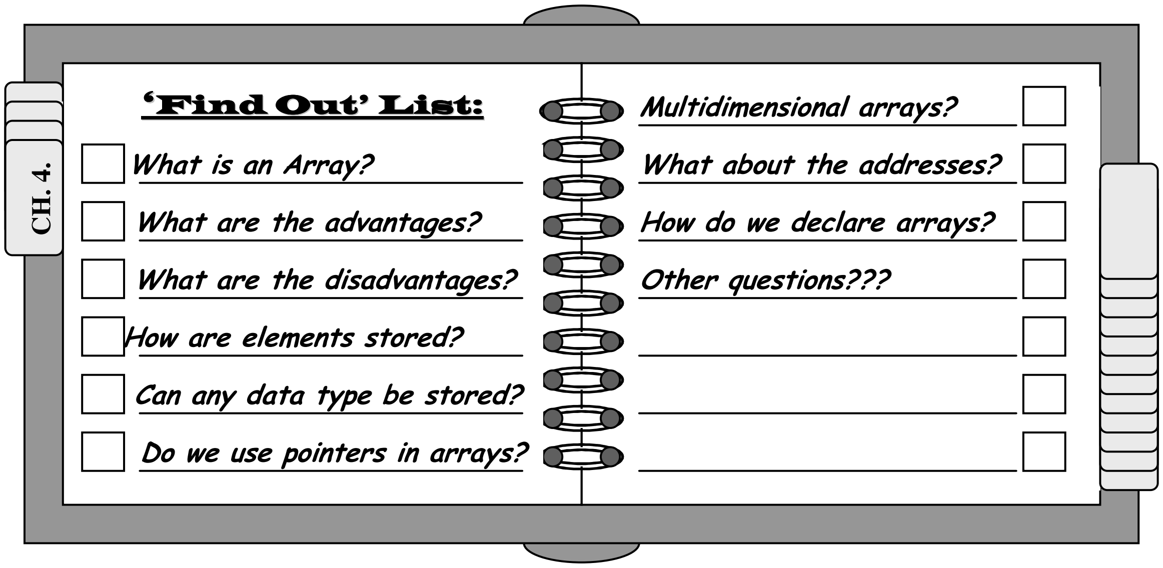
CHAPTER 4: NUMERIC ARRAYS

"All things begin in order, so shall they end"
Thomas Browne (1605 - 1682)



CH. 4.

'Find Out' List:

- What is an Array?
- What are the advantages?
- What are the disadvantages?
- How are elements stored?
- Can any data type be stored?
- Do we use pointers in arrays?
- Multidimensional arrays?
- What about the addresses?
- How do we declare arrays?
- Other questions???

Introduction

In this chapter, we introduce our first simple abstract data structure: ***Numeric Arrays*** (arrangements of integers and real numbers) You may be wondering why we chose numeric arrays as our first abstract data structure. In some respects, strings are much simpler. In fact, strings *are* actually numeric arrays (which we could have concluded since we know that characters are really as numbers), but there are a few extra considerations with strings that we do not wish to deal with here (nothing major; just not anything that we need to be concerned with at the present time). We will deal with strings in the following chapter.

One consideration in which abstract data type to introduce first was simplicity. Numeric arrays certainly fill this criteria (although, as we get further into the subject matter, we find that arrays *can* become somewhat complicated). Another consideration was structure usefulness. Arrays are, arguably, the most used data structure in existence. Finally, another concern was introducing structures that would serve as building blocks for later data structures. Arrays can be (and frequently are) incorporated into a number of additional data structures.

Numeric Arrays

An array is a data structure which contains a ***fixed*** number of ***contiguous*** storage elements ***all of the same data type***. A one-dimensional array is referred to as a ***vector.*** Multi-dimensional arrays are referred to as ***Matrices*** (pl.) or a ***Matrix*** (sg.).

Def

✓ ***What is an Array?***

Suppose that we wished to store the first 10 prime numbers (1, 2, 3, 5, 7, 11, 13, 17, 19, 23) in RAM. Of course, we could do this by storing them as individual integer variables (i.e., reserving 10 individual integer addresses) one for each number. For example, using the c declaration:

C/C++ Code 4.1

```
int prime1 = 1, prime2 = 2, prime3 = 3, prime4 = 5, prime5 = 7, prime6 = 11,
    prime7 = 13, prime8 = 17, prime9 = 19, prime10 = 23;
```

The problems with declaring variables in this manner, however, are:

NOTE: Since all values are less than 255, we could have declared these variables using the data type CHAR

1. It is very tedious to have to declare each of the individual variables
2. If we need to find a particular value and we don't know which variable location it is stored in, searching for it becomes cumbersome (we'll see why a little later).

In this case we wish to store all the variables in order, and since each prime number takes up the same amount of storage (2-bytes per integer), all we really need to do is establish a *base address* for the first element in the array. Remember, this was the definition we gave earlier: an array is a ***fixed*** number of ***contiguous*** storage elements ***all of the same data type***. In RAM (assuming that the first available contiguous block (= 10 elements * 2-bytes/element = 20 bytes of storage) was at address 20000), it might look like:

Figure 4.1

20000 (*prime1*)	20001 (*prime1*)	20002 (*prime2*)	20003 (*prime2*)
00000000	**00000001**	**00000000**	**00000010**
20004 (*prime3*)	20005 (*prime3*)	20006 (*prime4*)	2007 (prime4)
00000000	**00000011**	**00000000**	**00000101**
20008 (*prime5*)	20009 (*prime5*)	20010 (*prime6*)	20011 (*prime6*)
00000000	**00000111**	**00000000**	**00001011**
20012 (*prime7*)	20013 (*prime7*)	20014 (*prime8*)	20015 (*prime8*)
00000000	**000001101**	**00000000**	**00010001**
20016 (*prime9*)	20017 (*prime9*)	20018 (*prime10*)	20019 (*prime10*)
00000000	**000010111**	**00000000**	**00011011**

where:

Table 4.1

Variable Name	address(es)	Decimal Value	Binary Value
prime1	20000/20001	1	0000000000000001
prime2	20002/20003	2	0000000000000011
prime3	20004/20005	3	0000000000000101
prime4	20006/20007	5	0000000000000111
prime5	20008/20009	7	0000000000001011
prime6	20010/20011	11	0000000000001101
prime7	20012/20013	13	0000000000010001
prime8	20014/20015	15	0000000000010011
prime9	20016/20017	19	0000000000010111
prime10	20018/20019	23	0000000000011011

Let's take the second problem first. ***Why is it cumbersome to search for an element in the block?***

Suppose we were looking for the value 19 (which is actually associated with variable *prime8*, or address(es) 20014/20015) but we did not know where it was located, we would have to enter (the relevant portion of) the c code:

C/C++ Code 4.2.

```
if (prime1 == 19) then printf("Stored in prime1\n");
  else if (prime2 ==19) then printf("Stored in prime2\n");
      else if (prime3 == 19) then printf("Stored in prime3\n");
           else if (prime4 == 19) then printf("Stored in prime4\n");
                else if (prime5 == 19) then printf("Stored in prime5\n");
                     else if (prime6 == 19) then printf("Stored in prime6\n");
                           else if (prime7 == 19) then printf("Stored in prime7\n");
                                 else if (prime8 == 19) then printf("Stored in prime8\n");
                                       else if (prime9 == 19) then printf("Stored in prime9\n");
                                            else if (prime10 == 10) then printf("Stored in prime10\n");
                                                   else printf("The value does not exist\n");
```

? ***How do we store the elements on an array (vector)???***

The c programming language (as do all third-generation languages provide for convenient declaration of an array:

C/C++ Code 4.3

```
int prime[10] = {1, 3, 5, 7, 11, 13, 17, 19, 23, 27};h
```

Because we know that the elements are stored ***contiguously***, and that each element in the array takes 2-bytes (16-bits) of storage, all we really have to know is the ***base address*** of the array. Once we have established a base address, we can calculate where every other element in the array is located (since each requires 2-bytes of storage). For example if the base address of the array (= the first element in the array) is at address 20000 (and 20001), then:

✓ ***How are elements stored ?***

Table 4.2

Element no/Position/Index	**Address(es)**	**Reference**
1	20000 (and 20001)	prime[0]
2	20000 + 2 = 20002 (and 20003)	prime[1]
3	20002 + 2 = 20004 (and 20005)	prime[2]
4	20004 + 2 = 20006 (and 20007)	prime[3]
5	20006 + 2 = 20008 (and 20009)	prime[4]
6	20008 + 2 = 20010 (and 20011)	prime[5]
7	20010 + 2 = 20012 (and 20013)	prime[6]
8	20012 + 2 = 20014 (and 20015)	prime[7]
9	20014 + 2 = 20016 (and 20017)	prime[8]
10	20016 + 2 = 20018 (and 20019)	prime[9]

? ***What is the column to the right? What does reference mean?***

Calculating Array Addresses

One of the advantages of an array is that we are able to readily find the address of any element in that array using very simple calculations. All we need is two components:

✓ ***What are the advantages ?***

1. The ***base address*** of the array (the location in memory where the array begins)
2. The array ***offset*** (often also referred to as the *subscript* or *index*)

Just as we did with the scalar variables in Figure 4.1., we know that the elements in the array will be stored contiguously (i.e., one right after the other). As before, the same amount of storage will be required. Assuming we can start storing the numbers at location 20000, the first number will be stored at location 20000 (and 20001), and the last number will be stored at location 200018 (and 20019). In this case that means that *prime*[0] = 1 will be stored in addresses 20000 and 20001, *prime*[1] = 3 will be stored in addresses 20002 and 20003, and *prime*[9] = 27 will be stored in addresses 20018 and 20019. The data is allocated in the same order as before, but the variable names have changed (see figure 4.2).

Figure 4.2

20000 (*prime*[0])	20001 (*prime*[0])	20002 (*prime*[1])	20003 (*prime*[1])
00000000	**00000001**	**00000000**	**00000011**
20004 (*prime*[2])	20005 (*prime*[2])	20006 (*prime*[3])	2007 (*prime*[3])
00000000	**00000101**	**00000000**	**00000111**
20008 (*prime*[4])	20009 (*prime*[4])	20010 (*prime*[5])	20011 (*prime*[5])
00000000	**00001011**	**00000000**	**00001101**
20012 (*prime*[6])	20013 (*prime*[6])	20014 (*prime*[7])	20015 (*prime*[7])
00000000	**000100001**	**00000000**	**00010011**
20016 (*prime*[8])	20017 (*prime*[8])	20018 (*prime*[9])	20019 (*prime*[9])
00000000	**000010111**	**00000000**	**00011011**

?

Why did we start with prime[0] instead of prime[1] (which would indicate that it is the first element in the array)???

In some programming languages, we could have. In c, however, we determine the position of an element in an array by determining its ***offset*** from the ***base address***. The formula is very simple. Any element's address in the array can be calculated as:

Formula 4.1.

element address = base address + offset * (number bytes/element)

This calculation is only possible because we are working with arrays, which have the certain constraints. Remember what we did: When we made the declaration **int** *prime*[10], we requested that 20-bytes of *contiguous* storage be set aside (10 elements, *each* requiring 2-bytes of storage). Once we know where the first element in the array is located (in this case at address 20000), we know where every other element in the array is located by calculating their ***offset*** from the ***base address*** (i.e., 20000). Thus, the address of *prime*[4] would be 20000 + (4 *2) = 20008; *prime*[8] would be 20000 + (8 * 2) = 20016; *prime*[0] would be 20000 + (0 * 2) = 200000.

