Bradley Jones
Peter Aitken

SEVENTH EDITION

Covers C11

Sams Teach Yourself C in One Hour a Day

SAMS

Bradley L. Jones
Peter Aitken
Dean Miller

Sams **Teach Yourself**

C Programming

in **One Hour a Day**

Seventh Edition

SAMS | 800 East 96th Street, Indianapolis, Indiana 46240

What Is a Pointer?

To understand pointers, you need a basic knowledge of how your computer stores information in memory. The following is a somewhat simplified account of PC memory storage.

Your Computer's Memory

A PC's memory (RAM) consists of many millions of sequential storage locations, and each location is identified by a unique address. The memory addresses in a given computer range from zero to a maximum value that depends on the amount of memory installed.

When you use your computer, the operating system uses some of the system's memory. When you run a program, the program's code (the machine-language instructions for the program's various tasks) and data (the information the program uses) also use some of the system's memory. This section examines the memory storage for program data.

When you declare a variable in a C program, the compiler sets aside a memory location with a unique address to store that variable. The compiler associates that address with the variable's name. When your program uses the variable name, it automatically accesses the proper memory location. The location's address is used, but it is hidden from you, and you need not be concerned with it.

Figure 9.1 shows this schematically. A variable named `rate` has been declared and initialized to `100`. The compiler has set aside storage at address `1004` for the variable and has associated the name `rate` with the address `1004`.

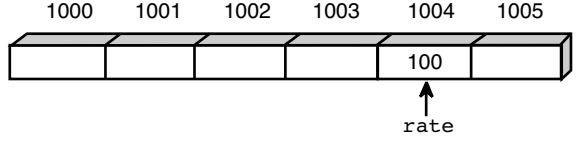

FIGURE 9.1
A program variable is stored at a specific memory address.

Creating a Pointer

You should note that the address of the variable `rate` (or any other variable) is a number and can be treated like any other number in C. If you know a variable's address, you can create a second variable in which to store the address of the first. The first step is to declare a variable to hold the address of `rate`. Give it the name `p_rate`, for example. At first, `p_rate` is uninitialized. Storage has been allocated for `p_rate`, but its value is undetermined, as shown in Figure 9.2.

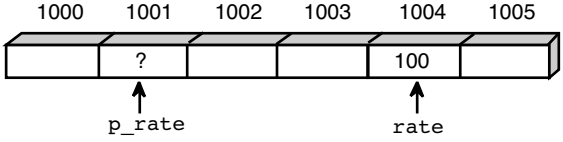

FIGURE 9.2
Memory storage space has been allocated for the variable `p_rate`.

The next step is to store the address of the variable `rate` in the variable `p_rate`. Because `p_rate` now contains the address of `rate`, it indicates the location in which `rate` is stored in memory. In C parlance, `p_rate` *points* to `rate`, or is a pointer to `rate`. This is shown in Figure 9.3.

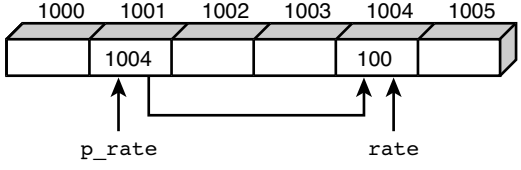

FIGURE 9.3
The variable `p_rate` contains the address of the variable `rate` and is therefore a pointer to `rate`.

To summarize, a pointer is a variable that contains the address of another variable. Now you can get down to the details of using pointers in your C programs.

Pointers and Simple Variables

In the example just given, a pointer variable pointed to a simple (that is, nonarray) variable. This section shows you how to create and use pointers to simple variables.

Declaring Pointers

A pointer is a numeric variable and, like all variables, must be declared before it can be used. Pointer variable names follow the same rules as other variables and must be unique. This lesson uses the convention that a pointer to the variable `name` is called `p_name`. This isn't necessary, however; you can name pointers anything you want as long as they follow C's naming rules.

A pointer declaration takes the following form:

```
typename *ptrname;
```

where *`typename`* is any of C's variable types and indicates the type of the variable that the pointer points to. The asterisk (`*`) is the indirection operator, and it indicates that *`ptrname`* is a pointer to type *`typename`* and not a variable of type *`typename`*. Pointers can be declared along with nonpointer variables. Here are some more examples:

```
char *ch1, *ch2;          /* ch1 and ch2 both are pointers to type char */
float *value, percent;    /* value is a pointer to type float, and
                          /* percent is an ordinary float variable */
```

NOTE

The * symbol is used as both the indirection operator and the multiplication operator. Don't worry about the compiler becoming confused. The context in which * is used always provides enough information for the compiler to figure out whether you mean indirection or multiplication.

Initializing Pointers

Now that you've declared a pointer, what can you do with it? You can't do anything with it until you make it point to something. Like regular variables, uninitialized pointers can be used, but the results are unpredictable and potentially disastrous. Until a pointer holds the address of a variable, it isn't useful. The address doesn't get stored in the pointer by magic; your program must put it there by using the address-of operator, the ampersand (&).When placed before the name of a variable, the address-of operator returns the address of the variable. Therefore, you initialize a pointer with a statement of the form

```
pointer = &variable;
```

Refer to the example in Figure 9.3. The program statement to initialize the variable `p_rate` to point at the variable `rate` would be

```
p_rate = &rate;     /* assign the address of rate to p_rate */
```

This statement assigns the address of `rate` to `p_rate`. Before the initialization, `p_rate` didn't point to anything in particular. After the initialization, `p_rate` is a pointer to `rate`.

