Chapter 18 Getting More from Functions

18-1. Passing Pointers to Functions

The default method of passing an argument to a function is by value. *Passing by value* means that the function is passed a copy of the argument's value. This method has three steps:

1. The argument expression is evaluated.
2. The result is copied onto the *stack,* a temporary storage area in memory.
3. The function retrieves the argument's value from the stack.

Figure 18.1 illustrates passing an argument by value. In this case, the argument is a simple type int variable, but the principle is the same for other variable types and more complex expressions.

When a variable is passed to a function by value, the function has access to the variable's value but not to the original copy of the variable. As a result, the code in the function can't modify the original variable. This is the main reason why passing by value is the default method of passing arguments: Data outside a function is protected from inadvertent modification.

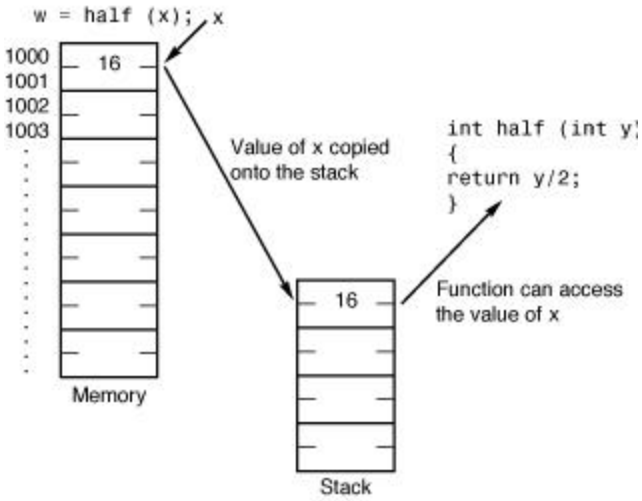

Figure 18-1. Passing an argument by value. The function can't modify the original argument variable.

Passing arguments by value is possible with the basic data types (char, int, long, float, and double) and structures. There is another way to pass an argument to a function, however: by passing a pointer to the argument variable rather than the value of the variable itself. This method of passing an argument is called *passing by reference*.

As you learned on Day 9, "Understanding Pointers," passing by reference is the only way to pass an array to a function; passing an array by value is not possible. With other data types, however, you can use either method. If your program uses large structures, passing them by value might cause your program to run out of stack space. Aside from this consideration, passing an argument by reference instead of by value offers an advantage as well as a disadvantage:

- The advantage of passing by reference is that the function can modify the value of the argument variable.
- The disadvantage of passing by reference is that the function can modify the value of the argument variable.

"What?" you might be saying. "An advantage that's also a disadvantage?" Yes. It all depends on the specific situation. If your program requires that a function modify an argument variable, passing by reference is an advantage. If there is no such need, it is a disadvantage because of the possibility of inadvertent modifications.

You might be wondering why you don't use the function's return value to modify the argument variable. You can do this, of course, as shown in the following example:

```
x = half(x);
int half(int y)
{
return y/2;
}
```

Remember, however, that a function can return only a single value. By passing one or more arguments by reference, you allow a function to "return" more than one value to the calling program. Figure 18.2 illustrates passing by reference for a single argument.

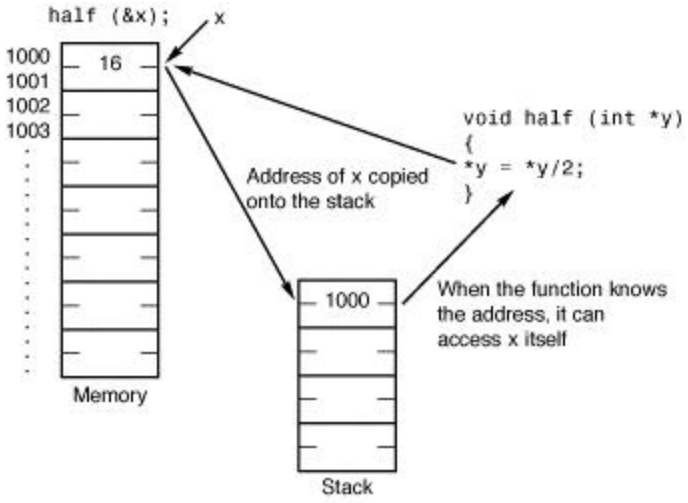

Figure 18-2. Passing by reference allows the function to modify the original argument's variable.

