PEARSON

C Primer Plus

第6版

英文版（下册）

[美] Stephen Prata 著

C Primer Plus (6th Edition)

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

C Primer Plus

第6版

英文版（下册）

[美] Stephen Prata 著

C Primer Plus (6th Edition)

人民邮电出版社
北 京

目录

12

Storage Classes, Linkage, and Memory Management

You will learn about the following in this chapter:

- Keywords:

  `auto`, `extern`, `static`, `register`, `const`, `volatile`, `restricted`, `_Thread_local`, `_Atomic`
- Functions:

  `rand()`, `srand()`, `time()`, `malloc()`, `calloc()`, `free()`
- How C allows you to determine the scope of a variable (how widely known it is) and the lifetime of a variable (how long it remains in existence)
- Designing more complex programs

One of C's strengths is that it enables you to control a program's fine points. C's memory management system exemplifies that control by letting you determine which functions know which variables and for how long a variable persists in a program. Using memory storage is one more element of program design.

Storage Classes

C provides several different models, or *storage classes*, for storing data in memory. To understand the options, it's helpful to go over a few concepts and terms first.

Every programming example in this book stores data in memory. There is a hardware aspect to this—each stored value occupies physical memory. C literature uses the term *object* for such a chunk of memory. An object can hold one or more values. An object might not yet actually have a stored value, but it will be of the right size to hold an appropriate value. (The phrase *object-oriented programming* uses the word *object* in a more developed sense to indicate class

objects, whose definitions encompass both data and permissible operations on the data; C is not an object-oriented programming language.)

There also is a software aspect—the program needs a way to access the object. This can be accomplished, for instance, by declaring a variable:

```
int entity = 3;
```

This declaration creates an *identifier* called `entity`. An identifier is a name, in this case one that can be used to designate the contents of a particular object. Identifiers satisfy the naming conventions for variables discussed in Chapter 2, "Introducing C." In this case, the identifier `entity` is how the software (the C program) designates the object that's stored in hardware memory. This declaration also provides a value to be stored in the object.

A variable name isn't the only way to designate an object. For instance, consider the following declarations:

```
int * pt = &entity;
int ranks[10];
```

In the first case, `pt` is an identifier. It designates an object that holds an address. Next, the expression `*pt` is not an identifier because it's not a name. However, it does designate an object, in this case the same object that `entity` designates. In general, as you may recall from Chapter 3, "Data and C," an expression that designates an object is called an lvalue. So `entity` is an identifier that is an lvalue, and `*pt` is an expression that is an lvalue. Along the same lines, the expression `ranks + 2 * entity` is neither an identifier (not a name) nor an lvalue (doesn't designate the contents of a memory location). But the expression `*(ranks + 2 * entity)` is an lvalue because it does designate the value of a particular memory location, the seventh element of the `ranks` array. The declaration of `ranks`, by the way, creates an object capable of holding ten `int`s, and each member of the array also is an object.

If, as with all these examples, you can use the lvalue to change the value in an object, it's a *modifiable lvalue*. Now consider this declaration:

```
const char * pc = "Behold a string literal!";
```

This causes the program to store the string literal contents in memory, and that array of character values is an object. Each character in the array also is an object, as it can be accessed individually. The declaration also creates an object having the identifier `pc` and holding the address of that string. The identifier `pc` is a modifiable lvalue because it can be reset to point to a different string. The `const` prevents you from altering the contents of a pointed-to string but not from changing which string is pointed to. So `*pc`, which designates the data object holding the `'B'` character, is an lvalue, but not a modifiable lvalue. Similarly, the string literal itself, because it designates the object holding the character string, is an lvalue, but not a modifiable one.

You can describe an object in terms of its *storage duration*, which is how long it stays in memory. You can describe an identifier used to access the object by its *scope* and its *linkage*, which together indicate which parts of a program can use it. The different storage classes offer

different combinations of scope, linkage, and storage duration. You can have identifiers that can be shared over several files of source code, identifiers that can be used by any function in one particular file, identifiers that can be used only within a particular function, and even identifiers that can be used only within a subsection of a function. You can have objects that exist for the duration of a program and objects that exist only while the function containing them is executing. With concurrent programming, you can have objects that exist for the duration of a particular thread. You also can store data in memory that is allocated and freed explicitly by means of function calls.

Next, let's investigate the meaning of the terms *scope*, *linkage*, and *storage duration*. After that, we'll return to specific storage classes.

Scope

Scope describes the region or regions of a program that can access an identifier. A C variable has one of the following scopes: block scope, function scope, function prototype scope, or file scope. The program examples to date have used block scope almost exclusively for variables. A *block*, as you'll recall, is a region of code contained within an opening brace and the matching closing brace. For instance, the entire body of a function is a block. Any compound statement within a function also is a block. A variable defined inside a block has *block scope*, and it is visible from the point it is defined until the end of the block containing the definition. Also, formal function parameters, even though they occur before the opening brace of a function, have block scope and belong to the block containing the function body. So the local variables we've used to date, including formal function parameters, have block scope. Therefore, the variables `cleo` and `patrick` in the following code both have block scope extending to the closing brace:

```
double blocky(double cleo)
{
    double patrick = 0.0;
    ...
    return patrick;
}
```

Variables declared in an inner block have scope restricted just to that block:

```
double blocky(double cleo)
{
    double patrick = 0.0;
    int i;
    for (i = 0; i < 10; i++)
    {
        double q = cleo * i; // start of scope for q
        ...
        patrick *= q;
    }                        // end of scope for q
    ...
```

```
    return patrick;
}
```

In this example, the scope of `q` is limited to the inner block, and only code within that block can access `q`.

Traditionally, variables with block scope had to be declared at the beginning of a block. C99 relaxed that rule, allowing you to declare variables anywhere in a block. One new possibility is in the control section of a `for` loop. That is, you now can do this:

```
for (int i = 0; i < 10; i++)
    printf("A C99 feature: i = %d", i);
```

As part of this new feature, C99 expanded the concept of a block to include the code controlled by a `for` loop, `while` loop, `do while` loop, or `if` statement, even if no brackets are used. So in the previous `for` loop, the variable `i` is considered to be part of the `for` loop block. Therefore, its scope is limited to the `for` loop. After execution leaves the `for` loop, the program will no longer see that `i`.

Function scope applies just to labels used with `goto` statements. This means that even if a label first appears inside an inner block in a function, its scope extends to the whole function. It would be confusing if you could use the same label inside two separate blocks, and function scope for labels prevents this from happening.

Function prototype scope applies to variable names used in function prototypes, as in the following:

```
int mighty(int mouse, double large);
```

Function prototype scope runs from the point the variable is defined to the end of the prototype declaration. What this means is that all the compiler cares about when handling a function prototype argument is the types; the names you use, if any, normally don't matter, and they needn't match the names you use in the function definition. One case in which the names matter a little is with variable-length array parameters:

```
void use_a_VLA(int n, int m, ar[n][m]);
```

If you use names in the brackets, they have to be names declared earlier in the prototype.

A variable with its definition placed outside of any function has *file scope*. A variable with file scope is visible from the point it is defined to the end of the file containing the definition. Take a look at this example:

```
#include <stdio.h>
int units = 0;          /* a variable with file scope */
void critic(void);
int main(void)
{
    ...
}
```