CHAPTER 7

Pointers

You had a glimpse of pointers in the last chapter and just a small hint at what you can use them for. Here, you'll delve a lot deeper into the subject of pointers and see what else you can do with them.

I'll cover a lot of new concepts here, so you may need to repeat some things a few times. This is a long chapter, so spend some time on it and experiment with the examples. Remember that the basic ideas are very simple, but you can apply them to solving complicated problems. By the end of this chapter, you'll be equipped with an essential element for effective C programming.

In this chapter you'll learn the following:

- What a pointer is and how it's used
- What the relationship between pointers and arrays is
- How to use pointers with strings
- How you can declare and use arrays of pointers
- How to write an improved calculator program

A First Look at Pointers

You have now come to one of the most extraordinarily powerful tools in the C language. It's also potentially the most confusing, so it's important you get the ideas straight in your mind at the outset and maintain a clear idea of what's happening as you dig deeper.

Back in Chapters 2 and 5 I discussed memory. I talked about how your computer allocates an area of memory when you declare a variable. You refer to this area in memory using the variable name in your program, but once your program is compiled and running, your computer references it by the address of the memory location. This is the number that the computer uses to refer to the "box" in which the value of the variable is stored.

Look at the following statement:

```
int number = 5;
```

Here an area of memory is allocated to store an integer, and you can access it using the name `number`. The value 5 is stored in this area. The computer references the area using an address. The specific address where this data will be stored depends on your computer and what operating system and compiler you're using. Even though the variable name is fixed in the source program, the address is likely to be different on different systems.

Variables that can store addresses are called **pointers**, and the address that's stored in a pointer is usually that of another variable, as illustrated in Figure 7-1. You have a pointer `P` that contains the address of another variable, called `number`, which is an integer variable containing the value 5. The address that's stored in `P` is the address of the first byte of `number`.

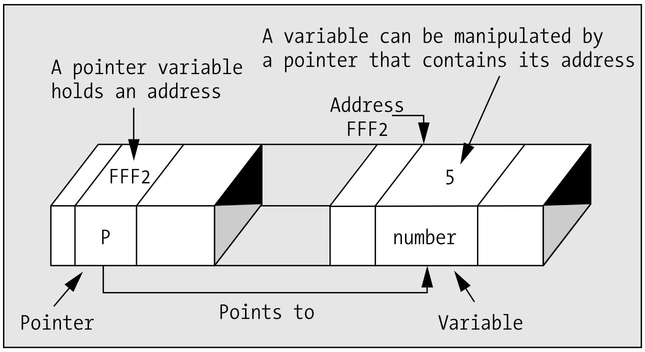
A variable can be manipulated by a pointer that contains its address
A pointer variable holds an address
Address FFF2
FFF2
5
P
number
Pointer
Points to
Variable

Figure 7-1. *How a pointer works*

The first thing to appreciate is that it's not enough to know that a particular variable, such as `P`, is a pointer. You, and more importantly, the compiler, must know the *type* of data stored in the variable to which it points. Without this information it's virtually impossible to know how to handle the contents of the memory to which it points. A pointer to a value of type `char` is pointing to a value occupying 1 byte, whereas a pointer to a value of type `long` is usually pointing to the first byte of a value occupying 4 bytes. This means that every pointer will be associated with a specific variable type, and it can be used only to point to variables of that type. So pointers of type "pointer to `int`" can point only to variables of type `int`, pointers of type "pointer to `float`" can point only to variables of type `float`, and so on. In general a pointer of a given type is written `type *` for any given type name `type`.

The type name `void` means absence of any type, so a pointer of type `void *` can contain the address of a data item of any type. Type `void *` is often used as an argument type or return value type with functions that deal with data in a type-independent way. Any kind of pointer can be passed around as a value of type `void *` and then cast to the appropriate type when you come to use it. The address of a variable of type `int` can be stored in a pointer variable of type `void *` for example. When you want to access the integer value at the address stored in the `void *` pointer, you must first cast the pointer to type `int *`. You'll meet the `malloc()` library function later in this chapter that returns a pointer of type `void *`.

Declaring Pointers

You can declare a pointer to a variable of type `int` with the following statement:

```
int *pointer;
```

The type of the variable with the name `pointer` is `int *`. It can store the address of any variable of type `int`. This statement just creates the `pointer` but doesn't initialize it. Uninitialized pointers are particularly hazardous, so you should always initialize a pointer when you declare it. You can initialize `pointer` so that it doesn't point to anything by rewriting the declaration like this:

```
int *pointer = NULL;
```

NULL is a constant that's defined in the standard library and is the equivalent of zero for a pointer. NULL is a value that's guaranteed not to point to any location in memory. This means that it implicitly prevents the accidental overwriting of memory by using a pointer that doesn't point to anything specific. NULL is defined in the header files <stddef.h>, <stdlib.h>, <stdio.h>, <string.h>, <time.h>, <wchar.h>, and <locale.h>, and you must have at least one of these headers included in your source file for NULL to be recognized by the compiler.

If you want to initialize your variable pointer with the address of a variable that you've already declared, you use the address of operator &:

```
int number = 10;
int *pointer = &number;
```

Now the initial value of pointer is the address of the variable number. Note that the declaration of number must precede the declaration of the pointer. If this isn't the case, your code won't compile. The compiler needs to have already allocated space and thus an address for number to use it to initialize the pointer variable.

There's nothing special about the declaration of a pointer. You can declare regular variables and pointers in the same statement, for example

```
double value, *pVal, fnum;
```

This statement declares two double precision floating-point variables, value and fnum, and a variable, pVal of type "pointer to double." With this statement it is obvious that only the second variable, pVal, is a pointer, but consider this statement:

```
int *p, q;
```

This declares a pointer, p, and a variable, q, that is of type int. It is a common mistake to think that both p and q are pointers.

Accessing a Value Through a Pointer

You use the **indirection operator**, *, to access the value of the variable pointed to by a pointer. This operator is also referred to as the **dereference operator** because you use it to "dereference" a pointer. Suppose you declare the following variables:

```
int number = 15;
int *pointer = &number;
int result = 0;
```

The pointer variable contains the address of the variable number, so you can use this in an expression to calculate a new value for total, like this:

```
result = *pointer + 5;
```

The expression *pointer will evaluate to the value stored at the address contained in the pointer. This is the value stored in number, 15, so result will be set to 15 + 5, which is 20.

So much for the theory. Let's look at a small program that will highlight some of the characteristics of this special kind of variable.

