Arrays - Chapter Summary

1 C Program Structure

A scalar data type, such as int, float or char, contains a single item of data: a value, of whatever type.

An aggregate data type contains more than one related data item. The three major aggregate data types in C are the array, the structure and the union. The data stored in these three types is organized according to clear rules defined in the language, and they are therefore sometimes referred to as structured data types.

2 Arrays

An array is a collection of related data items all of the same type. The items of data in an array, known as the elements of the array, are stored in adjacent locations in memory.

Arrays need to be allocated appropriate memory storage by the compiler, and therefore, like variables, must be declared before use. The general form of an array declaration is:

type name[size]

In this declaration, size refers to the number of elements in the array, and it must be an integer constant. The declaration therefore means that the array variable name contains size number of elements, each of type type.

The type declaration indicates how many bytes will be needed to store each element, and the variable name is a symbolic representation of the address of the first element in the array. Once this information has been declared, it is possible in later operations to access any element in the array.

The following are all possible array declarations:

```
long planets[9];          /* an array of 9 long ints */
double radii[23];         /* an array of 23 doubles */
int index[MAX_RECORDS];   /* an array of MAX_RECORDS ints */
```

In the last example, MAX_RECORDS would be an integer constant given in a #define statement at the top of the program. This and other preprocessor commands are covered in a later chapter.

3 Initialising an Array

The data held in an array will have no useful initial values, and needs to be initialised before use. The whole array can be initialised as part of the original declaration statement. All the elements to be assigned should be in a list, separated by commas, within braces { and }:

```
short form[3] = { 7, 6, 13 };
char punctuation[5] = { '!', ',', '.', '?', '"' };
double length[] = { 3.45, 6.7, 2.987, 1.98 };
```

Note that in the last example above, the size of the array is not explicitly declared, but is implicitly declared in the assignment list. Four floating-point numbers are assigned to the array, so its size must be four elements.

4 Accessing the Elements of an Array

An element of an array may be accessed by specifying the name of the array and the element number, thus:

name[expression]

The name is the name of the array variable, and the expression, which specifies the element number, must evaluate to an integer value.

Note that the first element in an array is always numbered zero. This means that in an array of six elements, for instance, the elements are numbered 0, 1, 2, 3, 4 and 5.

```
int main(void)
{
     char vowels[5] = { 'a', 'e', 'i', 'o', 'u' };
     char letter;

     letter = vowels[3];
     vowels[2] = 'I';

     printf("\nThe variable letter has the value: %c.",
             letter);

     printf("\nThe third element of vowels[] is: %c.",
              vowels[2]);
```

```
        return 0;
}
```

When the above example is compiled and run, it will print out:

The variable letter has the value: o.

The third element of vowels[] is: I

The program has assigned the variable letter the value of element 3 in the array called vowels, which is the fourth element, 'o'. Similarly, element number 2 in the array, the third element, has been assigned a new value, a capital 'I'.

Note that there is no bounds checking for arrays in C. It is the programmer's responsibility not to attempt to assign more elements to an array than have been allowed for in the array declaration. It is quite possible to do this, and the additional elements will be assigned memory in adjacent locations as if they were legal elements of the array, but they will in practice be spilling over the bounds of the array and over-writing space assigned to other data, which will therefore be corrupted and lost. The program may run and produce incorrect output, or in some instances it may simply crash.

5 Array Operations

C does not support whole-array operations. You cannot write anything like records[]++; and expect each of the elements in the array called records to be incremented by one. If you attempt to compile such a statement, you simply generate an error message.

There are ways of achieving the equivalent of whole-array operations, however, either through the use of pointers, as will be explained in later chapters, or through the use of for loops.

Although whole arrays may not be operated on, array elements may be treated exactly like scalar data, and for loops provide a convenient mechanism for handling multiple repetitive operations.

The for loop in the following program fragment initialises all the elements of an array to zero:

```
int records[10];
int count;

for (count=0; count < 10; count++)
     records[count] = 0;
```

In the next example, the for loop assigns the value of each element in the array records to the corresponding element in the array new_copy:

```
int records[10], new_copy[10];
int count;

for (count=0; count < 10; count++)
     new_copy[count] = records[count];
```

In the final example, the first for loop requests and assigns values to each of the ten array elements in records in turn, while the second for loop prints back the newly-assigned values of each of the ten elements in a comma-separated list on a single line.

```
int records[10];
int count;

for (count=0; count < 10; count++)
{
     printf("\nWhat value for element no.%d?\n",
               count);
     scanf("%d%*c", &records[count]);
}

for (count=0; count < 10; count++)
{
     printf(" %d", records[count]);
     if (count < 9)
          printf(",");
}
```

6. Array Names and Functions

6.1 Indexing

When an array name is used as an argument to a function, the value that is passed to the function is the address of the beginning of the array.

The address of any element in the array may then be calculated by adding the element's displacement from the start of the array to the array's starting address; a method known as indexing. The displacement is found by multiplying the number of bytes used to store data of the type contained in the array by the number of the element, which is one reason why it is important that the first element in an array is numbered zero. The calculation might be expressed thus:

```
element_address =
            array_address + (bytes_per_element * element_no);
```

6.2 Call by Reference

By using this method of indexing, a function can access and alter any element in an array that is passed to it as an argument. This method of sharing data between functions is referred to as call by reference, and is a rare exception to the normal practice in C of calling by value. When a scalar variable, such as an element of an array, is used as an argument to a function, for instance, it is only the value of the variable that is actually passed across to the function being called.

Care must be taken when an array is used as an argument to a function, because the function is effectively given the power to alter the contents of an array used in the function that called it.

6.3 Declaring an Array as a Formal Parameter

There are three ways to declare an array as a formal parameter to a function: as an array of a given number of elements, as an array of unknown size, or as a pointer. Pointers and the indirection operator * will be explained in later chapters, but the following are all possible declarations for the formal parameters to a function called increase, which returns a long integer value and which includes an array of twelve long integers called debts as one of its arguments:

```
long increase(double inflation, long debts[12])
long increase(double inflation, long debts[])
long increase(double inflation, long * debts)
```

The second form of declaration is possible (and preferable to the first), because there is no bounds checking on arrays in C, and the program will index from the starting address for the array, whether or not this takes it beyond the actual limits of the array. It is left to the programmer to ensure that the array bounds are not exceeded.

7 Strings

In programming terms, a string is a series of logically-connected characters whose significance depends on all the characters being present and in the correct order. The most common examples of strings are normal English words, phrases and sentences.

7.1 The String Convention in C

Unlike other languages, such as BASIC, C has no language support for strings, and does not have a string type of data item.

It will be seen from the above definition, however, that an array of char type variables would offer some of the features required for a string, namely a series of characters held in a fixed order. The lack of bounds checking for arrays means that there is no automatic way of determining whether all the characters are present, however.

Strings are therefore implemented in C by overlaying a convention on char arrays; the convention being that the last element in a string must be the null character '\0'. The string will always be terminated when the null character, the null terminator, is encountered.

One practical effect of this convention is that the number of elements in a char array storing a string is always one greater than the total number of characters, including spaces, in the string itself.

7.2 Convention Support: `strcmp` and `strcpy`

The string convention is supported by library functions such as strcmp and strcpy, which take strings as arguments and perform whole-string operations.

The function strcmp, for instance, takes two strings as arguments and compares the first string to the second, returning the value zero if the strings are identical.

Note that strcmp returns a value equal to the difference between the character codes of the first non-identical characters in the two strings. This value is positive if the first non-identical character in the first string has a higher character code value than its counterpart in the second string, and negative if it has a lower value. Since character sets assign sequential character codes to letters of the alphabet, this means that strcmp can be used for sorting lists of words and names into alphabet order: if strcmp returns a positive value, the first string comes after the second string in an alphabetical list.

The following program fragment prints apple and banana in alphabetical order:

```
char fruit1[7] = "banana";
char fruit2[7] = "apple";

if (strcmp(fruit2, fruit1) > 0)
     printf("%s, %s", fruit1, fruit2);
else
     printf("%s, %s", fruit2, fruit1);
```

The function strcpy also takes two strings as its arguments and copies the contents of the second string into the first.

The standard library also contains the functions gets and puts, which input a string from the user and output a string to the screen, or standard device, respectively.

In the case of gets, the user types in the string, and the string is inputted and the null terminator added when the return key is pressed.

7.3 String Literals

C also supports string literals, which are the contents of a string excluding the null terminator; that is to say, the actual words that appear outputted to screen or printer.

String literals are enclosed in double quotes, may be assigned to character arrays and may be used in functions that can take strings as arguments, as in the following two program statements:

```
char description[] = "A beautiful view of Cirencester.";
if (strcmp(password, "Open Sesame!")==0)
     open_cave();
```

7.4 Strings in `printf` and `scanf`

Both printf and scanf support the string convention through the %s conversion specification, as illustrated in the following program:

```
int main(void)
{
     char name[30];

     printf("\n%s\n", "What is your name?");
     scanf("%s", name);
     printf("\nHello, %s!", name);

     return 0;
}
```

Note that in the scanf statement in the above example, the variable name was not preceded by the address operator & (although ISO C allows you to use it). In order to be assigned a string, the variable name must already be an address. This is indeed the case, since name was declared as an array of char variables and will therefore have been assigned the address of the beginning of the array.

8 Multidimensional Arrays

C does not specify any limits for the number of elements that may be held in an array. The only practical constraint is the amount of memory that is available.

C also allows multidimensional arrays, that is to say, arrays of arrays. Again, there is no defined limit to the number of dimensions possible. The general form for the declaration of a multidimensional array of n dimensions is:

```
type array_name[dim1size][dim2size]...   ...[dimnsize];
```

A two-dimensional static array of exam marks for 100 students examined in 5 subjects would be declared:

```
int results[100][5];
```

This statement would be more easily readable, however, if symbolic constants, defined by #define statements at the top of the program, were used instead of numbers for the sizes of the arrays:

```
int results[STUDENTS][EXAM_MARKS];
```

The mark student number 65 achieved in subject 3 (starting from 0) could be referred to as results[65][3]. Remember, however, that the students are numbered from 0 to 99, and the exam marks from 0 to 4.

The initialisation statement for the array would be written like this:

```
int results[STUDENTS][EXAM_MARKS] =
{
     { 60, 45, 76, 34, 87 },
     { 56, 67, 55, 64, 72 },
     ..................
     { 76, 54, 23, 88, 44 }
}
```

There would be 100 lists of five exam marks each. Each list of five comma-separated marks would be enclosed in braces { and }, each set of braces would be separated from the others by a comma, and the whole list of 100 sets of marks would be enclosed in opening and closing braces. Note that there is no comma after the last exam mark in each list, nor is there a comma after the very last set of exam marks.