Arrays Chapter Four

4.1 Chapter Overview

This chapter discusses how to declare and use arrays in your assembly language programs. This is probably the most important chapter on composite data structures in this text. Even if you elect to skip the chapters on Strings, Character Sets, Records, and Dates and Times, be sure you read and understand the material in this chapter. Much of the rest of the text depends on your understanding of this material.

4.2 Arrays

Along with strings, arrays are probably the most commonly used composite data type. Yet most beginning programmers have a very weak understanding of how arrays operate and their associated efficiency trade-offs. It's surprising how many novice (and even advanced!) programmers view arrays from a completely different perspective once they learn how to deal with arrays at the machine level.

Abstractly, an array is an aggregate data type whose members (elements) are all the same type. Selection of a member from the array is by an integer index[1]. Different indices select unique elements of the array. This text assumes that the integer indices are contiguous (though this is by no means required). That is, if the number *x* is a valid index into the array and *y* is also a valid index, with $x < y$, then all *i* such that $x < i < y$ are valid indices into the array.

Whenever you apply the indexing operator to an array, the result is the specific array element chosen by that index. For example, *A[i]* chooses the i^{th} element from array *A*. Note that there is no formal requirement that element *i* be anywhere near element *i+1* in memory. As long as *A[i]* always refers to the same memory location and *A[i+1]* always refers to its corresponding location (and the two are different), the definition of an array is satisfied.

In this text, we will assume that array elements occupy contiguous locations in memory. An array with five elements will appear in memory as shown in Figure 4.1

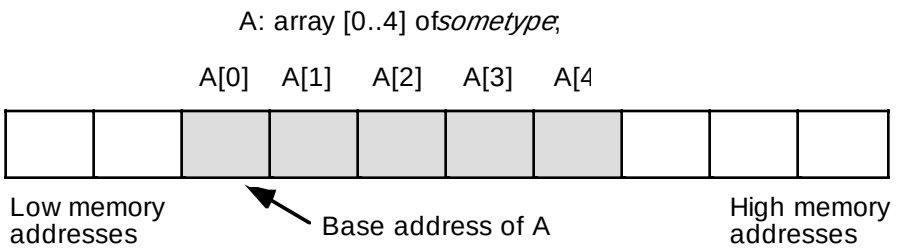

Figure 4.1 Array Layout in Memory

The *base address* of an array is the address of the first element on the array and always appears in the lowest memory location. The second array element directly follows the first in memory, the third element follows the second, etc. Note that there is no requirement that the indices start at zero. They may start with any number as long as they are contiguous. However, for the purposes of discussion, it's easier to discuss accessing array elements if the first index is zero. This text generally begins most arrays at index zero unless

1. Or some value whose underlying representation is integer, such as character, enumerated, and boolean types.

there is a good reason to do otherwise. However, this is for consistency only. There is no efficiency benefit one way or another to starting the array index at zero.

To access an element of an array, you need a function that translates an array index to the address of the indexed element. For a single dimension array, this function is very simple. It is

```
Element_Address = Base_Address + ((Index - Initial_Index) * Element_Size)
```

where *Initial_Index* is the value of the first index in the array (which you can ignore if zero) and the value *Element_Size* is the size, in bytes, of an individual element of the array.

4.3 Declaring Arrays in Your HLA Programs

Before you access elements of an array, you need to set aside storage for that array. Fortunately, array declarations build on the declarations you've seen thus far. To allocate *n* elements in an array, you would use a declaration like the following in one of the variable declaration sections:

ArrayName: *basetype*[n];

ArrayName is the name of the array variable and *basetype* is the type of an element of that array. This sets aside storage for the array. To obtain the base address of the array, just use *ArrayName*.

The "[n]" suffix tells HLA to duplicate the object *n* times. Now let's look at some specific examples:

```
static

    CharArray: char[128];       // Character array with elements 0..127.
    IntArray: integer[ 8 ];     // "integer" array with elements 0..7.
    ByteArray: byte[10];        // Array of bytes with elements 0..9.
    PtrArray: dword[4];         // Array of double words with elements 0..3.
```

The second example, of course, assumes that you have defined the *integer* data type in the TYPE section of the program.

These examples all allocate storage for uninitialized arrays. You may also specify that the elements of the arrays be initialized to a single value using declarations like the following in the STATIC and READONLY sections:

```
RealArray: real32[8] := [ 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0 ];
IntegerAry: integer[8] := [ 1, 1, 1, 1, 1, 1, 1, 1 ];
```

These definitions both create arrays with eight elements. The first definition initializes each four-byte real value to 1.0, the second declaration initializes each integer element to one. Note that the number of constants within the square brackets must match the size you declare for the array.

This initialization mechanism is fine if you want each element of the array to have the same value. What if you want to initialize each element of the array with a (possibly) different value? No sweat, just specify a different set of values in the list surrounded by the square brackets in the example above:

```
RealArray: real32[8] := [ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 ];
IntegerAry: integer[8] := [ 1, 2, 3, 4, 5, 6, 7, 8 ];
```

4.4 HLA Array Constants

The last few examples in the last section demonstrate the use of HLA array constants. An HLA array constant is nothing more than a list of values (all the same time) surrounded by a pair of brackets. The following are all legal array constants:

```
[ 1, 2, 3, 4 ]
[ 2.0, 3.14159, 1.0, 0.5 ]
```



```
        [ 'a', 'b', 'c', 'd' ]
[ "Hello", "world", "of", "assembly" ]
```

(note that this last array constant contains four double word pointers to the four HLA strings appearing elsewhere in memory.)

As you saw in the previous section you can use array constants in the STATIC and READONLY sections to provide initial values for array variables. Of course, the number of comma separated items in an array constant must exactly match the number of array elements in the variable declaration. Likewise, the type of the array constant's elements must match the type of the elements in the array variable.