TRY IT OUT: DECLARING POINTERS

In this example, you're simply going to declare a variable and a pointer. You'll then see how you can output their addresses and the values they contain.

```
/* Program 7.1 A simple program using pointers */
#include <stdio.h>

int main(void)
{
  int number = 0;                 /* A variable of type int initialized to 0  */
  int *pointer = NULL;            /* A pointer that can point to type int     */

  number = 10;
  printf("\nnumber's address: %p", &number);            /* Output the address */
  printf("\nnumber's value: %d\n\n", number);           /* Output the value   */

  pointer = &number;              /* Store the address of number in pointer   */

  printf("pointer's address: %p", &pointer);            /* Output the address */
  printf("\npointer's size: %d bytes", sizeof(pointer)); /* Output the size   */
  printf("\npointer's value: %p", pointer);  /* Output the value (an address) */
  printf("\nvalue pointed to: %d\n", *pointer);       /* Value at the address */
  return 0;
}
```

The output from the program will look something like the following. Remember, the actual address is likely to be different on your machine:

```
number's address: 0012FEE4
number's value: 10

pointer's address: 0012FEE0
pointer's size: 4 bytes
pointer's value: 0012FEE4
value pointed to: 10
```

How It Works

You first declare a variable of type `int` and a pointer:

```
  int number = 0;                 /* A variable of type int initialized to 0 */
  int *pointer = NULL;            /* A pointer that can point to type int     */
```

The pointer called `pointer` is of type "pointer to `int`." Pointers need to be declared just like any other variable. To declare the pointer called `pointer`, you put an asterisk (*) in front of the variable name in the declaration. The asterisk defines `pointer` as a pointer, and the type, `int`, fixes it as a pointer to integer variables. The initial value, `NULL`, is the equivalent of 0 for a pointer—it doesn't point to anything.

After the declarations, you store the value 10 in the variable called `number` and then output its address and its value with these statements:

```
  number = 10;
  printf("\nnumber's address: %p", &number);           /* Output the address */
  printf("\nnumber's value: %d\n\n", number);          /* Output the value   */
```

To output the address of the variable called `number`, you use the output format specifier `%p`. This outputs the value as a memory address in hexadecimal form.

The next statement obtains the address of the variable `number` and stores that address in `pointer`, using the `address of` operator `&`:

```
  pointer = &number;             /* Store the address of number in pointer   */
```

Remember, the only kind of value that you should store in `pointer` is an address.

Next, you have four `printf()` statements that output, respectively, the address of `pointer` (which is the first byte of the memory location that `pointer` occupies), the number of bytes that the pointer occupies, the value stored in `pointer` (which is the address of `number`), and the value stored at the address that `pointer` contains (which is the value stored in `number`).

Just to make sure you're clear about this, let's go through these line by line. The first output statement is as follows:

```
  printf("pointer's address: %p", &pointer);
```

Here, you output the address of `pointer`. Remember, a pointer itself has an address, just like any other variable. You use `%p` as the conversion specifier to display an address, and you use the `&` (`address of`) operator to reference the address that the `pointer` variable occupies.

Next you output the size of `pointer`:

```
  printf("\npointer's size: %d bytes", sizeof(pointer)); /* Output the size   */
```

You can use the `sizeof` operator to obtain the number of bytes a pointer occupies, just like any other variable, and the output on my machine shows that a pointer occupies 4 bytes, so a memory address on my machine is 32 bits.

The next statement outputs the value stored in `pointer`:

```
  printf("\npointer's value: %p", pointer);
```

The value stored in `pointer` is the address of `number`. Because this is an address, you use `%p` to display it and you use the variable name, `pointer`, to access the address value.

The last output statement is as follows:

```
  printf("\nvalue pointed to: %d", *pointer);
```

Here, you use the pointer to access the value stored in `number`. The effect of the `*` operator is to access the data contained in the address stored at `pointer`. You use `%d` because you know it's an integer value. The variable `pointer` stores the address of `number`, so you can use that address to access the value stored in `number`. As I said, the `*` operator is called the **indirection operator**, or sometimes the **dereferencing operator**.

While we've noted that the addresses shown will be different on different computers, they'll often be different at different times on the same computer. The latter is due to the fact that your program won't always be loaded at the same place in memory. The addresses of `number` and `pointer` are where in the computer the variables are stored. Their values are what is actually stored at those addresses. For the variable called `number`, it's an actual integer value (10), but for the variable called `pointer`, it's the address of `number`. Using `*pointer` actually gives you access to the value of `number`. You're accessing the value of the variable, `number`, indirectly.

You'll certainly have noticed that your indirection operator, `*`, is also the symbol for multiplication. Fortunately, there's no risk of confusion for the compiler. Depending on where the asterisk appears, the compiler will understand whether it should interpret it as an indirection operator or as a multiplication sign.

Figure 7-2 illustrates using a pointer.

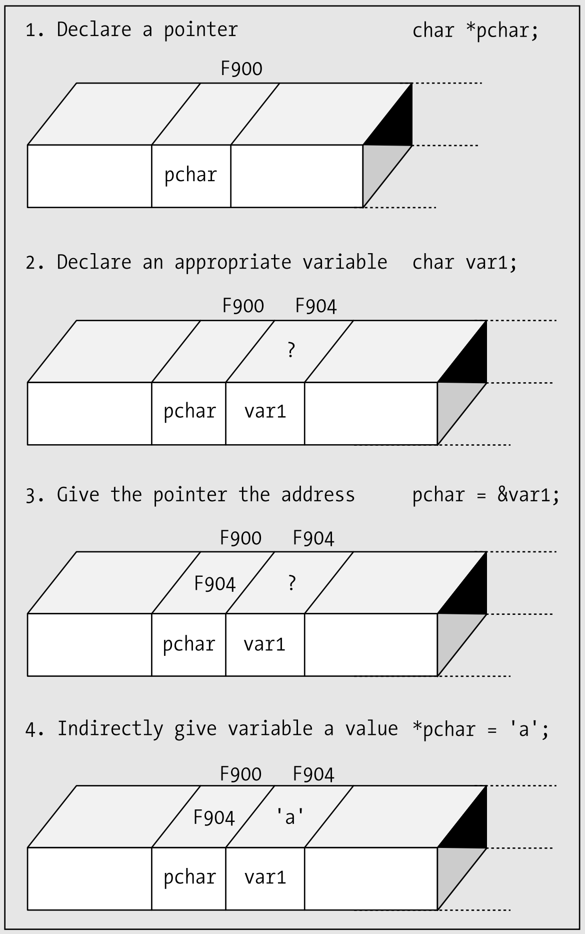

Figure 7-2. *Using a pointer*

Using Pointers

Because you can access the contents of `number` through the pointer `pointer`, you can use a dereferenced pointer in arithmetic statements. For example

```
*pointer += 25;
```

This statement increments the value of whatever variable pointer currently addresses by 25. The * indicates you're accessing the contents of whatever the variable called `pointer` is pointing to. In this case, it's the contents of the variable called `number`.

The variable `pointer` can store the address of any variable of type `int`. This means you can change the variable that `pointer` points to by a statement such as this:

```
pointer = &another_number;
```

If you repeat the same statement that you used previously:

```
*pointer +=  25;
```

the statement will operate with the new variable, `another_number`. This means that a pointer can contain the address of any variable of the same type, so you can use one pointer variable to change the values of many other variables, as long as they're of the same type as the pointer.

