LESSON 8

ARRAYS

This lesson discusses an aggregate data type common to programmers, arrays. Arrays are declared just like a fundamental data type but the size of the array is added by use of subscripts. Arrays of all types are possible, including arrays of arrays. Strings are just arrays of characters, but they are sufficiently important to be treated separately a lesson by themselves. A typical array declaration allocates memory starting from a base address. The array name is in effect a pointer constant to this base address. In this lesson this relationship of array to address will be explained. Another key point is how to pass arrays as parameters to functions. A number of examples will illustrate these concepts. Programs often use homogeneous data, i.e. data that is the same type. For example, if you want to manipulate some grades, you might declare int grade0, gradel, grade2. However, if the number of grades is large, it is tedious to represent and manipulate the data by means of a group unique identifiers. Instead, an array, which is a derived type, can be used. Individual elements of the array are accessed using a subscript, also called an index.

The brackets [ ] are used to contain the subscripts of an array. To use grade[0], grade[1], and grade[2] in a program, you declare

```
int grade[3];
```

where the integer 3 in the declaration represents the size of the array, or the number of elements in the array. The indexing of array elements always starts at 0. This is one of the characteristic features of the C language. So there are three elements and the index goes from 0, 1 and 2. Notice that the dimension 3 is not used as an index.

A one-dimensional array declaration is a type followed by an identifier with a bracketed constant integral expression. The value of the constant expression, which must be positive, is the size of the array; it specifies the number of elements in the array, starting at 0. To store the elements in the array, the compiler assigns an appropriate amount of memory starting from a base address.

A small example program illustrates how to:

- fill an array
- prints out array values
- sum the elements of the array.

```
#define   SIZE 5
main(    )
{
  int a[SIZE]; /* space for a[0],...a [4] is allocated */
  int i, sum = 0;
  for (i = 0;   i < SIZE;   ++i  )    /* fill the array */
     a[i]   =   i * i;
 for (i = 0;   i < SIZE;   ++i  )   /* print values */
   printf("a[%d] = %d ", i, a[i]);
 for (i = 0; i   < SIZE;   ++i  )/* sum elements */
  sum  += a[i];
 printf("\nsum  =  %d\n", sum);
}
```

The output of the program is:

```
a[0] = 0 a[1] = 1 a[2] = 4 a[3] = 9 a[4] = 16
sum = 30
```

The a array required memory to store five integer values. Thus if a [0] is stored at location f1000, then on a system needing 4 bytes for an int, the remaining array elements are successively stored at locations f1004, f1008, f1012, and f1016.

It is considered good programming practice to define the size of an array as a symbolic constant. In this example SIZE. Since much of the code may depend on this value, it is convenient to be able to change a single #define line to process different size arrays. The three parts of the for statement provide a concise notation for dealing with array computations.

INITIALIZATION

Arrays may be of storage class automatic, external, or static, but not register. External and static arrays can be initialized using an array initializer. An example is:

```
static float x[7] = {-1.1, 0.2, 33.0, 4.4, 5.05, 0.0, 7.7};
```

This initializes x[0] to-1.1, x[1] to 0.2, and so forth, until x[6] = 7.7. When a list of initializers is shorter than the number of array elements to be initialized, the remaining elements are initialized to zero. If an external or static array is not initialized, then the C standard is to initialize all elements to zero automatically.

In contrast, automatic arrays cannot be initialzed under K&R compilers. But ANSI C allows the use of the initialization form just described on all storage class arrays (register arrays are still not permitted). ANSI C while allowing automatic arrays to be initialized on the declaration line does not initialize all elements to zero automatically. The elements of an automatic array always start with "garbage" values, that is, with the arbitrary values that happen to be in memory when the array is allocated.

If an array is declared without a dimension size and is initialized to a series of values, the array is implicitly given the size of the number of initializers. Example:

```
int a[] = {33, 44, 55, 66};    and    int a[4] = {33, 44, 55, 66}:
```

are equivalent external declarations.

SUBSCRIPTING

Given that a declaration of the form

```
int i, a[SIZE];
```

has been made. Then you can write a[i] to access an element of the array. Here i is the index not the dimension variable. You may also write a[exp], where exp is a positive integral expression used to access a single element of the array. Again exp is a subscript, or index, of the array a. The value of a subscript must lie in the range 0 to (SIZE - 1).

An array subscript value outside this range will cause a run-time error. When this happens, the condition is called "overrunning the bounds of the array" or "subscript out of bounds." It is a common programming error. The effect of the error is system dependent and can be quite confusing. One frequent result is that the value of some unrelated variable will be returned or modified. Thus you, the programmer, must ensure that all subscripts stay within bounds.

In previous lessons examples showed how to count digits, letters, etc. By using an array, you can easily count the occurrence of each uppercase letter separately, using more concise code. Here a program that does that.

```
/* Count each uppercase letter separately. */

#include <stdio.h>
#include <ctype.h>
main(   )
{
  int  c, i, letter [26];
  for (i = 0; i < 26; ++i)                        /* initialize array to zero*/
     letter [i] = 0;
 while ((c= getchar( )) != EOF)                   /*count letters*/
       if (isupper(c))
       ++letter[  c - 'A'  ];
       for (i = 0; i < 26; ++i) {                 /* print results */
```

```
                if ( i % 6  == 0)                        /* modulo operator  */
                     printf("\n") ;
                printf(" %c:%9d", 'A'+i, letter [i]) ;
              }/* end for loop */
        printf("\n\n") ;
        }/* end main */
```

