Operators	Associativity	Type
`()` `[]` `++` *(postfix)* `--` *(postfix)*	left to right	postfix
`+` `-` `++` `--` `!` `*` `&` `(`*type*`)`	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `s>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 7.5 | Precedence and associativity of the operators discussed so far.

7.4 Passing Arguments to Functions by Reference

There are two ways to pass arguments to a function—**pass-by-value** and **pass-by-reference**. *However, all arguments in C are passed by value*. Functions often require the capability to *modify variables in the caller* or receive a pointer to a large data object to avoid the overhead of receiving the object by value (which incurs the time and memory overheads of making a copy of the object). As we saw in Chapter 5, `return` may be used to return *one* value from a called function to a caller (or to return control from a called function without passing back a value). Pass-by-reference also can be used to enable a function to "return" multiple values to its caller by modifying variables in the caller.

*Use & and * to Accomplish Pass-By-Reference*

In C, you use pointers and the indirection operator to accomplish pass-by-reference. When calling a function with arguments that should be modified, the *addresses* of the arguments are passed. This is normally accomplished by applying the address operator (`&`) to the variable (in the caller) whose value will be modified. As we saw in Chapter 6, arrays are *not* passed using operator `&` because C automatically passes the starting location in memory of the array (the name of an array is equivalent to `&arrayName[0]`). When the address of a variable is passed to a function, the indirection operator (`*`) may be used in the function to modify the value at that location in the caller's memory.

Pass-By-Value

The programs in Figs. 7.6 and 7.7 present two versions of a function that cubes an integer—`cubeByValue` and `cubeByReference`. Line 14 of Fig. 7.6 passes the variable `number` by value to function `cubeByValue`. The `cubeByValue` function cubes its argument and passes the new value back to `main` using a `return` statement. The new value is assigned to `number` in `main` (line 14).

```c
// Fig. 7.6: fig07_06.c
// Cube a variable using pass-by-value.
#include <stdio.h>

int cubeByValue(int n); // prototype

int main(void)
{
   int number = 5; // initialize number

   printf("The original value of number is %d", number);

   // pass number by value to cubeByValue
   number = cubeByValue(number);

   printf("\nThe new value of number is %d\n", number);
}

// calculate and return cube of integer argument
int cubeByValue(int n)
{
   return n * n * n; // cube local variable n and return result
}
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 7.6 | Cube a variable using pass-by-value.

Pass-By-Reference

Figure 7.7 passes the variable `number` by reference (line 15)—the address of `number` is passed—to function `cubeByReference`. Function `cubeByReference` takes as a parameter a pointer to an `int` called `nPtr` (line 21). The function *dereferences* the pointer and cubes the value to which `nPtr` points (line 23), then assigns the result to `*nPtr` (which is really `number` in `main`), thus changing the value of `number` in `main`. Figures 7.8 and 7.9 analyze graphically and step-by-step the programs in Figs. 7.6 and 7.7, respectively.

```c
// Fig. 7.7: fig07_07.c
// Cube a variable using pass-by-reference with a pointer argument.

#include <stdio.h>

void cubeByReference(int *nPtr); // function prototype

int main(void)
{
```

Fig. 7.7 | Cube a variable using pass-by-reference with a pointer argument. (Part 1 of 2.)

```
10      int number = 5; // initialize number
11
12      printf("The original value of number is %d", number);
13
14      // pass address of number to cubeByReference
15      cubeByReference(&number);
16
17      printf("\nThe new value of number is %d\n", number);
18   }
19
20   // calculate cube of *nPtr; actually modifies number in main
21   void cubeByReference(int *nPtr)
22   {
23      *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24   }
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 7.7 | Cube a variable using pass-by-reference with a pointer argument. (Part 2 of 2.)

Use a Pointer Parameter to Receive an Address

A function receiving an *address* as an argument must define a *pointer parameter* to receive the address. For example, in Fig. 7.7 the header for function `cubeByReference` (line 21) is:

```
void cubeByReference(int *nPtr)
```

The header specifies that `cubeByReference` *receives* the *address* of an integer variable as an argument, stores the address locally in `nPtr` and does not return a value.

Pointer Parameters in Function Prototypes

The function prototype for `cubeByReference` (Fig. 7.7, line 6) specifies an `int *` parameter. As with other variable types, it's *not* necessary to include names of pointers in function prototypes. Names included for documentation purposes are ignored by the C compiler.