TRY IT OUT: USING POINTERS

Let's exercise this newfound facility in an example. You'll use pointers to increase values stored in some other variables.

```
/* Program 7.2  What's the pointer */
#include <stdio.h>

int main(void)
{
  long num1 = 0L;
  long num2 = 0L;
  long *pnum = NULL;

  pnum = &num1;                              /* Get address of num1       */
  *pnum = 2;                                 /* Set num1 to 2             */
  ++num2;                                    /* Increment num2            */
  num2 += *pnum;                             /* Add num1 to num2          */

  pnum = &num2;                              /* Get address of num2       */
  ++*pnum;                                   /* Increment num2 indirectly */

  printf("\nnum1 = %ld  num2 = %ld  *pnum = %ld  *pnum + num2 = %ld\n",
                                    num1, num2, *pnum, *pnum + num2);
  return 0;
}
```

When you run this program, you should get the following output:

```
num1 = 2  num2 = 4  *pnum = 4  *pnum + num2 = 8
```

How It Works

The comments should make the program easy to follow up to the `printf()`. First, in the body of `main()`, you have these declarations:

```
  long num1 = 0;
  long num2 = 0;
  long *pnum = NULL;
```

This ensures that you set out with initial values for the two variables, `num1` and `num2`, at 0. The third statement above declares an integer pointer, `pnum`, which is initialized with `NULL`.

■ **Caution** You should always initialize your pointers when you declare them. Using a pointer that isn't initialized to store an item of data is dangerous. Who knows what you might overwrite when you use the pointer to store a value?

The next statement is an assignment:

```
pnum = &num1;                                   /* Get address of num1       */
```

The pointer `pnum` is set to point to `num1` here, because you take the address of `num1` using the & operator.

The next two statements are the following:

```
*pnum = 2;                                       /* Set num1 to 2             */
++num2;                                          /* Increment num2            */
```

The first statement exploits your newfound power of the pointer, and you set the value of `num1` to 2 indirectly by dereferencing `pnum`. Then the variable `num2` gets incremented by 1 in the normal way, using the increment operator.

The statement is the following:

```
num2 += *pnum;                                  /* Add num1 to num2          */
```

This adds the contents of the variable pointed to by `pnum`, to `num2`. Because `pnum` still points to `num1`, `num2` is being increased by the value of `num1`.

The next two statements are the following:

```
pnum = &num2;                                    /* Get address of num2      */
++*pnum;                                         /* Increment num2 indirectly */
```

First, the pointer is reassigned to point to `num2`. The variable `num2` is then incremented indirectly through the pointer. You can see that the expression `++*pnum` increments the value pointed to by `pnum` without any problem However, if you want to use the postfix form, you have to write `(*pnum)++`. The parentheses are essential—assuming that you want to increment the value rather than the address. If you omit them, the increment would apply to the address contained in `pnum`. This is because the operators ++ and unary * (and unary &, for that matter) share the same precedence level and are evaluated right to left. The compiler would apply the ++ to `pnum` first, incrementing the address, and only then dereference it to get the value. This is a common source of error when incrementing values through pointers, so it's probably a good idea to use parentheses in any event.

Finally, before the `return` statement that ends the program, you have the following `printf()` statement:

```
printf("\nnum1 = %ld  num2 = %ld  *pnum = %ld  *pnum + num2 = %ld",
                                    num1, num2, *pnum, *pnum + num2);
```

This displays the values of `num1`, `num2`, `num2` incremented by 1 through `pnum` and, lastly, `num2` in the guise of `pnum`, with the value of `num2` added.

Pointers can be confusing when you encounter them for the first time. It's the multiple levels of meaning that are the source of the confusion. You can work with addresses or values, pointers or variables, and sometimes it's hard to work out what exactly is going on. The best thing to do is to keep writing short programs that use the things I've described: getting values using pointers, changing values, printing addresses, and so on. This is the only way to really get confident about using pointers.

I've mentioned the importance of operator precedence again in this discussion. Don't forget that Table 3-2 in Chapter 3 shows the precedence of all the operators in C, so you can always refer back to it when you are uncertain about the precedence of an operator.

Let's look at an example that will show how pointers work with input from the keyboard.

TRY IT OUT: USING A POINTER WITH SCANF()

Until now, when you've used `scanf()` to input values, you've used the & operator to obtain the address to be transferred to the function. When you have a pointer that already contains an address, you simply need to use the pointer name as a parameter. You can see this in the following example:

```
/* Program 7.3  Pointer argument to scanf */
#include <stdio.h>

int main(void)
{
  int value = 0;
  int *pvalue = NULL;

  pvalue = &value;                    /* Set pointer to refer to value  */

  printf ("Input an integer: ");
  scanf(" %d", pvalue);              /* Read into value via the pointer */

  printf("\nYou entered %d\n", value);      /* Output the value entered */
  return 0;
}
```

This program will just echo what you enter. How unimaginative can you get? Typical output could be something like this:

```
Input an integer: 10
You entered 10
```

How It Works

Everything should be pretty clear up to the `scanf()` statement:

```
  scanf(" %d", pvalue);
```

You normally store the value entered by the user at the address of the variable. In this case, you could have used `&value`. But here, the pointer `pvalue` is used to hand over the address of `value` to `scanf()`. You already stored the address of `value` in `pvalue` with this assignment:

```
  pvalue = &value;                    /* Set pointer to refer to value  */
```

`pvalue` and `&value` are the same, so you can use either.

You then just display `value`:

```
  printf("\nYou entered %d", value);
```

Although this is a rather pointless example, it isn't pointerless, as it illustrates how pointers and variables can work together.

Testing for a NULL Pointer

The pointer declaration in the last example is the following:

```
  int *pvalue = NULL;
```

Here, you initialize pvalue with the value NULL. As I said previously, NULL is a special constant in C, and it's the pointer equivalent to 0 with ordinary numbers. The definition of NULL is contained in <stdio.h> as well as a number of other header files, so if you use it, you must ensure that you include one of these header files.

When you assign 0 to a pointer, it's the equivalent of setting it to NULL, so you could write the following:

```
   int *pvalue = 0;
```

Because NULL is the equivalent of zero, if you want to test whether the pointer pvalue is NULL, you can write this:

```
if(!pvalue)
{
  ...
}
```

When pvalue is NULL, !pvalue will be true, so the block of statement will be executed only if pvalue is NULL. Alternatively you can write the test as follows:

```
if(pvalue == NULL)
{
  ...
}
```

Pointers to Constants

You can use the const keyword when you declare a pointer to indicate that the value pointed to must not be changed. Here's an example of a declaration of a const pointer:

```
long value = 9999L;
const long *pvalue = &value;      /* Defines a pointer to a constant */
```

Because you have declared the value pointed to by pvalue to be const, the compiler will check for any statements that attempt to modify the value pointed to by pvalue and flag such statements as an error. For example, the following statement will now result in an error message from the compiler:

```
*pvalue = 8888L;                  /* Error - attempt to change const location */
```

You have only asserted that what pvalue points to must not be changed. You are quite free to do what you want with value:

```
value = 7777L;
```

The value pointed to has changed but you did not use the pointer to make the change. Of course, the pointer itself is not constant, so you can still change what it points to:

```
long number = 8888L;
pvalue = &number;                 /* OK - changing the address in pvalue     */
```

This will change the address stored in pvalue to point to number. You still cannot use the pointer to change the value that is stored though. You can change the address stored in the pointer as much as you like but using the pointer to change the value pointed to is not allowed.

