Chapter 19: Handling of Arrays, Strings and Other Data Structures

Up to this point, we have studied simple data types and basic arrays built on those simple data types. Some of the simple data types studied include:

a) Integers: both halfword and fullword.
b) Packed decimal
c) Character data.

This lecture will cover the following:
1. A generalized “self describing” array that includes limits on the permitted index values. Only 1–D and 2–D arrays will be considered.
2. Options for a string data type and how that differs from a character array.
3. Use of indirect addressing with pointer structures generalized to include descriptions of the data item pointed to.

Structures of Arrays

We first consider the problem of converting an index in a one–dimensional array into an byte displacement. We then consider two ways of organizing a two–dimensional array, and proceed to convert the index pair into a byte displacement.

The simple array type has two variants:

0–based: The first element in the array is either AR[0] for a singly dimensioned array or AR[0][0] for a 2–D array.

1–based: The first element in the array is either AR[1] for a singly dimensioned array or AR[1][1] for a 2–D array.

We shall follow the convention of using only 0–based arrays. One reason is that it allows for efficient conversion from a set of indices into a displacement from the base address. By definition, the base address of an array will be the address of its first element: either the address of AR[0] or AR[0][0].

Byte Displacement and Addressing Array Elements: The General Case

We first consider addressing issues for an array that contains either character halfword, or fullword data. It will be constrained to one of these types. The addressing issue is well illustrated for a singly dimensioned array.

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fullwords</td>
<td>FW[0]</td>
<td>FW[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each of these examples, suppose that the array begins at address X. In other words, the address declared for the array is that of its element 0.

The character entries would be: C[0] at X, C[1] at X + 1, C[2] at X + 2, etc.
The halfword entries would be: HW[0] at X, HW[1] at X + 2, etc.
The fullword entries would be: FW[0] at X, FW[1] at X + 4, etc.
Byte Displacement and Addressing Array Elements: Our Case

I have decided not to write macros that handle the general case, but to concentrate on arrays that store 4-byte fullwords. The goal is to focus on array handling and not macro writing. The data structure for such an array will be designed under the following considerations.

1. It must have a descriptor specifying the maximum allowable index.
   In this data structure, I store the size and derive the maximum index.

2. It might store a descriptor specifying the minimum allowable index.
   For a 0-based array, that index is 0.

3. It should be created by a macro that allows the size to be specified at assembly time. Once specified, the array size will not change.

In this design, I assume the following:

1. The array is statically allocated; once loaded, its size is set.
2. The array is “zero based”; its first element has index 0. I decide to include this “base value” in the array declaration, just to show how to do it.
3. The array is self-describing for its maximum size.

Here is an example of the proposed data structure as it would be written in System 370 Assembler. The array is named “ARRAY”.

```assembly
ARBASE   DC F'0'      THE FIRST INDEX IS 0
ARSIZE   DC F'100'    SIZE OF THE ARRAY
ARRAY    DC 100F'0'   STORAGE FOR THE ARRAY
```

I want to generalize this to allow for a macro construction that will specify both the array name and its size.

The Constructor for a One-Dimensional Array

Here is the macro I used to construct a one-dimensional array while following the design considerations listed above.

```assembly
33 MACRO
34 &L1 ARMAKE &NAME,&SIZE
35 &L1 B X&SYSNDX
36 &NAME.B DC F'0'  ZERO BASED ARRAY
37 &NAME.S DC F'&SIZE'
38 &NAME.V DC &SIZE.F'0'
39 X&SYSNDX SLA R3,0
40 MEND
```

Line 34: The macro is named “ARMAKE” for “Array Make”.
   It takes two arguments: the array name and array size.
   A typical invocation: ARMAKE XX,20 creates an array called XX.

Note the “&L1” on line 34 and repeated on line 35. This allows a macro definition to be given a label that will persist into the generated code.
More on the 1–D Constructor

33 MACRO
34 &L1 ARMAKE &NAME,&SIZE
35 &L1 B X&SYSNDX
36 &NAME.B DC F'0' ZERO BASED ARRAY
37 &NAME.S DC F'&SIZE'
38 &NAME.V DC &SIZE.F'0'
39 X&SYSNDX SLA R3,0
40 MEND

Line 35: A macro is used to generate a code sequence. Since I am using it to create a data structure, I must provide code to jump around the data, so that the data will not be executed. While it might be possible to place all invocations of this macro in a program location that will not be executed, I do not assume that.