Once again, notice that the address of the variable *prime* (without a subscript) is identical to the (base) address of the individual element address for *prime*[0]**.**

***prime* == &*prime*[0]**

?

Why do we multiply the offset by 2 to determine the element's address???

Because this is an integer array. As we will see, if it were a character array, we would multiple the offset by 1 and add it to the base address; if it were a float array, or an array of longs, we would multiply the offset by 4 and add it to the base array. If it were an array of data type double, we would multiply the offset by 8; if it were an array of data type long double, we would multiply the offset by 16.

?

Why is finding an element in the array easier using this notation???

As we have seen, we can quickly calculate the address of every element in the array, we can immediately examine the contents of that location. Review the code necessary (C/C++ Code 4.2) we needed to find an element when it was stored on one of ten scalar variables. Using arrays, the same code[1] might look like:

C/C++ Code 4.4.

```
for (i = 0; i < 10; i++)                      // for the offset values from 0 to 9
  if (prime[i] == 19)                    // does the address contain the value 19?
      printf("Stored in prime[%d]\n",i);      // if so, print the variable offset
  else if (i == 9)                            // otherwise, if we haven't found it …
      printf("The value does not exist\n");   // display the not found message
```

Simpler? Quite a bit. And much more flexible. If we had a 100 elements, or even a 1000000000 elements in the array, the number of lines of code necessary would not change.

? ***Are there any disadvantages to arrays???***

✓ ***What are the disadvantages ?*** *Yes*. As we mentioned earlier, we still need **contiguous** memory.

? ***What if we were storing real numbers (floats)???***

Arrays of Data Type Float

There are very few difference between integer arrays and float arrays, except that we would be storing the values using 4-bytes (32-bits).

Assume that we were storing the square-roots of the (integer) numbers from 1 to 10. In this case, we would store (Figure 4.3.):

[1] This section of code assumes that we have made the declaration: **int** *i*;

Figure 4.3.

Integer Value	1	2	3	4	5	6	7	8	9	10
Square root :	1.000	1.414	1.732	2.000	2.236	2.449	2.646	2.828	3.000	3.162
array (*sqrt*) offset:	0	1	2	3	4	5	6	7	8	9

Let's also assume that we have already declared our real array and associated it with the variable name *sqrt.* That is assume we issued the command:

C/C++ Code 4.5.

```
float sqrt[10];
```

In this case, we would need 40 contiguous bytes of storage, since each element in the (**float**) array requires 4-bytes of storage. If this block of memory was available starting at (for example) location 15012, (i.e., *sqrt* == &*sqrt*[0] == 15012) the relevant portion of RAM might appear as it does in figures 4.4. and as outlined in table 4.3. [2]:

Figure 4.4.

15012 (*sqrt*[0])	15013 (*sqrt*[0])	15014 (*sqrt*[0])	15015 (*sqrt*[0])
01000000	**00000000**	**00000000**	**00000001**
15016 (*sqrt*[1])	15017 (*sqrt*[1])	15018 (*sqrt*[1])	15019 (*sqrt*[1])
01000000	**00000000**	**00000101**	**10000110**
15020 (*sqrt*[2])	15021 (*sqrt*[2])	15022 (*sqrt*[2])	15023 (*sqrt*[2])
01000000	**00000000**	**0000110**	**11000100**
15024 (*sqrt*[3])	15025 (*sqrt*[3])	15026 (*sqrt*[3])	15027 (*sqrt*[3])
01000000	**00000000**	**00000000**	**00000010**
15028 (*sqrt*[4])	15029 (*sqrt*[4])	15030 (*sqrt*[4])	15031 (*sqrt*[4])
01000000	**00000000**	**00001000**	**10111100**
15032 (*sqrt*[5])	15033 (*sqrt*[5])	15034 (*sqrt*[5])	15035 (*sqrt*[5])
01000000	**00000000**	**00001001**	**10010001**
15036 (*sqrt*[6])	15037 (*sqrt*[6])	15038 (*sqrt*[6])	15039 (*sqrt*[6])
01000000	**00000000**	**00001010**	**01010110**
15040 (*sqrt*[7])	15041 (*sqrt*[7])	15042 (*sqrt*[7])	15043 (*sqrt*[7])
01000000	**00000000**	**00001011**	**00001100**
15044 (*sqrt*[8])	15045 (*sqrt*[8])	15046 (*sqrt*[8])	15047 (*sqrt*[8])
01000000	**00000000**	**00000000**	**00000011**
15048 (*sqrt*[9])	15049 (*sqrt*[9])	15050 (*sqrt*[9])	15051 (*sqrt*[9])
01000000	**00000000**	**00001100**	**01011010**

[2] Binary numbers entered are not true representations of the values

Where:

Table 4.3.