Constant Pointers

Of course, you might also want to ensure that the address stored in a pointer cannot be changed. You can arrange for this to be the case by using the `const` keyword slightly differently in the declaration of the pointer. Here's how you could ensure that a pointer always points to the same thing:

```
int count = 43;
int *const pcount = &count;          /* Defines a constant */
```

The second statement declares and initializes `pnumber` and indicates that the address stored must not be changed. The compiler will therefore check that you do not inadvertently attempt to change what the pointer points to elsewhere in your code, so the following statements will result in an error message when you compile:

```
int item = 34;
pcount = &item;                    /* Error - attempt to change a constant pointer */
```

You can still change the value that `pcount` points to using `pcount` though:

```
*pcount = 345;                     /* OK - changes the value of count */
```

This references the value stored in `count` through the pointer and changes its value to 345. You could also use `count` directly to change the value.

You can create a constant pointer that points to a value that is also constant:

```
int item = 25;
const int *const pitem = &item;
```

`pitem` is a constant pointer to a constant so everything is fixed. You cannot change the address stored in `pitem` and you cannot use `pitem` to modify what it points to.

Naming Pointers

You've already started to write some quite large programs. As you can imagine, when your programs get even bigger, it's going to get even harder to remember which variables are normal variables and which are pointers. Therefore, it's quite a good idea to use names beginning with `p` for use as pointer names. If you follow this method religiously, you stand a reasonable chance of knowing which variables are pointers.

Arrays and Pointers

You'll need a clear head for this bit. Let's recap for a moment and recall what an array is and what a pointer is:

> An **array** is a collection of objects of the same type that you can refer to using a single name. For example, an array called `scores[50]` could contain all your basketball scores for a 50-game season. You use a different index value to refer to each element in the array. `scores[0]` is your first score and `scores[49]` is your last. If you had ten games each month, you could use a multidimensional array, `scores[12][10]`. If you start play in January, the third game in June would be referenced by `scores[5][2]`.

> A **pointer** is a variable that has as its value the address of another variable or constant of a given type. You can use a pointer to access different variables at different times, as long as they're all of the same type.

These seem quite different, and indeed they are, but arrays and pointers are really very closely related and they can sometimes be used interchangeably. Let's consider strings. A string is just an array of elements of type `char`. If you want to input a single character with `scanf()`, you could use this:

```
char single;
scanf("%c", &single);
```

Here you need the `address of` operator for `scanf()` to work because `scanf()` needs the address of the location where the input data is to be stored.

However, if you're reading in a string, you can write this:

```
char multiple[10];
scanf("%s", multiple);
```

Here you don't use the `&` operator. You're using the array name just like a pointer. If you use the array name in this way without an index value, it refers to the address of the first element in the array.

Always keep in mind, though, that arrays are *not* pointers, and there's an important difference between them. You can change the address contained in a pointer, but you can't change the address referenced by an array name.

Let's go through several examples to see how arrays and pointers work together. The following examples all link together as a progression. With practical examples of how arrays and pointers can work together, you should find it fairly easy to get a grasp of the main ideas behind pointers and their relationship to arrays.

TRY IT OUT: ARRAYS AND POINTERS

Just to further illustrate that an array name by itself refers to an address, try running the following program:

```
/* Program 7.4  Arrays and pointers - A simple program*/
#include <stdio.h>

int main(void)
{
  char multiple[] = "My string";

  char *p = &multiple[0];
  printf("\nThe address of the first array element  : %p", p);

  p = multiple;
  printf("\nThe address obtained from the array name: %p\n", p);
  return 0;
}
```

On my computer, the output is as follows:

```
The address of the first array element  : 0x0013ff62
The address obtained from the array name: 0x0013ff62
```

How It Works

You can conclude from the output of this program that the expression `&multiple[0]` produces the same value as the expression `multiple`. This is what you might expect because `multiple` evaluates to the address of the first byte of the array, and `&multiple[0]` evaluates to the first byte of the first element of the array, and it would be surprising if these were not the same. So let's take this a bit further. If `p` is set to `multiple`, which has the same value as `&multiple[0]`, what does `p + 1` equal? Let's try the following example.

TRY IT OUT: ARRAYS AND POINTERS TAKEN FURTHER

This program demonstrates the effect of adding an integer value to a pointer.

```
/* Program 7.5 Arrays and pointers taken further */
#include <stdio.h>

int main(void)
{
  char multiple[] = "a string";
  char *p = multiple;

  for(int i = 0 ; i<strlen(multiple) ; i++)
    printf("\nmultiple[%d] = %c  *(p+%d) = %c  &multiple[%d] = %p  p+%d = %p",
                       i, multiple[i], i, *(p+i), i, &multiple[i], i, p+i);
  return 0;
}
```

The output is the following:

```
multiple[0] = a  *(p+0) = a  &multiple[0] = 0x0013ff63  p+0 = 0x0013ff63
multiple[1] =    *(p+1) =    &multiple[1] = 0x0013ff64  p+1 = 0x0013ff64
multiple[2] = s  *(p+2) = s  &multiple[2] = 0x0013ff65  p+2 = 0x0013ff65
multiple[3] = t  *(p+3) = t  &multiple[3] = 0x0013ff66  p+3 = 0x0013ff66
multiple[4] = r  *(p+4) = r  &multiple[4] = 0x0013ff67  p+4 = 0x0013ff67
multiple[5] = i  *(p+5) = i  &multiple[5] = 0x0013ff68  p+5 = 0x0013ff68
multiple[6] = n  *(p+6) = n  &multiple[6] = 0x0013ff69  p+6 = 0x0013ff69
multiple[7] = g  *(p+7) = g  &multiple[7] = 0x0013ff6a  p+7 = 0x0013ff6a
```

How It Works

Look at the list of addresses to the right in the output. Because `p` is set to the address of `multiple`, `p + n` is essentially the same as `multiple + n`, so you can see that `multiple[n]` is the same as `*(multiple + n)`. The addresses differ by 1, which is what you would expect for an array of elements that each occupy one byte. You can see from the two columns of output to the left that `*(p + n)`, which is dereferencing the address that you get by adding an integer `n` to the address in `p`, evaluates to the same thing as `multiple[n]`.

TRY IT OUT: DIFFERENT TYPES OF ARRAYS

That's interesting, but you already knew that the computer could add numbers together without much problem. So let's change to a different type of array and see what happens:

```
/* Program 7.6 Different types of arrays */
#include <stdio.h>

int main(void)
{
  long multiple[] = {15L, 25L, 35L, 45L};
  long * p = multiple;

  for(int i = 0 ; i<sizeof(multiple)/sizeof(multiple[0]) ; i++)
    printf("\naddress p+%d (&multiple[%d]): %d   *(p+%d) value: %d",
                        i,          i,    p+i,      i,      *(p+i));
  printf("\n    Type long occupies: %d bytes\n", sizeof(long));
  return 0;
}
```

If you compile and run this program, you get an entirely different result:

```
address p+0 (&multiple[0]): 1310552   *(p+0) value: 15
address p+1 (&multiple[1]): 1310556   *(p+1) value: 25
address p+2 (&multiple[2]): 1310560   *(p+2) value: 35
address p+3 (&multiple[3]): 1310564   *(p+3) value: 45
    Type long occupies: 4 bytes
```

How It Works

I have spaced out the second and subsequent arguments to the `printf()` function so you can more easily see the correspondence between format specifiers and the arguments. This time the pointer, `p`, is set to the address that results from `multiple`, where `multiple` is an array of elements of type `long`. The pointer will initially contain the address of the first byte in the array, which is also the first byte of the element `multiple[0]`. This time the addresses are displayed using the `%d` specifier so they will be decimal values. This will make is easier to see the difference between successive addresses.

