Pointers - Chapter Summary

1 What is a structure?

The pointer in C is a powerful and flexible tool that allows the programmer direct access to memory locations and the data they contain. It is one of the features of the language that really justifies the claim that C is "a high-level language that gives low-level access".

Pointers allow the programmer to allocate memory dynamically; that is, minimise the amount of memory a program uses and maximise the speed at which it runs by keeping track of memory use and reallocating storage during program execution.

The use of pointers provides a mechanism for call by reference, where functions can access and manipulate the actual data passed to them as arguments, and not just the values of those data items.

Pointers provide the means for dealing concisely and efficiently with aggregate data types, which allows whole-array and structure operations to be performed. They may also be used to construct complex data structures such as trees and linked lists, which streamline the access of data through multiple levels of choice.

Used wisely and very carefully, pointers can be used to achieve programming clarity, simplicity and efficiency.

The care is needed because an error in a pointer is difficult to trace because the problem appears to be connected with another data item altogether. A pointer contains the address of another data item, and if this address is wrong, reading from it will result in inaccurate or nonsensical data being assigned. Writing to the wrong address may overwrite existing data, with potentially disastrous results!

Wisdom is needed, because pointers provide a mechanism to change data created and stored elsewhere in the program, thus effectively sidestepping all the barriers that C erects to protect variables and ensure data integrity.

2 Pointer Types

A pointer is a scalar data item that contains the address of another data item.

The value stored in a pointer is an address. The type of a pointer, however, is not the type of the value it stores, but the type of the data at which it points; that is to say, the type of the value stored at the address contained in the pointer. Thus, a pointer is declared as a pointer to a particular type of data and cannot then be used to point to any other type of data.

A pointer may, however, be itself used in any context where a scalar data item of the appropriate type may be used.

3 Declaring Pointers

The general form for declaring a pointer is:

```
type * name;
```

For example, the following declarations are for a pointer to `short ints` and a pointer to `doubles` respectively:

```
short  * score;
double * fraction;
```

4 Manipulating Pointers

4.1 The 'address of' operator &

The address of operator & is a unary operator, which, when followed by a data name, gives the address of that data.

The use of this operator in the `scanf` function has already been demonstrated. It is also used for assigning addresses to pointers:

```
short  * sptr = &score;
double * fraction = &conversion;
```

In this example, the pointers `sptr` and `fraction` are declared and assigned the addresses of the variables `score` and `conversion` respectively. The pointers `sptr` and `fraction` now point at `score` and `conversion`.

4.2 The 'contents of' operator *

The contents of operator * is a unary operator, which, when followed by a pointer, gives the value stored at the pointed-to address. This operator also sometimes called the reference operator, or the indirection operator.

There are three `printf` statements in the following program. In the first, the `printf` function has the pointer `fraction` as its argument, and this contains the address of the variable `conversion`. The second `printf` has the expression `*fraction` as its argument, and this evaluates to the value stored in the address pointed to, which is the value stored in the variable `conversion`.

```
int main(void)
{
     double conversion = 0.45359237;
     double * fraction;

     fraction = &conversion;

     printf("\nAddress of 'conversion' is %p.",
               (unsigned int) fraction);
     printf("\nThe value of 'conversion': %f.", *fraction);

     *fraction *= 5;

     printf("\nThe new value of conversion is %f.",
               conversion);

     return 0;
}
```

When compiled and run, this program will first print out the address of the variable "`conversion`" and the original value assigned to it, the latter being `0.45359237` (which happens to be the conversion factor for changing pounds into kilograms).

The penultimate line in the program changes the actual value stored in the variable `conversion`, multiplying it by `5`. This change is confirmed by printing the new value to the screen by means of the final `printf` statement, which takes `conversion` as an argument.

4.3 Indirection

The above program illustrates the important principle of indirection. A pointer contains the address of an item of data, and by using this address, it is possible to access the value stored in the data.

In the example given in 4.2 above, the expression `*fraction *= 5` was used to change the value stored by the variable `conversion`. The pointer `fraction` had been assigned the address of `conversion`, and the expression `*fraction` therefore referred to the contents of that data. The action carried out on `*fraction` indirectly affected the data that `fraction` pointed to.

5 Pointers and Precedence

5.1 Precedence and Pointer Reference

As the example in 4.3 showed, pointer references may be used in any context where data of the corresponding type might be found, including on the left-hand side of assignment statements.