Array offset	Address(es)
0	15012 (through 15015)
1	15012 + 1 * 4 = 15016 (through 15019)
2	15012 + 2 * 4 = 15020 (through 15023)
3	15012 + 3 * 4 = 15024 (through 15027)
4	15012 + 4 * 4 = 15028 (through 15031)
5	15012 + 5 * 4 = 15032 (through 15035)
6	15012 + 6 * 4 = 15036 (through 15039)
7	15012 + 7 * 4 = 15040 (through 15043)
8	15012 + 8 * 4 = 15044 (through 15047)
9	15012 + 9 * 4 = 15048 (through 15051)

? ***What c code is necessary to store and locate an element in the array (for example, the value 2.646)???***

Basically, the same code as we used previously (see C/C++ Code 4.6):

C/C++ Code 4.6.

```
#include <stdio.h>
void main()
{ float sqrts[10] = {1.0, 1.414, 1.732, 2.0, 2.236, 2.449, 2.646, 2.828, 3.0, 3.162};
  int i;                                         // offset index
  for (i = 0; i < 10; i++)                       // set the loop parameters
      if (sqrts[i] == 2.646)                     // address contains 2.646?
              printf("Stored in sqrts[%d]\n",i);  // if so, print the variable offset
      else if (i == 9)                           // if we haven't found it …
             printf(""The value does not exist\n"); // display the not found message
  }
```

? ***Can any data type be stored in an array???***

Yes, provided ***all of the elements of the array are of the same (basic) data type*** (Let's not forget our definition of an array). In each case, the amount of contiguous storage required will be equal to the number of elements in the array times the amount of storage required for the data type. For the available c programming language data types:

✓ ***Can any data type be stored ?***

Table 4.4

Data Type	No. bytes in array
char	1 * n
short, int	2 * n
long, float	4 * n
double	8 * n
long double	16 * n

Where ***n*** is the number of elements in an array.

?

What if I need to store a large number of variables, say 100,000 elements of data type long double ???

In this case, we would need a total of 1,600,000 ***contiguous*** bytes of storage. It might not be available (we will address these issues when we get to dynamic memory allocation.

?

Does that mean we can NEVER mix data types???

In the manner we have discussed arrays – NO. However, we can create a new data type which will allow is to do so. We will discuss these data structures in chapter 7.

Accessing Array Elements Using Pointers

Because we know that the base address of an array is actually the address of the first element, and we know that each element in the array requires the same amount of storage, we can actually determine the contents of any element in the array without the use of subscripts. Take, for example, the following C/C++ program(Code 4.7)[3]:

[3] The code provided is not intended to be efficient; it does not stop once the value is found, nor will it print anything if the value is not found.

C/C++ Code 4.7.

```
#include <stdio.h>
void main()
{  float sqrts[10] = {1.0, 1.414, 1.732, 2.0, 2.236, 2.449, 2.645, 2.828, 3.0, 3.162};
   float *sqrtsptr;                              // pointer to the array
   sqrtsptr = sqrts;                             // set in the base address of the array
   while (sqrtsptr <= &sqrts[9])                 // search entire array
   {  if (*sqrtsptr == 2.236)                    // search for value 2.236
         printf("%5.3f found at address %p\n",*sqrtsptr, sqrtsptr);   // if found
      sqrtsptr++;   } }                          // increment the pointer value
```

In this program, the pointer to the array (*sqrtsptr*) first gets the base address of the array (*sqrtsptr* = *sqrts*: REMEMBER: referring to the variable name of an array without using any subscripts (offset) is the same as referring the base address of the first element in the array(e.g., *sqrt* == &*sqrt*[0])). According to our previous example, this is the address 15012. As long as the address contained in *sqrtsptr* is less than or equal to the address of the last element in the array (*sqrtsptr*[9] = 15048) we will check to see if the value of the floating-point number contained at that address is equal to the value we are looking for (in this case, 2.236). If it is, we will print out the address where the number is located (*sqrtsptr*). We then move on to the next element in the array (*sqrtsptr*++). In the first pass we increment the value of *sqrtsptr* from 15012 to 15016 (which is the address of *sqrt*[1]). The program will stop only when it contains the address 15052.

? ***Why isn't the value of sqrtsptr incremented from 15012 to 15013 after the first pass???***

Remember, we declared the pointer *sqrtsptr* to be of type **float**. Since floats require 4-bytes of storage, incrementing the pointer will increment the value of the pointer by 4 as well.

? ***How could the address contained in the variable (address) sqrtsptr ever become 15052? That address is not part of the array sqrts.***

The address 15052 may not be part of the array, but it is a valid address. In fact, if we were to enter the commands:

C/C++ Code 4.8.

```
sqrtsptr = sqrts[10];
printf("The value %7.3f is located in address %p\n", *sqrtsptr, sqrtsptr);
```

Something would be printed out for the value of **sqrtsptr* at location 15052. It would be pure jibberish (as we've see when when we try to print out values for uninitialized variables), but there would be something located in addresses 15052 through 15055

(REMEMBER: pointers require 4-bytes of storage), and the c program would attempt to interpret it as a floating-point number.

? ***How would this appear in RAM???***

Let's assume that the base address of the float array *sqrt* (*sqrt* == *sqrt*[0] == 15012; refer to figure 4.4. for a visual representation of how these values would be store in binary). Let's further assume that when we issued the declaration:

float **sqrtsptr*; (see code 4.7 for origin)

We were assigned the address 15126 (through 15129) which we associate with the variable name *sqrtsptr*, and where we will store an address (which, if we were to go to, would contain a value of type **float**). If we were to look at that section of RAM (addresses 15126 through 15129) we might (at the time of assignment) find:

Figure 4.5.

15126	15127	15128	15129
00001000	**00010001**	**00011000**	**00010011**

Which would translate (in simplified, decimal terms) as:

Figure 4.6.

15126 through 15129
135338003

since $2^{27} + 2^{20} + 2^{16} + 2^{12} + 2^{11} + 2^{4} + 2^{1} + 2^{0}$ = 134,217,728 + 1,048,576 + 65,536 + 4,096 + 2,048 +16 + 2 + 1 = 135,338,003

? **How could we have a RAM address of 135 Million something ??? My computer at home only has 64 Megabytes??**

Remember, we haven't initialized the variable (location) *sqrtsptr* yet. Whatever was in that location previously, is still there (regardless of whether the address exists or not).

After we issue the command: ***sqrtsptr = sqrts*;** (see Code 4.7.)

The relevant portion of RAM might appear as:

Figure 4.7.