Look at the output. With this example, `p` is 1310552 and `p+1` is equal to 1310556. You can see that 1310556 is 4 greater than 1310552 although you only added 1. This isn't a mistake. The compiler realizes that when you add 1 to an address value, what you actually want to do is access the next variable of that type. This is why, when you declare a pointer, you have to specify the *type* of variable that's to be pointed to. Remember that `char` data is stored in 1 byte and that variables declared as `long` typically occupy 4 bytes. As you can see, on my computer variables declared as `long` are 4 bytes. Incrementing a pointer to type `long` by 1 on my computer increments the address by 4, because a value of type `long` occupies 4 bytes. On a computer that stores type `long` in 8 bytes, incrementing a pointer to `long` by 1 will increase the address value by 8.

Note that you could use the array name directly in this example. You could write the `for` loop as

```
  for(int i = 0 ; i<sizeof(multiple)/sizeof(multiple[0]) ; i++)
    printf(
      "\naddress multiple+%d (&multiple[%d]): %d   *(multiple+%d) value: %d",
                          i,           i,  multiple+i,       i, *(multiple+i));
```

This works because the expressions `multiple` and `multiple+i` both evaluate to an address. We output the values of these addresses and output the value at these addresses by using the `*` operator. The arithmetic with addresses works the same here as it did with the pointer `p`. Incrementing `multiple` by 1 results in the address of the next element in the array, which is 4 bytes further along in memory. However, don't be misled; an array name is just a fixed address and is not a pointer.

Multidimensional Arrays

So far, you've looked at one-dimensional arrays; but is it the same story with arrays that have two or more dimensions? Well, to some extent it is. However, the differences between pointers and array names start to become more apparent. Let's consider the array that you used for the tic-tac-toe program at the end of Chapter 5. You declared the array as follows:

```
char board[3][3] = {
                     {'1','2','3'},
                     {'4','5','6'},
                     {'7','8','9'}
                   };
```

You'll use this array for the examples in this section, to explore multidimensional arrays in relation to pointers.

TRY IT OUT: USING TWO-DIMENSIONAL ARRAYS

You'll look first at some of the addresses related to your array, `board`, with this example:

```
/* Program 7.7 Two-Dimensional arrays and pointers */
#include <stdio.h>

int main(void)
{
  char board[3][3] = {
                       {'1','2','3'},
                       {'4','5','6'},
                       {'7','8','9'}
                     };

  printf("address of board       : %p\n", board);
  printf("address of board[0][0]  : %p\n", &board[0][0]);
  printf("but what is in board[0] : %p\n", board[0]);
  return 0;
}
```

The output might come as a bit of a surprise to you:

```
address of board        : 0x0013ff67
address of board[0][0]  : 0x0013ff67
but what is in board[0] : 0x0013ff67
```

How It Works

As you can see, all three output values are the same, so what can you deduce from this? The answer is quite simple. When you declare a one-dimensional array, placing `[n1]` after the array name tells the compiler that it's an array with `n1` elements. When you declare a two-dimensional array by placing `[n2]` for the second dimension after the `[n1]` for the first dimension, the compiler creates an array of size `n1`, in which each element is an array of size `n2`.

As you learned in Chapter 5, when you declare a two-dimensional array, you're creating an array of subarrays. So when you access this two-dimensional array using the array name with a single index value, `board[0]` for example, you're actually referencing the address of one of the subarrays. Using the two-dimensional array name by itself references the address of the beginning of the whole array of subarrays, which is also the address of the beginning of the first subarray.

To summarize

```
board
board[0]
&board[0][0]
```

all have the same value, but they aren't the same thing.

This also means that the expression `board[1]` results in the same address as the expression `board[1][0]`. This should be reasonably easy to understand because the latter expression is the first element of the second subarray, `board[1]`.

The problems start when you use pointer notation to get to the values within the array. You still have to use the indirection operator, but you must be careful. If you change the preceding example to display the value of the first element, you'll see why:

```
/* Program 7.7 A Two-Dimensional arrays */
#include <stdio.h>

int main(void)
{
  char board[3][3] = {
                       {'1','2','3'},
                       {'4','5','6'},
                       {'7','8','9'}
                     };

  printf("value of board[0][0] : %c\n", board[0][0]);
  printf("value of *board[0]   : %c\n", *board[0]);
  printf("value of **board     : %c\n", **board);
  return 0;
}
```

The output from this program is as follows:

```
value of board[0][0] : 1
value of *board[0]   : 1
value of **board     : 1
```

As you can see, if you use board as a means of obtaining the value of the first element, you need to use two indirection operators to get it: **board. You were able to use just one * in the previous program because you were dealing with a one-dimensional array. If you used only the one *, you would get the address of the first element of the array of arrays, which is the address referenced by board[0].
The relationship between the multidimensional array and its subarrays is shown in Figure 7-3.

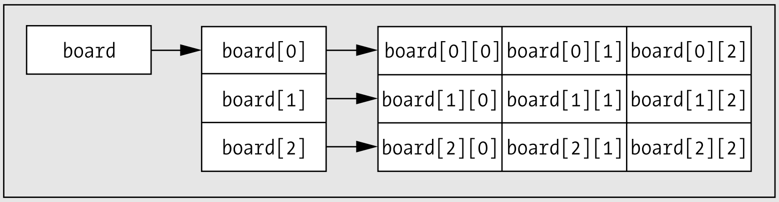

Figure 7-3. *Referencing an array, its subarrays, and its elements*

As Figure 7-3 shows, board refers to the address of the first element in the array of subarrays, and board[0], board[1], and board[2] refer to the addresses of the first element in the corresponding subarrays. Using two index values accesses the value stored in an element of the array. So, with this clearer picture of what's going on in your multidimensional array, let's see how you can use board to get to all the values in that array. You'll do this in the next example.

TRY IT OUT: GETTING ALL THE VALUES IN A TWO-DIMENSIONAL ARRAY

This example takes the previous example a bit further using a for loop:

```
/* Program 7.8  Getting the values in a two-dimensional array */
#include <stdio.h>

int main(void)
{
  char board[3][3] = {
                       {'1','2','3'},
                       {'4','5','6'},
                       {'7','8','9'}
                     };

   /* List all elements of the array */
  for(int i = 0; i < 9; i++)
    printf(" board: %c\n", *(*board + i));
  return 0;
}
```

The output from the program is as follows:

```
board: 1
board: 2
board: 3
board: 4
board: 5
board: 6
board: 7
board: 8
board: 9
```

How It Works

The thing to notice about this program is the way you dereference `board` in the loop:

```
printf(" board: %c\n", *(*board + i));
```

As you can see, you use the expression `*(*board + i)` to get the value of an array element. The expression between the parentheses, `*board + i`, produces the address of the element in the array that is at offset `i`. Dereferencing this results in the value at this address. It's important that the brackets are included. Leaving them out would give you the value pointed to by `board` (i.e., the value stored in the location referenced by the address stored in `board`) with the value of `i` added to this value. So if `i` had the value 2, you would simply output the value of the first element of the array plus 2. What you actually want to do, and what your expression does, is to add the value of `i` to the *address* contained in `board`, and then dereference this new address to obtain a value.