Although a pointer reference such as `*fraction` is equivalent to a data item, it is important to remember that it is also an expression, being composed of a pointer and an operator. The contents-of operator has its own level of precedence, and associates right to left. Great care must be exercised in writing and reading expressions that include pointer references if obscure and elusive programming errors are to be avoided.

In an example given in this chapter, for instance, the final line of the program reads:

```
(*px)++;
```

The contents-of operator * and the increment operator ++ are of equal precedence. Since they both associate from right to left, the increment operator would have acted first if no brackets had been included in the statement. The statement

```
*px++;
```

would thus have meant: post-increment the pointer `px`, and obtain the value pointed to before the increment takes place. That is, the pointer (not the thing being pointed to) would be incremented!

Brackets have a higher precedence than either the contents-of or the increment operator, so the brackets around the pointer reference `(*px)` ensure that it is the value pointed to by `px` that is incremented, and not the value of the address itself.

Note that the C Language Summaries in Appendix include a complete table of operator precedence and associativity.

5.2. The Unary and Binary * Operators

The binary multiplication operator * has the same symbol as the unary pointer-reference operator *. This means that some expressions may be visually confusing:

```
answer = *pointer1**pointer2;
/* Strongly recommend *pointer1 * *pointer2   */
```

Precedence and compiler design ensure that the ambiguity is more apparent than real, however. The pointer-reference operator has a higher precedence than the multiplication operator and will always be executed first. Thus, the above statement will assign the product of the value pointed to by `pointer1` and that pointed to by `pointer2` to the variable `answer`.

Compiler design will ensure that even such an obscure statement as

```
answer = *pointer1*pointer2;
```

will be evaluated correctly. The expression `*pointer1*pointer2` would make no sense if the second * was a pointer-reference operator, since this is a unary operator and therefore cannot link two operands. The compiler will therefore interpret the statement to mean that the value pointed to by `pointer1` is to be multiplied by the address held by `pointer2` and the result assigned to the variable `answer`.

6 Address Constants, Pointer Reference and Pointers

6.1 Addresses and Pointers

The expression `&example` gives the address of the data item called `example`, and this address may be assigned to a pointer:

```
int example;
int * pointer1;

pointer1 = &example;
```

However, remember that although `pointer1` and `&name` both evaluate to the same address, `&example` is a constant, whereas `pointer1` is a data item, in this case a variable. Although `example` is a variable and may be assigned different values, its address remains constant throughout its existence. Pointers are usually variables and may be assigned different addresses at different times.

Please note that a `const` pointer would be defined thus:

```
int * const fixed = &example; /* always points at example */
...
fixed = &somethingelse;       /* ERROR, fixed is const */
```

6.2. Pointer Reference and Pointers

A pointer contains the address of a data item. A pointer reference is an expression that evaluates to the value stored at the address contained in the pointer. As has been demonstrated, the pointer and the pointer reference may often be of different types, as addresses are always unsigned integers.

Although pointer names are always prefixed by a * in declaration statements, pointers can only be assigned addresses, even if the assignment takes place as part of the declaration statement.

7 Limitations to the Use of Pointers

Pointers point to data with fixed addresses. They cannot point to constant literals, like 5, 'c', etc. , which have no addresses, nor to expressions, which, although their constituent variables have addresses, do not in themselves exist as data objects stored in memory.

Pointers cannot point to register variables, as these do not have fixed and predictable addresses.

8 Pointers to Pointers

It is perfectly legitimate in C to declare pointers to pointers. A pointer to a pointer will contain the address of the pointer that is pointed to. Thus, even double indirection is possible:

```
int main(void)
{
     int sample = 100, answer;
     int * pointer1;
     int ** point2point; /* i.e. a pointer to an int pointer */

     pointer1 = &sample;
     point2point = &pointer1;

     answer = *(*point2point);

     printf("\nThe value of 'answer' is %d.", answer);

     return 0;
}
```

When compiled and run, this program will print out:

```
     The value of 'answer' is 100.
```

The expression `*point2point` evaluates to the content of the data pointed to by `point2point`, which, as this data pointed to is another pointer, is an address. This address is the address of the variable `sample`. The expression `*(*point2point)` therefore refers to the content of the variable sample, and this is `100`, which is the value assigned to the variable `answer`.