15126 through 15129
15012

since *sqrts* == &*sqrts*[0] == 15012.

For the subsequent commands (again, refer to C/C++ code 4.7):

C/C++ Code 4.9.

while (*sqrtsptr* <= &*sqrts*[9])

/* As long as the contents of address 15126 (location *sqrtsptr*, which presently contains the value 15012) are less than 15048 (the address associated with *sqrts*[9] */

{ **if** (**sqrtsptr* == 2.236)

/* search for value the 2.236. The expression **sqrtsptr* means redirect to the address contained at location *sqrtsptr* (i.e., 15012) and evaluate the contents of that location (as a **float**, since that is how we declared it). The first time that means we should examine the contents of location 15012 (the address presently contained in location sqrtsptr) to see if contains the value 2.236. Since we know that address 15012 contains the value 1.000, the statement evaluates to false. That means we will consider the statement within the if statement: */

printf("%5.3f found at address %p\n",**sqrtsptr*, *sqrtsptr*);

/ * print out the address ONLY if found. In this case, we skip the command */

sqrtsptr++; } }

/* which will increment the contents of *sqrtsptr* (address 15126) by 4 since we defined *sqrtsptr* as a location in memory which will contain an address at which we can find the data type **float**. Since a **float** requires 4-bytes of storage, when we increment, we increment by 4 */

At the end of our first pass through the loop (we will continue as long as the contents of location *sqrtsptr* are less than or equal to 15048 (==&*sqrts*[9])), the portion of RAM associated with *sqrtsptr* would appear as:

Figure 4.7.

15126 through 15129
15016

Meaning that we are now pointing to address 15016 (through 15019, since we are pointing at a **float**), or to variable *sqrts*[1].

Table 4.5. shows how the program would progress, and what all of the variable values would be, as we continue through the loop.

Table 4.5.

Pass Number	Contents of *sqrtsptr*	Contents at **sqrtsptr*	(*sqrtsptr* <= &*sqrts*[9])	(**sqrtsptr* == 2.236)	Print??
1	15012	1.000	15012 <= 15048: NO	NO	NO
2	15016	1.414	15016 <= 15048: NO	NO	NO
3	15020	1.732	15020 <= 15048: NO	NO	NO
4	15024	2.000	15024 <= 15048: NO	NO	NO
5	15026	2.236	15028 <= 15048: NO	YES	YES

Notice that we did find a match (i.e., the condition **sqrtsptr* == 2.236 turned out to be true in pass number 5) so we printed out the expression[4]:

printf("%5.3f found at address %p\n",sqrtsptr*, *sqrtsptr*);**

Which yielded the output:

2.236 found at address 15026

The procedure we went through might also help to explain why if we issued the commands:

C/C++ Code 4.10.

```
sqrtsptr = sqrts[9];
sqrtsptr++;
printf("The value %7.3f is located in address %p\n", *sqrtsptr, sqrtsptr);
```

We would indeed (for our example) that the new contents (address) of *sqrtsptr* would be 15052.

✓ ***Do we use pointers in arrays ?***

? ***What about the contents of location 15052, or the value of *sqrtsptr ???***

Absolutely no idea. We could figure it out, however, as we did before, if we were to see the relevant portion of RAM.

Multidimensional Arrays

Arrays greater than one-dimension are also stored in RAM in a similar fashion, although at first glance they seem a little more complicated. Assume that we wished to store the squares and the cubes of the numbers from 1 to 5[5]. If we were to enter these values in a table they might appear as (Table 4.6):

[4] Notice that in this case, we would have continued checking ALL the elements in the array, eve, if we found a the number we were looking for. A better conditional check would have been:

while ((*sqrtsptr* <= &*sqrts*[9]) && (**sqrtsptr* != 2.236))

which would have exited the loop as soon as the value was found.

[5] This is obviously a trivial example, since it would be easier to calculate these values when needed.

Table 4.6.

Decimal Number	Number Squared	Number Cubed	Row:
1	1	1	1
2	4	8	2
3	9	81	3
4	16	256	4
5	25	625	5
Column:	1	2	

We could refer to any element in this table by its row *and* column number. For example, the value 4 (the square of 2) is located in row2, column 1; the value 256 (the cube of 4) is located in row 4, column 2. If we were to store these values as a two-dimensional array (a table) in RAM, it would basically be done in the same manner, although RAM obviously is not sectioned off into rows and columns. Following the same format as we have been using, the c declaration would be:

C/C++ Code 4.11.

```
int numtable[5][2] = {(1, 1), (4, 8), (9, 81), (16, 256), (25, 625)};
```

Once again, this statement reserves a section of memory, in this case 20 (*contiguous*) bytes of RAM.

? ***Why 20-bytes???***

Because we have 5 rows each with 2 columns each holding a 2-byte integer (5 * 2 * 2 = 20). In this case, assuming we had 20-bytes of contiguous storage available starting at address 9500 (and ending at address 9519; see figure 4.8).

Figure 4.8.

9500	9501	9502	9503
00000000	00000001	00000000	00000001
9504	9505	9506	9507
00000000	00000100	00000000	00001000
9508	9509	9510	9511
00000000	00001001	00000000	01010001
9512	9513	9514	9515
00000000	00010000	00000001	00000000
9516	9517	9518	9519
00000000	00011001	00000010	01110001

Where:

Table 4.7.

Array Offset	Decimal Value	RAM Address(es)	Binary Value
[0][0]	1	9500 (& 9501)	0000000000000001
[0][1]	1	9502 (& 9503)	0000000000000001
[1][0]	4	9504 (& 9505)	0000000000000100
[1][1]	8	9506 (& 9507)	0000000000001000
[2][0]	9	9508 (& 9509)	0000000001010001
[2][1]	81	9510 (& 9511)	0000000001010001
[3][0]	16	9512 (& 9513)	0000000000010000
[3][1]	256	9514 (& 9515)	0000000100000000
[4][0]	25	9516 (& 9517)	0000000000011001
[4][0]	625	9518 (& 9519)	0000001001110001

To put this in more understandable terms, the relevant portion of RAM might appear as (Figure 4.9):

Figure 4.9.