To make this clearer, let's see what happens if you omit the parentheses in the example. Try changing the initial values for the array so that the characters go from `'9'` to `'1'`. If you leave out the brackets in the expression in the `printf()` call, so that it reads like this

```
printf(" board: %c\n", **board + i);
```

you should get output that looks something like this:

```
board: 9
board: :
board: ;
board: <
board: =
board: >
board: ?
board: @
board: A
```

This output results because you're adding the value of `i` to the contents of the first element of the array, `board`. The characters you get come from the ASCII table, starting at `'9'` and continuing to `'A'`.

Also, if you us the expression `**(board + i)`, this too will give erroneous results. In this case, `**(board + 0)` points to `board[0][0]`, whereas `**(board + 1)` points to `board[1][0]`, and `**(board + 2)` points to `board[2][0]`. If you use higher increments, you access memory locations outside the array, because there isn't a fourth element in the array of arrays.

Multidimensional Arrays and Pointers

So now that you've used the array name using pointer notation for referencing a two-dimensional array, let's use a variable that you've declared as a pointer. As I've already stated, this is where there's a significant difference. If you declare a pointer and assign the address of the array to it, then you can use that pointer to access the members of the array.

TRY IT OUT: MULTIDIMENSIONAL ARRAYS AND POINTERS

You can see this in action here:

```
/* Program 7.9  Multidimensional arrays and pointers*/
#include <stdio.h>

int main(void)
{
  char board[3][3] = {
                       {'1','2','3'},
                       {'4','5','6'},
                       {'7','8','9'}
                     };

  char *pboard = *board;             /* A pointer to char */

  for(int i = 0; i < 9; i++)
    printf(" board: %c\n", *(pboard + i));
  return 0;
}
```

Here, you get the same output as before:

```
board: 1
board: 2
board: 3
board: 4
board: 5
board: 6
board: 7
board: 8
board: 9
```

How It Works

Here, you initialize `pboard` with the address of the first element of the array, and then you just use normal pointer arithmetic to move through the array:

```
  char *pboard = *board;             /* A pointer to char */

  for(int i = 0; i < 9; i++)
   printf(" board: %c\n", *(pboard + i));
```

Note how you dereference board to obtain the address you want (with *board), because board, by itself, is the address of the array board[0], not the address of an element. You could have initialized pboard by using the following:

```
char *pboard = &board[0][0];
```

This amounts to the same thing. You might think you could initialize pboard using this statement:

```
pboard = board;                    /* Wrong level of indirection! */
```

This is wrong. You should at least get a compiler warning if you do this. Strictly speaking, this isn't legal, because pboard and board have different levels of **indirection**. That's a great jargon phrase that just means that pboard refers to an address that contains a value of type char, whereas board refers to an address *that refers to an address* containing a value of type char. There's an extra level with board compared to pboard. Consequently, pboard needs one * to get to the value and board needs two. Some compilers will allow you to get away with this and just give you a warning about what you've done. However, it is an error, so you shouldn't do it!

Accessing Array Elements

Now you know that, for a two-dimensional array, you have several ways of accessing the elements in that array. Table 7-1 lists these ways of accessing your board array. The left column contains row index values to the board array, and the top row contains column index values. The entry in the table corresponding to a given row index and column index shows the various possible expressions for referring to that element.

Table 7-1. *Pointer Expressions for Accessing Array Elements*

board	0	1	2
0	board[0][0] *board[0] **board	board[0][1] *(board[0]+1) *(*board+1)	board[0][2] *(board[0]+2) *(*board+2)
1	board[1][0] *(board[0]+3) *board[1] *(*board+3)	board[1][1] *(board[0]+4) *(board[1]+1) *(*board+4)	board[1][2] *(board[0]+5) *(board[1]+2) *(*board+5)
2	board[2][0] *(board[0]+6) *(board[1]+3) *board[2] *(*board+6)	board[2][1] *(board[0]+7) *(board[1]+4) *(board[2]+1) *(*board+7)	board[2][2] *(board[0]+8) *(board[1]+5) *(board[2]+2) *(*board+8)

Let's see how you can apply what you've learned so far about pointers in a program that you previously wrote without using pointers. Then you'll be able to see how the pointer-based implementation differs. You'll recall that in Chapter 5 you wrote an example that worked out your hat size. Let's see how you could have done things a little differently.

TRY IT OUT: KNOW YOUR HAT SIZE REVISITED

Here's a rewrite of the hat sizes example using pointer notation:

```
/* Program 7.10  Understand pointers to your hat size - if you dare */
#include <stdio.h>
#include <stdbool.h>

int main(void)
{
  char size[3][12] = {               /* Hat sizes as characters */
       {'6', '6', '6', '6', '7', '7', '7', '7', '7', '7', '7', '7'},
       {'1', '5', '3', '7', ' ', '1', '1', '3', '1', '5', '3', '7'},
       {'2', '8', '4', '8', ' ', '8', '4', '8', '2', '8', '4', '8'}
                     };

   int headsize[12] =                 /* Values in 1/8 inches    */
       {164,166,169,172,175,178,181,184,188,191,194,197};

  char *psize = *size;
  int *pheadsize = headsize;

  float cranium = 0.0;               /* Head circumference in decimal inches */
  int your_head = 0;                 /* Headsize in whole eighths            */
  bool hat_found = false;            /* Indicates when a hat is found to fit */
  bool too_small = false;            /* Indicates headsize is too small      */

  /* Get the circumference of the head */
  printf("\nEnter the circumference of your head above your eyebrows"
                                    " in inches as a decimal value: ");
  scanf(" %f", &cranium);
  /* Convert to whole eighths of an inch */
  your_head = (int)(8.0*cranium);
  /* Search for a hat size */
  for(int i = 0 ; i < 12 ; i++)
  {
    /* Find head size in the headsize array */
    if(your_head > *(pheadsize+i))
      continue;

    /* If it is the first element and the head size is  */
    /* more than 1/8 smaller then the head is too small */
    /*  for a hat                                       */
    if((i == 0) && (your_head < (*pheadsize)-1))
    {
      printf("\nYou are the proverbial pinhead. No hat for"
                                           "you I'm afraid.\n");
      too_small = true;
      break;                         /* Exit the loop */
    }

    /* If head size is more than 1/8 smaller than the current */
    /* element in headsize array, take the next element down  */
```

```
      /* as the head size                                         */
    if( your_head < *(pheadsize+i)-1)
      i--;

    printf("\nYour hat size is %c %c%c%c\n",
                     *(psize + i),         /* First row of size  */
                     *(psize + 1*12 + i),  /* Second row of size */
                     (i==4) ?' ' : '/',
                     *(psize+2*12+i));     /* Third row of size  */
    hat_found=true;
    break;
  }
  if(!hat_found && !too_small)
    printf("\nYou, in technical parlance, are a fathead."
                                  " No hat for you, I'm afraid.\n");
  return 0;
}
```

The output from this program is the same as in Chapter 5, so I won't repeat it. It's the code that's of interest, so let's look at the new elements in this program.

