Variables and Data Structures Chapter Five

Chapter One discussed the basic format for data in memory. Chapter Three covered how a computer system physically organizes that data. This chapter finishes this discussion by connecting the concept of *data representation* to its actual physical representation. As the title implies, this chapter concerns itself with two main topics: variables and data structures. This chapter does not assume that you've had a formal course in data structures, though such experience would be useful.

5.0 Chapter Overview

This chapter discusses how to declare and access scalar variables, integers, reals, data types, pointers, arrays, and structures. You must master these subjects before going on to the next chapter. Declaring and accessing arrays, in particular, seems to present a multitude of problems to beginning assembly language programmers. However, the rest of this text depends on your understanding of these data structures and their memory representation. Do not try to skim over this material with the expectation that you will pick it up as you need it later. You will need it right away and trying to learn this material along with later material will only confuse you more.

5.1 Some Additional Instructions: LEA, LES, ADD, and MUL

The purpose of this chapter is not to present the 80x86 instruction set. However, there are four additional instructions (above and beyond mov) that will prove handy in the discussion throughout the rest of this chapter. These are the *load effective address* (lea), *load es and general purpose register* (les), *addition* (add), and *multiply* (mul). These instructions, along with the mov instruction, provide all the necessary power to access the different data types this chapter discusses.

The lea instruction takes the form:

```
lea       reg16, memory
```

reg16 is a 16 bit general purpose register. *Memory* is a memory location represented by a mod/reg/rm byte[1] (except it must be a memory location, it cannot be a register).

This instruction loads the 16 bit register with the offset of the location specified by the memory operand. lea ax,1000h[bx][si], for example, would load ax with the address of the memory location pointed at by 1000h[bx][si]. This, of course, is the value 1000h+bx+si. Lea is also quite useful for obtaining the address of a variable. If you have a variable I somewhere in memory, lea bx,I will load the bx register with the address (offset) of I.

The les instruction takes the form

```
les       reg16, memory32
```

This instruction loads the es register and one of the 16 bit general purpose registers from the specified memory address. Note that any memory address you can specify with a mod/reg/rm byte is legal but like the lea instruction it must be a memory location, not a register.

The les instruction loads the specified general purpose register from the word at the given address, it loads the es register from the following word in memory. This instruction, and it's companion lds (which loads ds) are the only instructions on pre-80386 machines that manipulate 32 bits at a time.

1. Or by the mod/reg/rm -- sib addressing mode bytes on the 80386.

The add instruction, like it's x86 counterpart, adds two values on the 80x86. This instruction takes several forms. There are five forms that concern us here. They are

```
add     reg, reg
add     reg, memory
add     memory, reg
add     reg, constant
add     memory, constant
```

All these instructions add the second operand to the first leaving the sum in the first operand. For example, add bx,5 computes bx := bx + 5.

The last instruction to look at is the mul (multiply) instruction. This instruction has only a single operand and takes the form:

```
mul     reg/memory
```

There are many important details concerning mul that this chapter ignores. For the sake of the discussion that follows, assume that the register or memory location is a 16 bit register or memory location. In such a case this instruction computes dx:ax :=ax*reg/mem[2]. Note that there is no immediate mode for this instruction.

5.2 Declaring Variables in an Assembly Language Program

Although you've probably surmised that memory locations and variables are somewhat related, this chapter hasn't gone out of its way to draw strong parallels between the two. Well, it's time to rectify that situation. Consider the following short (and useless) Pascal program:

```
program useless(input,output);
var i,j:integer;
begin
                i := 10;
                write('Enter a value for j:');
                readln(j);
                i := i*j + j*j;
                writeln('The result is ',i);
end.
```

When the computer executes the statement i:=10;[3] it makes a copy of the value 10 and somehow remembers this value for use later on. To accomplish this, the compiler sets aside a memory location specifically for the exclusive use of the variable i. Assuming the compiler arbitrarily assigned location DS:10h for this purpose it could use the instruction mov ds:[10h],10 to accomplish this[4]. If i is a 16 bit word, the compiler would probably assign the variable j to the word starting at location 12h or 0Eh. Assuming it's location 12h, the second assignment statement in the program might wind up looking like the following:

```
mov     ax, ds:[10h]            ;Fetch value of I
mul     ds:[12h]                ;Multiply by J
mov     ds:[10h], ax            ;Save in I (ignore overflow)
mov     ax, ds:[12h]            ;Fetch J
mul     ds:[12h]                ;Compute J*J
add     ds:[10h], ax            ;Add I*J + J*J, store into I
```

2. Any time you multiply two 16 bit values you could get a 32 bit result. The 80x86 places this 32 bit result in dx:ax with the H.O. word in dx and the L.O. word in ax.

3. Actually, the computer executes the *machine code* emitted by the Pascal compiler for this statement; but you need not worry about such details here.

4. But don't try this at home, folks! There is one minor syntactical detail missing from this instruction. MASM will complain bitterly if you attempt to assemble this particular instruction.

Although there are a few details missing from this code, it is fairly straightforward and you can easily see what is going on in this program.

Now imagine a 5,000 line program like this one using variables like ds:[10h], ds:[12h], ds:[14h], etc. Would you want to locate the statement where you accidentally stored the result of a computation into j rather than i? Indeed, why should you even care that the variable i is at location 10h and j is at location 12h? Why shouldn't you be able to use names like i and j rather than worrying about these numerical addresses? It seems reasonable to rewrite the code above as:

```
mov     ax, i
mul     j
mov     i, ax
mov     ax, j
mul     j
add     i, ax
```

Of course you can do this in assembly language! Indeed, one of the primary jobs of an assembler like MASM is to let you use symbolic names for memory locations. Furthermore, the assembler will even assign locations to the names automatically for you. You needn't concern yourself with the fact that variable i is really the word at memory location DS:10h unless you're curious.

It should come as no surprise that ds will point to the dseg segment in the SHELL.ASM file. Indeed, setting up ds so that it points at dseg is one of the first things that happens in the SHELL.ASM main program. Therefore, all you've got to do is tell the assembler to reserve some storage for your variables in dseg and attach the offset of said variables with the names of those variables. This is a very simple process and is the subject of the next several sections.

5.3 Declaring and Accessing Scalar Variables

Scalar variables hold single values. The variables i and j in the preceding section are examples of scalar variables. Examples of data structures that are not scalars include arrays, records, sets, and lists. These latter data types are made up from scalar values. They are the *composite types*. You'll see the composite types a little later; first you need to learn to deal with the scalar types.

To declare a variable in dseg, you would use a statement something like the following:

```
ByteVar         byte    ?
```

ByteVar is a *label*. It should begin at column one on the line somewhere in the dseg segment (that is, between the dseg segment and dseg ends statements). You'll find out all about labels in a few chapters, for now you can assume that most legal Pascal/C/Ada identifiers are also valid assembly language labels.

If you need more than one variable in your program, just place additional lines in the dseg segment declaring those variables. MASM will automatically allocate a unique storage location for the variable (it wouldn't be too good to have i and j located at the same address now, would it?). After declaring said variable, MASM will allow you to refer to that variable *by name* rather than by location in your program. For example, after inserting the above statement into the data segment (dseg), you could use instructions like mov ByteVar,al in your program.

The first variable you place in the data segment gets allocated storage at location DS:0. The next variable in memory gets allocated storage just beyond the previous variable. For example, if the variable at location zero was a byte variable, the next variable gets allocated storage at DS:1. However, if the first variable was a word, the second variable gets allocated storage at location DS:2. MASM is always careful to allocate variables in such a manner that they do not overlap. Consider the following dseg definition:

```
dseg            segment  para public 'data'
bytevar         byte     ?                      ;byte allocates bytes
wordvar         word     ?                      ;word allocates words
dwordvar        dword    ?                      ;dword allocs dbl words
byte2           byte     ?
word2           word     ?
dseg            ends
```

MASM allocates storage for *bytevar* at location DS:0. Because *bytevar* is one byte long, the next available memory location is going to be DS:1. MASM, therefore, allocates storage for *wordvar* at location DS:1. Since words require two bytes, the next available memory location after *wordvar* is DS:3 which is where MASM allocates storage for *dwordvar. Dwordvar* is four bytes long, so MASM allocates storage for *byte2* starting at location DS:7. Likewise, MASM allocates storage for *word2* at location DS:8. Were you to stick another variable after *word2*, MASM would allocate storage for it at location DS:0A.

Whenever you refer to one of the names above, MASM automatically substitutes the appropriate offset. For example, MASM would translate the mov ax,wordvar instruction into mov ax,ds:[1]. So now you can use symbolic names for your variables and completely ignore the fact that these variables are really memory locations with corresponding offsets into the data segment.

5.3.1 Declaring and using BYTE Variables

So what are byte variables good for, anyway? Well you can certainly represent any data type that has less than 256 different values with a single byte. This includes some very important and often-used data types including the character data type, boolean data type, most enumerated data types, and small integer data types (signed and unsigned), just to name a few.

Characters on a typical IBM compatible system use the eight bit ASCII/IBM character set (see "A: ASCII/IBM Character Set" on page 1345). The 80x86 provides a rich set of instructions for manipulating character data. It's not surprising to find that most byte variables in a typical program hold character data.

The boolean data type represents only two values: true or false. Therefore, it only takes a single bit to represent a boolean value. However, the 80x86 really wants to work with data at least eight bits wide. It actually takes extra code to manipulate a single bit rather than a whole byte. Therefore, you should use a whole byte to represent a boolean value. Most programmers use the value zero to represent false and anything else (typically one) to represent true. The 80x86's zero flag makes testing for zero/not zero very easy. Note that this choice of zero or non-zero is mainly for convenience. You could use *any* two different values (or two different sets of values) to represent true and false.

Most high level languages that support enumerated data types convert them (internally) to unsigned integers. The first item in the list is generally item zero, the second item in the list is item one, the third is item two, etc. For example, consider the following Pascal enumerated data type:

```
colors = (red, blue, green, purple, orange, yellow, white, black);
```

Most Pascal compilers will assign the value zero to red, one to blue, two to green, etc.

Later, you will see how to actually create your own enumerated data types in assembly language. All you need to learn now is how to allocate storage for a variable that holds an enumerated value. Since it's unlikely there will be more than 256 items enumerated by the data type, you can use a simple byte variable to hold the value. If you have a variable, say *color* of type *colors*, using the instruction mov color,2 is the same thing as saying color:=green in Pascal. (Later, you'll even learn how to use more meaningful statements like mov color,green to assign the color green to the color variable).

Of course, if you have a small unsigned integer value (0…255) or small signed integer (-128…127) a single byte variable is the best way to go in most cases. Note that most pro-

grammers treat all data types except small signed integers as unsigned values. That is, characters, booleans, enumerated types, and unsigned integers are all usually unsigned values. In some very special cases you might want to treat a character as a signed value, but most of the time even characters are unsigned values.

There are three main statements for declaring byte variables in a program. They are

```
identifier      db      ?
identifier      byte    ?
and
identifier      sbyte   ?
```

identifier represents the name of your byte variable. "db" is an older term that predates MASM 6.x. You will see this directive used quite a bit by other programmers (especially those who are not using MASM 6.x or later) but Microsoft considers it to be an obsolete term; you should always use the byte and sbyte declarations instead.

The byte declaration declares unsigned byte variables. You should use this declaration for all byte variables *except* small signed integers. For signed integer values, use the sbyte (signed byte) directive.

Once you declare some byte variables with these statements, you may reference those variables within your program by their names:

```
i               db      ?
j               byte    ?
k               sbyte   ?
                  .
                  .
                  .
                mov     i, 0
                mov     j, 245
                mov     k, -5
                mov     al, i
                mov     j, al
                etc.
```

Although MASM 6.x performs a small amount of type checking, you should not get the idea that assembly language is a strongly typed language. In fact, MASM 6.x will only check the values you're moving around to verify that they will *fit* in the target location. All of the following are legal in MASM 6.x:

```
                mov     k, 255
                mov     j, -5
                mov     i, -127
```

Since all of these variables are byte-sized variables, and all the associated constants will fit into eight bits, MASM happily allows each of these statements. Yet if you look at them, they are logically incorrect. What does it mean to move -5 into an unsigned byte variable? Since signed byte values must be in the range -128…127, what happens when you store the value 255 into a signed byte variable? Well, MASM simply converts these values to their eight bit equivalents (-5 becomes 0FBh, 255 becomes 0FFh [-1], etc.).

Perhaps a later version of MASM will perform stronger type checking on the values you shove into these variables, perhaps not. However, you should always keep in mind that it will always be possible to circumvent this checking. It's up to you to write your programs correctly. The assembler won't help you as much as Pascal or Ada will. Of course, even if the assembler disallowed these statements, it would still be easy to get around the type checking. Consider the following sequence:

```
                mov     al, -5
                  .
        ; Any number of statements which do not affect AL
                  .
                mov     j, al
```

There is, unfortunately, no way the assembler is going to be able to tell you that you're storing an illegal value into j[5]. The registers, by their very nature, are neither signed nor unsigned. Therefore the assembler will let you store a register into a variable regardless of the value that may be in that register.

Although the assembler does not check to see if both operands to an instruction are signed or unsigned, it most certainly checks their size. If the sizes do not agree the assembler will complain with an appropriate error message. The following examples are all illegal:

```
mov     i, ax       ;Cannot move 16 bits into eight
mov     i, 300      ;300 won't fit in eight bits.
mov     k, -130     ;-130 won't fit into eight bits.
```

You might ask "if the assembler doesn't really differentiate signed and unsigned values, why bother with them? Why not simply use db all the time?" Well, there are two reasons. First, it makes your programs easier to read and understand if you explicitly state (by using *byte* and *sbyte*) which variables are signed and which are unsigned. Second, who said anything about the assembler ignoring whether the variables are signed or unsigned? The mov instruction ignores the difference, but there are other instructions that do not.

One final point is worth mentioning concerning the declaration of byte variables. In all of the declarations you've seen thus far the operand field of the instruction has always contained a question mark. This question mark tells the assembler that the variable should be left uninitialized when DOS loads the program into memory[6]. You may specify an initial value for the variable, that will be loaded into memory before the program starts executing, by replacing the question mark with your initial value. Consider the following byte variable declarations:

```
i               db      0
j               byte    255
k               sbyte   -1
```

In this example, the assembler will initialize i, j, and k to zero, 255, and -1, respectively, when the program loads into memory. This fact will prove quite useful later on, especially when discussing tables and arrays. Once again, the assembler only checks the sizes of the operands. It does not check to make sure that the operand for the byte directive is positive or that the value in the operand field of sbyte is in the range -128…127. MASM will allow any value in the range -128…255 in the operand field of any of these statements.

In case you get the impression that there isn't a real reason to use byte vs. sbyte in a program, you should note that while MASM sometimes ignores the differences in these definitions, Microsoft's CodeView debugger does not. If you've declared a variable as a signed value, CodeView will display it as such (including a minus sign, if necessary). On the other hand, CodeView will always display db and byte variables as positive values.

5.3.2 Declaring and using WORD Variables

Most 80x86 programs use word values for three things: 16 bit signed integers, 16 bit unsigned integers, and offsets (pointers). Oh sure, you can use word values for lots of other things as well, but these three represent most applications of the word data type. Since the word is the largest data type the 8086, 8088, 80186, 80188, and 80286 can handle, you'll find that for most programs, the word is the basis for most computations. Of course, the 80386 and later allow 32 bit computations, but many programs do not use these 32 bit instructions since that would limit them to running on 80386 or later CPUs.

You use the dw, word, and sword statements to declare word variables. The following examples demonstrate their use:

5. Actually, for this simple example you *could* modify the assembler to detect this problem. But it's easy enough to come up with a slightly more complex example where the assembler could *not* detect the problem on.
6. DOS actually initializes such variables to zero, but you shouldn't count on this.

```
NoSignedWord     dw       ?
UnsignedWord     word     ?
SignedWord       sword    ?
Initialized0     word     0
InitializedM1    sword    -1
InitializedBig   word     65535
InitializedOfs   dw       NoSignedWord
```

Most of these declarations are slight modifications of the byte declarations you saw in the last section. Of course you may initialize any word variable to a value in the range -32768…65535 (the union of the range for signed and unsigned 16 bit constants). The last declaration above, however, is new. In this case a label appears in the operand field (specifically, the name of the NoSignedWord variable). When a label appears in the operand field the assembler will substitute the offset of that label (within the variable's segment). If these were the only declarations in *dseg* and they appeared in this order, the last declaration above would initialize *InitializedOfs* with the value zero since *NoSignedWord*'s offset is zero within the data segment. This form of initialization is quite useful for initializing *pointers*. But more on that subject later.

The CodeView debugger differentiates dw/word variables and sword variables. It always displays the unsigned values as positive integers. On the other hand, it will display sword variables as signed values (complete with minus sign, if the value is negative). Debugging support is one of the main reasons you'll want to use word or sword as appropriate.

5.3.3 Declaring and using DWORD Variables

You may use the dd, dword, and sdword instructions to declare four-byte integers, pointers, and other variables types. Such variables will allow values in the range -2,147,483,648…4,294,967,295 (the union of the range of signed and unsigned four-byte integers). You use these declarations like the `word` declarations:

```
NoSignedDWord    dd       ?
UnsignedDWord    dword    ?
SignedDWord      sdword   ?
InitBig          dword    4000000000
InitNegative     sdword   -1
InitPtr          dd       InitBig
```

The last example initializes a double word pointer with the segment:offset address of the InitBig variable.

Once again, it's worth pointing out that the assembler doesn't check the types of these variables when looking at the initialization values. If the value fits into 32 bits, the assembler will accept it. Size checking, however, is strictly enforced. Since the only 32 bit `mov` instructions on processors earlier than the 80386 are les and lds, you will get an error if you attempt to access dword variables on these earlier processors using a mov instruction. Of course, even on the 80386 you cannot move a 32 bit variable into a 16 bit register, you must use the 32 bit registers. Later, you'll learn how to manipulate 32 bit variables, even on a 16 bit processor. Until then, just pretend that you can't.

Keep in mind, of course, that CodeView differentiates between dd/dword and sdword. This will help you see the actual values your variables have when you're debugging your programs. CodeView only does this, though, if you use the proper declarations for your variables. Always use sdword for signed values and dd or dword (dword is best) for unsigned values.

5.3.4 Declaring and using FWORD, QWORD, and TBYTE Variables

MASM 6.x also lets you declare six-byte, eight-byte, and ten-byte variables using the df/fword, dq/qword, and dt/tbyte statements. Declarations using these statements were originally intended for floating point and BCD values. There are better directives for the floating point variables and you don't need to concern yourself with the other data types you'd use these directives for. The following discussion is for completeness' sake.

The df/fword statement's main utility is declaring 48 bit pointers for use in 32 bit protected mode on the 80386 and later. Although you could use this directive to create an arbitrary six byte variable, there are better directives for doing that. You should only use this directive for 48 bit far pointers on the 80386.

dq/qword lets you declare *quadword* (eight byte) variables. The original purpose of this directive was to let you create 64 bit double precision floating point variables and 64 bit integer variables. There are better directives for creating floating point variables. As for 64 bit integers, you won't need them very often on the 80x86 CPU (at least, not until Intel releases a member of the 80x86 family with 64 bit general purpose registers).

The dt/tbyte directives allocate ten bytes of storage. There are two data types indigenous to the 80x87 (math coprocessor) family that use a ten byte data type: ten byte BCD values and extended precision (80 bit) floating point values. This text will pretty much ignore the BCD data type. As for the floating point type, once again there is a better way to do it.

5.3.5 Declaring Floating Point Variables with REAL4, REAL8, and REAL10

These are the directives you should use when declaring floating point variables. Like dd, dq, and dt these statements reserve four, eight, and ten bytes. The operand fields for these statements may contain a question mark (if you don't want to initialize the variable) or it may contain an initial value in floating point form. The following examples demonstrate their use:

```
x               real4    1.5
y               real8    1.0e-25
z               real10   -1.2594e+10
```

Note that the operand field must contain a valid floating point constant using either decimal or scientific notation. In particular, *pure integer constants are not allowed.* The assembler will complain if you use an operand like the following:

```
x               real4      1
```

To correct this, change the operand field to "1.0".

Please note that it takes special hardware to perform floating point operations (e.g., an 80x87 chip or an 80x86 with built-in math coprocessor). If such hardware is not available, you must write software to perform operations like floating point addition, subtraction, multiplication, etc. In particular, you cannot use the 80x86 add instruction to add two floating point values. This text will cover floating point arithmetic in a later chapter (see "Floating Point Arithmetic" on page 771). Nonetheless, it's appropriate to discuss how to declare floating point variables in the chapter on data structures.

MASM also lets you use dd, dq, and dt to declare floating point variables (since these directives reserve the necessary four, eight, or ten bytes of space). You can even initialize such variables with floating point constants in the operand field. But there are two major drawbacks to declaring variables this way. First, as with bytes, words, and double words, the CodeView debugger will only display your floating point variables properly if you use the real4, real8, or real10 directives. If you use dd, dq, or dt, CodeView will display your values as four, eight, or ten byte unsigned integers. Another, potentially bigger, problem with using dd, dq, and dt is that they allow both integer and floating point constant initializers (remember, real4, real8, and real10 do not). Now this might seem like a good feature

at first glance. However, the integer representation for the value one is *not* the same as the floating point representation for the value 1.0. So if you accidentally enter the value "1" in the operand field when you really meant "1.0", the assembler would happily digest this and then give you incorrect results. Hence, you should always use the real4, real8, and real10 statements to declare floating point variables.

5.4 Creating Your Own Type Names with TYPEDEF

Let's say that you simply do not like the names that Microsoft decided to use for declaring byte, word, dword, real, and other variables. Let's say that you prefer Pascal's naming convention or, perhaps, C's naming convention. You want to use terms like *integer, float, double, char, boolean,* or whatever. If this were Pascal you could redefine the names in the **type** section of the program. With C you could use a "**#define**" or a **typedef** statement to accomplish the task. Well, MASM 6.x has it's own *typedef* statement that also lets you create aliases of these names. The following example demonstrates how to set up some Pascal compatible names in your assembly language programs:

```
integer         typedef     sword
char            typedef     byte
boolean         typedef     byte
float           typedef     real4
colors          typedef     byte
```

Now you can declare your variables with more meaningful statements like:

```
i               integer     ?
ch              char        ?
FoundIt         boolean     ?
x               float       ?
HouseColor      colors      ?
```

If you are an Ada, C, or FORTRAN programmer (or any other language, for that matter), you can pick type names you're more comfortable with. Of course, this doesn't change how the 80x86 or MASM reacts to these variables one iota, but it does let you create programs that are easier to read and understand since the type names are more indicative of the actual underlying types.

Note that CodeView still respects the underlying data type. If you define *integer* to be an sword type, CodeView will display variables of type integer as signed values. Likewise, if you define *float* to mean real4, CodeView will still properly display *float* variables as four-byte floating point values.

5.5 Pointer Data Types

Some people refer to pointers as scalar data types, others refer to them as composite data types. This text will treat them as scalar data types even though they exhibit some tendencies of both scalar and composite data types (for a complete description of composite data types, see "Composite Data Types" on page 206).

Of course, the place to start is with the question "What is a pointer?" Now you've probably experienced pointers first hand in the Pascal, C, or Ada programming languages and you're probably getting worried right now. Almost everyone has a real bad experience when they first encounter pointers in a high level language. Well, fear not! Pointers are actually *easier* to deal with in assembly language. Besides, most of the problems you had with pointers probably had nothing to do with pointers, but rather with the linked list and tree data structures you were trying to implement with them. Pointers, on the other hand, have lots of uses in assembly language that have nothing to do with linked lists, trees, and other scary data structures. Indeed, simple data structures like arrays and records often involve the use of pointers. So if you've got some deep-rooted fear about

pointers, well forget everything you know about them. You're going to learn how *great* pointers really are.