Using Pointers

Now that you know how to declare and initialize pointers, you're probably wondering how to use them. The indirection operator (*) comes into play again. When the * precedes the name of a pointer, it refers to the variable pointed to.

Consider the previous example, in which the pointer `p_rate` has been initialized to point to the variable `rate`. If you write `*p_rate`, this pointer variable refers to the contents of the variable `rate`. If you want to print the value of `rate` (which is `100` in the example), you could write

```
printf("%d", rate);
```

or you could write this statement:

```
printf("%d", *p_rate);
```

In C, these two statements are equivalent. Accessing the contents of a variable by using the variable name is called *direct access*. Accessing the contents of a variable by using a pointer to the variable is called *indirect access* or *indirection*. Figure 9.4 shows that a pointer name preceded by the indirection operator refers to the value of the pointed-to variable.

FIGURE 9.4
Use of the indirection operator with pointers.

Pause a minute and think about this material. Pointers are an integral part of the C language, and it's essential that you understand them. Pointers have confused many people, so don't worry if you feel a bit puzzled. If you need to review, that's fine. Maybe the following summary can help.

If you have a pointer named `ptr` that has been initialized to point to the variable `var`, the following are true:

- `*ptr` and `var` both refer to the contents of `var` (that is, whatever value the program has stored there).
- `ptr` and `&var` refer to the address of `var`.

As you can see, a pointer name without the indirection operator accesses the pointer value itself, which is, of course, the address of the variable pointed to.

Listing 9.1 demonstrates basic pointer use. You should enter, compile, and run this program.

Input ▼

LISTING 9.1 pointer.c: Basic Pointer Use

```
1:  /* Demonstrates basic pointer use. */
2:
3:  #include <stdio.h>
4:
5:  /* Declare and initialize an int variable */
6:
7:  int var = 1;
```

```
8:
9:  /* Declare a pointer to int */
10:
11: int *ptr;
12:
13: int main( void )
14: {
15:     /* Initialize ptr to point to var */
16:
17:     ptr = &var;
18:
19:     /* Access var directly and indirectly */
20:
21:     printf("\nDirect access, var = %d", var);
22:     printf("\nIndirect access, var = %d", *ptr);
23:
24:     /* Display the address of var two ways */
25:
26:     printf("\n\nThe address of var = %p", &var);
27:     printf("\nThe address of var = %p\n", ptr);
28:
29:     return 0;
30: }
```

Output ▼

```
Direct access, var = 1
Indirect access, var = 1

The address of var = 4202496
The address of var = 4202496
```

NOTE

The address reported for `var` will probably not be `4202496` on your system.

Analysis ▼

In this listing, two variables are declared. On line 7, `var` is declared as an `int` and initialized to `1`. On line 11, a pointer to a variable of type `int` is declared and named `ptr`. On line 17, the pointer `ptr` is assigned the address of `var` using the address-of operator (`&`). The rest of the program prints the values from these two variables to the screen. Line 21 prints the value of `var`, whereas line 22 prints the value stored in the location pointed to by `ptr`. In this program, this value is `1`. Line 26 prints the address of `var` using the address-of operator. This is the same value printed by line 27 using the pointer variable, `ptr`.

This listing is good to study. It shows the relationship between a variable, its address, a pointer, and the dereferencing of a pointer.

DO	DON'T
DO understand what pointers are and how they work. The mastering of C requires mastering pointers.	**DON'T** use an uninitialized pointer until an address has been assigned. Results can be disastrous if you do.

Pointers and Variable Types

The previous discussion ignores the fact that different variable types occupy different amounts of memory. For the more common PC operating systems, a `short` takes 2 bytes, a `float` takes 4 bytes, and so on. Each individual byte of memory has its own address, so a multibyte variable actually occupies several addresses.

How, then, do pointers handle the addresses of multibyte variables? Here's how it works: The address of a variable is actually the address of the first (lowest) byte it occupies. This can be illustrated with an example that declares and initializes three variables:

```
short vshort = 12252;
char vchar = 90;
float vfloat = 1200.156004;
```

These variables are stored in memory, as shown in Figure 9.5. In this figure, the `short` variable occupies 2 bytes, the `char` variable occupies 1 byte, and the `float` variable occupies 4 bytes.

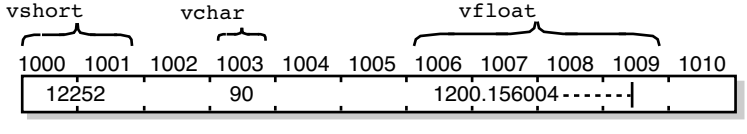

FIGURE 9.5 Different types of numeric variables occupy different amounts of storage space in memory.

Now declare and initialize pointers to these three variables:

```
int *p_vshort;
char *p_vchar;
float *p_vfloat;
/* additional code goes here */
p_vshort = &vshort;
p_vchar = &vchar;
p_vfloat = &vfloat;
```