How It Works

This program works in essentially the same way as the example from Chapter 5. The differences arise because the implementation is now in terms of the pointers `pheadsize` and `psize` that contain the addresses of the start of the `headsize` and `size` arrays respectively. The value in `your_head` is compared with the values in the array in the following statement:

```
    if(your_head > *(pheadsize+i))
      continue;
```

The expression on the right side of the comparison, `*(pheadsize+i)`, is equivalent to `headsize[i]` in array notation. The bit between the parentheses adds `i` to the address of the beginning of the array. Remember that adding an integer `i` to an address will add `i` times the length of each element. Therefore, the subexpression between parentheses produces the address of the element corresponding to the index value `i`. The dereference operator `*` then obtains the contents of this element for the comparison operation with the value in the variable `your_head`.

If you examine the `printf()` in the middle, you'll see the effect of two array dimensions on the pointer expression that access an element in a particular row:

```
    printf("\nYour hat size is %c %c%c%c\n",
                     *(psize + i),         /* First row of size  */
                     *(psize + 1*12 + i),  /* Second row of size */
                     (i==4) ?' ' : '/',
                     *(psize+2*12+i));     /* Third row of size  */
```

The first expression is `*(psize + i)`that accesses the `i`th element in the first row of size so this is equivalent to `size[0][i]`. The second expression is `*(psize + 1*12 + i)` that accesses the `i`th element in the second row of size so it is equivalent to `size[1][i]`. I have written the expression to show that the address of the start of the second row is obtained by adding the row size to `psize`. You then add `i` to that to get the element within the second row. To get the element in the third row of the size array you use the expression `*(psize + 2*12 + i)`, which is equivalent to `size[2][i]`.

Using Memory As You Go

Pointers are an extremely flexible and powerful tool for programming over a wide range of applications. The majority of programs in C use pointers to some extent. C also has a further facility that enhances the power of pointers and provides a strong incentive to use them in your code; it permits memory to be allocated dynamically when your program executes. Allocating memory dynamically is possible only because you have pointers available.

Think back to the program in Chapter 5 that calculated the average scores for a group of students. At the moment, it works for only ten students. Suppose you want to write the program so that it works for any number of students without knowing the number of students in the class in advance, and so it doesn't use any more memory than necessary for the number of student scores specified. **Dynamic memory allocation** allows you to do just that. You can create arrays at runtime that are large enough to hold the precise amount of data that you require for the task.

When you explicitly allocate memory at runtime in a program, space is reserved for you in a memory area called the **heap**. There's another memory area called the **stack** in which space to store function arguments and local variables in a function is allocated. When the execution of a function is finished, the space allocated to store arguments and local variables is freed. The memory in the heap is controlled by you. As you'll see in this chapter, when you allocate memory on the heap, it is up to you to keep track of when the memory you have allocated is no longer required and free the space you have allocated to allow it to be reused.

Dynamic Memory Allocation: The malloc() Function

The simplest standard library function that allocates memory at runtime is called `malloc()`. You need to include the `<stdlib.h>` header file in your program when you use this function. When you use the `malloc()` function, you specify the number of bytes of memory that you want allocated as the argument. The function returns the address of the first byte of memory allocated in response to your request. Because you get an address returned, a pointer is a useful place to put it.

A typical example of dynamic memory allocation might be this:

```
int *pNumber = (int *)malloc(100);
```

Here, you've requested 100 bytes of memory and assigned the address of this memory block to `pNumber`. As long as you haven't modified it, any time that you use the variable `pNumber`, it will point to the first `int` location at the beginning of the 100 bytes that were allocated. This whole block can hold 25 `int` values on my computer, where they require 4 bytes each.

Notice the cast, `(int *)`, that you use to convert the address returned by the function to the type "pointer to `int`." You've done this because `malloc()` is a general-purpose function that's used to allocate memory for any type of data. The function has no knowledge of what you want to use the memory for, so it actually returns a pointer of type "pointer to `void`," which, as I indicated earlier, is written as `void *`. Pointers of type `void *` can point to any kind of data. However, you can't dereference a pointer of type "pointer to `void`" because what it points to is unspecified. Many compilers will arrange for the address returned by `malloc()` to be automatically cast to the appropriate type, but it doesn't hurt to be specific.

You could request any number of bytes, subject only to the amount of free memory on the computer and the limit on `malloc()` imposed by a particular implementation. If the memory that you request can't be allocated for any reason, `malloc()` returns a pointer with the value `NULL`. Remember that this is the equivalent of 0 for pointers. It's always a good idea to check any dynamic memory request immediately using an `if` statement to make sure the memory is actually there before you try to use it. As with money, attempting to use memory you don't have is generally catastrophic. For that reason, writing

```
if(pNumber == NULL)
{
  /*Code to deal with no memory allocated */
}
```

with a suitable action if the pointer is `NULL` is a good idea. For example, you could at least display a message `"Not enough memory"` and terminate the program. This would be much better than allowing the program to continue, and crashing when it uses a `NULL` address to store something. In some instances, though, you may be able to free up a bit of memory that you've been using elsewhere, which might give you enough memory to continue.

Using the sizeof Operator in Memory Allocation

The previous example is all very well, but you don't usually deal in bytes; you deal in data of type `int`, type `double`, and so on. It would be very useful to allocate memory for 75 items of type `int`, for example. You can do this with the following statement:

```
pNumber = (int *) malloc(75*sizeof(int));
```

As you've seen already, `sizeof` is an operator that returns an unsigned integer of type `size_t` that's the count of the number of bytes required to store its argument. It will accept a type keyword such as `int` or `float` as an argument between parentheses, in which case the value it returns will be the number of bytes required to store an item of that type. It will also accept a variable or array name as an argument. With an array name as an argument, it returns the number of bytes required to store the whole array. In the preceding example, you asked for enough memory to store 75 data items of type `int`. Using `sizeof` in this way means that you automatically accommodate the potential variability of the space required for a value of type `int` between one C implementation and another.

TRY IT OUT: DYNAMIC MEMORY ALLOCATION

You can put the concept of dynamic memory allocation into practice by using pointers to help calculate prime numbers. In case you've forgotten, a prime number is an integer that's exactly divisible only by 1 or by the number itself.

The process for finding a prime is quite simple. First, you know by inspection that 2, 3, and 5 are the first three prime numbers, because they aren't divisible by any lower number other than 1. Because all the other prime numbers must be odd (otherwise they would be divisible by 2), you can work out the next number to check by starting at the last prime you have and adding 2. When you've checked out that number, you add another 2 to get the next to be checked, and so on.