9500 & 9501	9502 & 9503	9504 & 9505
1	**1**	**4**
9506 & 9507	9508 & 9509	9510 & 9511
8	**9**	**81**
9512 & 9513	9514 & 9515	9516 & 9517
16	**256**	**25**
9518 & 9519	9520 ….	
625		

✓ ***Multidimensional Arrays ?***

? ***How do the array offsets work with multidimensional arrays???***

Calculating Multidimensional Array Addresses

As we saw with uni-dimensional arrays, these indices (e.g., [0][1], or [3][0]) are really offset references to the *base address* of the array (in this case, numtable, or numtable[0][0], or for our example, address 9500). Conceptually, we apply them as we did when we referred to the locations in a table by their row and column numbers. The difference is that instead of starting at row 1 and column 1, we start at row 0 and column 0. Operationally, however, we use the offsets calculate individual element base addresses. For example:

Table 4.9.

Array offsets	Calculated Address
[0][0]	9500 + (0 * 4 + 0 * 2) = 9500 + 0 + 0 = 9500
[0][1]	9500 + (0 * 4 + 1 * 2) = 9500 + 0 + 2 = 9502
[1][0]	9500 + (1 * 4 + 0 * 2) = 9500 + 4 + 0 = 9504
[1][1]	9500 + (1 * 4 + 1 * 2) = 9500 + 4 + 2 = 9506
[2][0]	9500 + (2 * 4 + 0 * 2) = 9500 + 8 + 0 = 9508
[2][1]	9500 + (2 * 4 + 1 * 2) = 9500 + 8 + 2 = 9510
[3][0]	9500 + (3 * 4 + 0 * 2) = 9500 + 12 + 0 = 9512
[3][1]	9500 + (3 * 4 + 1 * 2) = 9500 + 12 + 2 = 9514
[4][1]	9500 + (4 * 4 + 0 * 2) = 9500 + 16 + 0 = 9516
[4][2]	9500 + (4 * 4 + 1 * 2) = 9500 + 16 + 2 = 9518

The general formula applied is:

Formula 4.2.

element address = base address
+ row reference * (number columns * bytes/element)
+ column reference * bytes/element

where (for our example): base address = 9500
number columns = 2
bytes/element = 2 (i.e., the elements are of type integer)

The C/C++ code which we would apply (C/C++ Code 4.11) to search for a number in memory corresponds to that which we have already seen:

C/C++ Code 4.12.

```
#include <stdio.h>
void main()
{ int numtable[5][2] = {(1, 1), (4, 8), (9, 81), (16, 256), (25, 625)};        // initialize
  int i, j;                                                  // offset indices
  for (i = 0; i < 5; i++)                                    // set the row parameters
      for (j =0; j < 2; j++)                                 // set the column parameters
          if (numtable[i][j] == 256)                         // 256?
                printf("Stored in numtable[%d][%d]\n",i, j); // print the variable offset
        else if (i == 4  &&  j == 2)                         // if we haven't found it
               printf("“The value does not exist\n");        // dnot found message
  }
```

What happens if we have more than 2 dimensions???

The procedures are the same, although the individual element address calculations become a little more complicated. Although we have trouble envisioning more than 3 dimensions (e.g., a cube which has height, length, and width), storing the individual elements into RAM is not a problem. If we were storing information into a 3-dimensional array, we might make the C/C++ declaration[6]:

C/C++ Code 4.12.

```
double cubetable[3][2][2];
```

? ***How much contiguous memory is required for this array???***

We have a total of 3 * 2 * 2 = 12 elements, each of data type double, which requires 8-bytes of storage. Therefore, we require 12 * 8 = 96 contiguous bytes of storage in RAM.

? ***How would the addresses of the individual elements in array <u>cubetable</u> be calculated???***

In a fashion similar to the manner in which we calculated the addresses for two dimensional arrays. If we assume that our first available block of RAM is at address 82000 (Table 4.10):

Table 4.10.

Array offsets	**Calculated Address**
[0][0][0]	82000 + (0 * 32 + 0 * 16 + 0 * 8) = 82000 + 0 + 0 + 0 = 82000
[0][0][1]	82000 + (0 * 32 + 0 * 16 + 1 * 8) = 82000 + 0 + 0 + 8 = 82008
[0][1][0]	82000 + (0 * 32 + 1 * 16 + 0 * 8) = 82000 + 0 + 16 + 0 = 82016
[0][1][1]	82000 + (0 * 32 + 1 * 16 + 1 * 8) = 82000 + 0 + 16 + 8 = 82024
[1][0][0]	82000 + (1 * 32 + 0 * 16 + 0 * 8) = 82000 + 32 + 0 + 0 = 82032
[1][0][1]	82000 + (1 * 32 + 0 * 16 + 1 * 8) = 82000 + 32 + 0 + 8 = 82040
[1][1][0]	82000 + (1 * 32 + 1 * 16 + 0 * 8) = 82000 + 32 + 16 + 0 = 82048
[1][1][1]	82000 + (1 * 32 + 1 * 16 + 1 * 8) = 82000 + 32 + 16 + 8 = 82056
[2][0][0]	82000 + (2 * 32 + 0 * 16 + 0 * 8) = 82000 + 64 + 0 + 0 = 82064
[2][0][1]	82000 + (2 * 32 + 0 * 16 + 1 * 8) = 82000 + 64 + 0 + 8 = 82072
[2][1][0]	82000 + (2 * 32 + 1 * 16 + 0 * 8) = 82000 + 64 + 16 + 0 = 82080
[2][1][1]	82000 + (2 * 32 + 1 * 16 + 1 * 8) = 82000 + 64 + 16 + 8 = 82088

[6] It is possible to initialize a three dimensional array along with its declaration. However, the syntax tends to be somewhat cumbersome, and confusing to new c programmers.