Line 36: I put in the lower bound on the index just to show a typical declaration.

Line 37: This holds the size of the array.

Label Concatenations in the Constructor

33 MACRO
34 &L1 ARMAKE &NAME,&SIZE
35 &L1 B X&SYSNDX
36 &NAME.B DC F'0' ZERO BASED ARRAY
37 &NAME.S DC F'&SIZE'
38 &NAME.V DC &SIZE.F'0'
39 X&SYSNDX SLA R3,0
40 MEND

Recall that the system variable symbol &SYSNDX in a counter that contains a four digit number unique to the macro expansion.

Line 39 uses one style of concatenation to produce a unique label. Note that in this more standard concatenation, the system variable symbol is the postfix of the generated symbol; it is the second part of a two–part concatenation. Recall that the symbol &SYSNDX counts total macro expansions. For the third macro expansion; the label would be “X0003”.

Lines 36, 37, and 38 use another type of concatenation, based on the dot. This is due to the fact that the symbolic parameter &NAME is the prefix of the generated symbol; it is the first part of a two–part concatenation. If &NAME is XX, then the labels are XXB, XXS, and XXV.

As always, it is desirable to provide a few macro expansions just to show that all of the process, as described above, really works. What follows is a sequence of two array constructions using the macro just discussed.
Sample Expansions of the 1–D Constructor Macro

```
90    ARMAKE XX,20
000014 47F0 C06A 00070 91+  B X0003
000018 00000000 92+XXB DC F'0'
00001C 00000014 93+XXS DC F'20'
000020 0000000000000000 94+XXV DC 20F'0'
000070 8B30 0000 00000 95+X0003 SLA R3,0
```

Notice the labels generated. Note also the unconditional branch statements in lines 91 and 97; these prevent the accidental execution of data. The target for each of the branch statements is a do–nothing shift of a register by zero spaces; it might have been written using another construct had your author known of that construct at the time. Anyway, this works.

Two More Macros for 1–D Arrays
I now define two macros to use the data structure defined above. I call these **ARPUT** and **ARGET**. Each will use R4 as a working register.

Macro **ARPUT &NAME, &INDX** stores the contents of register R4 into the indexed element of the named 1–D array. Consider the high–level language statement **A2[10] = Y**. This becomes

```
L R4,Y
ARPUT A2,=F'10'
CHANGE ELEMENT 10
```

Consider the high–level language statement **Y = A3[20]**. This becomes

```
ARGET A3,=F'20'
GET ELEMENT 20
ST R4,Y
```

**NOTE:** For some reason, I decided to implement the index as a fullword when I wrote the code. I just continue the practice, though a halfword index would make more sense.

Design of the Macros
The two macros, **ARPUT** and **ARGET**, share much of the same design. Much is centered on proper handling of the index, passed as a fullword. Here are the essential processing steps.

1. The index value is examined. If it is negative, the macro exits.
2. If the value in the index is not less than the number of elements in the array, the macro exits. For N elements, valid indices are 0 ≤ K < N.
3. Using the SLA instruction, the index value is multiplied by 4 in order to get a byte offset from the base address.
4. **ARPUT** stores the value in R4 into the indexed address. **ARGET** retrieves the value at the indexed address and loads R4.
The ARPUT Macro
Here is the definition of the macro to store a value into the named array.

```
44 MACRO
45 &L2 ARPUT &NAME, &INDX
46 &L2 ST R3, S&SYSNDX
47 L R3, &INDX
48 C R3, &NAME.B
49 BL Z&SYSNDX
50 C R3, &NAME.S
51 BNL Z&SYSNDX
52 SLA R3, 2
53 ST R4, &NAME.V(R3)
54 B Z&SYSNDX
55 S&SYSNDX DC F'0'
56 Z&SYSNDX L R3, S&SYSNDX
57 MEND
```

Note the two labels, S&SYSNDX and Z&SYSNDX, generated by concatenation with the System Variable Symbol &SYSNDX. This allows the macro to use conditional branching.