Compile the program into an executable "count" and then give the command:

```
        count < input_ex.txt
```

to read the input_ex.txt and the following appears on the screen:

```
        A:   25 B:   52 C;  219 D:   14 E:   121 F:   13
        G:   9 H:   131:    121 J:    1  K:    1  L   39
        M:   25 N:   440:   38 P:  2430:   1 R:   37
        S:   73 T:   96 U:    7V;    3  W:    17 X:   9
        Y-   117-   27
```

PROGRAM ANALYSIS

The count for each of the 26 capital letters will be stored in the array letter.

```
        int  c, i, letter [26];
```

It is important to remember that the elements of the array are letter[0], letter[1], . . ., letter[25]. Forgetting that array subscripting starts at 0 causes many beginning programmer errors. The variable i will be used as a subscript or index for the array letter.

```
        for (i = 0; i < 26; ++i)                        /* initialize array to zero*/
             letter [i] = 0;
```

Automatic arrays must be explicitly initialized. This for loop follows a standard pattern for processing all the elements of an array. The subscripted variable letter is initialized to 0. The termination test is to see if the upper bound (26) is exceeded.

```
        while ((c= getchar( )) != EOF)                  /*count letters*/
                if (isupper(c))
                ++letter[  c - 'A'  ];
```

The library function getchar( ) is used repeatedly to read a character in the Input stream and assign its value to c. The while loop is exited when the end-of-file sentinel is detected.

The macro isupper( ) from < ctype.h > is used to test whether c is an uppercase letter. If it is, then an appropriate element of the array letter is incremented.

```
++letter[  c - 'A'  ];
```

On an ASCII compiler the expression c - 'A' has the value 0 if c has the value 'A', the value 1 if c has the value 'B' , and so forth.

Thus the uppercase letter value of c is mapped into the range of values 0 to 25. Because brackets have higher precedence than ++ an equivalent statement is

```
+ +(letter[c - 'A']);
```

Thus you see, that letter[0] is incremented if c has the value 'A', letter[1] is incremented if c has the value 'B', and so forth.

```
for (i = 0; i < 26; ++i) {                    /* print results */
        if ( i % 6  == 0)                          /* modulo operator  */
            printf("\n") ;
        printf(" %c:%9d", 'A' + i, letter [i]) ;
      }/* end for loop */
```

The same for loop is used to proces; the array letter. Every sixth time through the loop a newline is printed. Here is an example of the modulo operator. As i runs from 0 to 25 the expression 'A' + i is used to print A through Z with each letter followed by a colon and the appropriate count.

An array name by itself is an address, or pointer value. In C pointers and arrays are almost identical in terms of how they are used to access memory. However, there are differences, and these differences are subtle and important. A pointer is a variable that takes addresses as values. An array name is a particular fixed address that can be thought of as a constant pointer.

When an array is declared, the compiler must allocate a base address and a sufficient amount of storage to contain all the elements of the array. The base address of the array is the initial location in memory where the array is stored; it is the address of the first element (index 0) of the array.

Suppose you make the declaration:

```
#define N       100
int a[N], *p;
p = a;      /* p is initialized to the base address of the array a*/
sum = 0;
for (p = a; p <&a[N]; ++p)
    sum   += *p;
```

In this loop, p is initialized to the base address of the array a. The successive values of p are the same as &a[1], &a[2],...&a(N]. In general if i is a variable of type int then p + i is the ith offset from the address p. In a similar manner if p = a then a + i is a similar offset.

This is shown in the following.

```
sum =0;
for ( i=0;  i < N;  ++i )
   sum   +=   *(a + i);
```

Just as the address *(a + i) is equivalent to a[i], so the expression *(p + i) is equivalent to p[i].

```
sum =0;
for(i = 0: i < N; ++i)
    sum += p[i];
```

Although in many ways arrays and pointers can be treated alike, there is one essential difference. Because the array a is a constant pointer and not a variable, expressions such as

```
a=p             ++a             a  +=  2
```

are illegal because you cannot change the address of a.

POINTER ARITHMETIC AND ELEMENT SIZE

Pointer arithmetic is one of the powerful features of C. If the variable p is a pointer to a particular type, then the expression p + 1 yields the correct machine address for storing or accessing the next variable of that type. In a similar fashion pointer expressions such as p + i and ++p and p += i all make sense to the C compiler and are executable. If p and q are both pointing to elements of an array, then p - q yields the int value representing the number of array elements between p and q.

Even though pointer expressions and arithmetic expressions have a similar appearance, there is a critical difference in interpretation between the two types of expressions. The following code illustrates the difference:

```
double a[2], *p, *q;
p  =  &a[0];
q = p+1;                                /* Equivalent to q = &a[1];  */
printf("%d\n", q - p):                  /* The value 1 is printed */
printf("%d\n", (int) q - (int) p);      /* The value 8 is printed */
```

What is printed by the last statement is machine dependent. On some computers, a double is stored in 8 bytes. Hence the difference of the two machine addresses interpreted as integers 8. You'll have to try this on your computer.

PASSING ARRAYS TO FUNCTIONS

A function definition is called a prototype. In the function definition, a formal parameter that is declared as an array is actually a pointer. When an array is being passed, its base address is passed call-by-value. The array elements themselves are not copied. As a notational convenience, the compiler allows array bracket notation to be used in declaring pointers as parameters.

This notation reminds the programmer and other readers of the code that the function should be called with an array. To illustrate this you write a function that sums the elements of an array of type int.

```
int sum (int a[ ], int n)
/* n is the size of the array */
{
   int i, s = 0;
   for(i = 0; i < n; ++i)
       s += a[i];
   return (s);
}
```

As part of the header of a function definition

```
int sum (int a[ ], int n);
```

the declaration

```
int  a[];   is equivalent to   int*a;
```

On the other hand, as declarations within the body of a function they are not equivalent.

The first will create a constant pointer (and no storage), whereas the second will create a pointer variable.