Pointers and Arrays - Chapter Summary

1 What is an Array?

An array is a collection of data items of the same type held in adjacent locations in memory. These items are referred to as the elements of the array. The amount of memory allocated to each element depends on the type of the data being stored.

One way of describing an array and accessing the data it holds is to give the array a symbolic name and declare the type and number of elements it contains. This is the familiar notation:

```
type arrayname[number]
```

In C a data item's identifier is used by the programmer as a representation of the address where the value of the data is stored. The name of the array represents the address of the start of the array; that is, the address of the first element in the array, `arrayname[0]`.

Given that the type of the elements in the array determines the number of bytes allocated to each, the address of an element could be calculated by adding its offset to the starting address of the array, where the offset would be given by the formula:

```
offset = index * sizeof(type)
```

Knowing the address of an element would then allow access to the value of that element.

This is essentially the mechanism used for accessing arrays in C, but the index notation enables the programmer to express these ideas more succinctly. As has been shown, the value of an element of an array may be represented by the expression `arrayname[index]`, and its address by the expression `&arrayname[index]`.

2 Pointers and Arrays

Pointers are always pointers to a specific type of data: a pointer to `int`, for example, or a pointer to `float`. Remembering that a variable type in C is associated with a fixed amount of memory storage, a pointer may be seen to convey two pieces of information: the address of the data at which it points and the number of bytes that are allocated to storing variables of that type.

It may be seen from the above what a close relationship exists between pointers and arrays; both a pointer and an array name represent an address associated with a unit of storage. In fact, any operation that can be effected by array indexing can also be effected with pointers. They are equivalent mechanisms, and since C compilers actually use pointers to implement array indexing anyway, pointers offer a more direct and efficient form of code to achieve the same results.

3 Indexing with Pointers

The following program uses two methods of accessing the elements of an array; the first using an index and the second using a pointer.

```
#include <stdio.h>

int main(void)
{
     int array[6] = { 9, 14, 5, 6, 7, 32 };

     int offset;
     int * ptr;

     ptr = array;   /* same as ptr = &array[0]; */
     offset = 5;    /* offset of the sixth element */

     printf("\nThird element in the array is %d.",
               array[2]);

     printf("\nSixth element in the array is %d.",
               *(ptr+offset));

     return 0;
}
```

When the program is compiled and run, it prints out:

```
     Third element in the array is 5.
     Sixth element in the array is 32.
```

The second `printf` statement uses a pointer and pointer arithmetic to find the address of the sixth element in the array, and then uses the contents of operator to find the value stored at this address.

Note that the statement assigning the starting address of the array to the pointer uses the actual name of the array to represent that address. This is possible because an array name is itself a pointer to the starting address of the array.

On systems that allocate 2 bytes of storage to `int` data types, the actual offset from the beginning of the array of the sixth element of an array of integers would be `10` bytes. So, on such a system, the expression `(ptr+offset)`, where `ptr` points to the beginning of the array and `offset` has the value `5`, would actually have the value:

```
     array_start_address + (sizeof(int) * 5)
```

or

```
     array_start_address + 10
```

However, the program simply assigned the value 5 to the variable `offset`, which is the same value as the index would have for the sixth element in the array.

Clearly, in arithmetic involving pointers, C implicitly and automatically scales numbers according to the size in storage of the data object that is pointed to. This is so whether the object is a scalar or an aggregate type.

4 Further Examples of Pointer Arithmetic

Any expression that can be written in terms of an array and an index can also be written in the form of a pointer to an array with an offset.

Given `ptr =&array[0]` then :

```
array[3] == *(array+3) == ptr[3] == *(ptr+3)

&array[3] == &*(array+3) == (array+3) == (ptr+3)
```

5 Differences between Pointers and Arrays

Declaring a pointer allocates memory to store the address that is pointed to. It does not in itself cause memory to be allocated for any object that might be stored at the address to which it points.

Only a full declaration of an array name, type and number of elements, allocates storage for an array. The number of elements may be declared explicitly, as an index, or implicitly, as the number of elements given in an initialisation statement.

An array name is a constant. It is a symbolic representation of an address in memory, but it will not be allocated a different address while the program is running.

A pointer, on the other hand, is usually a variable and may be assigned many addresses during the execution of a program. It always stores the values of these addresses at the same location in memory, of course, because the address of that location is also, like the array name, a constant.