Probably the best place to start is with the definition of a pointer. Just exactly what is a pointer, anyway? Unfortunately, high level languages like Pascal tend to hide the simplicity of pointers behind a wall of abstraction. This added complexity (which exists for good reason, by the way) tends to frighten programmers because *they don't understand what's going on*.

Now if you're afraid of pointers, well, let's just ignore them for the time being and work with an array. Consider the following array declaration in Pascal:

```
M: array [0..1023] of integer;
```

Even if you don't know Pascal, the concept here is pretty easy to understand. M is an array with 1024 integers in it, indexed from M[0] to M[1023]. Each one of these array *elements* can hold an integer value that is independent of all the others. In other words, this array gives you 1024 different integer variables each of which you refer to by number (the array index) rather than by name.

If you encountered a program that had the statement M[0]:=100 you probably wouldn't have to think at all about what is happening with this statement. It is storing the value 100 into the first element of the array M. Now consider the following two statements:

```
i := 0; (* Assume "i" is an integer variable *)
M [i] := 100;
```

You should agree, without too much hesitation, that these two statements perform the same exact operation as M[0]:=100;. Indeed, you're probably willing to agree that you can use any integer expression in the range 0…1023 as an index into this array. The following statements *still* perform the same operation as our single assignment to index zero:

```
i := 5;                 (* assume all variables are integers*)
j := 10;
k := 50;
m [i*j-k] := 100;
```

"Okay, so what's the point?" you're probably thinking. "Anything that produces an integer in the range 0…1023 is legal. So what?" Okay, how about the following:

```
M [1] := 0;
M [ M [1] ] := 100;
```

Whoa! Now that takes a few moments to digest. However, if you take it slowly, it makes sense and you'll discover that these two instructions perform the exact same operation you've been doing all along. The first statement stores zero into array element M[1]. The second statement fetches the value of M[1], which is an integer so you can use it as an array index into M, and uses that value (zero) to control where it stores the value 100.

If you're willing to accept the above as reasonable, perhaps bizarre, but usable nonetheless, then you'll have no problems with pointers. *Because m[1] is a pointer!* Well, not really, but if you were to change "M" to "memory" and treat this array as all of memory, this is the exact definition of a pointer.

A pointer is simply a memory location whose value is the address (or index, if you prefer) of some other memory location. Pointers are very easy to declare and use in an assembly language program. You don't even have to worry about array indices or anything like that. In fact, the only complication you're going to run into is that the 80x86 supports two kinds of pointers: *near* pointers and *far* pointers.

A near pointer is a 16 bit value that provides an offset into a segment. It could be any segment but you will generally use the data segment (*dseg* in SHELL.ASM). If you have a word variable p that contains 1000h, then p "points" at memory location 1000h in *dseg*. To access the word that p points at, you could use code like the following:

```
mov     bx, p       ;Load BX with pointer.
mov     ax, [bx]    ;Fetch data that p points at.
```

By loading the value of p into bx this code loads the value 1000h into bx (assuming p contains 1000h and, therefore, points at memory location 1000h in *dseg*). The second instruction above loads the ax register with the word starting at the location whose offset appears in bx. Since bx now contains 1000h, this will load ax from locations DS:1000 and DS:1001.

Why not just load ax directly from location 1000h using an instruction like mov ax,ds:[1000h]? Well, there are lots of reasons. But the primary reason is that this single instruction *always* loads ax from location 1000h. Unless you are willing to mess around with self-modifying code, you cannot change the location from which it loads ax. The previous two instructions, however, always load ax from the location that p points at. This is very easy to change under program control, without using self-modifying code. In fact, the simple instruction mov p,2000h will cause those two instructions above to load ax from memory location DS:2000 the next time they execute. Consider the following instructions:

```
        lea     bx, i       ;This can actually be done with
        mov     p, bx       ; a single instruction as you'll
         .                  ; see in Chapter Eight.
         .
< Some code that skips over the next two instructions >

        lea     bx, j       ;Assume the above code skips these
        mov     p, bx       ; two instructions, that you get
         .                  ; here by jumping to this point from
         .                  ; somewhere else.
        mov     bx, p       ;Assume both code paths above wind
        mov     ax, [bx]    ; up down here.
```

This short example demonstrates two execution paths through the program. The first path loads the variable p with the address of the variable i (remember, lea loads bx with the offset of the second operand). The second path through the code loads p with the address of the variable j. Both execution paths converge on the last two mov instructions that load ax with i or j depending upon which execution path was taken. In many respects, this is like a *parameter* to a procedure in a high level language like Pascal. Executing the same instructions accesses different variables depending on whose address (i or j) winds up in p.

Sixteen bit near pointers are small, fast, and the 80x86 provides efficient access using them. Unfortunately, they have one very serious drawback – you can only access 64K of data (one segment) when using near pointers[7]. Far pointers overcome this limitation at the expense of being 32 bits long. However, far pointers let you access any piece of data anywhere in the memory space. For this reason, and the fact that the UCR Standard Library uses far pointers exclusively, this text will use far pointers most of the time. But keep in mind that this is a decision based on trying to keep things simple. Code that uses near pointers rather than far pointers will be shorter and faster.

To access data referenced by a 32 bit pointer, you will need to load the offset portion (L.O. word) of the pointer into bx, bp, si, or di and the segment portion into a segment register (typically es). Then you could access the object using the register indirect addressing mode. Since the les instruction is so convenient for this operation, it is the perfect choice for loading es and one of the above four registers with a pointer value. The following sample code stores the value in al into the byte pointed at by the far pointer p:

```
        les     bx, p       ;Load p into ES:BX
        mov     es:[bx], al ;Store away AL
```

Since near pointers are 16 bits long and far pointers are 32 bits long, you could simply use the dw/word and dd/dword directives to allocate storage for your pointers (pointers are inherently unsigned, so you wouldn't normally use sword or sdword to declare a pointer).

7. Technically, this isn't true. A single pointer is limited to accessing data in one particular segment at a time, but you could have several near pointers each pointing at data in different segments. Unfortunately, you need to keep track of all this yourself and it gets out of hand very quickly as the number of pointers in your program increases.

However, there is a much better way to do this by using the typedef statement. Consider the following general forms:

```
typename        typedef   near ptr basetype
typename        typedef   far ptr basetype
```

In these two examples *typename* represents the name of the new type you're creating while *basetype* is the name of the type you want to create a pointer for. Let's look at some specific examples:

```
nbytptr         typedef   near ptr byte
fbytptr         typedef   far ptr byte
colorsptr       typedef   far ptr colors
wptr            typedef   near ptr word
intptr          typedef   near ptr integer
intHandle       typedef   near ptr intptr
```

(these declarations assume that you've previously defined the types *colors* and *integer* with the typedef statement). The typedef statements with the *near ptr* operands produce 16 bit near pointers. Those with the *far ptr* operands produce 32 bit far pointers. MASM 6.x ignores the base type supplied after the *near ptr* or *far ptr*. However, CodeView uses the base type to display the object a pointer refers to in its correct format.

Note that you can use *any* type as the base type for a pointer. As the last example above demonstrates, you can even define a pointer to another pointer (a *handle*). CodeView would properly display the object a variable of type intHandle points at as an address.

With the above types, you can now generate pointer variables as follows:

```
bytestr         nbytptr     ?
bytestr2        fbytptr     ?
CurrentColor    colorsptr   ?
CurrentItem     wptr        ?
LastInt         intptr      ?
```

Of course, you can initialize these pointers at assembly time if you know where they are going to point when the program first starts running. For example, you could initialize the *bytestr* variable above with the offset of *MyString* using the following declaration:

```
bytestr         nbytptr   MyString
```

5.6 Composite Data Types

Composite data types are those that are built up from other (generally scalar) data types. An array is a good example of a composite data type – it is an aggregate of elements all the same type. Note that a composite data type need not be composed of scalar data types, there are arrays of arrays for example, but ultimately you can decompose a composite data type into some primitive, scalar, types.

This section will cover two of the more common composite data types: arrays and records. It's a little premature to discuss some of the more advanced composite data types.

5.6.1 Arrays

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[8]. Different indices select unique elements of the array. This text assumes that the integer indices are contigu-

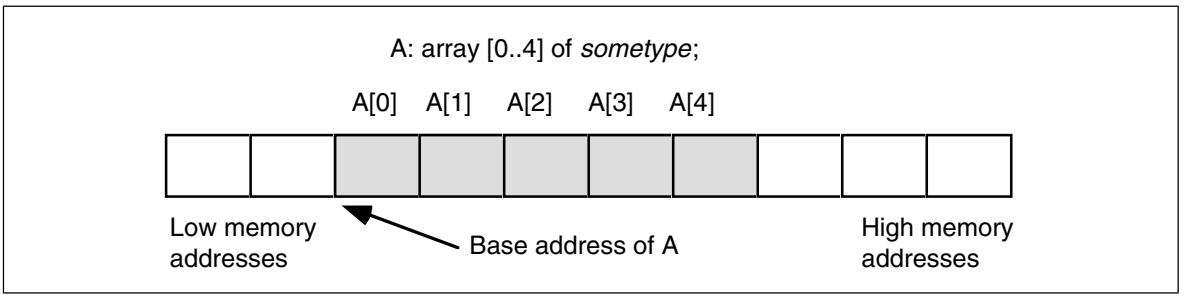

Figure 5.1 Single Dimension Array Implementation

ous (though it 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, arrays occupy contiguous locations in memory. An array with five elements will appear in memory as shown in Figure 5.1.

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 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 some value other than zero.

To access an element of an array, you need a function that converts an array index into 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.

5.6.1.1 Declaring Arrays in Your Data Segment

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 so far. To allocate *n* elements in an array, you would use a declaration like the following:

```
arrayname       basetype    n dup (?)
```

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 dup (?) operand tells the assembler to duplicate the object inside the parentheses *n* times. Since a question mark appears inside the parentheses, the definition above

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

would create *n* occurrences of an uninitialized value. Now let's look at some specific examples:

```
CharArray       char     128 dup (?)            ;array[0..127] of char
IntArray        integer  8 dup (?)              ;array[0..7] of integer
BytArray        byte     10 dup (?)             ;array[0..9] of byte
PtrArray        dword    4 dup (?)              ;array[0..3] of dword
```

The first two examples, of course, assume that you've used the typedef statement to define the char and integer data types.

