \text{Size of array level 2 substructure.} \\
\text{TEMPLATE}(20) & = 2 \quad \text{Number of non-pointer bytes.} \\
\text{TEMPLATE}(21) & = -2 \quad \text{Start level 2 structure SUBSTRUC} \\
\text{TEMPLATE}(22) & = 15 \quad \text{Size of array of SUBSTRUC.} \\
\text{TEMPLATE}(23) & = -3 \quad \text{Start level 3 substructure.} \\
\text{TEMPLATE}(24) & = 6 \quad \text{Size of array level 3 substructure} \\
\text{TEMPLATE}(25) & = 32000 \quad \text{Next entry contains number of pointer bytes.} \\
\text{TEMPLATE}(26) & = 4 \quad \text{POINTER1 has 4 pointer bytes.} \\
\text{TEMPLATE}(27) & = -3 \quad \text{Start level 3 substructure.} \\
\text{TEMPLATE}(28) & = 1 \quad \text{Size of array level 3 substructure.} \\
\text{TEMPLATE}(29) & = 32000 \quad \text{Next entry contain number of pointer bytes.} \\
\text{TEMPLATE}(30) & = 4 \quad \text{POINTER2 has 4 pointer bytes.} \\
\text{TEMPLATE}(31) & = -3 \quad \text{Level 3 substructure DATA1} \\
\text{TEMPLATE}(32) & = 1 \quad \text{Size of array} \\
\text{TEMPLATE}(33) & = 4 \quad \text{Number of non-pointer bytes} \\
\text{TEMPLATE}(34) & = -3 \quad \text{Level 3 substructure POINTER3} \\
\text{TEMPLATE}(35) & = 1 \quad \text{Size of array} \\
\text{TEMPLATE}(36) & = 32000 \quad \text{Number of pointer bytes follows.} \\
\text{TEMPLATE}(37) & = 4 \quad \text{POINTER3 has 4 pointer bytes.} \\
\text{TEMPLATE}(38) & = 0 \quad \text{End of template}
\end{align*}
\]

By applying the rule that element variables need not be preceded by level numbers if they precede all substructures at the same level we could obtain the following equivalent template:

\[
\begin{align*}
\text{TEMPLATE}(15) & = 542 \quad \text{Length of structure} \\
\text{TEMPLATE}(16) & = -1 \quad \text{Level 1 structure} \\
\text{TEMPLATE}(17) & = 1 \quad \text{Size of array}
\end{align*}
\]
TEMPLATE(18) = 2    Non-pointer bytes.
TEMPLATE(19) = -2   Level 2 substructure
TEMPLATE(20) = 15   Size of array of SUBSTRUC
TEMPLATE(21) = 32000 First entry this level is not
                      non-pointer.
TEMPLATE(22) = 28   Number of pointer bytes
TEMPLATE(23) = 4    Number of non-pointer bytes
TEMPLATE(24) = 4    Number of pointer bytes.
TEMPLATE(25) = 0    End of template.

We could obtain a template that was two entries shorter if
DATA1 had been declared before POINTER. In general, a smaller
template may be obtained by declaring all non-pointer data
before pointer data at a given level, and by having as few
logical levels as possible. Since the time of execution of the
subprograms (and space required for GARBCOL) goes up as the
length of the templates, the size of the template is important.

For Example 1 we could obtain the smallest template if we
had originally declared the structure to be:

    DCL 1 BASED1 BASED(P),
       2 TEMPLATE_NUMBER FIXED BIN (15,0),
       2 DATA1(15) CHAR(4),
       2 POINTER1(15,0:5) POINTER,
       2 POINTER2(15) POINTER,
       2 POINTER3(15) POINTER;

which would be equivalent for most purposes. The template would
be

    TEMPLATE(15) = 542
    TEMPLATE(16) = -1
    TEMPLATE(17) = 1
    TEMPLATE(18) = 62
    TEMPLATE(19) = 480
    TEMPLATE(20) = 0

EXAMPLE 2

Assume the following declaration:

    DCL 1 REFERSTRUC BASED(P),
       2 TEMPLATE_NUMBER FIXED BIN (15,0),
       2 N FIXED BIN (15,0),
       2 SARRAY (N REFER (M)),
       3 PTR1 POINTER,
       3 DATA1 CHAR(8);
Assuming that the number to be stored in TEMPLATE_NUMBER would be 30, we would obtain the following template within the array TEMPLATE.

\[
\begin{align*}
\text{TEMPLATE}(30) &= -2 & \text{Offset of refer variable is 2} \\
\text{TEMPLATE}(31) &= -1 & \text{Start level 1 structure} \\
\text{TEMPLATE}(32) &= -1 & \text{Size of array level 1 structure.} \\
\text{TEMPLATE}(33) &= 4 & \text{4 non-pointer bytes.} \\
\text{TEMPLATE}(34) &= -2 & \text{Start level 2 substructure.} \\
\text{TEMPLATE}(35) &= -2 & \text{Offset of REFER bound is 2} \\
\text{TEMPLATE}(36) &= 32000 & \text{First entry this level is not} \\
& & \text{non-pointer byte count} \\
\text{TEMPLATE}(37) &= 4 & \text{Number of pointer bytes.} \\
\text{TEMPLATE}(38) &= 8 & \text{Number of non-pointer bytes.} \\
\text{TEMPLATE}(39) &= 0 & \text{End of template}
\end{align*}
\]

**Example 3**

Assuming the following declaration and that 'REFAMT' will contain the refer-amount:

\[
\begin{align*}
\text{DCL} \ 1 \ \text{REFERSTR2 BASED (P),} \\
&2 \ \text{TNUMB FIXED BINARY (15),} \\
&2 \ \text{REFAMT FIXED BINARY (15),} \\
&2 \ \text{N FIXED BINARY (15),} \\
&2 \ \text{STRUC (O:M REFER (N), 7),} \\
&\quad 3 \ \text{DATA CHAR(4),} \\
&\quad 3 \ \text{FTR PTR POINTER;}
\end{align*}
\]

we may begin to construct the template. First we must construct the auxiliary structure used in constructing the template. We obtain:

\[
\begin{align*}
\text{DCL} \ 1 \ \text{REFERSTR2 BASED (P),} \\
&2 \ \text{TNUMB FIXED BINARY (15),} \\
&2 \ \text{REFAMT FIXED BINARY (15),} \\
&2 \ \text{N FIXED BINARY (15),} \\
&2 \ \text{DUMMYSTR (O:M REFER (N), 7),} \\
&\quad 3 \ \text{STRUC (7),} \\
&\quad \quad 4 \ \text{DATA CHAR(4),} \\
&\quad \quad 4 \ \text{FTR PTR POINTER;}
\end{align*}
\]

Now we assume that the template index to be stored in TNUMB is 1. We now obtain the following entries in the template array TEMPLATE:

\[
\begin{align*}
\text{TEMPLATE}(1) &= -2 & \text{Negative of offset refer-amount} \\
\text{TEMPLATE}(2) &= -1 & \text{Start level 1} \\
\text{TEMPLATE}(3) &= 1 & \text{Size of array level 1} \\
\text{TEMPLATE}(4) &= 6 & \text{Six non-pointer bytes} \\
\text{TEMPLATE}(5) &= -2 & \text{Start level 2 structure} \\
\text{TEMPLATE}(6) &= -2 & \text{Offset refer-amount var. is 2} \\
\text{TEMPLATE}(7) &= -3 & \text{Start level 3 structure.}
\end{align*}
\]
TEMPLATE(8) = 7    Size of array level 3 structure.
TEMPLATE(9) = 4    Four non-pointer bytes
TEMPLATE(10) = 4   Four pointer bytes
TEMPLATE(11) = 0   End of template

Note: For aligned structures, the PL/I compiler inserts extra padding between certain elements. This padding must be included in the template. Consider the following structure:

DCL 1 STRUC BASED(P),
    2 TEMP_NUMB FIXED BINARY (15,0),
    2 PTR1 POINTER,
    2 DATA1 CHAR (2),
    2 PTR2 POINTER;

Since pointer variables are fullword aligned, two bytes of padding are inserted after DATA1. The template would be: 14,-1,1,2,4,4,4,0.
ROUTINE DESCRIPTIONS

PUTAREA - written in Assembler. Accessible structures are defined to be all structures which are pointed to by the pointer parameters given to the subroutine and all structures pointed to by pointer variables within accessible structures. PUTAREA outputs the template array and all accessible structures in a format suitable for reading by the routine GETAREA. Normally all accessible structures would have been allocated within the area variable though this is not required. Thus when the area is emptied by PUTAREA all structures within the area are freed. The accessible structures are modified so that they should not be referenced after the call to PUTAREA.

PL/I declaration: DCL PUTAREA EXTERNAL ENTRY;

PL/I calling sequence:

CALL PUTAREA(filename, template, area, p1, p2, ... pn);

where:

filename - is a string specifying the ddname of the output file, which must have RECFM = U and a blocksize greater or equal to 80 bytes.

template - is a template array (see section on templates). The lower bound of the FIXED BINARY (15) array must be one.

area - an area variable. This area is assigned the value EMPTY by PUTAREA thus freeing all allocations made within the area.

p1, p2, ... pn - an arbitrary number of pointers (at least one). These provide a "handle" to the accessible structures. Each pointer parameter is assigned the value NULL by the routine PUTAREA.

GETAREA written in Assembler. Causes a file that was created by PUTAREA to be read and the based structures contained therein to be allocated within the area variable. The template array stored in the file is moved to the template parameter and all pointer parameters are given the values of the "handles".

PL/I declaration: DCL GETAREA EXTERNAL ENTRY;

PL/I calling sequence:

CALL GETAREA(filename, template, area, p1, p2, ... pn);

where:

filename - is a string specifying the ddname of the input file, which was created by the subprogram PUTAREA.
template - a template array (FIXED BINARY (15)). It must be at least as large as the template array that was passed to PUTAREA when the file was created. The lower bound of the array must be 1. The values of the template array passed to PUTAREA when the file was created are placed in this array.

area - the area which will receive the structures from the file. Structures already allocated within this area are not freed.

p1,p2,...,pn - the pointer variables which will receive the addresses of the structures pointed to by the corresponding pointers when the file was created. If there are more pointer parameters in the parameter list for GETAREA than for the corresponding call of PUTAREA the extra pointers will be given the value NULL.

MOVAREA - written in Assembler. MOVAREA causes all accessible structures to be copied into the first area parameter updating all accessible pointer variables. It empties the second area parameter thus freeing all allocations within the second area. It is logically equivalent to a call of PUTAREA followed by a call of GETAREA, except for the restriction that the target area (the first area parameter) may not be the same as the source area (the second area parameter).

PL/I declaration: DCL MOVAREA EXTERNAL ENTRY;

PL/I calling sequence:
CALL MOVAREA (template,area1,area2,p1,p2,...,pn);

where: template - is a template array.

area1 - is the target area, to receive the structures. Structures already allocated within area1 are not freed.

area2 - is the source area. It is emptied by MOVAREA thus freeing all structures allocated within it.

p1,p2,...,pn - an arbitrary number of pointer variables (but at least one) which are handles to the linked structures to be moved. They are updated to the new locations within area1 by MOVAREA.
GARBCOL - written in Assembler. GARBCOL performs an in-place garbage collection. All accessible structures (pointed to by the pointer parameters or by pointer variables within accessible structures) are found in the first pass. In the second phase all unaccessible structures within the area are "FREE'd". For use of GARBCOL, every based structure allocated within the area must be initialized by a call of 'INITIAL' immediately after allocation. Based structures moved into an area by MOVAREA or GETAREA are already properly initialized.

PL/I declaration:  DCL GARBCOL EXTERNAL ENTRY;

PL/I calling sequence:

CALL GARBCOL (template,area,p1,p2,...pn);

where:  template - is a template array.
        area - is the area to be collected.
        p1,p2,...pn - are an arbitrary number of pointer variables which provide a handle to the accessible structures.

INITIAL - written in Assembler. INITIAL initializes a based structure so that it may be used in area to be processed by GARBCOL. It does not store a template-index or initialize the pointer variables within the structure to null. This also must be done following the structure's allocation.

PL/I declaration:  DCL INITIAL EXTERNAL ENTRY (POINTER);

PL/I calling sequence:

CALL INITIAL (pntr);

where  pntr is a pointer to the recently allocated structure.

ISIZE - written in Assembler, returns as its value the size of an area variable.

PL/I declaration:  DCL ISIZE EXTERNAL ENTRY(AREA(*))
        RETURNS (FIXED BINARY (15));

PL/I call as function:  e.g.

    J = ISIZE (area);

where:  area is the area variable whose size is to be determined.
IEXTENT - written in Assembler, returns the value of the length of the current extent within an area variable.

PL/I declaration:

DCL IEXTENT EXTERNAL ENTRY (AREA(*))
  RETURNS (FIXED BINARY(15));

PL/I call as function: e.g.

I = IEXTENT(area);

where    area - is the area variable whose extent is to be determined.

ILARGE - written in Assembler, returns as its value the size (in bytes) of the largest based variable which may be allocated within a given area without causing the AREA condition. If the storage recycling routines are used within AREA on-units, this routine may be used to avoid infinite loops when there is not enough storage left in an area for allocation of a based structure.

PL/I declaration:

DCL ILARGE EXTERNAL ENTRY (AREA(*))
  RETURNS (FIXED BINARY(15));

PL/I call as function:

  e.g.    J = ILARGE(area);

where: area - is the area variable. ILARGE will return the size of the largest based variable which may be allocated within the area variable without causing the AREA condition.

DMPAREA - written in PL/I. Causes a hexadecimal dump of an area variable to be written to the STREAM file SYSPRINT.

PL/I declaration: DCL DMPAREA EXTERNAL ENTRY (AREA(*));

PL/I calling sequence:

CALL DMPAREA (area);

where: area - is the area variable to be dumped.

OPENIN, RDPFILE - PL/I routines called by GETAREA to open and read the file 'AREAFIL'.

OPENOUT, WRFILE - PL/I routines called by PUTAREA to open and write to file 'AREAFIL'.

-19-
ROUTINE RESTRICTIONS

1. All based structures allocated within an area variable to be processed must contain a valid template number in the leading halfword of the structure.

2. All pointer variables passed to the routines as parameters or within accessible based structures must either have the value NULL or the address of a level 1 based structure with the appropriate template number (depending on the type of structure) within its leading halfword.

3. The files used by PUTAREA and GETAREA must have RECFM=U and a blocksize of at least 80 bytes.

4. Programs using GETAREA and PUTAREA should not use a file named 'AREAFIL', as this name is used for the file used by GETAREA and PUTAREA.

5. The template array must be FIXED BINARY (15), 1 dimensional, have a lower bound of 1 and no more than 4095 elements.

6. No template may contain a nesting of deeper than 10 logical levels.
ERROR MESSAGES

The subroutines attempt to check their parameters for validity. Also if during the course of processing, an error is detected, an error indication is given. If an error is detected the subprogram signals the ERROR on-condition, and places an error code in the last element of the template array. This error code can be read if the calling program has established an on-ERROR block in which he prints the contents of the last element of the template array. The on-block may not return control to the subprogram, but must either end the program or make a branch to some other part of the main program. If the on-block attempts to return to the subprogram in which the error was detected the subprogram will cause an ABEND 0.

PUTAREA - error codes

1 - Lower bound of template array is not 1
2 - Output file blocksize less than 80 bytes
3 - Template-index within accessible structure is out of bounds of template array.
4 - Bad template. Detected for fixed length structure.
5 - Bad template. Detected for adjustable length structure.
6 - Error in length calculation, probably bad template.
999 - Unknown error, probably bad template.

GETAREA - error codes

11 - Lower bound of template array not 1.
12 - No records in input file.
13 - Template array parameter too small, can't receive old templates from file.
14 - Area too small to receive all structures from file.
999 - Unknown error, probably bad templates.

MOVAREA - error codes

21 - Lower bound of template array not 1.
22 - Template index within accessible structure out of range of template array.
23 - Target area is same as Source area.
24 - Target area too small to receive all accessible structures.
25 - Bad template. Probably bad level numbers.
26 - Bad template detected in length of structure calculation.
999 - Unknown error. Probably bad template.

GARBCL - error codes

30 - Bad template. Detected while calculating length.
31 - Lower bound of template array not 1
32 - Template-index within accessible structure not within bounds of template array.
33 - Bad template. Logical level.
34 - Accessible record was not within the area parameter.
35 - Unknown error. Detected within collection pass.
BIBLIOGRAPHY


Appendix

The following PL/I program illustrates how the subprograms GARBCOL, PUTAREA, GETAREA, and ILARGE might be used. The program builds and maintains a data base consisting of records containing the name and age of different individuals.