To check whether a number is actually prime rather than just odd, you could divide by all the odd numbers less than the number that you're checking, but you don't need to do as much work as that. If a number is *not* prime, it must be divisible by one of the primes lower than the number you're checking. Because you'll obtain the primes in sequence, it will be sufficient to check a candidate by testing whether any of the primes that you've already found is an exact divisor.

You'll implement this program using pointers and dynamic memory allocation:

```
/* Program 7.11  A dynamic prime example         */
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

int main(void)
{
  unsigned long *primes = NULL;       /* Pointer to primes storage area    */
  unsigned long trial = 0;            /* Integer to be tested              */
  bool found = false;                /* Indicates when we find a prime    */
  size_t total = 0;                  /* Number of primes required         */
  size_t count = 0;                  /* Number of primes found            */

  printf("How many primes would you like - you'll get at least 4?  ");
  scanf("%u", &total);               /* Total is how many we need to find */
  total = total<4U ? 4U:total;       /* Make sure it is at least 4        */

  /* Allocate sufficient memory to store the number of primes required */
  primes = (unsigned long *)malloc(total*sizeof(unsigned long));
  if(primes == NULL)
  {
     printf("\nNot enough memory. Hasta la Vista, baby.\n");
     return 1;
  }

  /* We know the first three primes     */
  /* so let's give the program a start. */
  *primes = 2UL;                     /* First prime                   */
  *(primes+1) = 3UL;                 /* Second prime                  */
  *(primes+2) = 5UL;                 /* Third prime                   */
  count = 3U;                        /* Number of primes stored       */
  trial = 5U;                        /* Set to the last prime we have */

  /* Find all the primes required */
  while(count<total)
  {
    trial += 2UL;                    /* Next value for checking       */

    /* Try dividing by each of the primes we have       */
    /* If any divide exactly - the number is not prime  */
    for(size_t i = 0 ; i < count ; i++)
      if(!(found = (trial % *(primes+i))))
        break;                       /* Exit if no remainder          */

    if(found)                       /* we got one - if found is true */
      *(primes+count++) = trial;    /* Store it and increment count  */
  }

  /* Display primes 5-up */
  for(size_t i = 0 ; i < total ; i ++)
  {
    if(!(i%5U))
      printf("\n");                  /* Newline after every 5         */
    printf ("%12lu", *(primes+i));
  }
  printf("\n");                      /* Newline for any stragglers    */
  return 0;
}
```

The output from the program looks something like this:

```
How many primes would you like - you'll get at least 4? 25

          2           3           5           7          11
         13          17          19          23          29
         31          37          41          43          47
         53          59          61          67          71
         73          79          83          89          97
```

How It Works

With this example, you can enter the number of prime numbers you want the program to generate. The pointer variable `primes` refers to a memory area that will be used to store the prime numbers as they're calculated. However, no memory is defined initially in the program. The space is allocated after you've entered the number of primes that you want:

```
printf("How many primes would you like - you'll get at least 4?  ");
scanf("%u", &total);               /* Total is how many we need to find */
total = total<4U ? 4U:total;       /* Make sure it is at least 4       */
```

After the prompt, the number that you enter is stored in `total`. The next statement then ensures that `total` is at least 4. This is because you'll define and store the three primes that you know (2, 3, and 5) by default.

You then use the value in `total` to allocate the appropriate amount of memory to store the primes:

```
primes = (unsigned long *)malloc(total*sizeof(unsigned long));
if (primes == NULL)
{
   printf("\nNot enough memory. Hasta la Vista, baby.\n");
   return 0;
}
```

Primes grow in size faster than the count so you store them as type `unsigned long` although if you want to maximize the range you can deal with you could use `unsigned long long`. Because you're going to store each prime as type `long`, the number of bytes you require is `total*sizeof(unsigned long)`. If the `malloc()` function returns `NULL`, no memory was allocated, so you display a message and end the program.

The maximum number of primes that you can specify depends on two things: the memory available on your computer, and the amount of memory that your compiler's implementation of `malloc()` can allocate at one time. The former is probably the major constraint. The argument to `malloc()` is of type `size_t` so the integer type that corresponds to `size_t` will limit the number of bytes you can specify. If `size_t` corresponds to a 4-byte unsigned integer, you will be able to allocate up to 4,294,967,295 bytes at one time.

Once you have the memory allocated for the primes, you define the first three primes and store them in the first three positions in the memory area pointed to by `primes`:

```
*primes = 2UL;                     /* First prime                  */
*(primes+1) =3UL;                  /* Second prime                 */
*(primes+2) = 5UL;                 /* Third prime                  */
```

As you can see, referencing successive memory locations is simple. Because `primes` is of type "pointer to `unsigned long`," `primes+1` refers to the address of the second location—the address being `primes` plus the number of bytes required to store one data item of type `unsigned long`. To store each value, you use the indirection operator; otherwise, you would be modifying the address itself.

Now that you have three primes, you set the variable count to 3 and initialize the variable trial with the last prime you stored:

```
count = 3U;                          /* Number of primes stored       */
trial = 5UL;                         /* Set to the last prime we have */
```

The value in trial will be incremented by 2 to get the next value to be tested when you start searching for the next prime.

All the primes are found in the while loop:

```
while(count<total)
{
  ...
 }
```

The variable count is incremented within the loop as each prime is found, and when it reaches the value total, the loop ends.

Within the while loop, you first increase the value in trial by 2UL, and then you test whether the value is prime:

```
  trial += 2UL;                      /* Next value for checking       */

  /* Try dividing by each of the primes we have       */
  /* If any divide exactly - the number is not prime  */
  for(size_t i = 0 ; i < count ; i++)
    if(!(found = (trial % *(primes+i))))
      break;                         /* Exit if no remainder          */
```

The for loop does the testing. Within this loop the remainder after dividing trial by each of the primes that you have so far is stored in found. If the division is exact, the remainder will be 0, and therefore found will be set to false. If you find any remainder is 0, this means that the value in trial isn't a prime and you can continue with the next candidate.

The value of an assignment expression is the value that's stored in the variable on the left of the assignment operator. Thus, the value of the expression (found = (trial % *(primes+i))) will be the value that's stored in found as a result of this. This will be false for an exact division, so the expression !(found = (trial % *(primes+i))) will be true in this case, and the break statement will be executed. Therefore, the for loop will end if any previously stored prime divides into trial with no remainder.

If none of the primes divides into trial exactly, the for loop will end when all the primes have been tried, and found will contain the result of converting the last remainder value, which will be some positive integer, to type bool. If trial had a factor, the loop will have ended via the break statement and found will contain false. Therefore, you can use the value stored in found at the completion of the for loop to determine whether you've found a new prime:

```
   if(found)                           /* we got one - if found is true */
     *(primes+count++) = trial;        /* Store it and increment count  */
```

If found is true, you store the value of trial in the next available slot in the memory area. The address of the next available slot is primes+count. Remember that the first slot is primes, so when you have count number of primes, the last prime occupies the location primes+count-1. The statement storing the new prime also increments the value of count after the new prime has been stored.

The while loop just repeats the process until you have all the primes requested. You then output the primes five on a line: