CHAPTER 5 POINTERS AND ARRAYS

Before we start Chapter 5, a quick note from your narrator. From time to time I have been adding some of my interpretation to this book. But I won't be adding anything to the first part of this chapter. I think that sections 5.1 through 5.6 contain some of the most elegantly written text in the book. The concepts are clearly stated and the example code is short, direct, and easy to understand. Pointers are the essential difference between C and any other modern programming languages. So pay close attention to this chapter and make sure you understand it before continuing.

This chapter is as strong now as it was in 1978 so without futher ado, we read as Kernighan and Ritchie teach us about pointers and arrays.

A pointer is a variable that contains the address of another variable. Pointers are very much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways.

Pointers have been lumped with the `goto` statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

5.1 Pointers and Addresses

Since a pointer contains the address of an object, it is possible to access the object "indirectly" through the pointer. Suppose that `x` is a variable, say an `int`, and that `px` is a pointer, created in some as yet unspecified way. The unary operator `&` gives the *address* of an object, so the statement

```
px = &x;
```

assigns the address of `x` to the variable `px`; `px` is now said to "point to" `x`. The `&` operator can be applied only to variables and array elements; constructs like `&(x+1)` and `&3` are illegal. It is also illegal to take the address of a `register` variable.

The unary operator `*` treats its operand as the address of the ultimate target, and accesses that address to fetch the contents. Thus if `y` is also an `int`,

```
y = *px;
```

assigns to `y` the contents of whatever `px` points to. So the sequence