

## An Introduction to Pointers in C and the use of Structures

### *Pointers*

As established, data is stored in memory and accessed by giving a name to that item of data, at a low level the computer uses memory address to refer to data. There are situation when it is advantageous to do the same thing in a program. For example, when it is not know how much data the user would like to enter into a program while it is running, variables cannot be declared at the compilation stage. In these situations it can be arranged so that the user enters the number of entries required and then storage is dynamically allocated before proceeding. Dynamic memory allocation requires the use of memory addresses. To hold memory address in a program a special variable called a pointer variable is used.

The declaration of a pointer variable is similar to a variable declaration, in that it instructs the complier to reserve storage large enough to hold an address. Syntactically, pointer declarations look just like variable declarations, except that there is a \* between the type and pointer name:

```
int* ip;          // a pointer to an integer
int *iptr;       // another pointer to an integer

char c, *cptr;   // a character variable c and
                // a character pointer cptr
```

All these forms are acceptable, but the preferred form should be the first, as strictly speaking a pointer to an **int** is a unique type, distinct from the **int** type, and this is made explicit by associating the \* with the type name rather than with the pointer variable name.

When a pointer is declared and has not yet been initialised, it contains whatever garbage was in that storage location when the program was started. In this respect a pointer is the same as any other variable. However, uninitialised pointers are particularly dangerous, as they could point to anywhere in memory, and using an uninitialised pointer variable can corrupt the memory in unexpected ways. To initialise a pointer, use the “address-of” operator & on a variable name. For example, if an integer pointer called *ticket* and an integer variable called *prize*, then the pointer *ticket* could be set to hold the address of *prize* with the statement `ticket = &prize;`. It is also possible when declaring a pointer to initialise it with a special value, called NULL, which is generally safer if you forget to initialise the pointer with a sensible value later in the program.

The other that may be required to do with a pointer is to get the value in the memory cell which the pointer is pointing to. This is called pointer dereferencing, and use the pointer to dereferencing operator \* to achieve this (note: the pointer dereference operator \* is nothing to do with the \* used in pointer variable declarations). For example, to write the value contained in *prize* to the screen, the following statement could be used:

```
printf("your prize is %d", *ticket);
```

this would have the same effect as:

```
printf("your prize is %d", prize);
```

Note that `printf("your prize is %d", *ticket);` would not work as this would output the memory address where *prize* is stored, rather than the value of *prize*.

It may be thought that an address is an address and should require the same amount of storage whether it points to a character or to an integer storage location. So, in a pointer declaration, why bother to explicitly declare the type that a pointer points at? The answer lies in pointer arithmetic, i.e. if a pointer is incremented or decremented, that pointer should now point to the next storage location of that type. To do this the compiler needs to know the size of the type of storage being pointed to, so that it can update the address by the correct amount. For example, if it is a **char** pointer, incrementing the pointer should move it along one byte in order to point at the next char; whereas if it is an **int** pointer, it would have to move along two bytes to point at the next int.

## *Structures*

Many programs require you to hold and access information about a single entity or object, say a DVD, which may occur several times, for example the data of 20 DVD's. In other languages this constitutes a 'record', in C it is called a 'struct' or structure. A record or structure is simply a way of holding information made up of different data types, under one name. You have already used something similar, which is an array. An array holds many different data elements, but they are all of the same type. The word 'struct' is a C key word that allows the programmer to define a 'user type'. In other words, when a structure is defined it becomes another data type much in the same way as an int or char are data types. The syntax for defining a structure is shown below. Elements of the syntax shown in square [ ] brackets are optional, elements inside the angled brackets < > are keyword identifiers.

### *struct syntax*

```
struct [<struct type name>] {
    [<type> <variable-name[, variable-name, ...]>] ;
    .....
    .....
    .....
} [<structure variables>] ;
```

<struct type name>    An optional tag name that refers to the structure type  
<structure variables>    The data definitions, also optional.

Although the <struct type name> and <structure variables> are optional, one of them must be included in the declaration. The elements in the record are defined by naming a data type, followed by one or more variable name. If more than one variable name of the same type is declared on the same line, each should be separated by commas.

### *Declaring struct variables*

The structure variable can be declared in two ways. If there is only to be one occurrence of the structure type then it can be declared within the definition:

```
struct{
    int hour;    //struct members, type int
    int minute;
    int second;
}time;            // struct name is time
```

When there may be more than one occurrence or instance of the structure type, the structure can be given a type name that can then be used to declare each instantiation.

```
struct ElectricMotor {
    char Model[5];
    int RPM;
    float NomV, CurrentA, OutputW;
};
```

A variable of the structure is then declared using the struct keyword, the structure tag or type name and a variable name:

```
struct ElectricMotor LeftMotor;
struct ElectricMotor RightMotor;
```

### *Accessing structure elements*

When you used an array you had to access each element of the array with an index or subscript to identify which element was required. With a structure, the elements are parts of the structure variable itself and are accessed by using the record selector operator.

The record selector operator is a full stop (.) between the structure variable name and the element to be accessed.

```
void main(void){
    struct ElectricMotor Mtr;

    strcpy(Mtr.Model, "RT76");
    Mtr.NomV = 12.5;
    printf("\nEnter Speed (RPM): ");
    scanf("%d", &Mtr.RPM);
}
```