Notice that the *inner-most* (the last dimension declared) array gets filled first.

Notice also that RAM is allocated in a linear fashion, even though we conceptually view it in a multidimensional fashion. We could, for example, make the declaration:

C/C++ Code 4.12.

```
long manydimensions[5][3][2][6][4][2];
```

Which would be in six-dimensions[7]. We know that the array would require:

4 * 5 * 3 * 2 * 6 * 4 * 2 = 5,670 Bytes of contiguous Storage

We need not go into exactly how to calculate each element's address, other than to say that the concepts applied above are the same.

✓ ***How do we calculate their addresses?***

? ***What happens if we want to mix variable types (for instance, integers and reals) in the same array???***

Can't be done, at least not within the definition of an array. Let's not forget the basic definition of an array:

> An array is a data structure which contains a ***fixed*** number of ***contiguous*** storage elements ***all of the same data type***.

However, as we will see (in Chapter 7), there is a data structure which has been developed for this purpose.

Ways of Declaring Arrays

The c programming language allows for **three** different array declarations:

Array Declaration #1: Automatic Arrays:

Automatic arrays: (local arrays)

- Defined within the function
- Local to the function in which declared (i.e., references to the array from other arrays cannot be made).
- Exist only during the existence of the function
- **Not** initialized

[7] If you can envision something in six dimensions, you are wasting your time here. Go home: You are a genius!

These are the most common types of declarations. For example, consider the following sections of c code (4.14):

C/C++ Code 4.15.

```
int anadditionalfunction (int *array2);          // function prototype
void main()
{ int array1[10];                                // array1 is an automatic array
   . . .
   anadditionalfunction(array1);                 // pass array1 to array2
   . . .   }
int anadditionalfunction (int *array2)           // the external function
 { . . .   }
```

In this example, both *array1* and *array2* are automatic (local) variables. Variable *array1* cannot be referred to from function *anadditionalfunction* (even though we passed the address of *array1* to *array2*, which duplicated the values into *array2*). Similarly, we cannot refer to *array1* from function *anadditionalfunction*. Because we are passing the address of the array (*array1* in function main and *array2* in function *anadditionalfunction*) we are calling by reference.

? ***How would this appear in RAM???***

Assume that the base address of the array (in function main, *array1*) is 78567 (REMEMBER: *array1* == &*array1*[0] == 78567). That means that the array takes up locations 78567 through 78586 (since we require 2 * 10 = 20 contiguous bytes of storage). There is one additional address we must consider in the program: *array2*, which is a location in RAM which will contain an address at which we expect to find an integer (on 2-bytes). Let's assume that *array2* is assigned the address 78591 (through 78594). If that is true, then the relevant portion of RAM (AFTER the call to *anadditionalfunction*) would appear as (Figure 4.10):

Figure 4.10.

78567 & 78568 (*array1*[0])	78569 & 78570 (*array1*[1])	78571 & 78572 (*array1*[2])
-------	**-------**	**-------**
78573 & 78574 (*array1*[3])	78575 & 78576 (*array1*[4])	78577 & 78578 (*array1*[5])
-------	**-------**	**-------**
78579 & 78580 (*array1*[6])	78581 & 78582 (*array1*[7])	78583 & 78584 (*array1*[9])
-------	**-------**	**-------**
78585 & 78586 (*array1*[9])	78587 78590	
-------	---	---
78591 through 78594 (*array2*)		
78567		

Notice that the only real piece of information that we pass (to function *anadditionalfunction*) is the address of *array1*. However, as we have stated repeatedly, is that ALL we need to know is:

1. An address in RAM
2. What type of data is stored at that address

Once we know that information, we can determine the information stored there (just as we have been emphasizing all along).

Array Declaration #2: External Arrays:

External arrays: (global arrays)

- Defined outside the function[8]
- Known to *ALL* functions
- Do *NOT* expire (as long as the program is running)
- Initialized when declared (set to 0 (zero) if numeric; **null** if character)

For example, consider the C/C++ Code given in 4.15::

C/C++ Code 4.16.

```
int array[10];                      // array is a GLOBAL array (available to ALL
                                    // functions in the program)
int anadditionalfunction ();        // function prototype
void main()
{  . . .
   anadditionalfunction();          // call to the external function
   . . .
   anadditionalfunction();
   . . .    }
int anadditionalfunction ()         // the external function
 {   . . .   }
```

In this case variable *array* could be reference from any function (including function **main**). We could refer to variable array from either main or from function *anadditionalfunction*, and access any element in the array (variable *array*). Note that declaring additional variable from within either of the functions (functions **main** or *anadditionalfunction*, as we did with

[8] The c programming language allows for the declaration *extern int array[10];* inside a function. This essentially has the same effect as declaring the array outside of a function.

array1 and *array2* in the previous example) does not mean that we could make reference to them from the other functions; they would still remain as automatic (local) variables.

Array Declaration #3: Static Arrays:

Static arrays:
them

- ■ Like automatic arrays, local to the function which declared
- ■ Like external arrays, retain values between calls
- ■ Like external arrays, initialized at the time of declaration (set to 0 (zero) if numeric; null if character)

? ***What's the difference between static and automatic arrays???***

If we had made the declaration (in function *anadditionalfunction*) **int** *array2*[10]; (automatic) instead of **static int** *array2*[10]; any values we established for *array2* after the first pass would have disappeared when we made the second call to the function. Notice, once again, that since *array1* is an automatic array (i.e., local to function **main** only) we could not make reference to it from function *anadditionalfunction*.