Using array constants to initialize small arrays is very convenient. Of course, if your array has several thousand elements in it, typing them all in will not be very much fun. Most arrays initialized this way have no more than a couple hundred entries, and generally far less than 100. It is reasonable to use an array constant to initialize such variables. However, at some point it will become far too tedious and error-prone to initialize arrays in this fashion. It is doubtful, for example, that you would want to manually initialize an array with 1,000 different elements using an array constant[2]. However, if you want to initialize all the elements of an array with the same value, HLA does provide a special array constant syntax for doing so. Consider the following declaration:

```
BigArray: uns32[ 1000 ] := 1000 dup [ 1 ];
```

This declaration creates a 1,000 element integer array initializing each element of the array with the value one. The "1000 dup [1]" expression tells HLA to create an array constant by duplicating the single value "[1]" one thousand times. You can even use the DUP operator to duplicate a series of values (rather than a single value) as the following example indicates:

```
SixteenInts: int32[16] := 4 dup [1,2,3,4];
```

This example initializes *SixteenInts* with four copies of the sequence "1, 2, 3, 4" yielding a total of sixteen different integers (i.e., 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4).

You will see some more possibilities with the DUP operator when looking at multidimensional arrays a little later.

4.5 Accessing Elements of a Single Dimension Array

To access an element of a zero-based array, you can use the simplified formula:

Element_Address = Base_Address + index * Element_Size

For the *Base_Address* entry you can use the name of the array (since HLA associates the address of the first element of an array with the name of that array). The *Element_Size* entry is the number of bytes for each array element. If the object is an array of bytes, the *Element_Size* field is one (resulting in a very simple computation). If each element of the array is a word (or other two-byte type) then *Element_Size* is two. And so on. To access an element of the *SixteenInts* array in the previous section, you'd use the formula:

```
Element_Address = SixteenInts + index*4
```

The 80x86 code equivalent to the statement "EAX:=SixteenInts[index]" is

```
mov( index, ebx );
shl( 2, ebx );                      //Sneaky way to compute 4*ebx
mov( SixteenInts[ ebx ], eax );
```

There are two important things to notice here. First of all, this code uses the SHL instruction rather than the INTMUL instruction to compute *4*index*. The main reason for choosing SHL is that it was more efficient. It turns out that SHL is a *lot* faster than INTMUL on many processors.

2. In the chapter on Macros and the HLA Run-Time Language you will learn how to automate the initialization of large array objects. So initializing large objects is not completely out of the question.

The second thing to note about this instruction sequence is that it does not explicitly compute the sum of the base address plus the index times two. Instead, it relies on the indexed addressing mode to implicitly compute this sum. The instruction "mov(SixteenInts[ebx], eax);" loads EAX from location *SixteenInts*+EBX which is the base address plus *index**4 (since EBX contains *index**4). Sure, you could have used

```
lea( eax, SixteenInts );
mov( index, ebx );
shl( 2, ebx );                      //Sneaky way to compute 4*ebx
add( eax, ebx );                    //Compute base address plus index*4
mov( SixteenInts[ ebx ], eax );
```

in place of the previous sequence, but why use five instructions where three will do the same job? This is a good example of why you should know your addressing modes inside and out. Choosing the proper addressing mode can reduce the size of your program, thereby speeding it up.

Of course, as long as we're discussing efficiency improvements, it's worth pointing out that the 80x86 scaled indexed addressing modes let you automatically multiply an index by one, two, four, or eight. Since this current example multiplies the index by four, we can simplify the code even farther by using the scaled indexed addressing mode:

```
mov( index, ebx );
mov( SixteenInts[ ebx*4 ], eax );
```

Note, however, that if you need to multiply by some constant other than one, two, four, or eight, then you cannot use the scaled indexed addressing modes. Similarly, if you need to multiply by some element size that is not a power of two, you will not be able to use the SHL instruction to multiply the index by the element size; instead, you will have to use INTMUL or some other instruction sequence to do the multiplication.

The indexed addressing mode on the 80x86 is a natural for accessing elements of a single dimension array. Indeed, it's syntax even suggests an array access. The only thing to keep in mind is that you must remember to multiply the index by the size of an element. Failure to do so will produce incorrect results.

Before moving on to multidimensional arrays, a couple of additional points about addressing modes and arrays are in order. The above sequences work great if you only access a single element from the *SixteenInts* array. However, if you access several different elements from the array within a short section of code, and you can afford to dedicate another register to the operation, you can certainly shorten your code and, perhaps, speed it up as well. Consider the following code sequence:

```
lea( ebx, SixteenInts );
mov( index, esi );
mov( [ebx+esi*4], eax );
```

Now EBX contains the base address and ESI contains the *index* value. Of course, this hardly appears to be a good trade-off. However, when accessing additional elements if *SixteenInts* you do not have to reload EBX with the base address of *SixteenInts* for the next access. The following sequence is a little shorter than the comparable sequence that doesn't load the base address into EBX:

```
lea( ebx, SixteenInts );
mov( index, esi );
mov( [ebx+esi*4], eax );
 .
 .                          //Assumption: EBX is left alone
 .                          //            through this code.
mov( index2, esi );
mov( [ebx+esi*4], eax );
```

This code is slightly shorter because the "mov([ebx+esi*4], eax);" instruction is slightly shorter than the "mov(SixteenInts[ebx*4], eax);" instruction. Of course the more accesses to *SixteenInts* you make without reloading EBX, the greater your savings will be. Tricky little code sequences such as this one sometimes pay off handsomely. However, the savings depend entirely on which processor you're using. Code

sequences that run faster on one 80x86 CPU might actually run *slower* on a different CPU. Unfortunately, if speed is what you're after there are no hard and fast rules. In fact, it is very difficult to predict the speed of most instructions on the 80x86 CPUs.

4.5.1 Sorting an Array of Values

Almost every textbook on this planet gives an example of a sort when introducing arrays. Since you've probably seen how to do a sort in high level languages already, it's probably instructive to take a quick look at a sort in HLA. The example code in this section will use a variant of the Bubble Sort which is great for short lists of data and lists that are nearly sorted, but horrible for just about everything else[3].

```
const
    NumElements:= 16;

static
    DataToSort: uns32[ NumElements ] :=
                    [
                        1, 2, 16, 14,
                        3, 9, 4,  10,
                        5, 7, 15, 12,
                        8, 6, 11, 13
                    ];

    NoSwap: boolean;

       .
       .
       .

    // Bubble sort for the DataToSort array:

    repeat

        mov( true, NoSwap );
        for( mov( 0, ebx ); ebx <= NumElements-2; inc( ebx )) do

            mov( DataToSort[ ebx*4], eax );
            if( eax > DataToSort[ ebx*4 + 4] ) then

                mov( DataToSort[ ebx*4 + 4 ], ecx );
                mov( ecx, DataToSort[ ebx*4 ] );
                mov( eax, DataToSort[ ebx*4 + 4 ] ); // Note: EAX contains
                mov( false, NoSwap );                // DataToSort[ ebx*4 ]

            endif;

        endfor;

    until( NoSwap );
```

The bubble sort works by comparing adjacent elements in an array. The interesting thing to note in this code fragment is how it compares adjacent elements. You will note that the IF statement compares EAX (which contains DataToSort[ebx*4]) against DataToSort[EBX*4 + 4]. Since each element of this array is four bytes (*uns32*), the index [EBX*4 + 4] references the next element beyond [EBX*4].

3. Fear not, you'll see some better sorting algorithms in the chapter on procedures and recursion.

As is typical for a bubble sort, this algorithm terminates if the innermost loop completes without swapping any data. If the data is already presorted, then the bubble sort is very efficient, making only one pass over the data. Unfortunately, if the data is not sorted (worst case, if the data is sorted in reverse order), then this algorithm is extremely inefficient. Indeed, although it is possible to modify the code above so that, on the average, it runs about twice as fast, such optimizations are wasted on such a poor algorithm. However, the Bubble Sort is very easy to implement and understand (which is why introductory texts continue to use it in examples). Fortunately, you will learn about more advanced sorts later in this text, so you won't be stuck with it for very long.

4.6 Multidimensional Arrays

The 80x86 hardware can easily handle single dimension arrays. Unfortunately, there is no magic addressing mode that lets you easily access elements of multidimensional arrays. That's going to take some work and lots of instructions.

Before discussing how to declare or access multidimensional arrays, it would be a good idea to figure out how to implement them in memory. The first problem is to figure out how to store a multi-dimensional object into a one-dimensional memory space.

Consider for a moment a Pascal array of the form "A:array[0..3,0..3] of char;". This array contains 16 bytes organized as four rows of four characters. Somehow you've got to draw a correspondence with each of the 16 bytes in this array and 16 contiguous bytes in main memory. Figure 4.2 shows one way to do this:

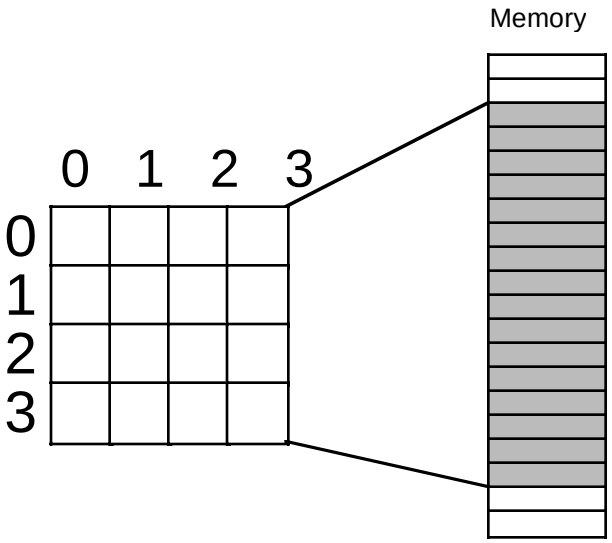

Figure 4.2 Mapping a 4x4 Array to Sequential Memory Locations

The actual mapping is not important as long as two things occur: (1) each element maps to a unique memory location (that is, no two entries in the array occupy the same memory locations) and (2) the mapping is consistent. That is, a given element in the array always maps to the same memory location. So what you really need is a function with two input parameters (row and column) that produces an offset into a linear array of sixteen memory locations.

Now any function that satisfies the above constraints will work fine. Indeed, you could randomly choose a mapping as long as it was unique. However, what you really want is a mapping that is efficient to compute at run time and works for any size array (not just 4x4 or even limited to two dimensions). While there are a

large number of possible functions that fit this bill, there are two functions in particular that most programmers and most high level languages use: *row major ordering* and *column major ordering.*

4.6.1 Row Major Ordering

Row major ordering assigns successive elements, moving across the rows and then down the columns, to successive memory locations. This mapping is demonstrated in Figure 4.3:

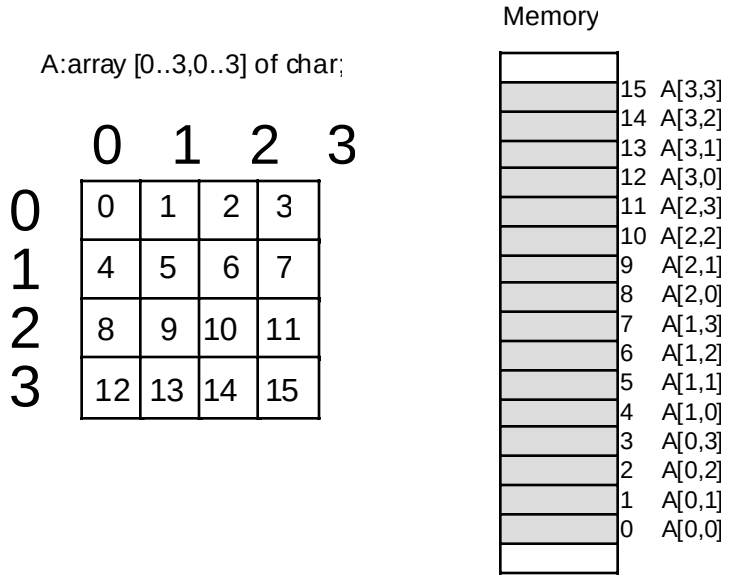

Figure 4.3 Row Major Array Element Ordering

Row major ordering is the method employed by most high level programming languages including Pascal, C/C++, Java, Ada, Modula-2, etc. It is very easy to implement and easy to use in machine language. The conversion from a two-dimensional structure to a linear array is very intuitive. You start with the first row (row number zero) and then concatenate the second row to its end. You then concatenate the third row to the end of the list, then the fourth row, etc. (see Figure 4.4).

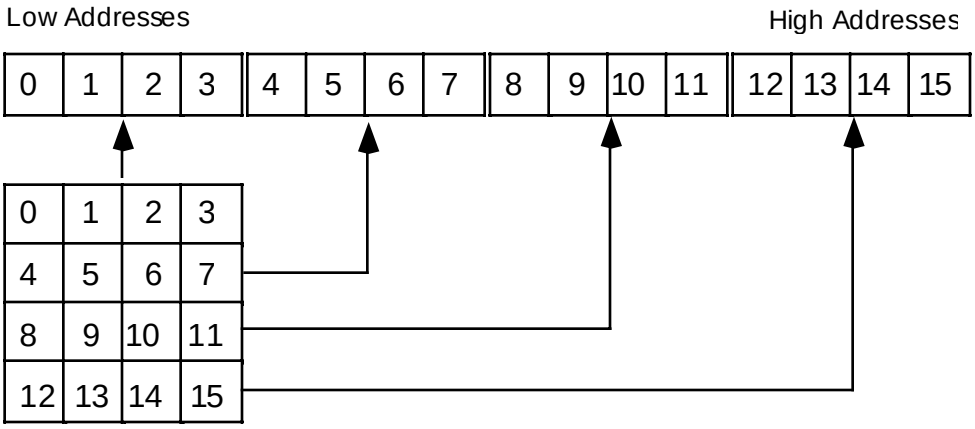

Figure 4.4 Another View of Row-Major Ordering for a 4x4 Array

For those who like to think in terms of program code, the following nested Pascal loop also demonstrates how row major ordering works:

```
index := 0;
for colindex := 0 to 3 do
    for rowindex := 0 to 3 do
    begin

        memory [index] := rowmajor [colindex][rowindex];
        index := index + 1;
    end;
```

The important thing to note from this code, that applies regardless of the number of dimensions, is that the rightmost index increases the fastest. That is, as you allocate successive memory locations you increment the rightmost index until you reach the end of the current row. Upon reaching the end, you reset the index back to the beginning of the row and increment the next successive index by one (that is, move down to the next row.). This works equally well for any number of dimensions[4]. The following Pascal segment demonstrates row major organization for a 4x4x4 array:

```
index := 0;
for depthindex := 0 to 3 do
    for colindex := 0 to 3 do
       for rowindex := 0 to 3 do begin

        memory [index] := rowmajor [depthindex][colindex][rowindex];
        index := index + 1;

       end;
```

The actual function that converts a list of index values into an offset doesn't involve loops or much in the way of fancy computations. Indeed, it's a slight modification of the formula for computing the address of an element of a single dimension array. The formula to compute the offset for a two-dimension row major ordered array declared in Pascal as "A:array [0..3,0..3] of integer" is

```
Element_Address = Base_Address + (colindex * row_size + rowindex) * Element_Size
```

As usual, *Base_Address* is the address of the first element of the array (*A[0][0]* in this case) and *Element_Size* is the size of an individual element of the array, in bytes. *Colindex* is the leftmost index, *rowindex* is the rightmost index into the array. *Row_size* is the number of elements in one row of the array (four, in

4. By the way, the number of dimensions of an array is its *arity.*

this case, since each row has four elements). Assuming *Element_Size* is one, this formula computes the following offsets from the base address:

```
Column      Row         Offset into Array
index       Index
0           0           0
0           1           1
0           2           2
0           3           3
1           0           4
1           1           5
1           2           6
1           3           7
2           0           8
2           1           9
2           2           10
2           3           11
3           0           12
3           1           13
3           2           14
3           3           15
```

For a three-dimensional array, the formula to compute the offset into memory is the following:

```
Address = Base + ((depthindex*col_size+colindex) * row_size + rowindex) * Element_Size
```

Col_size is the number of items in a column, *row_size* is the number of items in a row. In C/C++, if you've declared the array as "*type* A[i] [j] [k];" then *row_size* is equal to *k* and *col_size* is equal to *j*.

For a four dimensional array, declared in C/C++ as "type A[i] [j] [k] [m];" the formula for computing the address of an array element is

```
Address =
Base + (((LeftIndex * depth_size + depthindex)*col_size+colindex) * row_size +
rowindex) * Element_Size
```

Depth_size is equal to *j*, *col_size* is equal to *k*, and *row_size* is equal to *m*. *LeftIndex* represents the value of the leftmost index.

By now you're probably beginning to see a pattern. There is a generic formula that will compute the offset into memory for an array with *any* number of dimensions, however, you'll rarely use more than four.

Another convenient way to think of row major arrays is as arrays of arrays. Consider the following single dimension Pascal array definition:

```
A: array [0..3] of  sometype;
```

Assume that *sometype* is the type "sometype = array [0..3] of char;".

A is a single dimension array. Its individual elements happen to be arrays, but you can safely ignore that for the time being. The formula to compute the address of an element of a single dimension array is

```
Element_Address = Base + Index * Element_Size
```

In this case *Element_Size* happens to be four since each element of *A* is an array of four characters. So what does this formula compute? It computes the base address of each row in this 4x4 array of characters (see Figure 4.5):

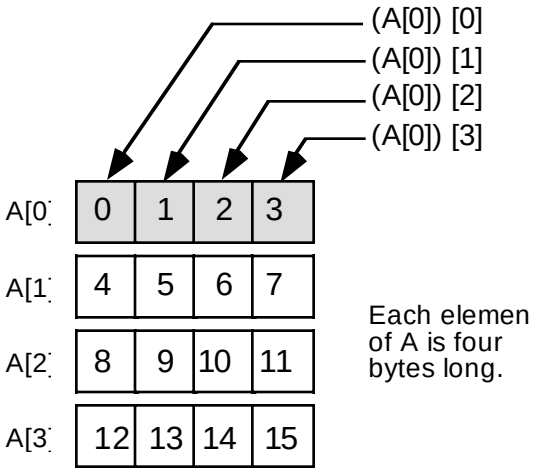

Figure 4.5 Viewing a 4x4 Array as an Array of Arrays

Of course, once you compute the base address of a row, you can reapply the single dimension formula to get the address of a particular element. While this doesn't affect the computation at all, conceptually it's probably a little easier to deal with several single dimension computations rather than a complex multidimensional array element address computation.