ARPUT Expanded
Here is an invocation of ARPUT and its expansion.

```
107 ARPUT XX, =F'10'

000126 5030 C146 0014C 108+ ST R3, S0005
00012A 5830 C46A 00470 109+ L R3, =F'10'
00012E 5930 C012 00018 110+ C R3, XXB
000132 4740 C14A 00150 111+ BL Z0005
000136 5930 C016 0001C 112+ C R3, XXS
00013A 47B0 C14A 00150 113+ BNL Z0005
00013E 8B30 0002 00002 114+ SLA R3, 2
000142 5043 C01A 00020 115+ ST R4, XXV(R3)
000146 47F0 C14A 00150 116+ B Z0005
00014A 0000
00014C 00000000 117+S0005 DC F'0'
000150 5830 C146 0014C 118+Z0005 L R3, S0005
```

Note the labels generated by use of the System Variable Symbol &SYSNDX.

We now examine the actions of the macro ARPUT.

```
108+ ST R3, S0005
109+ L R3, =F'10'
```

Register R3 will be used to hold the index into the array. This line saves the value so that it can be restored at the end of the macro. Here we note that the generated line does not have a label; this reflects the fact that line 107, the macro invocation, is not labeled.

Register R3 is loaded with the index to be used for the macro. As the index was specified as a literal in the invocation, this is copied in the macro expansion. Were this a keyword macro, the literal would have to be specified in a different manner. This is a positional macro, which does not require special handling of literals.
ARPUT: Checking the Index Value

We continue our analysis of the macro. At this point, the value of the array index has been loaded into register R3, the original contents having been saved.

```
110+   C   R3, XXB
111+   BL   Z0005
112+   C   R3, XXS
113+   BNL   Z0005
```

This code checks that the index value is within permissible bounds. The requirement is that \( XXB \leq \text{Index} < XXS \).

If this is not met, the macro restores the value of R3 and exits. If the requirement is met, the index is multiplied by 4 in order to convert it into a byte displacement from element 0.

```
114+   SLA   R3, 2
```

Here is the code to store the value into the array, called XXV.

```
115+   ST   R4, XXV (R3)
116+   B   Z0005
117+S0005   DC   F'0'
118+Z0005   L   R3, S0005
```

Line 115   This is the actual store command.

Line 116   Note the necessity of branching around the stored value, so that the data will not be executed as if it were code.

Line 117   The save area for the macro.

Line 118   This restores the original value of R3. The macro can now exit.

The ARGET Macro

Here is the definition of the macro to retrieve a value from the named array.

```
61   MACRO
62   &L3   ARGET &NAME, &INDX
63   &L3   ST   R3, S&SYSNDX
64   L   R3, &INDX
65   C   R3, &NAME.B
66   BL   Z&SYSNDX
67   C   R3, &NAME.S
68   BNL   Z&SYSNDX
69   SLA   R3, 2
70   L   R4, &NAME.V(R3)
71   B   Z&SYSNDX
72 S&SYSNDX   DC   F'0'
73 Z&SYSNDX   L   R3, S&SYSNDX
74   MEND
```
ARGET Expanded

Here is an invocation of the macro and its expansion.

```
119  ARGET YY, =F'20'
```

```
000154 5030 C172 00178 120+  ST  R3, S0006
000158 5830 C46E 00474 121+  L  R3, =F'20'
00015C 5930 C072 00078 122+  C  R3, YYB
000160 4740 C176 0017C 123+  BL  Z0006
000164 5930 C076 0007C 124+  C  R3, YYS
000168 47B0 C176 0017C 125+  BNL  Z0006
00016C 8B30 0002 00002 126+  SLA  R3, 2
000170 5843 C07A 00080 127+  L  R4, YYV (R3)
000174 47F0 C176 0017C 128+  B  Z0006
000178 00000000 129+S0006  DC  F'0'
00017C 5830 C172 00178 130+S0006  L  R3, S0006
```

The only difference between this macro and ARPUT occurs in line 127 of the expansion. Here the value is loaded into register R4. Note that, in the sequence of macro expansions, ARPUT was expansion number 5 and ARGET was expansion number six; hence the labels here are “S0006” and “Z0006” rather than “S0005” and “Z0005”.

Row–Major and Column–Major 2–D Arrays

The mapping of a one–dimensional array to linear address space is simple. How do we map a two–dimensional array? There are three standard options: The two that we shall consider are called row–major order and column–major order.

Consider the array declared as INT A[2][3], using 32–bit integers, which occupy four bytes. In this array the first index can have values 0 or 1 and the second 0, 1, or 2. Suppose the first element is found at address A. The following table shows the allocation of these elements to the linear address space.