✓ ***How do we declare arrays ?***

Summary

In this chapter, we introduced the concept of numeric arrays. It is somewhat deceptive to differentiate numeric arrays from our next chapter (strings) since ALL data stored in RAM is numeric (including strings), but we have done so because there are a few additional considerations to keep in mind when dealing with strings.

As mentioned earlier, arrays (including, as we will see, strings) are one of the most commonly used, and powerful, abstract data structures available to us. They allow for quick calculation of an address, and therefore allow us to carry out our prime directive:

Give me an address and tell me what type of data is stored there, and I will tell you the value of that data.

Arrays are still constraining, however. There are two major constraints:

1. ALL elements in an array MUST be of the same data type
2. ALL elements in an array MUST be stored in contiguous locations in RAM

In Chapter 7 we will ease the restrictions on constraint number 1. In Section 3, we will ease the restrictions on constraint number 2.

Chapter Terminology: Be able to fully describe these terms

& (before variable)
* (before variable)
Address
Address Calculation
Automatic Array
Base Address
Call by Reference
Call by Value
Column Offset
Contiguous Storage
External Array
Matrix
Multi-dimensional Array
Offset
Pointer Address
Pointer Contents
Row Offset
Static Array
Subscript
Vector

Review Questions

1. What is the main advantage of an array?
2. What is the main disadvantage of an array?
3. What are the major differences between automatic, static, and external arrays?
4. When I entered the c code:

```
int myarray[7], *mypointer;
mypointer = &myarray[3];
printf("%lu %lu",myarray, mypointer);
```

I received the output: **56871 56889**
Looking at RAM, I found:

56868	56869	56870	56871
00001100	00000001	00000000	00000001
56872	56873	56874	56875
00010011	00000100	00000000	00001000
56876	56877	56878	56879
00000000	00001001	00000000	00000001
56880	56881	56882	56883
00001001	00010000	00000001	00000000
56884	56885	56886	56887
11010111	00001010	01110101	11111100
56888	56889	56890	56891
11010110	00000000	00000000	11011110
56892	56893	56894	56895
00101101	00011001	00000010	01110001

a. What would be the output of the statement: printf("%lu", &*myarray*[2]);
b. What would be the output of the statement: printf("%d", *myarray*[7]);
c. What would be the output of the statement: printf("%d", **mypointer*);
d. What would be the output of the statement: printf("%lu", --*mypointer*);
e. <u>Disregarding</u> the previous question, What would be the output of the statement: printf("%d", **mypointer*); <u>***AFTER***</u> we issue the statement: ++*mypointer*;

Review Question Answers (NOTE: checking the answers before you have tried to answer the questions doesn't help you at all)

1. What is the main advantage of an array?

   The address of any element in the array can be quickly calculated.

2. What is the main disadvantage of an array?

   We must allocate <u>CONTIGUOUS</u> storage in RAM and all of the elements <u>MUST</u> be of the same data type

3. What are the major differences between automatic, static, and external arrays?

Characteristic\Array	Automatic	Static	External
Defined inside function?	YES	YES	NO
Local to function ?	YES	YES	NO
Values Lost after Return?	YES	NO	NO
Initialized when Declared?	NO	YES	YES

4.a. What would be the output of the statement: printf("%lu", &*myarray*[2]); ??

Since the base address of the array *myarray* is 56871, then &myarray[2] must be:

56871 + 2*2 = 56871 + 4 = <u>56875</u> *(since myarray is an integer array).*

4.b. What would be the output of the statement: printf("%d", *myarray*[7]);

The address of myarray[7] is: 56871 + 2*7 = 56871 + 14 = 56885.
At address 56885 we find: 0000101001110101 (on 16-bits), which equates to:

$2^{11} + 2^9 + 2^6 + 2^5 + 2^4 + 2^2 + 2^0 = 2{,}048 + 512 + 64 + 32 + 16 + 4 + 1 = \underline{2{,}677}$

4.c. What would be the output of the statement: printf("%d", **mypointer*);

Since *mypointer* = &*myarray*[3]; <u>AND</u> the base address of the array *myarray* is 56871, location *mypointer* CONTAINS the address: 56871 + 3*2 = 56871 + 6 = 56877.

(Notice also that if we went to location 56889 (location *mypointer*) we find: 00000000000000001101111000101101 (on 4-bytes or 36-bits) which equates to 56877).

If we REDIRECT to location 56877 we find 1111110011010110 (on 16-bits). Since the value is negative, we must compliment (assume a two's compliment machine):

```
1111110011010110 =>  0000001100101001   (one's compliment)
                    +               1
                           1100101010   (two's compliment)
```

= -($2^9 + 2^8 + 2^5 + 2^3 + 2^1$) = -(512 + 256 + 32 + 8 + 2) = <u>- 810</u>

4.d. What would be the output of the statement: printf("%lu", --*mypointer*);

Because (from above) we know that location mypointer contains the address 56877 AND mypointer is a pointer to the data type int, then DECREMENTING the pointer means that we decrease the value contained at location mypointer by 2-bytes. The new contents of mypointer would thus be: 56877 – 2 = 56875 (the address of myarray[2]).

4.e. <u>Disregarding</u> the previous question, What would be the output of the statement: printf("%d", **mypointer*); ***<u>AFTER</u>*** we issue the statement: ++*mypointer*;

Given that the original contents of mypointer was 56877 incrementing changes the address contained to 56877 + 2 = 56879 (the address of myarray[4]). At that address we find:

0000000100001001 (on 16-bits), which equates to: $2^8 + 2^3 + 2^0$ = 256 + 8 + 1 = <u>265</u>