The program reads data cards containing a code number and associated information for each code. The code number indicates operation is to be performed on the data base.

<table>
<thead>
<tr>
<th>CODE</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Start up. Read file name. Read file by using GETAREA.</td>
</tr>
<tr>
<td>11</td>
<td>Start up. Read file name, but no file exists. Initialize data base as empty.</td>
</tr>
<tr>
<td>1</td>
<td>Read name and age. Insert into data base.</td>
</tr>
<tr>
<td>2</td>
<td>Read name. Delete that entry from data base.</td>
</tr>
<tr>
<td>3</td>
<td>Read name. Print age of person with that name.</td>
</tr>
<tr>
<td>4</td>
<td>Print contents of data base.</td>
</tr>
<tr>
<td>5</td>
<td>End of run. Write file out using PUTAREA.</td>
</tr>
</tbody>
</table>

The data base is kept as a preorder (by name) binary tree.

Note particularly the on-units at line numbers 186 and 194 within the program.

The area on-unit will be activated whenever an allocation is attempted, but there is not enough space left in the area variable to satisfy it. The on-unit prints a message that it was called and then calls GARBCOL to reclaim space within the area. It then checks, by means of a call to ILARGE, if there is now enough space to allocate the based structure. If there isn't, the program is halted. This check is important because if there were not enough space, the program could get into an infinite loop continually activating the on-unit until time ran out.

The ERROR on-unit is used to print an error code produced by the subprograms. It first checks to see if the ERROR on-condition was caused by a SIGNAL statement (ONCODE = 9). If so it prints the appropriate message. If the ERROR on-condition was not signaled then it was not caused by error-detection within the subprograms, and so no message is printed. In either case the program is halted.
TESTPGM: PROCEDURE OPTIONS (MAIN);

INSERT: PROCEDURE (NAME, AGE);
/* INSERTS RECORD WITH NAME AND AGE IF NAME NOT ALREADY
   THERE. IF NAME THERE AGE ENTRY IS CHANGED TO NEW VALUE */
DCL NAME CHAR(12), AGE FIXED BINARY(15), TEMPP POINTER;

IF (ROOT = NULL) THEN
   DO; ALLOCATE RECORD IN (AREA) SET(ROOT);
      CALL INITIAL(ROOT);
      ROOT->RECTYPE = RTYPE;
      ROOT->RECAGE = AGE;
      ROOT->RECNME = NAME;
      ROOT->LPNT, ROOT->RPNT = NULL;
   RETURN;
END;
TEMPP = ROOT;
LP: IF (TEMPP -> RECNME = NAME) THEN
    DO; TEMPP->RECAGE = AGE;
    RETURN;
ELSE IF (TEMPP -> RECNME > NAME) THEN
   IF (TEMPP -> LPNT = NULL) THEN
      DU; TEMPP = TEMPP -> LPNT;
   END;
   ELSE DO; /* LPNT = NULL */
      ALLOCATE RECORD IN (AREA);
      CALL INITIAL (RECPNT);
      RECTYPE = RTYPE;
      RECAE = AGE;
      RECNME = NAME;
      LPNT, RPNT = NULL;
      TEMPP -> LPNT = RECPNT;
      RETURN;
   END;
ELSE IF (TEMPP -> RPNT = NULL) THEN
   DU; ALLOCATE RECORD IN (AREA);
   CALL INITIAL(RECPNT);
   RECTYPE = RTYPE;
   RECAE = AGE;
   RECNME = NAME;
   LPNT, KPNT = NULL;
   TEMPP -> RPNT = RECPNT;
   RETURN;
END;
ELSE DU; TEMPP = TEMPP -> RPNT;
   GO TO LP;
END;
END INSERT;
DELETE: PROCEDURE (NAME);
/* DELETE CAUSES THE DELETION OF THE RECORD WHOSE RECNAME FIELD IS NAME. */
DCL NAME CHAR(12), (P1, P2, PTRL, PTRR, TEMPP) POINTER;
DCL PNTR BASED (HOLE) POINTER;

IF ROOT = NULL THEN RETURN; /* NO RECORDS IN FILE */
TEMPP = ROOT; HOLE = ADDR(ROOT);
DO WHILE (TEMPP->RECNAME != NAME);
   IF (TEMPP->RECNAME > NAME) THEN
      DO; HOLE = ADDR(TEMPP->LPNT);
         TEMPP = TEMPP->LPNT;
      END;
   ELSE DO; HOLE = ADDR(TEMPP->RPNT);
      TEMPP = TEMPP->RPNT;
   END;
   IF (TEMPP = NULL) THEN RETURN; /* NOT THERE*/
END;

/* TEMPP POINTS TO RECORD TO BE DELETED */
PTRL = TEMPP->LPNT;
PTRR = TEMPP->RPNT;
IF (PTRL = NULL) THEN
   DO; /* NULL LEFT SUBTREE */
      HOLE -> PNTR = PTRR;
      RETURN;
END;
ELSE IF (PTRR = NULL) THEN
   DO; HOLE -> PNTR = PTRL;
      RETURN;
END;

/* FIND NEXT RECORD (ALPHABETICALLY) AFTER DELETED RECORD */
IF PTRR -> LPNT = NULL THEN
   DO; /* RIGHT SON OF DELETED NODE NEXT */
      PTRR -> LPNT = PTRL;
      HOLE -> PNTR = PTRR;
      RETURN;
END;

/* WILL HAVE TO SEARCH FOR IT */
P2 = PTRR;
DO WHILE (P2 -> LPNT = NULL);
   P1 = P2;
   P2 = P2 -> LPNT;
END;
P1 -> LPNT = P2 -> RPNT;
P2 -> LPNT = PTRL;
P2 -> RPNT = PTRR;
HOLE -> PNTR = P2;
RETURN;
END DELETE;

-A3-
IFIND:  PROCEDURE (NAME) RETURNS (FIXED BINARY(15));
   DCL NAME CHAK (12);
   DCL TEMPP POINTER;

   TEMPP = ROOT;
   DO WHILE (TEMPP /= NULL);
     IF (TEMPP->RECNME = NAME) THEN
       RETURN(TEMPP->RECAGE);
     ELSE IF (TEMPP->RECNME > NAME) THEN
       TEMPP = TEMPP->LPNT;
     ELSE TEMPP = TEMPP->RPNT;
   END;
   RETURN (-1); /* INDICATES NOT IN FILE */
END IFIND;

PRINTALL:  PROCEDURE;
   DCL STACK(1:20) POINTER;
   DCL STK FIXED BINARY (15), TEMPP POINTER;

   IF (ROOT = NULL) THEN
     DO; PUT EDIT (*FILE EMPTY*) (SKIP, A);
        RETURN;
     END;
   ELSE PUT EDIT (*CONTENTS OF FILE*) (SKIP(2), A);
   PUT SKIP(2);
   STK = 1;
   STACK(1) = ROOT;
   TEMPP = ROOT->LPNT;

   LOOP:  DO WHILE (TEMPP /= NULL);
     STK = STK + 1;
     IF (STK > 20) THEN
       DO; PUT EDIT (*STACK OVERFLOW, PRINT FAILS*) (SKIP, A);
          RETURN;
     END;
     STACK(STK) = TEMPP;
     TEMPP = TEMPP->LPNT;
   END;

   IF STK = 0 THEN RETURN; /* PRINTING COMPLETE */
   TEMPP = STACK(STK);
   STK = STK - 1;
   PUT EDIT ( TEMPP->RECNME, TEMPP->RECAGE)(SKIP, A, F(5));
   TEMPP = TEMPP->RPNT;
   GO TO LOOP;
END PRINTALL;
DCL INSERT ENTRY (CHAR(12), FIXED BINARY(15)),
DELETE ENTRY (CHAR(12)),
IFIND ENTRY (CHAR(12)) RETURNS (FIXED BINARY(15)),
PRINT ENTRY;

DCL (GETAREA, PUTAREA, GARBCOL) EXTERNAL ENTRY,
INITIAL EXTERNAL ENTRY (POINTER),
ILARGE EXTERNAL ENTRY (AREA(*)) RETURNS (FIXED BINARY(15));

