"To me the most important part of a program is laying out the data structure."

Dan Bricklin

8.1 INTRODUCTION

In this chapter we will be concerned with pointers, arrays and strings. If you have done any programming before beginning to study C, you will have already encountered and used arrays and strings. If you have arrived here via assembly language programming, you will already be familiar with the concept of pointers. If you have come to C by a different route and are uncertain about pointers then now’s your chance to find out! Pointers are one of the most important features of C and as such contribute to the flexibility and power for which the language is known. The concept of pointers is crucial to a clear understanding of the way arrays work, and strings are simply arrays of type char. It is for this reason that we have not so far concerned ourselves very much with strings. However, by the time you have worked through this chapter, you should be proficient in the basic uses of pointers, be able to use one- and two-dimensional arrays and be at home with strings.

8.2 POINTERS

We begin our discussion of pointers by giving a simple definition:

A pointer is ... a symbolic representation of an address in the computer’s memory

As we saw in Chapter 3 all variables have an address at which the value of the variable is located or stored. We have already encountered the idea of an address when we used the scanf function. This function, you will remember, requires the address of the variable to be given as a parameter to the function rather than the name of the variable. The address operator (&) is used for this purpose. Thus
if we have a variable called `first`, then `&first` is its address; `&first` is a pointer to the variable `first` and will have a numerical value within a function (or program) which will be constant. Thus `&first` is a pointer constant which cannot be changed by reassignment - just as 52 is a constant which cannot be assigned another value. So, although the value of `first` might change during the execution of a program, its address, `&first`, or the pointer to `first` will remain constant.

You might expect that, since we have pointer constants which are analogous to constants, there would also be pointer variables. If that is what you were thinking then you would be quite correct. A pointer variable can be assigned a value which will be the address of a variable. Thus

```plaintext
paddress = &first;
```

assigns to the pointer variable `paddress` the address of the variable `first` - `paddress` now 'points to' `first`. This variable can be assigned to the address of another variable, for example:

```plaintext
paddress = &second;
```

Any subsequent reference to `paddress` would be a reference to `second` rather than to `first`.

**The indirection operator**

Whilst it is useful, and necessary, to be able to use pointers to identify the address of a variable it is also necessary to be able to obtain access to the value stored at that address, i.e. the value at the address 'pointed to' by the pointer variable (or pointer constant). This is achieved by use of the *indirection* operator, or *dereferencing* operator (*). Thus if `paddress` is a pointer to the variable `first`, then the statement

```plaintext
second = *paddress;
```

will assign the current value of `first` to the variable `second`. A little thought will show that the pair of statements `paddress = &first;` and `second = *paddress;` put the same value in `second` as the single assignment statement `second = first;` as is illustrated in Figure 8.1. Thus the (unary) operator * together with the unary operator & allows us to indirectly achieve the same result as can be obtained by a single assignment statement - hence the term indirection operator.