# Week 3

# (and beyond) Advanced Features of C

Currently, this section discusses only one advanced feature: *pointers*. Pointers are useful in many ways. First, they are an important way in which a function can pass a variable to another function in such a way that the called function can modify this variable. (Remember that the em scope rules we discussed in week 2 ordinarily prohibit this.) Secondly, pointers are useful in creating *data structures* whose structure is *dynamic*; i.e., is changed by the program at run-time.

The reader also discusses pointers in detail.

### 3.0.1 Pointers

**Pointers** are one of the most crucial concepts in computer programming. Unfortunately, they are also one of the most difficult to understand for beginners. It took me years to fully understand pointers, so please don't feel discouraged if this is all a little confusing.

Run-time errors involving pointers can be extremely elusive. Therefore, I will try to dissect everything about pointers so that you will at least know where to begin when looking for a pointer related bug.

#### What is a pointer?

Technically, a pointer is an address into the computer's memory system. If this doesn't mean anything to you, don't worry, you don't have to know about memory systems to use pointers. Think of the computer's memory as a row of mailboxes along a suburban street. Each mailbox has a number printed on it so that the mailman knows what box to put mail into. These numbers are memory addresses, and a pointer in our model would be one of these numbers. Therefore it is possible to have a variable that represents a mailbox number, and you can read or write from that part of memory.

There are many reasons why this might be useful. Consider the problem of representing a string in C. Unlike most programming languages, C has no built in capability to represent a string. Instead, the convention is used that a string is a pointer to the first character in the string, and the string continues in memory until a NULL character is seen (character 0). This is equivalent to storing one letter of each string in each mailbox along the street and telling the mailman that the string starts in a specific mailbox, and ends when there is some special terminating character in a mailbox. Pointers have many other uses, which will all be seen later in this book. For now, you only need to be familiar with how to use them and how to declare them as variables.

### Using pointers

It is easier to explain how to declare a pointer if we explain how to use one first. There are two operators in C that deal with pointers: the star operator (*), and the ampersand operator (&).

The star operator is called the **dereferencing** operator. That is, it takes a pointer and gives you the value at that memory location. If you have a pointer `p` that points to some piece of memory that holds the value 10, `*p` will have the value 10.

The ampersand operator is called the **address** operator. It is the opposite of the star operator — it gives the address of a variable. Therefore, if `x` is a variable, `&x` is the address of `x`. The use of this operator will be seen later.

### Declaring pointers

Declaring pointers really isn't any different from declaring any other kind of variable, it's just sometimes hard to tell what's going on. To declare a variable called `p` that is a pointer to an integer, you would use a declaration as shown in figure 3.1.

```
int *p;
```

Figure 3.1: Declaring a pointer

If you're an alert reader, you're probably thinking, "Wait a minute, p is supposed to be a memory address, why are you using the star operator?" This is a perfectly reasonable question. A non-confusing way to think about the declaration is, "I'm declaring an integer variable called `*p`. This means that since `*p` is an integer, p is the address of an integer, or a pointer to an integer.

Pointers are the sort of thing you get used to. Don't worry if none of this makes sense. When you have to use them, it will.

### Pointer arithmetic

In C, you can add and subtract pointers of the same type. Most of the time you don't add two pointers, you just add integers to pointers. For example, if you have a pointer `str` that points to some string, and you want to advance the pointer one character down the string, you simply type `str++;`. This will cause str to "have one added to it", which really means that str will point to the next character in the string.

C is smart enough to do the right thing no matter what type the pointer is. This means that if you have a pointer to the beginning of a sequence of 32 bit integers or 8 bit characters, saying `p++` will always cause `p` to point to the next logical unit according to the type of the pointer.

The ability to add values to pointers allows you to use them just like arrays! If you have declared an integer pointer `p`, the statement `p[5]` is equivalent to `*(p+5)`. This is one of a few reasons why C is too dumb to do array bounds checking (can you see why?)

### Modifying function arguments

I mentioned before that the arguments to a function act as local variables within the function. It is possible to use them just as local variables and change them! It is very important to note, however, that **any change to an argument will not appear outside the function!** Consider the C program in figure 3.2.

```
#include <stdio.h>

void silly(int x)
{
  x *= 2;
}

void main(void)
{
  int y;

  y = 10;
  silly(y);
  printf ("The new value of y is %d\n", y);
}
```

Figure 3.2: A silly C program

What do you think it will print? If you followed what I just said, you know that it will print 10, not 20. The desired effect is possible using pointers! If you were to pass a function the memory address of one of your variables, the function can put data into that memory if it wants. Figure 3.3 shows a C program that achieves the desired effect.

```
#include <stdio.h>

void NotSoSilly(int *x)
{
  *x *= 2;
}

void main(void)
{
  int y;

  y = 10;

  NotSoSilly(&y);
  printf ("The new value of y is %d\n",y);
}
```

Figure 3.3: Passing pointers to functions

This program will print out 20 as desired. As you can see, we use the ampersand operator to pass the function `NotSoSilly` the *address* of one of its local variables. The function can then modify the contents of that address.