Like most variables, a pointer variable may be assigned values, and those values may be acted on by operators, including unary operators such as the increment and decrement operators.

Being a constant, an array name cannot itself be assigned values, nor can it be acted on by unary arithmetic operators such as the increment and decrement operators, because it stands for an address that is fixed for the duration of the program's execution.

6 Address Expressions and Indirection

The 'contents of' operator * may be used with any expression that yields an address, to give the value stored at that address.

As a variable, a pointer may be acted on by operators and form part of an expression.

When the contents of operator acts on an address that is represented by an expression involving a pointer, the pointer expression is first evaluated according to the rules of operator precedence to establish the value of the address pointed at. Any numbers in the pointer expression will be scaled according to the size of the object pointed at by the pointer. Then the whole expression is evaluated to yield the value stored at the address given by the pointer expression.

Accordingly, the expression `*(pointer - 9)` gives the value stored at the address nine storage units lower than the address pointed to by `pointer`. If `pointer` is a pointer to `float`, for instance, and the system being used assigns 4 bytes of storage to `float` types, then the 9 in the pointer expression will be scaled up to represent 36 bytes. The whole expression will therefore yield the value stored at the address 36 bytes below that pointed to by `pointer`.

Again, the expression `*(++pointer)` refers to the value stored at the address one above that originally pointed to by `pointer`. In this case, the value of pointer is incremented by one before the contents of operator acts. Conversely, `*(pointer++)` yields the value stored at the address pointed to before the value of `pointer` is incremented.

7 Pointer to Arrays and Loops

Just as loops can be written using the index notation to carry out whole-array operations, so they can also be written using pointers and indirection. The main practical difference between the two methods is that whereas using the index notation an end condition for the loop can be easily specified by comparing the loop counter to the number of elements in the array, when using pointers, the loop's end test usually requires a pointer expression that represents the address of the end of the array.

The following two functions both use loops to initialise the elements of an array of integers to zero. The `for` loop in `init1_array` uses an array name and an index, and the `while` loop in `init2_array` uses a pointer to an array and indirection:

```
void init1_array(int array[], size_t no_elements)
{
     size_t count;
     for (count = 0; count < no_elements; count++)
     {
          array[count] = 0;
```

```
        }
    }

    void init2_array(int * array, int no_elements)
    {
        int * array_end = array + no_elements;
        while(array < array_end)
        {
            array++ = 0;
        }
    }
```

The notes in this chapter illustrate other loop constructions for carrying out whole-array operations using pointers.

8 Passing Arrays as Arguments to Functions

When an array name is used as an argument to a function, it is the starting address of the array that is passed to the function, and this, contrary to the general practice in C, is an example of call by reference. Either index or pointer notation can be applied within the body of a function that takes an array as one of its arguments, since the name of an array is itself a pointer to the start of the array. The following are index and pointer versions of a function to print out a comma-separated list of the elements in an array of integers:

```
void print_elements(const int array[], size_t n)
{
    size_t i;

    putchar('\n');
    for (i = 0; i < n; i++)
    {   /* no comma after last element */
        if (count != i - 1)
            printf("%d, ", array[count]);
        else
            printf("%d", array[count]);

    }
}

void print_elements(const int * array, size_t n)
{
    size_t i;
    putchar('\n');
```

```
     for (i = 0; i < n; i++)
     {
          if (count != i - 1)
               printf("%d, ", *(array+count));
          else
               printf("%d", *(array+count));
     }
}
```

9 Subarrays

A subarray is an array that is itself part of a larger array.

A subarray can be given as an argument to a function by passing a pointer to the address of the first element of the subarray, using either index or pointer notation. It will also be necessary to indicate the length of the subarray, either by passing the number of elements in the subarray or by passing a pointer to the end of the subarray.

The following program prints a subarray from an array of characters by passing the address of the start and end of the subarray to the function `print_sub` :

```
#include <stdio.h>

void print_sub(const char *, const char *);

int main(void)
{
     char word[9] = "subarrays";
     print_sub(word + 3, word + 8);

     return 0;
}

void print_sub(const char * start, const char * end)
{
     putchar('\n');
     while (start < end)
     {
          printf("%c", *start++);
     }
}
```

When compiled and run, this program demonstrates that the word array is itself a subarray of the word subarray!