Consider a Pascal array defined as "A:array [0..3] [0..3] [0..3] [0..3] [0..3] of char;" You can view this five-dimension array as a single dimension array of arrays. The following Pascal code demonstrates such a definition:

```
type
        OneD = array [0..3] of char;
        TwoD = array [0..3] of OneD;
        ThreeD = array [0..3] of TwoD;
        FourD = array [0..3] of ThreeD;
var
        A : array [0..3] of FourD;
```

The size of *OneD* is four bytes. Since *TwoD* contains four *OneD* arrays, its size is 16 bytes. Likewise, *ThreeD* is four *TwoDs*, so it is 64 bytes long. Finally, *FourD* is four *ThreeDs*, so it is 256 bytes long. To compute the address of "A [b, c, d, e, f]" you could use the following steps:

- Compute the address of *A [b]* as "Base + *b* * size". Here size is 256 bytes. Use this result as the new base address in the next computation.
- Compute the address of *A [b, c]* by the formula "Base + *c**size", where *Base* is the value obtained immediately above and size is 64. Use the result as the new base in the next computation.
- Compute the address of *A [b, c, d]* by "Base + *d**size" with *Base* coming from the above computation and size being 16.
- Compute the address of *A [b, c, d, e]* with the formula "Base + *e**size" with Base from above and size being four. Use this value as the base for the next computation.
- Finally, compute the address of *A [b, c, d, e, f]* using the formula "Base + *f**size" where base comes from the above computation and size is one (obviously you can simply ignore this final multiplication). The result you obtain at this point is the address of the desired element.

Not only is this scheme easier to deal with than the fancy formulae given earlier, but it is easier to compute (using a single loop) as well. Suppose you have two arrays initialized as follows

A1 = [256, 64, 16, 4, 1] and A2 = [b, c, d, e, f]

then the Pascal code to perform the element address computation becomes:


```
for i := 0 to 4 do
    base := base + A1[i] * A2[i];
```

Presumably *base* contains the base address of the array before executing this loop. Note that you can easily extend this code to any number of dimensions by simply initializing *A1* and *A2* appropriately and changing the ending value of the for loop.

As it turns out, the computational overhead for a loop like this is too great to consider in practice. You would only use an algorithm like this if you needed to be able to specify the number of dimensions at run time. Indeed, one of the main reasons you won't find higher dimension arrays in assembly language is that assembly language displays the inefficiencies associated with such access. It's easy to enter something like "*A [b,c,d,e,f]*" into a Pascal program, not realizing what the compiler is doing with the code. Assembly language programmers are not so cavalier – they see the mess you wind up with when you use higher dimension arrays. Indeed, good assembly language programmers try to avoid two dimension arrays and often resort to tricks in order to access data in such an array when its use becomes absolutely mandatory. But more on that a little later.

4.6.2 Column Major Ordering

Column major ordering is the other function frequently used to compute the address of an array element. FORTRAN and various dialects of BASIC (e.g., older versions of Microsoft BASIC) use this method to index arrays.

In row major ordering the rightmost index increased the fastest as you moved through consecutive memory locations. In column major ordering the leftmost index increases the fastest. Pictorially, a column major ordered array is organized as shown in Figure 4.6:

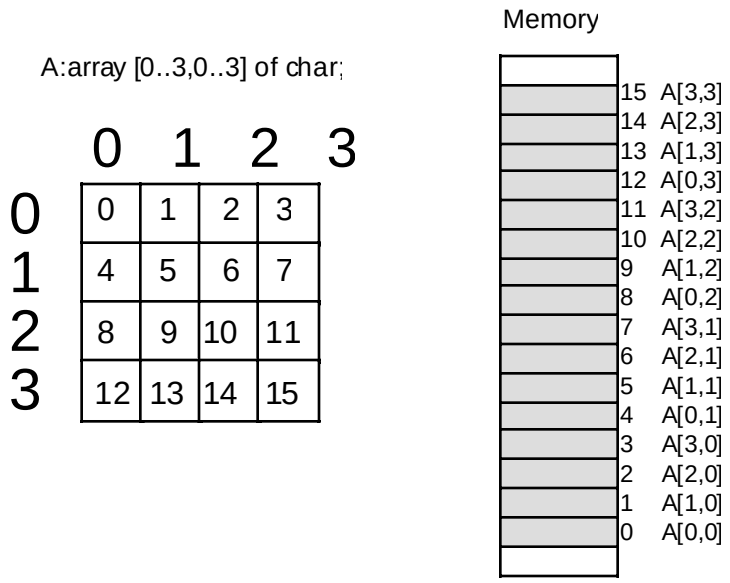

Figure 4.6 Column Major Array Element Ordering

The formulae for computing the address of an array element when using column major ordering is very similar to that for row major ordering. You simply reverse the indexes and sizes in the computation:

For a two-dimension column major array:

```
Element_Address = Base_Address + (rowindex * col_size + colindex) * Element_Size
```

For a three-dimension column major array:


```
Address = Base + ((rowindex*col_size+colindex) * depth_size + depthindex) *
Element_Size
```

For a four-dimension column major array:

```
Address =
    Base + (((rowindex * col_size + colindex)*depth_size + depthindex) *
        Left_size + Leftindex) * Element_Size
```

The single Pascal loop provided for row major access remains unchanged (to access A[b][c][d][e][f]):

```
    for i := 0 to 4 do
        base := base + A1[i] * A2[i];
```

Likewise, the initial values of the *A1* array remain unchanged:

```
A1 = {256, 64, 16, 4, 1}
```

The only thing that needs to change is the initial values for the *A2* array, and all you have to do here is reverse the order of the indices:

```
A2 = {f, e, d, c, b}
```

4.7 Allocating Storage for Multidimensional Arrays

If you have an *m* x *n* array, it will have *m * n* elements and require *m*n*Element_Size* bytes of storage. To allocate storage for an array you must reserve this amount of memory. As usual, there are several different ways of accomplishing this task. Fortunately, HLA's array declaration syntax is very similar to high level language array declaration syntax, so C/C++, BASIC, and Pascal programmers will feel right at home. To declare a multidimensional array in HLA, you use a declaration like the following:

ArrayName: *elementType* [*comma_separated_list_of_dimension_bounds*];

For example, here is a declaration for a 4x4 array of characters:

```
GameGrid: char[ 4, 4 ];
```

Here is another example that shows how to declare a three dimensional array of strings:

```
NameItems: string[ 2, 3, 3 ];
```

Remember, string objects are really pointers, so this array declaration reserves storage for 18 double word pointers (2*3*3=18).

As was the case with single dimension arrays, you may initialize every element of the array to a specific value by following the declaration with the assignment operator and an array constant. Array constants ignore dimension information; all that matters is that the number of elements in the array constant correspond to the number of elements in the actual array. The following example shows the *GameGrid* declaration with an initializer:

```
GameGrid: char[ 4, 4 ] :=
    [
        'a', 'b', 'c', 'd',
        'e', 'f', 'g', 'h',
        'i', 'j', 'k', 'l',
        'm', 'n', 'o', 'p'
    ];
```

Note that HLA ignores the indentation and extra whitespace characters (e.g., newlines) appearing in this declaration. It was laid out to enhance readability (which is always a good idea). HLA does not interpret the four separate lines as representing rows of data in the array. Humans do, which is why it's good to lay out the initial data in this manner, but HLA completely ignores the physical layout of the declaration. All that

matters is that there are 16 (4*4) characters in the array constant. You'll probably agree that this is much easier to read than

```
GameGrid: char[ 4,4 ] :=
    [ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
      'n', 'o', 'p' ];
```

Of course, if you have a large array, an array with really large rows, or an array with many dimensions, there is little hope for winding up with something reasonable. That's when comments that carefully explain everything come in handy.

As with single dimension arrays, you can use the DUP operator to initialize each element of a really large array with the same value. The following example initializes a 256x64 array of bytes so that each byte contains the value $FF:

```
StateValue: byte[ 256, 64 ] := 256*64 dup [$ff];
```

Note the use of a constant expression to compute the number of array elements rather than simply using the constant 16,384 (256*64). The use of the constant expression more clearly suggests that this code is initializing each element of a 256x64 element array than does the simple literal constant 16,384.

Another HLA trick you can use to improve the readability of your programs is to use *nested array constants*. The following is an example of an HLA nested array constant:

```
[ [0, 1, 2], [3, 4], [10, 11, 12, 13] ]
```

Whenever HLA encounters an array constant nested inside another array constant, it simply removes the brackets surrounding the nested array constant and treats the whole constant as a single array constant. For example, HLA converts the nested array constant above to the following:

```
[ 0, 1, 2, 3, 4, 10, 11, 12, 13 ]
```

You can take advantage of this fact to help make your programs a little more readable. For multidimensional array constants you can enclose each row of the constant in square brackets to denote that the data in each row is grouped and separate from the other rows. As an example, consider the following declaration for the *GameGrid* array that is identical (as far as HLA is concerned) to the previous declaration:

```
GameGrid: char[ 4, 4 ] :=
    [
        [ 'a', 'b', 'c', 'd' ],
        [ 'e', 'f', 'g', 'h' ],
        [ 'i', 'j', 'k', 'l' ],
        [ 'm', 'n', 'o', 'p' ]
    ];
```

This declaration makes it clearer that the array constant is a 4x4 array rather than just a 16-element one-dimensional array whose elements wouldn't fit all on one line of source code. Little aesthetic improvements like this are what separate mediocre programmers from good programmers.

4.8 Accessing Multidimensional Array Elements in Assembly Language

Well, you've seen the formulae for computing the address of an array element. You've even looked at some Pascal code you could use to access elements of a multidimensional array. Now it's time to see how to access elements of those arrays using assembly language.

The MOV, SHL, and INTMUL instructions make short work of the various equations that compute offsets into multidimensional arrays. Let's consider a two dimension array first:

```
static
```



```
    i:     int32;
    j:     int32;
    TwoD:  int32[ 4, 8 ];

         .
         .
         .

// To peform the operation TwoD[i,j] := 5; you'd use code like the following.
// Note that the array index computation is (i*8 + j)*4.

        mov( i, ebx );
        shl( 3, ebx );       // Multiply by eight (shl by 3 is a multiply by 8).
        add( j, ebx );
        mov( 5, TwoD[ ebx*4 ] );
```

Note that this code does *not* require the use of a two register addressing mode on the 80x86. Although an addressing mode like *TwoD[ebx][esi]* looks like it should be a natural for accessing two dimensional arrays, that isn't the purpose of this addressing mode.

Now consider a second example that uses a three dimension array:

```
static
    i:         int32;
    j:         int32;
    k:         int32;
    ThreeD:    int32[ 3, 4, 5 ];
        .
        .
        .

// To peform the operation ThreeD[i,j,k] := ESI; you'd use the following code
// that computes ((i*4 + j)*5 + k )*4 as the address of ThreeD[i,j,k].

        mov( i, ebx );
        shl( 2, ebx );       // Four elements per column.
        add( j, ebx );
        intmul( 5, ebx );    // Five elements per row.
        add( k, ebx );
        mov( esi, ThreeD[ ebx*4 ] );
```

Note that this code uses the INTMUL instruction to multiply the value in EBX by five. Remember, the SHL instruction can only multiply a register by a power of two. While there are ways to multiply the value in a register by a constant other than a power of two, the INTMUL instruction is more convenient[5].

4.9 Large Arrays and MASM

There is a defect in later versions of MASM v6.x that create some problems when you declare large static arrays in your programs. Now you may be wondering what this has to do with you since we're using HLA, but don't forget that HLA v1.x compiles to MASM assembly code and then runs MASM to assemble this output. Therefore, any defect in MASM is going to be a problem for HLA users.

The problem occurs when the total number of array elements you declare in a static section (STATIC, READONLY, or STORAGE) starts to get large. Large in this case is CPU dependent, but it falls somewhere between 128,000 and one million elements for most systems. MASM, for whatever reason, uses a very slow algorithm to emit array code to the object file; by the time you declare 64K array elements, MASM starts to produce a noticeable delay while compiling your code. After that point, the delay grows linearly with the

5. A full discussion of multiplication by constants other than a power of two appears in the chapter on arithmetic.

number of array elements (i.e., as you double the number of array elements you double the assembly time) until the data saturates MASM's internal buffers and the cache. Then there is a big jump in execution time. For example, on a 300 MHz Pentium II processor, compiling a program with an array with 256,000 elements takes about 30 seconds, compiling a program with an array having 512,000 element takes several minutes. Compiling a program with a one-megabyte array seems to take forever.

There are a couple of ways to solve this problem. First, of course, you can limit the size of your arrays in your program. Unfortunately, this isn't always an option available to you. The second possibility is to use MASM v6.11; the defect was introduced in MASM after this version. The problem with MASM v6.11 is that it doesn't support the MMX instruction set, so if you're going to compile MMX instructions (or other instructions that MASM v6.11 doesn't support) with HLA you will not be able to use this option. A third option is to put your arrays in a VAR section rather than a static declaration section; HLA processes arrays you declare in the VAR section so MASM never sees them. Hence, arrays you declare in the VAR section don't suffer from this problem.

4.10 Dynamic Arrays in Assembly Language

One problem with arrays up to this point is that their size is static. That is, the number of elements in all of the examples is chosen when writing the program, it is not set while the program is running (i.e., dynamically). Alas, sometimes you simply don't know how big an array needs to be when you're writing the program; you can only determine the size of the array while the program is running. This section describes how to allocate storage for arrays dynamically so you can set their size at run time.

Allocating storage for a single dimension array, and accessing elements of that array, is a nearly trivial task at run time. All you need to do is call the HLA Standard Library *malloc* routine specifying the size of the array, in bytes. *Malloc* will return a pointer to the base address of the new array in the EAX register. Typically, you would save this address in a pointer variable and use that value as the base address of the array in all future array accesses.

To access an element of a single dimensional dynamic array, you would generally load the base address into a register and compute the index in a second register. Then you could use the based indexed addressing mode to access elements of that array. This is not a whole lot more work than accessing elements of a statically allocated array. The following code fragment demonstrates how to allocate and access elements of a single dimension dynamic array:

```
static
    ArySize:      uns32;
    BaseAdrs:     pointer to uns32;
        .
        .
        .
    stdout.put( "How many elements do you want in your array? " );
    stdin.getu32();
    mov( eax, ArySize;       // Save away the upper bounds on this array.
    shl( 2, eax );           // Multiply eax by four to compute the number of bytes.
    malloc( eax );           // Allocate storage for the array.
    mov( eax, BaseAdrs );    // Save away the base address of the new array.
        .
        .
        .

    // Zero out each element of the array:

    mov( BaseAdrs, ebx );
    mov( 0, eax );
    for( mov(0, esi); esi < ArySize; inc( esi )) do

        mov( eax, [ebx + esi*4 ]);
```



```
endfor;
```

Dynamically allocating storage for a multidimensional array is fairly straight-forward. The number of elements in a multidimensional array is the product of all the dimension values; e.g., a 4x5 array has 20 elements. So if you get the bounds for each dimension from the user, all you need do is compute the product of all of these bound values and multiply the final result by the size of a single element. This computes the total number of bytes in the array, the value that *malloc* expects.

Accessing elements of multidimensional arrays is a little more problematic. The problem is that you need to keep the dimension information (that is, the bounds on each dimension) around because these values are needed when computing the row major (or column major) index into the array[6]. The conventional solution is to store these bounds into a static array (generally you know the *arity*, or number of dimensions, at compile-time, so it is possible to statically allocate storage for this array of dimension bounds). This array of dynamic array bounds is known as a *dope vector*. The following code fragment shows how to allocate storage for a two-dimensional dynamic array using a simple dope vector.

```
var
    ArrayPtr:   pointer to uns32;
    ArrayDims:  uns32[2];
        .
        .
        .
    // Get the array bounds from the user:

    stdout.put( "Enter the bounds for dimension #1: " );
    stdin.get( ArrayDims[0] );

    stdout.put( "Enter the bounds for dimension #2: " );
    stdin.get( ArrayDims[1*4] );

    // Allocate storage for the array:

    mov( ArrayDims[0], eax );
    intmul( ArrayDims[1*4], eax );
    shl( 2, eax );          // Multiply by four since each element is 4 bytes.
    malloc( eax );          // Allocate storage for the array and
    mov( eax, ArrayPtr );   //  save away the pointer to the array.


    // Initialize the array:

    mov( 0, edx );
    mov( ArrayPtr, edi );
    for( mov( 0, ebx ); ebx < ArrayDims[0]; inc( ebx )) do

        for( mov( 0, ecx ); ecx < ArrayDims[1*4]; inc( ecx )) do

            // Compute the index into the array
            // as esi := ( ebx * ArrayDims[1*4] + ecx ) * 4
            // (Note that the final multiplication by four is
            //  handled by the scaled indexed addressing mode below.)

            mov( ebx, esi );
            intmul( ArrayDims[1*4], esi );
            add( ecx, esi );

            // Initialize the current array element with edx.
```

6. Technically, you don't need the value of the left-most dimension bound to compute an index into the array, however, if you want to check the index bounds using the BOUND instruction (or some other technique), you will need this value around at run-time as well.


```
            mov( edx, [edi+esi*4] );
            inc( edx );

        endfor;

    endfor;
```

4.11 HLA Standard Library Array Support

The HLA Standard Library provides an array module that helps reduce the effort needed to support static and dynamic arrays in your program. The "arrays.hhf" library module provides code to declare and allocate dynamic arrays, compute the index into an array, copy arrays, perform row reductions, transpose arrays, and more. This section will explore some of the more useful features the arrays module provides.

One of the more interesting features of the HLA Standard Library arrays module is that most of the array manipulation procedures support both statically allocated and dynamically allocated arrays. In fact, the HLA array procedures can automatically figure out if an array is static or dynamic and generate the appropriate code for that array. There is one catch, however. In order for HLA to be able to differentiate statically and dynamically allocated arrays, you must use the dynamic array declarations found in the arrays package. This won't be a problem because HLA's dynamic array facilities are powerful and very easy to use.

To declare a dynamic array with the HLA arrays package, you use a variable declaration like the following:

```
variableName: array.dArray( elementType, Arity );
```

The *elementType* parameter is a regular HLA data type identifier (e.g., *int32* or some type identifier you've defined in the TYPE section). The *Arity* parameter is a constant that specifies the number of dimensions for the array (*arity* is the formal name for "number of dimensions"). Note that you do not specify the bounds of each dimension in this declaration. Storage allocation occurs later, at run time. The following is an example of a declaration for a dynamically allocated two-dimensional matrix:

```
ScreenBuffer: array.dArray( char, 2 );
```

The *array.dArray* data type is actually an HLA macro[7] that expands the above to the following:

```
ScreenBuffer: record
                dataPtr:      dword;
                dopeVector:   uns32[ 2 ];
                elementType:  char;
              endrecord;
```

The *dataPtr* field will hold the base address of the array once the program allocates storage for it. The *dopeVector* array has one element for each array dimension (the macro uses the second parameter of the *array.dArray* type as the number of dimensions for the *dopeVector* array). The *elementType* field is a single object that has the same type as an element of the dynamic array. HLA provides a couple of built-in functions that you can use on these fields to extract important information. The *@Elements* function returns the number of elements in an array. Therefore, "@Elements(ScreenBuffer.dopeVector)" will return the number of elements (two) in the *ScreenBuffer.dopeVector* array. Since this array contains one element for each dimension in the dynamic array, you can use the *@Elements* function with the *dopeVector* field to determine the arity of the array. You can use the HLA *@Size* function on the *ScreenBuffer.elementType* field to determine the size of an array element in the dynamic array. Most of the time you will know the arity and type of your dynamic arrays (after all, you declared them), so you probably won't use these functions often until you start writing macros that process dynamic arrays.

7. See the chapter on Macros and the HLA Compile-Time Language for details on macros.

After you declare a dynamic array, you must initialize the dynamic array object before attempting to use the array. The HLA Standard Library *array.daAlloc* routine handles this task for you. This routine uses the following syntax:

```
array.daAlloc( arrayName, comma_separated_list_of_array_bounds );
```

To allocate storage for the *ScreenBuffer* variable in the previous example you could use a call like the following:

```
array.daAlloc( ScreenBuffer, 20, 40 );
```

This call will allocate storage for a 20x40 array of characters. It will store the base address of the array into the *ScreenBuffer.dataPtr* field. It will also initialize *ScreenBuffer.dopeVector[0]* with 20 and *ScreenBuffer.dopeVector[1*4]* with 40. To access elements of the *ScreenBuffer* array you can use the techniques of the previous section, or you could use the *array.index* function.

The *array.index* function automatically computes the address of an array element for you. This function uses the following call syntax:

```
array.index( reg32, arrayName, comma_separated_list_of_index_values );
```

The first parameter must be a 32-bit register. The *array.index* function will store the address of the specified array element in this register. The second *array.index* parameter must be the name of an array; this can be either a statically allocated array or an array you've declared with *array.dArray* and allocated dynamically with *array.daAlloc*. Following the array name parameter is a list of one or more array indices. The number of array indices must match the arity of the array. These array indices can be constants, *dword* memory variables, or registers (however, you must not specify the same register that appears in the first parameter as one of the array indices). Upon return from this function, you may access the specified array element using the register indirect addressing mode and the register appearing as the first parameter.

One last routine you'll want to know about when manipulating HLA dynamic arrays is the *array.daFree* routine. This procedure expects a single parameter that is the name of an HLA dynamic array. Calling *array.daFree* will free the storage associated with the dynamic array. The following code fragment is a rewrite of the example from the previous section that uses HLA dynamic arrays:

```
var
    da:     array.dArray( uns32, 2 );
    Bnd1:   uns32;
    Bnd2:   uns32;
        .
        .
        .
    // Get the array bounds from the user:

    stdout.put( "Enter the bounds for dimension #1: " );
    stdin.get( Bnd1 );

    stdout.put( "Enter the bounds for dimension #2: " );
    stdin.get( Bnd2 );

    // Allocate storage for the array:

    array.daAlloc( da, Bnd1, Bnd2 );


    // Initialize the array:

    mov( 0, edx );
    for( mov( 0, ebx ); ebx < Bnd1; inc( ebx )) do

        for( mov( 0, ecx ); ecx < Bnd2; inc( ecx )) do
```



```
            // Initialize the current array element with edx.
            // Use array.index to compute the address of the array element.

            array.index( edi, da, ebx, ecx );
            mov( edx, [edi] );
            inc( edx );

        endfor;

    endfor;
```

Another extremely useful library module is the *array.cpy* routine. This procedure will copy the data from one array to another. The calling syntax is:

```
array.cpy( sourceArrayName, destArrayName );
```

The source and destination arrays can be static or dynamic arrays. The *array.cpy* automatically adjusts and emits the proper code for each combination of parameters. With most of the array manipulation procedures in the HLA Standard Library, you pay a small performance penalty for the convenience of these library modules. Not so with *array.cpy*. This procedure is very, very fast; much faster than writing a loop to copy the data element by element.

4.12 Putting It All Together

Accessing elements of an array is a very common operation in assembly language programs. This chapter provides the basic information you need to efficiently access array elements. After mastering the material in this chapter you should know how to declare arrays in HLA and access elements of those arrays in your programs.