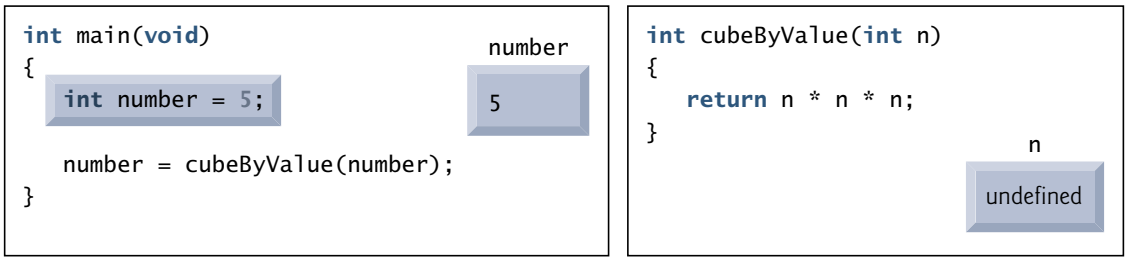
Functions That Receive One-Dimensional Arrays

For a function that expects a one-dimensional array as an argument, the function's prototype and header can use the pointer notation shown in the parameter list of function `cubeByReference` (line 21). The compiler does not differentiate between a function that receives a pointer and one that receives a one-dimensional array. This, of course, means that the function must "know" when it's receiving an array or simply a single variable for which it's to perform pass-by-reference. When the compiler encounters a function parameter for a one-dimensional array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`. The two forms are interchangeable.

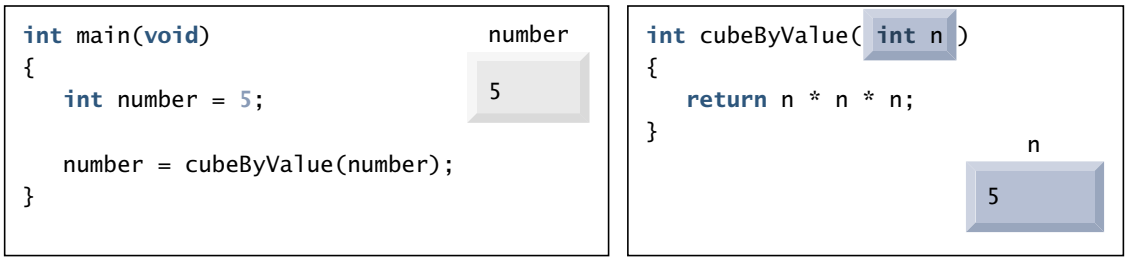
Error-Prevention Tip 7.2

Use pass-by-value to pass arguments to a function unless the caller explicitly requires the called function to modify the value of the argument variable in the caller's environment. This prevents accidental modification of the caller's arguments and is another example of the principle of least privilege.

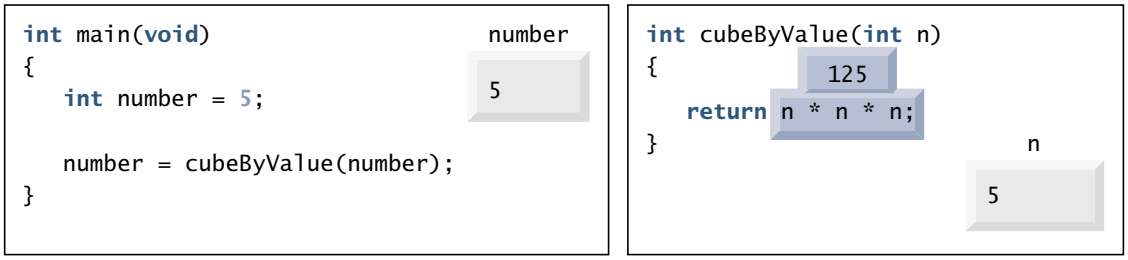
Step 1: Before `main` calls `cubeByValue`:



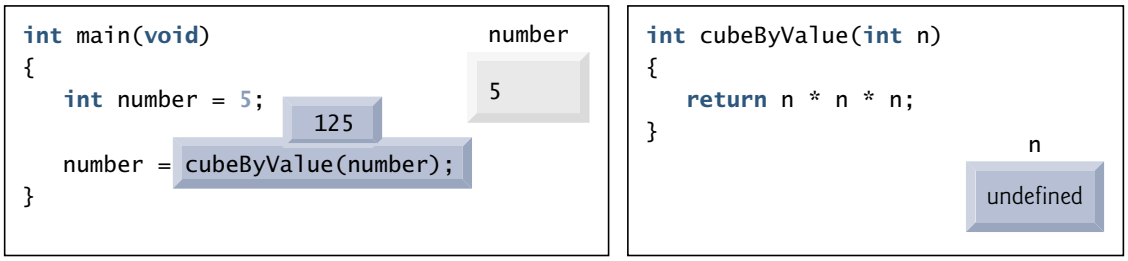
Step 2: After `cubeByValue` receives the call:



Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:



Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:



Step 5: After `main` completes the assignment to `number`:

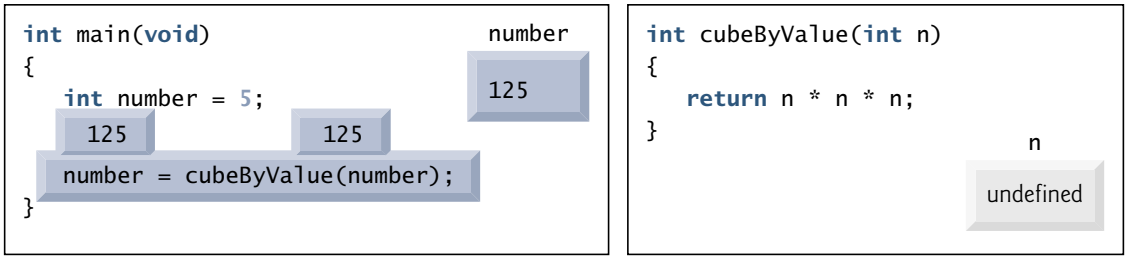


Fig. 7.8 | Analysis of a typical pass-by-value.