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:

```
RealArray       real4    8 dup (1.0)
IntegerAry      integer  8 dup (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.

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? Well, that is easily handled as well. The variable declaration statements you've seen thus far offer yet another initialization form:

name *type* $value_1$, $value_2$, $value_3$, …, $value_n$

This form allocates *n* variables of type *type*. It initializes the first item to $value_1$, the second item to $value_2$, etc. So by simply enumerating each value in the operand field, you can create an array with the desired initial values. In the following integer array, for example, each element contains the square of its index:

```
Squares         integer  0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
```

If your array has more elements than will fit on one line, there are several ways to continue the array onto the next line. The most straight-forward method is to use another integer statement *but without a label*:

```
Squares         integer  0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
                integer  121, 144, 169, 196, 225, 256, 289, 324
                integer  361, 400
```

Another option, that is better in some circumstances, is to use a backslash at the end of each line to tell MASM 6.x to continue reading data on the next line:

```
Squares         integer  0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, \
                         121, 144, 169, 196, 225, 256, 289, 324, \
                         361, 400
```

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.

You need to learn about one final technique for initializing single dimension arrays before moving on. Consider the following declaration:

```
BigArray        word     256 dup (0,1,2,3)
```

This array has 1024 elements, not 256. The n dup (xxxx) operand tells MASM to duplicate xxxx *n* times, not create an array with *n* elements. If xxxx consists of a single item, then the dup operator will create an *n* element array. However, if xxxx contains two items separated by a comma, the dup operator will create an array with 2**n* elements. If xxxx contains three items separated by commas, the dup operator creates an array with 3**n* items, and so on. Since there are four items in the parentheses above, the dup operator creates 256*4 or 1024 items in the array. The values in the array will initially be 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 ...

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

5.6.1.2 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 MASM associates the address of the first operand with the label). 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 integer, or other two-byte type) then Element_Size is two. And so on. To access an element of the Squares array in the previous section, you'd use the formula:

Element_Address = Squares + index*2

The 80x86 code equivalent to the statement AX:=Squares[index] is

```
mov     bx, index
add     bx, bx                  ;Sneaky way to compute 2*bx
mov     ax, Squares [bx]
```

There are two important things to notice here. First of all, this code uses the add instruction rather than the mul instruction to compute 2*index. The main reason for choosing add is that it was more convenient (remember, mul doesn't work with constants and it only operates on the ax register). It turns out that add is a *lot* faster than mul on many processors, but since you probably didn't know that, it wasn't an overriding consideration in the choice of this instruction.

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 ax, Squares[bx] loads ax from location Squares+bx which is the base address plus index*2 (since bx contains index*2). Sure, you could have used

```
lea     ax, Squares
add     bx, ax
mov     ax, [bx]
```

in place of the last instruction, but why use three instructions where one 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.

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.

If you are using an 80386 or later, you can take advantage of the scaled indexed addressing mode to speed up accessing an array element even more. Consider the following statements:

```
mov     ebx, index              ;Assume a 32 bit value.
mov     ax, Squares [ebx*2]
```

This brings the instruction count down to two instructions. You'll soon see that two instructions aren't necessarily faster than three instructions, but hopefully you get the idea. Knowing your addressing modes can surely help.

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 Squares 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. The mov ax,Squares[BX] instruction is four bytes long (assuming you need a two-byte displacement to hold the offset to Squares in the data segment). You can reduce this to a two byte instruction by using the base/indexed addressing mode as follows:

```
lea     bx, Squares
mov     si, index
add     si, si
mov     ax, [bx][si]
```

Now bx contains the base address and `si` contains the index*2 value. Of course, this just replaced a single four-byte instruction with a three-byte and a two-byte instruction, hardly a good trade-off. However, you do not have to reload bx with the base address of Squares for the next access. The following sequence is one byte shorter than the comparable sequence that doesn't load the base address into `bx`:

```
lea     bx, Squares
mov     si, index
add     si, si
mov     ax, [bx][si]
 .
 .                  ;Assumption: BX is left alone
 .                  ; through this code.
mov     si, index2
add     si, si
mov     cx, [bx][si]
```

Of course the more accesses to Squares you make without reloading bx, 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 an 8086 might actually run *slower* on an 80486 (and vice versa). 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 simple 8086, even more so on processors like the 80486 and Pentium/80586 that offer pipelining, on-chip caches, and even superscalar operation.

5.6.2 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 5.2 shows one way to do this.

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 bytes.

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 possi-

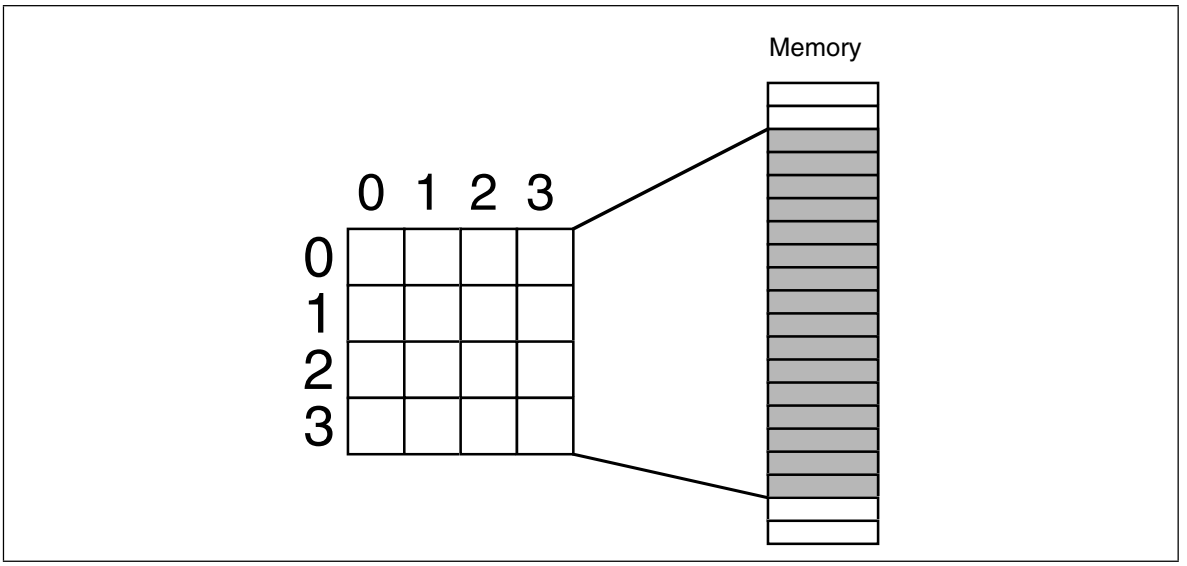

Figure 5.2 Mapping a 4 x 4 Array to Memory

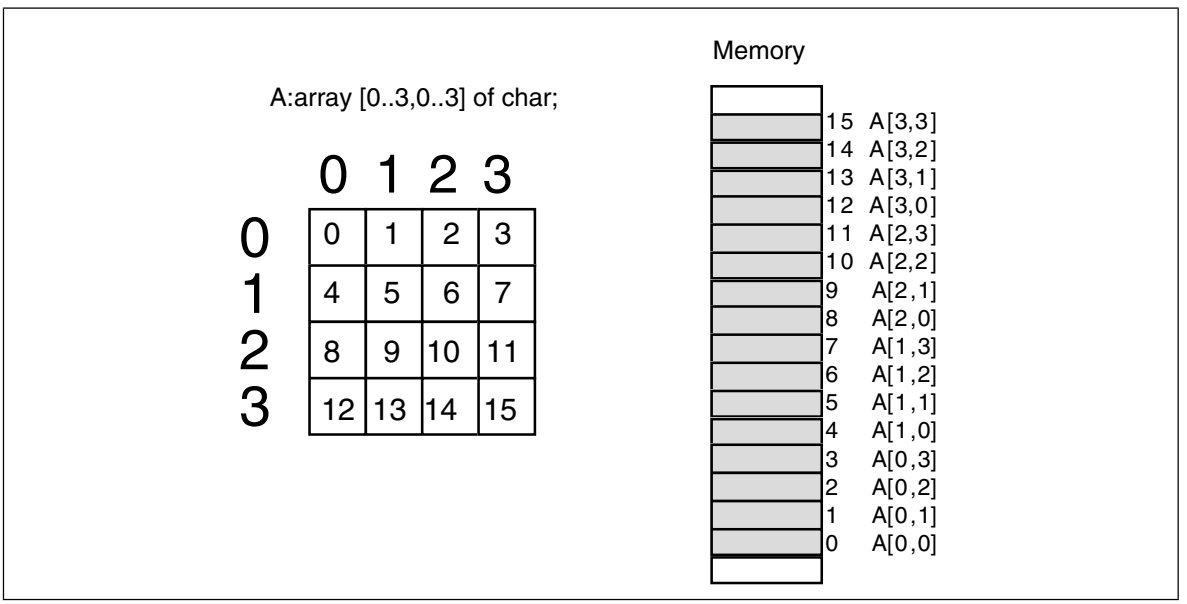

Figure 5.3 Row Major Element Ordering

ble 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*.

5.6.2.1 Row Major Ordering

Row major ordering assigns successive elements, moving across the rows and then down the columns, to successive memory locations. The mapping is best described in Figure 5.3.

Row major ordering is the method employed by most high level programming languages including Pascal, C, Ada, Modula-2, etc. It is very easy to implement and easy to use in machine language (especially within a debugger such as CodeView). The conversion from a two-dimensional structure to a linear array is very intuitive. You start with the

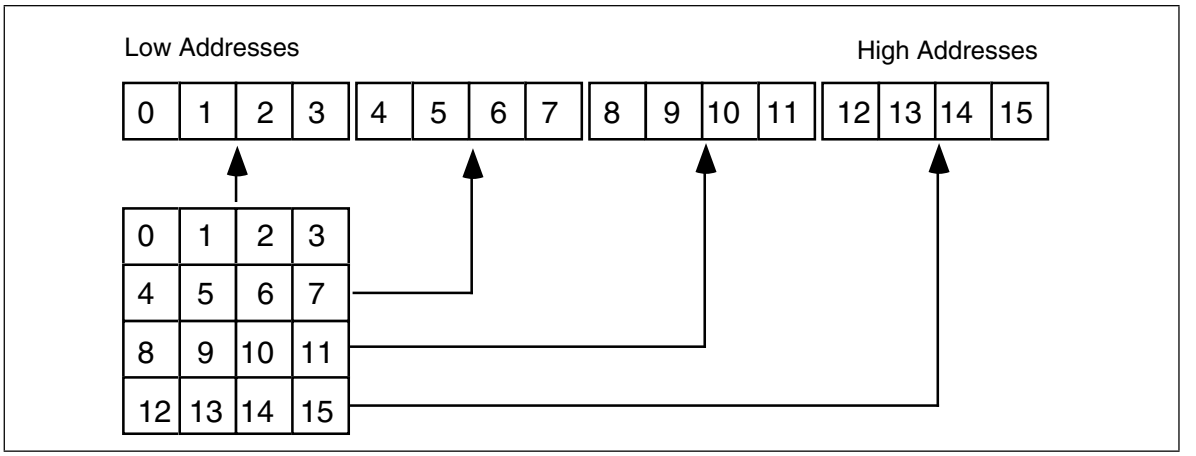

Figure 5.4 Another View of Row Major Ordering for a 4x4 Array

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 5.4).

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 across the board to row major order no matter how many dimensions it has, 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[9]. 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 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

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

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

Column Index	Row Index	Offset into Array
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 Pascal, if you've declared the array as "A:array [i..j] [k..l] [m..n] of *type*;" then row_size is equal to n-m+1 and col_size is equal to l-k+1.

For a four dimensional array, declared as "A:array [g..h] [i..j] [k..l] [m..n] of *type*;" 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 i-j+1, col_size and row_size are the same as before. 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* 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 5.5).

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.

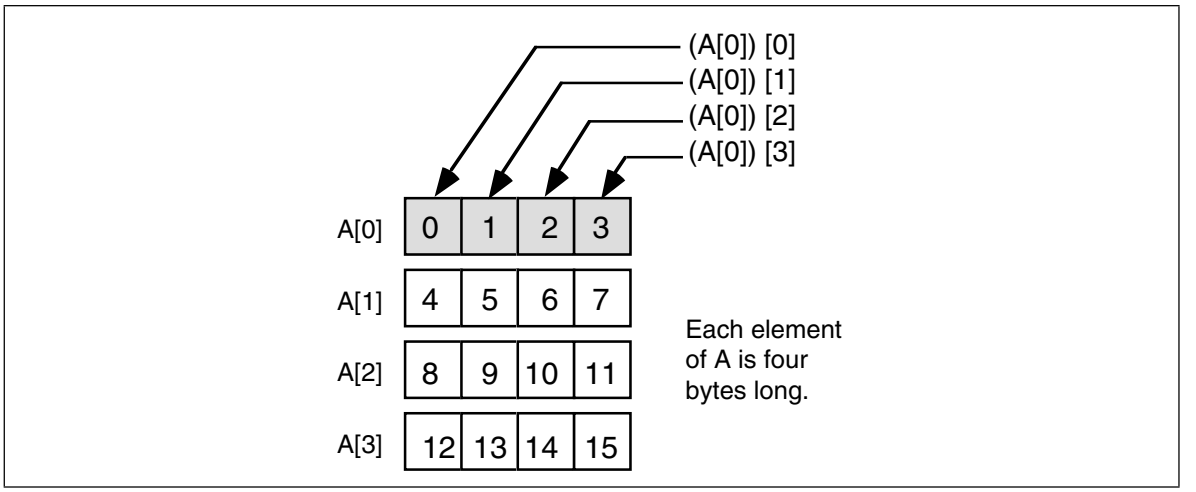

Figure 5.5 Viewing a 4x4 Array as an Array of Arrays

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:

```
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 from above, 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.