<table>
<thead>
<tr>
<th>Address</th>
<th>Row Major</th>
<th>Column Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A[0][0]</td>
<td>A[0][0]</td>
</tr>
<tr>
<td>A + 4</td>
<td>A[0][1]</td>
<td>A[1][0]</td>
</tr>
<tr>
<td>A + 8</td>
<td>A[0][2]</td>
<td>A[0][1]</td>
</tr>
<tr>
<td>A + 12</td>
<td>A[1][0]</td>
<td>A[1][1]</td>
</tr>
<tr>
<td>A + 16</td>
<td>A[1][1]</td>
<td>A[0][2]</td>
</tr>
</tbody>
</table>

The mechanism for Java arrays is likely to be somewhat different.

Addressing Elements in Arrays of 32–Bit Fullwords

Consider first a singly dimensioned array that holds 4–byte fullwords. The addressing is simple: \textbf{Address ( A[K] ) = Address ( A[0] ) + 4*K.}

Suppose that we have a two dimensional array declared as A[M][N], where each of M and N has a fixed positive integer value. Again, we assume 0–based arrays and ask for the address of an element A[K][J], assuming that \(0 \leq K < M\) and \(0 \leq J < N\). At this point, I must specify either row–major or column–major ordering.
As FORTRAN is the only major language to use column–major ordering, I shall assume row–major. The formula is as follows.

Element offset = \( K \cdot N + J \), which leads to a byte offset of \( 4 \cdot (K \cdot N + J) \); hence

Address \((A[K][J]) = \text{Address} \((A[0][0]) + 4 \cdot (K \cdot N + J)\)

Suppose that the array is declared as \( A[2][3] \) and that element \( A[0][0] \) is at address \( A \).

Address \((A[K][J]) = \text{Address} \((A[0][0]) + 4 \cdot (K \cdot 3 + J)\).

Element \( A[0][0] \) is at offset \( 4 \cdot (0 \cdot 3 + 0) = 0 \), or address \( A + 0 \).

Element \( A[0][1] \) is at offset \( 4 \cdot (0 \cdot 3 + 1) = 4 \), or address \( A + 4 \).

Element \( A[0][2] \) is at offset \( 4 \cdot (0 \cdot 3 + 2) = 8 \), or address \( A + 8 \).

Element \( A[1][0] \) is at offset \( 4 \cdot (1 \cdot 3 + 0) = 12 \), or address \( A + 12 \).

Element \( A[1][1] \) is at offset \( 4 \cdot (1 \cdot 3 + 1) = 16 \), or address \( A + 16 \).

Element \( A[1][2] \) is at offset \( 4 \cdot (1 \cdot 3 + 1) = 20 \), or address \( A + 20 \).

Here is a first cut at what we might want the data structure to look like.

\[
\begin{align*}
\text{ARRB} & \quad \text{DC F'0'} \quad \text{ROW INDEX STARTS AT 0} \\
\text{ARRCNT} & \quad \text{DC F'30'} \quad \text{NUMBER OF ROWS} \\
\text{ARCB} & \quad \text{DC F'0'} \quad \text{COLUMN INDEX STARTS AT 0} \\
\text{ARCCNT} & \quad \text{DC F'20'} \quad \text{NUMBER OF COLUMNS} \\
\text{ARRAY} & \quad \text{DC 600F '0'} \quad \text{STORAGE FOR THE ARRAY}
\end{align*}
\]

\textbf{NOTE:} The number 600 in the declaration of the storage for the array is not independent of the row and column count. It is the product of the row and column count.

We need a way to replace the number 600 by 30\( \times \)20, indicating that the size of the array is a computed value. This leads us to the Macro feature called “SET Symbols”.

\textbf{SET Symbols}

The feature called “SET Symbols” allows for computing values in a macro, based on the values or attributes of the symbolic parameters. There are three basic types of SET symbols.

1. \textbf{Arithmetic} These are 32–bit numeric values, initialized to 0.
2. \textbf{Binary} These are 1–bit values, initialized to 0.
3. \textbf{Character} These are strings of characters, initialized to the null string.

Each of these comes in two varieties: local and global.

The \textbf{local} SET symbols have meaning only within the macro in which they are defined. In terms used by programming language textbooks, these symbols have scope that is local to the macro invocation. Declarations in different macro expansions are independent.