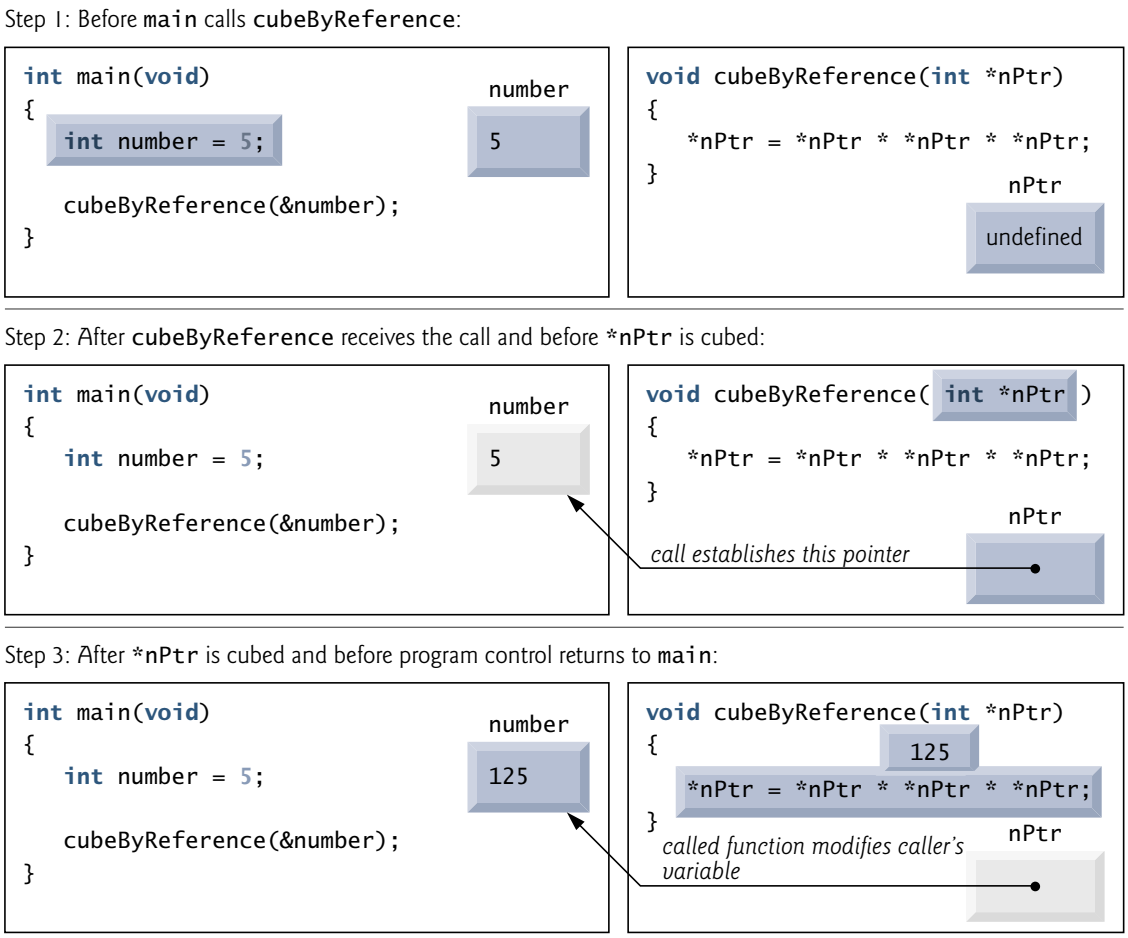


Fig. 7.9 | Analysis of a typical pass-by-reference with a pointer argument.

7.5 Using the const Qualifier with Pointers

The **const qualifier** enables you to inform the compiler that the value of a particular variable should not be modified.

Software Engineering Observation 7.1

The const qualifier can be used to enforce the principle of least privilege in software design. This can reduce debugging time and prevent unintentional side effects, making a program easier to modify and maintain.

Over the years, a large base of legacy code was written in early versions of C that did not use const because it was not available. For this reason, there are significant opportunities for improvement by reengineering old C code.

Six possibilities exist for using (or not using) const with function parameters—two with pass-by-value parameter passing and four with pass-by-reference parameter passing. How do you choose one of the six possibilities? Let the **principle of least privilege** be your guide—always award a function enough access to the data in its parameters to accomplish its specified task, but absolutely no more.