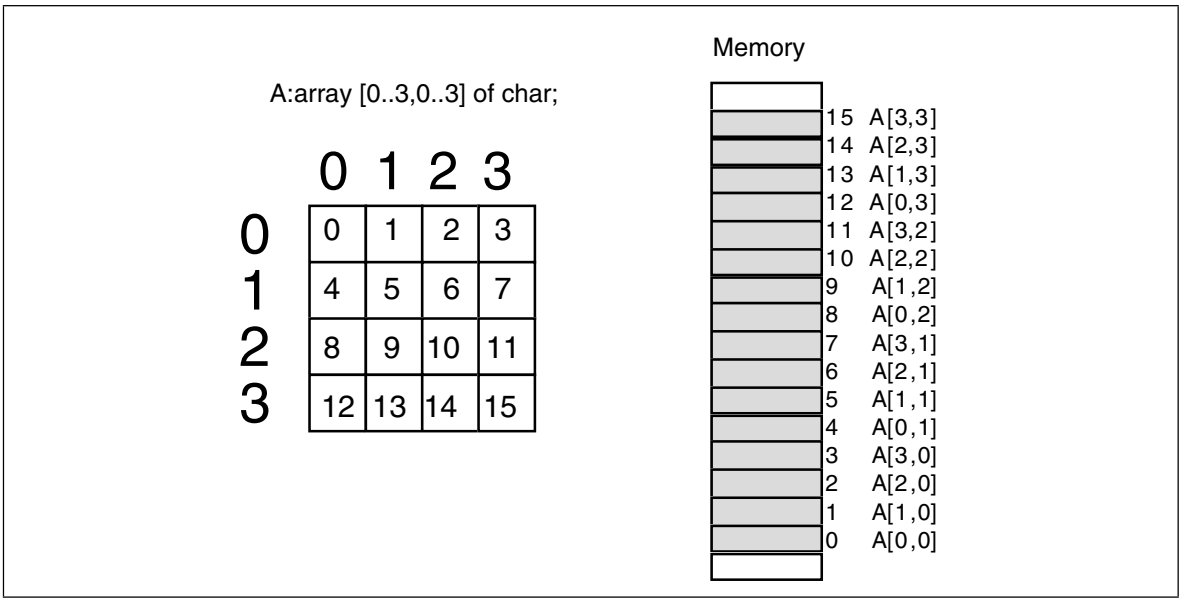

Figure 5.6 Column Major Element Ordering

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.

5.6.2.2 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., Microsoft) 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 5.6.

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}
```

5.6.2.3 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. This text will try to take the approach that is easiest to read and understand in your programs.

Reconsider the dup operator for reserving storage. n dup (xxxx) replicates xxxx *n* times. As you saw earlier, this dup operator allows not just one, but several items within the parentheses and it duplicates everything inside the specified number of times. In fact, the dup operator allows *anything* that you might normally expect to find in the operand field of a byte statement *including additional occurrences of the DUP operator.* Consider the following statement:

```
A               byte     4 dup (4 dup (?))
```

The first dup operator repeats everything inside the parentheses four times. Inside the parentheses the 4 DUP (?) operation tells MASM to set aside storage for four bytes. Four copies of four bytes yields 16 bytes, the number necessary for a 4 x 4 array. Of course, to reserve storage for this array you could have just as easily used the statement:

```
A               byte     16 dup (?)
```

Either way the assembler is going to set aside 16 contiguous bytes in memory. As far as the 80x86 is concerned, there is no difference between these two forms. On the other hand, the former version provides a better indication that A is a 4 x 4 array than the latter version. The latter version looks like a single dimension array with 16 elements.

You can very easily extend this concept to arrays of higher arity as well. The declaration for a three dimension array, A:array [0..2, 0..3, 0..4] of integer might be

```
A               integer  3 dup (4 dup (5 dup (?)))
```

(of course, you will need the integer typedef word statement in your program for this to work.)

As was the case with single dimension arrays, you may initialize every element of the array to a specific value by replacing the question mark (?) with some particular value. For example, to initialize the above array so that each element contains one you'd use the code:

```
A               integer  3 dup (4 dup (5 dup (1)))
```

If you want to initialize each element of the array to a different value, you'll have to enter each value individually. If the size of a row is small enough, the best way to approach this task is to place the data for each row of an array on its own line. Consider the following 4x4 array declaration:

```
A               integer  0,1,2,3
                integer    1,0,1,1
                integer    5,7,2,2
                integer    0,0,7,6
```

Once again, the assembler doesn't care where you split the lines, but the above is much easier to identify as a 4x4 array than the following that emits the exact same data:

```
A               integer  0,1,2,3,1,0,1,1,5,7,2,2,0,0,7,6
```

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.

5.6.2.4 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, add, and mul instructions make short work of the various equations that compute offsets into multidimensional arrays. Let's consider a two dimension array first:

```
; Note: TwoD's row size is 16 bytes.

TwoD            integer  4 dup (8 dup (?))
i               integer  ?
j               integer  ?

                 .       .
                 .       .
                 .       .

; To peform the operation TwoD[i,j] := 5; you'd use the code:

                mov      ax, 8        ;8 elements per row
                mul      i
                add      ax, j
                add      ax, ax       ;Multiply by element size (2)
                mov      bx, ax       ;Put in a register we can use
                mov      TwoD [bx], 5
```

Of course, if you have an 80386 chip (or better), you could use the following code[10]:

```
                mov      eax, 8       ;Zeros H.O. 16 bits of EAX.
                mul      i
                add      ax, j
                mov      TwoD[eax*2], 5
```

Note that this code does *not* require the use of a two register addressing mode on the 80x86. Although an addressing mode like TwoD [bx][si] 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:

```
ThreeD          integer  4 dup (4 dup (4 dup (?)))
i               integer  ?
j               integer  ?
k               integer  ?

                 .       .
                 .       .
                 .       .

; To peform the operation ThreeD[i,j,k] := 1; you'd use the code:

                mov      bx, 4        ;4 elements per column
                mov      ax, i
                mul      bx
                add      ax, j
```

10. Actually, there is an even *better* 80386 instruction sequence than this, but it uses instructions yet to be discussed.

```
mul     bx          ;4 elements per row
add     ax, k
add     ax, ax      ;Multiply by element size (2)
mov     bx, ax      ;Put in a register we can use
mov     ThreeD [bx], 1
```

Of course, if you have an 80386 or better processor, this can be improved somewhat by using the following code:

```
mov     ebx, 4
mov     eax, ebx
mul     i
add     ax, j
mul     bx
add     k
mov     ThreeD[eax*2], 1
```

5.6.3 Structures

The second major composite data structure is the Pascal *record* or C *structure*[11]. The Pascal terminology is probably better, since it tends to avoid confusion with the more general term *data structure*. However, MASM uses "structure" so it doesn't make sense to deviate from this. Furthermore, MASM uses the term *record* to denote something slightly different, furthering the reason to stick with the term structure.

Whereas an array is homogeneous, whose elements are all the same, the elements in a structure can be of any type. Arrays let you select a particular element via an integer index. With structures, you must select an element (known as a *field*) by name.

The whole purpose of a structure is to let you encapsulate different, but logically related, data into a single package. The Pascal record declaration for a student is probably the most typical example:

```
student = record
                Name: string [64];
                Major: integer;
                SSN:   string[11];
                Midterm1: integer;
                Midterm2: integer;
                Final: integer;
                Homework: integer;
                Projects: integer;
        end;
```

Most Pascal compilers allocate each field in a record to contiguous memory locations. This means that Pascal will reserve the first 65 bytes for the name[12], the next two bytes hold the major code, the next 12 the Social Security Number, etc.

In assembly language, you can also create structure types using the MASM struct statement. You would encode the above record in assembly language as follows:

```
student         struct
Name            char     65 dup (?)
Major           integer  ?
SSN             char     12 dup (?)
Midterm1        integer  ?
Midterm2        integer  ?
Final           integer  ?
Homework        integer  ?
Projects        integer  ?
student         ends
```

11. It also goes by some other names in other languages, but most people recognize at least one of these names.
12. Strings require an extra byte, in addition to all the characters in the string, to encode the length.

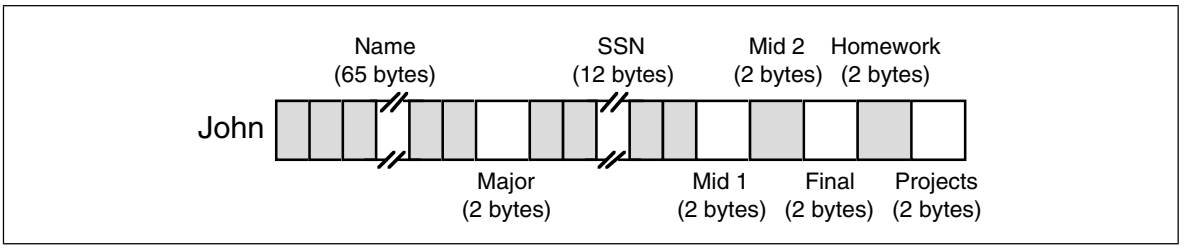

Figure 5.7 Student Data Structure Storage in Memory

Note that the structure ends with the ends (for *end structure*) statement. The label on the ends statement must be the same as on the struct statement.

The field names within the structure must be unique. That is, the same name may not appear two or more times in the same structure. However, all field names are local to that structure. Therefore, you may reuse those field names elsewhere in the program[13].

The struct directive only defines a structure type. It does *not* reserve storage for a structure variable. To actually reserve storage you need to declare a variable using the structure name as a MASM statement, e.g.,

```
John            student     {}
```

The braces must appear in the operand field. Any initial values must appear between the braces. The above declaration allocates memory as shown in Figure 5.7. :

If the label John corresponds to the *base address* of this structure, then the Name field is at offset John+0, the Major field is at offset John+65, the SSN field is at offset John+67, etc.

To access an element of a structure you need to know the offset from the beginning of the structure to the desired field. For example, the Major field in the variable John is at offset 65 from the base address of John. Therefore, you could store the value in ax into this field using the instruction mov John[65], ax. Unfortunately, memorizing all the offsets to fields in a structure defeats the whole purpose of using them in the first place. After all, if you've got to deal with these numeric offsets why not just use an array of bytes instead of a structure?

Well, as it turns out, MASM lets you refer to field names in a structure using the same mechanism C and Pascal use: the dot operator. To store ax into the Major field, you could use mov John.Major,ax instead of the previous instruction. This is much more readable and certainly easier to use.

Note that the use of the dot operator does *not* introduce a new addressing mode. The instruction mov John.Major,ax still uses the displacement only addressing mode. MASM simply adds the base address of John with the offset to the Major field (65) to get the actual displacement to encode into the instruction.

You may also specify default initial values when creating a structure. In the previous example, the fields of the student structure were given indeterminate values by specifying "?" in the operand field of each field's declaration. As it turns out, there are two different ways to specify an initial value for structure fields. Consider the following definition of a "point" data structure:

```
Point           struct
x               word    0
y               word    0
z               word    0
Point           ends
```

Whenever you declare a variable of type point using a statement similar to

```
CurPoint        Point   {}
```

13. You *cannot* redefine a fieldname as an equate or macro label. You may, however, reuse a field name as a statement label. Also, note that versions of MASM prior to 6.0 do not support the ability to reuse structure field names.

MASM automatically initializes the CurPoint.x, CurPoint.y, and CurPoint.z variables to zero. This works out great in those cases where your objects usually start off with the same initial values[14]. Of course, it might turn out that you would like to initialize the X, Y, and Z fields of the points you declare, but you want to give each point a different value. That is easily accomplished by specifying initial values inside the braces:

```
Point1          point    {0,1,2}
Point2          point    {1,1,1}
Point3          point    {0,1,1}
```

MASM fills in the values for the fields in the order that they appear in the operand field. For Point1 above, MASM initializes the X field with zero, the Y field with one, and the Z field with two.

The type of the initial value in the operand field must match the type of the corresponding field in the structure definition. You cannot, for example, specify an integer constant for a real4 field, nor could you specify a value greater than 255 for a byte field.

MASM does not require that you initialize all fields in a structure. If you leave a field blank, MASM will use the specified default value (undefined if you specify "?" rather than a default value).

5.6.4 Arrays of Structures and Arrays/Structures as Structure Fields

Structs may contain other structures or arrays as fields. Consider the following definition:

```
Pixel           struct
Pt              point    {}
Color           dword    ?
Pixel           ends
```

The definition above defines a single point with a 32 bit color component. When initializing an object of type Pixel, the first initializer corresponds to the Pt field, *not the x-coordinate field*. **The following definition is incorrect:**

```
ThisPt          Pixel    {5,10}
```

The value of the first field ("5") is not an object of type point. Therefore, the assembler generates an error when encountering this statement. MASM will allow you to initialize the fields of ThisPt using declarations like the following:

```
ThisPt          Pixel    {,10}

ThisPt          Pixel    {{},10}

ThisPt          Pixel    {{1,2,3}, 10}

ThisPt          Pixel    {{1,,1}, 12}
```

The first and second examples above use the default values for the Pt field (x=0, y=0, z=0) and set the Color field to 10. Note the use of braces to surround the initial values for the point type in the second, third, and fourth examples. The third example above initializes the x, y, and z fields of the Pt field to one, two, and three, respectively. The last example initializes the x and z fields to one and lets the y field take on the initial value specified by the Point structure (zero).

Accessing Pixel fields is very easy. Like a high level language you use a single period to reference the Pt field and a second period to access the x, y, and z fields of point:

14. Note, of course, that the initial values for the x, y , and z fields need not all be zero. You could have initialized the fields to 1, 2, and 3 just as easily.

```
                mov     ax, ThisPt.Pt.X
                 .
                 .
                 .
                mov     ThisPt.Pt.Y, 0
                 .
                 .
                 .
                mov     ThisPt.Pt.Z, di
                 .
                 .
                 .
                mov     ThisPt.Color, EAX
```

You can also declare *arrays* as structure fields. The following structure creates a data type capable of representing an object with eight points (e.g., a cube):

```
Object8         struct
Pts             point   8 dup (?)
Color           dword   0
Object8         ends
```

This structure allocates storage for eight different points. Accessing an element of the Pts array requires that you know the size of an object of type point (remember, you must multiply the index into the array by the size of one element, six in this particular case). Suppose, for example, that you have a variable CUBE of type Object8. You could access elements of the Pts array as follows:

```
; CUBE.Pts[i].X := 0;

                mov     ax, 6
                mul     i           ;6 bytes per element.
                mov     si, ax
                mov     CUBE.Pts[si].X, 0
```

The one unfortunate aspect of all this is that you must know the size of each element of the Pts array. Fortunately, MASM provides an operator that will compute the size of an array element (in bytes) for you, more on that later.

5.6.5 Pointers to Structures

During execution, your program may refer to structure objects directly or indirectly using a pointer. When you use a pointer to access fields of a structure, you must load one of the 80x86's pointer registers (si, di, bx, or bp on processors less than the 80386) with the offset and es, ds, ss, or cs[15] with the segment of the desired structure. Suppose you have the following variable declarations (assuming the Object8 structure from the previous section):

```
Cube            Object8 {}
CubePtr         dword   Cube
```

CubePtr contains the address of (i.e., it is a pointer to) the Cube object. To access the Color field of the Cube object, you could use an instruction like mov eax,Cube.Color. When accessing a field via a pointer you need to load the address of the object into a segment:pointer register pair, such as es:bx. The instruction les bx,CubePtr will do the trick. After doing so, you can access fields of the Cube object using the disp+bx addressing mode. The only problem is "How do you specify which field to access?" Consider briefly, the following *incorrect* code:

```
                les     bx, CubePtr
                mov     eax, es:[bx].Color
```

15. Add FS or GS to this list for the 80386 and later.

There is one major problem with the code above. Since field names are local to a structure and it's possible to reuse a field name in two or more structures, how does MASM determine which offset Color represents? When accessing structure members directly (.e.g., mov eax,Cube.Color) there is no ambiguity since Cube has a specific type that the assembler can check. es:bx, on the other hand, can point at *anything*. In particular, it can point at any structure that contains a Color field. So the assembler cannot, on its own, decide which offset to use for the Color symbol.

MASM resolves this ambiguity by requiring that you explicitly supply a type in this case. Probably the easiest way to do this is to specify the structure name as a *pseudo-field*:

```
les     bx, CubePtr
mov     eax, es:[bx].Object8.Color
```

By specifying the structure name, MASM knows which offset value to use for the Color symbol[16].

5.7 Sample Programs

The following short sample programs demonstrate many of the concepts appearing in this chapter.

5.7.1 Simple Variable Declarations

```
; Sample variable declarations
; This sample file demonstrates how to declare and access some simple
; variables in an assembly language program.
;
; Randall Hyde
;
;
; Note: global variable declarations should go in the "dseg" segment:

dseg            segment  para public 'data'

; Some simple variable declarations:

character       byte     ?           ;"?" means uninitialized.
UnsignedIntVar  word     ?
DblUnsignedVar  dword    ?

;You can use the typedef statement to declare more meaningful type names:

integer         typedef  sword
char            typedef  byte
FarPtr          typedef  dword

; Sample variable declarations using the above types:

J               integer  ?
c1              char     ?
PtrVar          FarPtr   ?


; You can tell MASM & DOS to initialize a variable when DOS loads the
; program into memory by specifying the initial value in the operand
```

16. Users of MASM 5.1 and other assemblers should keep in mind that field names are *not* local to the structure. Instead, they must all be unique within a source file. As a result, such programs do not require the structure name in the "dot path" for a particular field. Keep this in mind when converting older code to MASM 6.x.

```
; field of the variable's declaration:

K               integer  4
c2              char     'A'
PtrVar2         FarPtr   L           ;Initializes PtrVar2 with the
                                     ; address of L.


; You can also set aside more than one byte, word, or double word of
; storage using these directives.  If you place several values in the
; operand field, separated by commas, the assembler will emit one byte,
; word, or dword for each operand:

L               integer  0, 1, 2, 3
c3              char     'A', 0dh, 0ah, 0
PtrTbl          FarPtr   J, K, L

; The BYTE directive lets you specify a string of characters byte enclosing
; the string in quotes or apostrophes.  The directive emits one byte of data
; for every character in the string (not including the quotes or apostrophes
; that delimit the string):

string          byte     "Hello world",0dh,0ah,0


dseg            ends




; The following program demonstrates how to access each of the above
; variables.

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg;These statements are provided by
                mov      ds, ax       ; shell.asm to initialize the
                mov      es, ax       ; segment register.


; Some simple instructions that demonstrate how to access memory:

                lea      bx, L        ;Point bx at first word in L.
                mov      ax, [bx];Fetch word at L.
                add      ax, 2[bx];Add in word at L+2 (the "1").
                add      ax, 4[bx];Add in word at L+4 (the "2").
                add      ax, 6[bx];Add in word at L+6 (the "3").
                mul      K            ;Compute (0+1+2+3)*123.
                mov      J, ax        ;Save away result in J.

                les      bx, PtrVar2;Loads es:di with address of L.
                mov      di, K        ;Loads 4 into di
                mov      ax, es:[bx][di];Fetch value of L+4.


; Examples of some byte accesses:

                mov      c1, ' '      ;Put a space into the c1 var.
                mov      al, c2       ;c3 := c2
                mov      c3, al
```

```
Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

5.7.2 Using Pointer Variables

```
; Using Pointer Variables in an Assembly Language Program
;
; This short sample program demonstrates the use of pointers in
; an assembly language program.
;
; Randall Hyde

dseg            segment para public 'data'


; Some variables we will access indirectly (using pointers):

J               word    0, 0, 0, 0
K               word    1, 2, 3, 4
L               word    5, 6, 7, 8

; Near pointers are 16-bits wide and hold an offset into the current data
; segment (dseg in this program).  Far pointers are 32-bits wide and hold
; a complete segment:offset address.  The following type definitions let
; us easily create near and far pointers

nWrdPtr         typedef near ptr word
fWrdPtr         typedef far ptr word


; Now for the actual pointer variables:

Ptr1            nWrdPtr ?
Ptr2            nWrdPtr K           ;Initialize with K's address.
Ptr3            fWrdPtr L           ;Initialize with L's segmented adrs.

dseg            ends



cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg    ;These statements are provided by
                mov     ds, ax      ; shell.asm to initialize the
                mov     es, ax      ; segment register.
```

```
; Initialize Ptr1 (a near pointer) with the address of the J variable.

                lea     ax, J
                mov     Ptr1, ax

; Add the four words in variables J, K, and L together using pointers to
; these variables:

                mov     bx, Ptr1    ;Get near ptr to J's 1st word.
                mov     si, Ptr2    ;Get near ptr to K's 1st word.
                les     di, Ptr3    ;Get far ptr to L's 1st word.



                mov     ax, ds:[si] ;Get data at K+0.
                add     ax, es:[di] ;Add in data at L+0.
                mov     ds:[bx], ax ;Store result to J+0.

                add     bx, 2       ;Move to J+2.
                add     si, 2       ;Move to K+2.
                add     di, 2       ;Move to L+2.



                mov     ax, ds:[si] ;Get data at K+2.
                add     ax, es:[di] ;Add in data at L+2.
                mov     ds:[bx], ax ;Store result to J+2.

                add     bx, 2       ;Move to J+4.
                add     si, 2       ;Move to K+4.
                add     di, 2       ;Move to L+4.



                mov     ax, ds:[si] ;Get data at K+4.
                add     ax, es:[di] ;Add in data at L+4.
                mov     ds:[bx], ax ;Store result to J+4.

                add     bx, 2       ;Move to J+6.
                add     si, 2       ;Move to K+6.
                add     di, 2       ;Move to L+6.



                mov     ax, ds:[si] ;Get data at K+6.
                add     ax, es:[di] ;Add in data at L+6.
                mov     ds:[bx], ax ;Store result to J+6.



Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

5.7.3 Single Dimension Array Access

```
; Sample variable declarations
; This sample file demonstrates how to declare and access some single
; dimension array variables in an assembly language program.
;
; Randall Hyde


                .386                            ;Need to use some 80386
                option  segment:use16           ; addressing modes.

dseg            segment para public 'data'

J               word    ?
K               word    ?
L               word    ?
M               word    ?

JD              dword   0
KD              dword   1
LD              dword   2
MD              dword   3

; Some simple uninitialized array declarations:

ByteAry         byte    4 dup (?)
WordAry         word    4 dup (?)
DwordAry        dword   4 dup (?)
RealAry         real8   4 dup (?)


; Some arrays with initialized values:

BArray          byte    0, 1, 2, 3
WArray          word    0, 1, 2, 3
DWArray         dword   0, 1, 2, 3
RArray          real8   0.0, 1.0, 2.0, 3.0


; An array of pointers:

PtrArray        dword   ByteAry, WordAry, DwordAry, RealAry

dseg            ends

; The following program demonstrates how to access each of the above
; variables.

cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg    ;These statements are provided by
                mov     ds, ax      ; shell.asm to initialize the
                mov     es, ax      ; segment register.