The \textbf{global} SET symbols specify values that are to be known in other macro expansions within the same assembly. In other words, the scope of such a symbol is probably the entire unit that is assembled independently; usually this is a CSECT.

A proper use of a global SET symbol demands the use of conditional assembly to insure that the symbol is defined once and only once.
Local and Global Set Declarations

Here are the instructions used to declare the SET symbols.

<table>
<thead>
<tr>
<th>Type</th>
<th>Local Instruction</th>
<th>Example</th>
<th>Global Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>LCLA</td>
<td>LCLA &amp;F1</td>
<td>GBLA</td>
<td>GBLA &amp;G1</td>
</tr>
<tr>
<td>Binary</td>
<td>LCLB</td>
<td>LCLB &amp;F2</td>
<td>GBLB</td>
<td>GBLB &amp;G2</td>
</tr>
<tr>
<td>Character</td>
<td>LCLC</td>
<td>LCLC &amp;F3</td>
<td>GBLC</td>
<td>GBLC &amp;G3</td>
</tr>
</tbody>
</table>

Each of these instructions declares a SET symbol that can have its value assigned by one of the SET instructions. There are three SET instructions.

- **SETA** SET Arithmetic Use with LCLA or GBLA SET symbols.
- **SETB** SET Binary Use with LCLB or GBLB SET symbols.
- **SETC** SET Character String Use with LCLC or GBLC SET symbols.

The requirements for placement of these instructions depend on the Operating System being run. The following standards have two advantages:

1. They are the preferred practice for clear programming, and
2. They seem to be accepted by every version of the Operating System.

Here is the sequence of declarations.

1. The macro prototype statement.
2. The global declarations used: GBLA, GBLB, or GBLC
3. The local declarations used: LCLA, LCLB, or LCLC
4. The appropriate SET instructions to give values to the SET symbols
5. The macro body.

Example of the Preferred Sequence

The following silly macro is not even complete. It illustrates the sequence for declaration, but might be incorrect in some details.

```assembly
MACRO &NAME HEDNG &HEAD, &PAGE
  GBLC &DATES HOLDS THE DATE
  GBLB &DATEP HAS DATES BEEN DEFINED
  LCLA &LEN, &MID HERE IS A LOCAL DECLARATION
  AIF (&DATEP).N20 IS DATE DEFINED?
  &DATES DC C`&SYSDATE` SET THE DATE
  &DATEP SETB (1) DECLARE IT SET
  .N20 ANOP
  &LEN SETA L`&HEAD LENGTH OF THE HEADER
  &MID SETA (120-&LEN)/2 MID POINT
  &NAME Start of macro body.
```
A Constructor Macro for the 2–D Array

This macro uses one arithmetic SET symbol to calculate the array size. Note that this definition does away with the base for the row and column numbers, as I never use them.

```
  30 *
  31 * MACRO DEFINITIONS
  32 *
  33 MACRO
  34 &L1 ARMAK2D &NAME,&ROWS,&COLS
  35 LCLA &SIZE
  36 &SIZE SETA (&ROWS*&COLS)
  37 &L1 B X&SYSNDX
  38 &NAME.RS DC H'&ROWS'
  39 &NAME.CS DC H'&COLS'
  40 &NAME.V DC &SIZE.F'0'
  41 X&SYSNDX SLA R3,0
  42 MEND
```

Here are two invocations of the macro `ARMAK2D` and their expansions.

```
86 ARMAK2D XX,10,20
00004A 47F0 C36E 00374 87+ B X0009
00004E 000A 88+XXRS DC H'10'
000050 0014 89+XXCS DC H'20'
000052 0000
000054 0000000000000000 90+XXV DC 200F'0'
000374 8B30 00000 00000 91+X0009 SLA R3,0
92 ARMAK2D YY,4,8
000378 47F0 C3FA 00400 93+ B X0010
00037C 0004 94+YRS DC H'4'
00037E 0008 95+YCS DC H'8'
000380 0000000000000000 96+YYV DC 32F'0'
000400 8B30 0000 00000 97+X0010 SLA R3,0
```

Please note the hexadecimal addresses at the left of the listing. Line 90 corresponds to byte address `X'0054'` and line 91 to byte address `X'0374'`. The difference is given by `X'320'`, which is decimal 800. Line 90 reserves 200 fullwords, which occupy 800 bytes.

The storage allocation indicated by lines 96 and 97 is similar. Line 96 sets aside thirty–two fullwords, for an allocation of 128 bytes. `X'400' – X'320' = X'80' = 128 in decimal.

The `ARGET2D` and `ARPUT2D` macros would be based on the similar macros discussed above. Each would take three arguments, and have prototypes as follows:

- `ARGET2D &NAME,&ROW,&COL`
- `ARPUT2D &NAME,&ROW,&COL`

Each macro would insure that the row and column numbers were within bounds, and then calculate the offset into the block of storage set aside for the array. The writing of the actual code for each of these macros is left as an exercise for the reader.
**Strings vs. Arrays of Characters**

While a string may be considered an array of characters, this is not the normal practice. A string is a sequence of characters with a fixed length. A string is stored in “string space”, which may be considered to be a large dynamically allocated array that contains all of the strings used. There are two ways to declare the length of a string.

1. Allot a special “end of string” character, such as the character with code X‘00’, as done in C and C++.
2. Store an explicit string length code, usually as a single byte that prefixes the string. A single byte can store an unsigned integer in the range 0 through 255 inclusive.

   In this method, the maximum string length is 255 characters.

There are variants on these two methods; some are worth consideration.

**Example String**

In this example, I used strings of digits that are encoded in EBCDIC. The character sequence “12345” would be encoded as F1 F2 F3 F4 F5. This is a sequence of five characters. In either of the above methods, it would require six bytes to be stored.

Here is the string, as would be stored by C++.

<table>
<thead>
<tr>
<th>Byte number</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
<td>F5</td>
<td>00</td>
</tr>
</tbody>
</table>

Here is the string, as might be stored by Visual Basic Version 6.

<table>
<thead>
<tr>
<th>Byte number</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>05</td>
<td>F1</td>
<td>F2</td>
<td>F3</td>
<td>F4</td>
<td>F5</td>
</tr>
</tbody>
</table>

Each method has its advantages. The main difficulty with the first approach, as it is implemented in C and C++, is that the programmer is responsible for the terminating X‘00’. Failing to place that can lead to strange run–time errors.

**Sharing String Space**

String variables usually are just pointers into string space. Consider the following example in the style of Visual Basic.

- **C1**
  - 0
  - 4
  - F1
  - F3
  - F0
  - F1
  - 0
  - 4
  - F1
  - 0

- **C2**
  - 0
  - 4
  - F2
  - F1
  - 0
  - 4
  - F2
  - F1
  - 0

- **C3**
  - 0
  - 4
  - F2
  - F1
  - 0
  - 4
  - F8

Here, each of the symbols C1, C2, and C3 references a string of length 4.

  - C1 references the string “1301”
  - C2 references the string “1302”
  - C3 references the string “2108”
Using Indirect Pointers with Attributes
Another string storage method uses indirect pointers, as follows.

| P1  | 0 2 | C 3  |
| P2  | 0 8 | D 7  |
| P3  |    | E 2  |
| P4  |    | C 3  |
| P5  | 0 3 | F 2  |
|     | 0 8 | F 1  |

Here the intermediate node has the following structure.
1. A reference count
2. The string length
3. A pointer into string space.

There are two references to the first string, of length 8: “CPSC 2105”.

There are three references to the second string, also of length 8: “CPSC 2108”.

There are many advantages to this method of indirect reference, with attributes stored in the intermediate node. It may be the method used by Java.

There are a number of advantages associated with this approach to string storage. In general the use of indirection in pointers, as illustrated above, simplifies system programming. Here are a few illustrations of some of the standard considerations:

1. If pointer P1 is deallocated, the reference count for the string “CPSC 2105” is reduced by 1, but the string is not removed.
2. If pointer P2 is deallocated, then the reference count for the string goes to 0, and the string space can be reclaimed.

Reclamation of dynamic memory (string space in this example) is a tricky business and often is simply not done. However, the existence of the intermediate pointer facilitates such an operation. Suppose that the string “CPSC 2105” is removed and the string “CPSC2108” is moved up by eight positions. There is only one value that needs to be reassigned, and it is not the addresses associated with pointers P3, P4, or P5. It is the address in the intermediate node that each of P3, P4, and P5 reference. This intermediate node is easy to locate.