DCL 1 RECORD BASED (RECPNT),
2 RECTYPE FIXED BINARY (15),
2 RECAGE FIXED BINARY (15),
2 RECNAME CHAR (12),
2 LPNT POINTER,
2 RPNT POINTER;
DCL KTYPE FIXED BINARY (15) INITIAL(1);
DCL ROOT POINTER;
DCL (CODE, AGE, CKOCOUNT) FIXED BINARY (15),
(FILENAME, NAME) CHAR (12),
TEMPLATE (7) FIXED BINARY (15)
INITIAL (24,-1,1,16,8,0);
DCL AREA AREA(1000);
DCL AREA244 LABEL;

/* GET FILENAME AND CODE TELLING WHETHER NEW FILE OR OLD FILE.
 CODE = 10 OLD FILE
 CODE = 11 NEW FILE
*/
GET LIST (CODE, FILENAME);
IF (CODE = 10) THEN /* OLD FILE */
    CALL GETAREA (FILENAME, TEMPLATE, AREA, ROOT);
ELSE IF (CODE = 11) THEN /* NEW FILE */
    ROOT = NULL;
ELSE DO; PUT EDIT (' CODE ON FIRST DATA CARD INVALID. HALT')
      (SKIP, A);
    STOP;
END;
ON AREA
BEGIN; PUT EDIT ('AREA OVERFLOW ')(SKIP, A);
CALL GARBCOL (TEMPLATE, AREA, ROOT);
IF (ILARGE(AREA) < 24) THEN
    DO; PUT EDIT ('AREA TOO SMALL TO CONTINUE. HALT')
      (SKIP, A);
    STOP;
END;
ELSE PUT EDIT('SUCCESSFUL GARBAGE COLLECTION.')(SKIP, A);
END;
ON ERROR
BEGIN; IF (UNCODE = 9) THEN /* CAUSED BY SIGNAL ERROR */
    PUT EDIT ('ERROR IN STORAGE MANAGEMENT. ERRORCODE =',
      TEMPLATE(7))(SKIP, A, F(5));
    STOP;
END;
/* INITIALIZATION COMPLETE NOW UPDATE FILE
 CODE = 1 INSERT NAME, AGE
 CODE = 2 DELETE NAME
 CODE = 3 PRINT AGE
 CODE = 4 PRINT ALL
 CODE = 5 END WRITE OUT FILE
*/
GET LIST (CODE) SKIP;
CRDCOUNT = 2;
DO WHILE (CODE #= 5):
    IF (CODE < 0) OR (CODE > 4) THEN
        DO; PUT EDIT('*** INVALID CODE = ', CODE, '. CARD NUMBER ',
                    CRDCOUNT, ' IGNORED.') (SKIP, A, F(5), A, F(5), A);
        GO TO NEXTCODE;
    END;
    GO TO TAB(CODE);
TAB(1): /* INSERT */
    GET LIST (NAME, AGE);
    PUT EDIT ('INSERT ', NAME, AGE) (SKIP, A, A, F(5));
    CALL INSERT(NAME, AGE);
    GO TO NEXTCODE;
TAB(2): /* DELETE */
    GET LIST (NAME);
    PUT EDIT ('DELETE ', NAME) (SKIP, A, A);
    CALL DELETE(NAME);
    GO TO NEXTCODE;
TAB(3): /* PRINT AGE */
    GET LIST (NAME);
    PUT EDIT ('AGE OF ', NAME, ' IS ') (SKIP, A, A, A);
    I = IFIND(NAME);
    IF (I = -1) THEN PUT EDIT ('UNKNOWN') (A);
    ELSE PUT EDIT (I) (F(5));
    GO TO NEXTCODE;
TAB(4): /* PRINTALL */
    CALL PRINTALL;
NEXTCODE: GET LIST (CODE) SKIP;
    CRDCOUNT = CRDCOUNT +1;
END;

/* FINISHED NOW WRITE OUT FILE */
CALL PUTAREA( FILENAME, TEMPLATE, AREA, ROOT);
PUT EDIT ('FILE = ', FILENAME, ' WRITTEN OUT.') (SKIP, A, A, A);
PUT LIST ('TASK COMPLETE ') SKIP;
END TESTPGM;

-A-6-