; Initialize the index variables.  Note that these variables provide
; logical indices into the arrays.  Don't forget that we've got to
; multiply these values by the element size when accessing elements of
; an array.
```

```
                mov     J, 0
                mov     K, 1
                mov     L, 2
                mov     M, 3

; The following code shows how to access elements of the arrays using
; simple 80x86 addressing modes:

                mov     bx, J       ;AL := ByteAry[J]
                mov     al, ByteAry[bx]

                mov     bx, K       ;AX := WordAry[K]
                add     bx, bx      ;Index*2 since this is a word array.
                mov     ax, WordAry[bx]

                mov     bx, L       ;EAX := DwordAry[L]
                add     bx, bx      ;Index*4 since this is a double
                add     bx, bx      ; word array.
                mov     eax, DwordAry[bx]

                mov     bx, M       ;BX := address(RealAry[M])
                add     bx, bx      ;Index*8 since this is a quad
                add     bx, bx      ; word array.
                add     bx, bx
                lea     bx, RealAry[bx];Base address + index*8.

; If you have an 80386 or later CPU, you can use the 386's scaled indexed
; addressing modes to simplify array access.

                mov     ebx, JD
                mov     al, ByteAry[ebx]

                mov     ebx, KD
                mov     ax, WordAry[ebx*2]

                mov     ebx, LD
                mov     eax, DwordAry[ebx*4]

                mov     ebx, MD
                lea     bx, RealAry[ebx*8]

Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

5.7.4 Multidimensional Array Access

```
; Multidimensional Array declaration and access
;
```

```
; Randall Hyde


                .386                            ;Need these two statements to
                option  segment:use16           ; use the 80386 register set.


dseg            segment para public 'data'


; Indices we will use for the arrays.

J               word    1
K               word    2
L               word    3

; Some two-dimensional arrays.
; Note how this code uses the "dup" operator to suggest the size
; of each dimension.

B2Ary           byte    3 dup (4 dup (?))
W2Ary           word    4 dup (3 dup (?))
D2Ary           dword   2 dup (6 dup (?))



; 2D arrays with initialization.
; Note the use of data layout to suggest the sizes of each array.

B2Ary2          byte    0, 1, 2, 3
                byte    4, 5, 6, 7
                byte    8, 9, 10, 11

W2Ary2          word    0,  1,  2
                word    3,  4,  5
                word    6,  7,  8
                word    9, 10, 11

D2Ary2          dword   0,  1,  2,  3,  4,  5
                dword   6,  7,  8,  9, 10, 11

; A sample three dimensional array.

W3Ary           word    2 dup (3 dup (4 dup (?)))

dseg            ends


cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg    ;These statements are provided by
                mov     ds, ax      ; shell.asm to initialize the
                mov     es, ax      ; segment register.

; AL := B2Ary2[j,k]

                mov     bx, J       ;index := (j*4+k)
                add     bx, bx      ;j*2
                add     bx, bx      ;j*4
                add     bx, K       ;j*4+k
                mov     al, B2Ary2[bx]
```

```
; AX := W2Ary2[j,k]

                mov     ax, J       ;index := (j*3 + k)*2
                mov     bx, 3
                mul     bx          ;(j*3)-- This destroys DX!
                add     ax, k       ;(j*3+k)
                add     ax, ax      ;(j*3+k)*2
                mov     bx, ax
                mov     ax, W2Ary2[bx]


; EAX := D2Ary[i,j]

                mov     ax, J       ;index := (j*6 + k)*4
                mov     bx, 6
                mul     bx          ;DX:AX := j*6, ignore overflow in DX.
                add     ax, k       ;j*6 + k
                add     ax, ax      ;(j*6 + k)*2
                add     ax, ax      ;(j*6 + k)*4
                mov     bx, ax
                mov     eax, D2Ary[bx]


; Sample access of a three dimensional array.
;
; AX := W3Ary[J,K,L]

                mov     ax, J       ;index := ((j*3 + k)*4 + l)*2
                mov     bx, 3
                mul     bx          ;j*3
                add     ax, K       ;j*3 + k
                add     ax, ax      ;(j*3 + k)*2
                add     ax, ax      ;(j*3 + k)*4
                add     ax, l       ;(j*3 + k)*4 + l
                add     ax, ax      ;((j*3 + k)*4 + l)*2
                mov     bx, ax
                mov     ax, W3Ary[bx]


Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

5.7.5 Simple Structure Access

```
; Sample Structure Definitions and Accesses.
;
; Randall Hyde
```

```
dseg            segment  para public 'data'


; The following structure holds the bit values for an 80x86 mod-reg-r/m byte.

mode            struct
modbits         byte     ?
reg             byte     ?
rm              byte     ?
mode            ends


Instr1Adrs      mode     {};All fields uninitialized.
Instr2Adrs      mode     {}


; Some structures with initialized fields.

axbx            mode     {11b, 000b, 000b}       ;"ax, ax" adrs mode.
axdisp          mode     {00b, 000b, 110b}       ;"ax, disp" adrs mode.
cxdispbxsi      mode     {01b, 001b, 000b}       ;"cx, disp8[bx][si]" mode.


; Near pointers to some structures:

sPtr1           word     axdisp
sPtr2           word     Instr2Adrs

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg     ;These statements are provided by
                mov      ds, ax       ; shell.asm to initialize the
                mov      es, ax       ; segment register.


; To access fields of a structure variable directly, just use the "."
; operator like you would in Pascal or C:

                mov      al, axbx.modbits
                mov      Instr1Adrs.modbits, al

                mov      al, axbx.reg
                mov      Instr1Adrs.reg, al

                mov      al, axbx.rm
                mov      Instr1Adrs.rm, al


; When accessing elements of a structure indirectly (that is, using a
; pointer) you must specify the structure type name as the first
; "field" so MASM doesn't get confused:

                mov      si, sPtr1
                mov      di, sPtr2

                mov      al, ds:[si].mode.modbits
                mov      ds:[di].mode.modbits, al
```

```
                mov     al, ds:[si].mode.reg
                mov     ds:[di].mode.reg, al

                mov     al, ds:[si].mode.rm
                mov     ds:[di].mode.rm, al


Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

5.7.6 Arrays of Structures

```
; Arrays of Structures
;
; Randall Hyde


dseg            segment para public 'data'


; A structure that defines an (x,y) coordinate.
; Note that the Point data type requires four bytes.

Point           struct
X               word    ?
Y               word    ?
Point           ends



; An uninitialized point:

Pt1             Point   {}

; An initialized point:

Pt2             Point   {12,45}


; A one-dimensional array of uninitialized points:

PtAry1          Point   16 dup ({})  ;Note the "{}" inside the parens.


; A one-dimensional array of points, all initialized to the origin.

PtAry1i         Point   16 dup ({0,0})
```

```
; A two-dimensional array of points:

PtAry2          Point    4 dup (4 dup ({}))


; A three-dimensional array of points, all initialized to the origin.

PtAry3          Point    2 dup (3 dup (4 dup ({0,0})))



; A one-dimensional array of points, all initialized to different values:

iPtAry          Point    {0,0}, {1,2}, {3,4}, {5,6}


; Some indices for the arrays:

J               word     1
K               word     2
L               word     3

dseg            ends




; The following program demonstrates how to access each of the above
; variables.

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg     ;These statements are provided by
                mov      ds, ax       ; shell.asm to initialize the
                mov      es, ax       ; segment register.

; PtAry1[J] := iPtAry[J]

                mov      bx, J        ;Index := J*4 since there are four
                add      bx, bx       ; bytes per array element (each
                add      bx, bx       ; element contains two words).

                mov      ax, iPtAry[bx].X
                mov      PtAry1[bx].X, ax

                mov      ax, iPtAry[bx].Y
                mov      PtAry1[bx].Y, ax

; CX := PtAry2[K,L].X;  DX := PtAry2[K,L].Y

                mov      bx, K        ;Index := (K*4 + J)*4
                add      bx, bx       ;K*2
                add      bx, bx       ;K*4
                add      bx, J        ;K*4 + J
                add      bx, bx       ;(K*4 + J)*2
                add      bx, bx       ;(K*4 + J)*4

                mov      cx, PtAry2[bx].X
                mov      dx, PtAry2[bx].Y

; PtAry3[j,k,l].X := 0
```

```
                        mov     ax, j       ;Index := ((j*3 +k)*4 + l)*4
                        mov     bx, 3
                        mul     bx          ;j*3
                        add     ax, k       ;j*3 + k
                        add     ax, ax      ;(j*3 + k)*2
                        add     ax, ax      ;(j*3 + k)*4
                        add     ax, l       ;(j*3 + k)*4 + l
                        add     ax, ax      ;((j*3 + k)*4 + l)*2
                        add     ax, ax      ;((j*3 + k)*4 + l)*4
                        mov     bx, ax
                        mov     PtAry3[bx].X, 0

Quit:                   mov     ah, 4ch     ;Magic number for DOS
                        int     21h         ; to tell this program to quit.
Main                    endp
cseg                    ends

sseg                    segment  para stack 'stack'
stk                     byte     1024 dup ("stack   ")
sseg                    ends

zzzzzzseg               segment  para public 'zzzzzz'
LastBytes               byte     16 dup (?)
zzzzzzseg               ends
                        end      Main
```

5.7.7 Structures and Arrays as Fields of Another Structure

```
; Structures Containing Structures as fields
; Structures Containing Arrays as fields
;
; Randall Hyde


dseg                    segment  para public 'data'

Point                   struct
X                       word     ?
Y                       word     ?
Point                   ends

; We can define a rectangle with only two points.
; The color field contains an eight-bit color value.
; Note: the size of a Rect is 9 bytes.

Rect                    struct
UpperLeft               Point    {}
LowerRight              Point    {}
Color                   byte     ?
Rect                    ends

; Pentagons have five points, so use an array of points to
; define the pentagon.  Of course, we also need the color
; field.
; Note: the size of a pentagon is 21 bytes.

Pent                    struct
Color                   byte     ?
Pts                     Point    5 dup ({})
Pent                    ends
```

```
; Okay, here are some variable declarations:

Rect1           Rect    {}
Rect2           Rect    {{0,0}, {1,1}, 1}

Pentagon1       Pent    {}
Pentagons       Pent    {}, {}, {}, {}

Index           word    2

dseg            ends


cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg    ;These statements are provided by
                mov     ds, ax      ; shell.asm to initialize the
                mov     es, ax      ; segment register.

; Rect1.UpperLeft.X := Rect2.UpperLeft.X

                mov     ax, Rect2.Upperleft.X
                mov     Rect1.Upperleft.X, ax

; Pentagon1 := Pentagons[Index]

                mov     ax, Index;Need Index*21
                mov     bx, 21
                mul     bx
                mov     bx, ax

; Copy the first point:

                mov     ax, Pentagons[bx].Pts[0].X
                mov     Pentagon1.Pts[0].X, ax

                mov     ax, Pentagons[bx].Pts[0].Y
                mov     Pentagon1.Pts[0].Y, ax

; Copy the second point:

                mov     ax, Pentagons[bx].Pts[2].X
                mov     Pentagon1.Pts[4].X, ax

                mov     ax, Pentagons[bx].Pts[2].Y
                mov     Pentagon1.Pts[4].Y, ax

; Copy the third point:

                mov     ax, Pentagons[bx].Pts[4].X
                mov     Pentagon1.Pts[8].X, ax

                mov     ax, Pentagons[bx].Pts[4].Y
                mov     Pentagon1.Pts[8].Y, ax

; Copy the fourth point:

                mov     ax, Pentagons[bx].Pts[6].X
                mov     Pentagon1.Pts[12].X, ax
```

```
                mov     ax, Pentagons[bx].Pts[6].Y
                mov     Pentagon1.Pts[12].Y, ax

; Copy the fifth point:

                mov     ax, Pentagons[bx].Pts[8].X
                mov     Pentagon1.Pts[16].X, ax

                mov     ax, Pentagons[bx].Pts[8].Y
                mov     Pentagon1.Pts[16].Y, ax

; Copy the Color:

                mov     al, Pentagons[bx].Color
                mov     Pentagon1.Color, al


Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp
cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

5.7.8 Pointers to Structures and Arrays of Structures

```
; Pointers to structures
; Pointers to arrays of structures
;
; Randall Hyde


                .386                            ;Need these two statements so
                option  segment:use16           ; we can use 80386 registers

dseg            segment para public 'data'

; Sample structure.
; Note: size is seven bytes.

Sample          struct
b               byte    ?
w               word    ?
d               dword   ?
Sample          ends


; Some variable declarations:

OneSample       Sample  {}
SampleAry       Sample  16 dup ({})

; Pointers to the above
```

```
OnePtr          word    OneSample   ;A near pointer.
AryPtr          dword   SampleAry


; Index into the array:

Index           word    8

dseg            ends




; The following program demonstrates how to access each of the above
; variables.

cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg    ;These statements are provided by
                mov     ds, ax      ; shell.asm to initialize the
                mov     es, ax      ; segment register.

; AryPtr^[Index] := OnePtr^


                mov     si, OnePtr  ;Get pointer to OneSample
                les     bx, AryPtr  ;Get pointer to array of samples
                mov     ax, Index   ;Need index*7
                mov     di, 7
                mul     di
                mov     di, ax

                mov     al, ds:[si].Sample.b
                mov     es:[bx][di].Sample.b, al

                mov     ax, ds:[si].Sample.w
                mov     es:[bx][di].Sample.w, ax

                mov     eax, ds:[si].Sample.d
                mov     es:[bx][di].Sample.d, eax


Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

5.8 Laboratory Exercises

In these laboratory exercises you will learn how to step through a program using CodeView and observe the results. Knowing how to trace through a program is an important skill to posses. There is no better way to learn assembly language than to single step through a program and observe the actions of each instruction. Even if you already know assembly language, tracing through a program with a debugger like CodeView is one of the best ways to verify that your program is working properly.

In these lab exercises you will assemble the sample program provided in the previous section. Then you will run the assembled program under CodeView and step through each instruction in the program. **For your lab report:** you will include a listing of each program and describe the operation of each statement including data loaded into any affected registers or values stored away into memory.

The following paragraphs describe one experimental run – stepping through the pgm5_1.asm program. Your lab report should contain similar information for all eight sample programs.

To assemble your programs, use the ML command with the /Zi option. For example, to assemble the first sample program you would use the following DOS command:

```
ml /Zi pgm5_1.asm
```

This command produces the pgm5_1.exe file that contains CodeView debugging information. You can load this program into the CodeView debugger using the following command:

```
cv pgm5_1
```

Once you are inside CodeView, you can single step through the program by repeatedly pressing the F8 key. Each time you press the F8 key, CodeView executes a single instruction in the program.

To better observe the results while stepping through your program, you should open the register window. If it is not open already, you can open it by pressing the F2 key. As the instructions you execute modify the registers, you can observe the changes.

All the sample programs begin with a three-instruction sequence that initializes the DS and ES registers; pressing the F8 key three times steps over these instructions and (on one system) loads the AX, ES, and DS registers with the value 1927h (this value will change on different systems).

Single stepping over the `lea bx, L` instruction loads the value 0015h into `bx`. Single stepping over the group of instructions following the `lea` produces the following results:

```
mov     ax, [bx]            ;AX = 0
add     ax, 2[bx]           ;AX = 1
add     ax, 4[bx]           ;AX = 3
add     ax, 6[bx]           ;AX = 6
mul     K                   ;AX = 18 (hex)
mov     J, ax               ;J is now equal to 18h.
```

Comments on the above instructions: this code loads bx with the base address of array `L` and then proceeds to compute the sum of `L[i]`, i=0..3 (0+1+2+3). It then multiples this sum by K (4) and stores the result into J. Note that you can use the "dw J" command in the command window to display J's current value (the "J" must be upper case because CodeView is case sensitive).

```
les     bx, PtrVar2         ;BX = 0015, ES = 1927
mov     di, K               ;DI = 4
mov     ax, es:[bx][di]     ;AX = 2
```

Comments on the above code: The les instruction loads es:bx with the pointer variable PtrVar2. This variable contains the address of the L variable. Then this code loads di with the value of K and completes by loading the second element of L into ax.

```
mov     c1, ' '
mov     al, c2
mov     c3, al
```

These three instructions simply store a space into byte variable c1 (verify with a "da c1" command in the command window) and they copy the value in c2 ("A") into the AL register and the c3 variable (verified with "da c3" command).

For your lab report: assemble and step through pgm5_2.asm, pgm5_3.asm, pgm5_4.asm, pgm5_5.asm, pgm5_6.asm, pgm5_7.asm, and pgm5_8.asm. Describe the results in a fashion similar to the above.

5.9 Programming Projects

1) The PC's video display is a *memory mapped I/O device*. That is, the display adapter maps each character on the text display to a word in memory. The display is an 80x25 array of words declared as follows:

```
display:array[0..24,0..79] of word;
```

Display[0,0] corresponds to the upper left hand corner of the screen, display[0,79] is the upper right hand corner, display[24,0] is the lower left hand corner, and display[24,79] is the lower right hand corner of the display.

The L.O. byte of each word holds the ASCII code of the character to appear on the screen. The H.O. byte of each word contains the *attribute* byte (see "The PC Video Display" on page 1247 for more details on the attribute byte). The base address of this array is B000:0 for monochrome displays and B800:0 for color displays.

The Chapter Five subdirectory contains a file named PGM5_1.ASM. This file is a skeletal program that manipulates the PC's video display. This program, when complete, writes a series of period to the screen and then it writes a series of blues spaces to the screen. It contains a main program that uses several instructions you probably haven't seen yet. These instructions essentially execute a for loop as follows:

```
for i:= 0 to 79 do
          for j := 0 to 24 do
                    putscreen(i,j,value);
```

Inside this program you will find some comments that instruct you to supply the code that stores the value in AX to location display[i,j]. Modify this program as described in its comments and test the result.

For this project, you need to declare two word variables, I and J, in the data segment. Then you will need to modify the "PutScreen" procedure. Inside this procedure, as directed by the comments in the file, you will need to compute the index into the screen array and then store the value in the ax register to location es:[bx+0] (assuming you've computed the index into bx). Note that es:[0] is the base address of the video display in this procedure. Check your code carefully before attempting to run it. If your code malfunctions, it may crash the system and you will have to reboot. This program, if operating properly, will fill the screen with periods and wait until you press a key. Then it will fill the screen with blue spaces. You should probably execute the DOS "CLS" (clear screen) command after this program executes properly. Note that there is a working version of this program named p5_1.exe in the Chapter Five directory. You can run this program to check out it's operation if you are having problems.

2) The Chapter Five subdirectory contains a file named PGM5_2.ASM. This file is a program (except for two short subroutines) that generates mazes and solves them on the screen. This program requires that you complete two subroutines: MazeAdrs and ScrnAdrs. These two procedures appear at the beginning of the file; you should ignore the remainder

of this program. When the program calls the MazeAdrs function, it passes an X coordinate in the dx register and a Y-coordinate in the cx register. You need to compute the index into an 27x82 array defined as follows:

```
maze:array[0..26, 0..81] of word;
```

Return the index in the ax register. *Do not access the maze array; the calling code will do that for you.*

The ScrnAdrs function is almost identical to the MazeAdrs function except it computes an index into a 25x80 array rather than a 27x82 array. As with MazeAdrs, the X-coordinate will be in the dx register and the Y-coordinate will be in the cx register.

Complete these two functions, assemble the program, and run it. Be sure to check your work over carefully. If you make any mistakes you will probably crash the system.

3) Create a program with a single dimension array of structures. Place at least four fields (your choice) in the structure. Write a code segment to access element "i" ("i" being a word variable) in the array.

4) Write a program which copies the data from a 3x3 array and stores the data into a second 3x3 array. For the first 3x3 array, store the data in row major order. For the second 3x3 array, store the data in column major order. Use nine sequences of instructions which fetch the word at location (i,j) (i=0..2, j=0..2).

5) Rewrite the code sequence above just using MOV instructions. Read and write the array locations directly, do not perform the array address computations.

5.10 Summary

This chapter presents an 80x86-centric view of memory organization and data structures. This certainly isn't a complete course on data structures. This chapter discussed the primitive and simple composite data types and how to declare and use them in your program. Lots of additional information on the declaration and use of simple data types appears later in this text.

One of the main goals of this chapter is to describe how to declare and use *variables* in an assembly language program. In an assembly language program you can easily create byte, word, double word, and other types of variables. Such scalar data types support boolean, character, integer, real, and other single data types found in typical high level languages. See:

- "Declaring Variables in an Assembly Language Program" on page 196
- "Declaring and using BYTE Variables" on page 198
- "Declaring and using WORD Variables" on page 200
- "Declaring and using DWORD Variables" on page 201
- "Declaring and using FWORD, QWORD, and TBYTE Variables" on page 202
- "Declaring Floating Point Variables with REAL4, REAL8, and REAL10" on page 202

For those who don't like using variable type names like `byte`, `word`, etc., MASM lets you create your own type names. You want to call them *Integers* rather than *Words*? No problem, you can define your own type names use the `typedef` statement. See:

- "Creating Your Own Type Names with TYPEDEF" on page 203

Another important data type is the *pointer*. Pointers are nothing more than memory addresses stored in variables (word or double word variables). The 80x86 CPUs support two types of pointers – near and far pointers. In real mode, near pointers are 16 bits long and contain the offset into a known segment (typically the data segment). Far pointers are 32 bits long and contain a full segment:offset logical address. Remember that you must use one of the register indirect or indexed addressing modes to access the data referenced by a pointer. For those who want to create their own pointer types (rather than simply

using `word` and `dword` to declare near and far pointers), the `typedef` instruction lets you create named pointer types. See:

- "Pointer Data Types" on page 203

A *composite data type* is one made up from other data types. Examples of composite data types abound, but two of the more popular composite data types are arrays and structures (records). An array is a group of variables, all the same type. A program selects an element of an array using an integer index into that array. Structures, on the other hand, may contain fields whose types are different. In a program, you select the desired field by supplying a field name with the *dot operator*. See:

- "Arrays" on page 206
- "Multidimensional Arrays" on page 210
- "Structures" on page 218
- "Arrays of Structures and Arrays/Structures as Structure Fields" on page 220
- "Pointers to Structures" on page 221

5.11 Questions

1) In what segment (8086) would you normally place your variables?

2) Which segment in the SHELL.ASM file normally corresponds to the segment containing your variables?

3) Describe how to declare byte variables. Give several examples. What would you normally use byte variables for in a program?

4) Describe how to declare word variables. Give several examples. Describe what you would use them for in a program.

5) Repeat question 21 for double word variables.

6) Explain the purpose of the TYPEDEF statement. Give several examples of its use.

7) What is a pointer variable?

8) What is the difference between a *near* and a *far* pointer?

9) How do you access the object pointed at by a far pointer. Give an example using 8086 instructions.

10) What is a composite data type?

11) How do you declare arrays in assembly language? Give the code for the following arrays:

a) A two dimensional 4x4 array of bytes b) An array containing 128 double words

c) An array containing 16 words d) A 4x5x6 three dimensional array of words

12) Describe how you would access a single element of each of the above arrays. Provide the necessary formulae and 8086 code to access said element (assume variable I is the index into single dimension arrays, I & J provide the index into two dimension arrays, and I, J, & K provide the index into the three dimensional array). Assume row major ordering, where appropriate.

13) Provide the 80386 code, using the scaled indexing modes, to access the elements of the above arrays.

14) Explain the difference between row major and column major array ordering.

15) Suppose you have a two-dimensional array whose values you want to initialize as follows:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Provide the variable declaration to accomplish this. Note: Do not use 8086 machine instructions to initialize the array. Initialize the array in your data segment.

```
Date=           Record
                          Month:integer;
                          Day:integer;
                          Year:integer;
                end;

Time=           Record
                          Hours:integer;
                          Minutes:integer;
                          Seconds:integer;
                end;

VideoTape =             record
                          Title:string[25];
                          ReleaseDate:Date;
                          Price:Real; (* Assume four byte reals *)
```

```
                                    Length: Time;
                                    Rating:char;
                                  end;

TapeLibrary : array [0..127] of VideoTape; (*This is a variable!*)
```

17) Suppose ES:BX points at an object of type VideoTape. What is the instruction that properly loads the `Rating` field into AL?