The function used in Figure 18.2 is not a good example of something you would use passing by reference for in a real program, but it does illustrate the concept. When you pass by reference, you must ensure that the function definition and prototype reflect the fact that the argument passed to the function is a pointer. Within the body of the function, you must also use the indirection operator to access the variable(s) passed by reference.

Listing 18.1 demonstrates passing by reference and the default passing by value. Its output clearly shows that a variable passed by value can't be changed by the function, whereas a variable passed by reference can be changed. Of course, a function doesn't need to modify a variable passed by reference. In such as case, there's no reason to pass by reference.

Listing 18.1. Passing by value and passing by reference.

```
1: /* Passing arguments by value and by reference. */
2:
3: #include <stdio.h>
4:
5: void by_value(int a, int b, int c);
6: void by_ref(int *a, int *b, int *c);
7:
8: main()
9: {
10:     int x = 2, y = 4, z = 6;
11:
12:     printf("\nBefore calling by_value(), x = %d, y = %d, z = %d.",
13:         x, y, z);
14:
15:     by_value(x, y, z);
16:
17:     printf("\nAfter calling by_value(), x = %d, y = %d, z = %d.",
18:            x, y, z);
19:
20:     by_ref(&x, &y, &z);
21:     printf("\nAfter calling by_ref(), x = %d, y = %d, z = %d.\n",
22:            x, y, z);
23:     return(0);
24: }
25:
26: void by_value(int a, int b, int c)
27: {
28:     a = 0;
```

```
29:     b = 0;
30:     c = 0;
31: }
32:
33: void by_ref(int *a, int *b, int *c)
34: {
35:     *a = 0;
36:     *b = 0;
37:     *c = 0;
38: }
Before calling by_value(), x = 2, y = 4, z = 6.
After calling by_value(), x = 2, y = 4, z = 6.
After calling by_ref(), x = 0, y = 0, z = 0.
```

ANALYSIS: This program demonstrates the difference between passing variables by value and passing them by reference. Lines 5 and 6 contain prototypes for the two functions called in the program. Notice that line 5 describes three arguments of type int for the by_value() function, but by_ref() differs on line 6 because it requires three pointers to type int variables as arguments. The function headers for these two functions on lines 26 and 33 follow the same format as the prototypes. The bodies of the two functions are similar, but not the same. Both functions assign 0 to the three variables passed to them. In the by_value() function, 0 is assigned directly to the variables. In the by_ref() function, pointers are used, so the variables must be dereferenced.

Each function is called once by main(). First, the three variables to be passed are assigned values other than 0 on line 10. Line 12 prints these values to the screen. Line 15 calls the first of the two functions, by_value(). Line 17 prints the three variables again. Notice that they are not changed. The by_value() function received the variables by value and therefore couldn't change their original content. Line 20 calls by_ref(), and line 22 prints the values again. This time, the values have all changed to 0. Passing the variables by reference gave by_ref() access to the actual contents of the variables.

You can write a function that receives some arguments by reference and others by value. Just remember to keep them straight inside the function, using the indirection operator (*) to dereference arguments passed by reference.

DON'T pass large amounts of data by value if it isn't necessary. You could run out of stack space.

DO pass variables by value if you don't want the original value altered.

DON'T forget that a variable passed by reference should be a pointer. Also, use the indirection operator to dereference the variable in the function.

18-2. Type void Pointers

You've seen the void keyword used to specify that a function either doesn't take arguments or doesn't return a value. The void keyword can also be used to create a generic pointer--a pointer that can point to any type of data object. For example, the statement

```
void *x;
```

declares x as a generic pointer. x points to something; you just haven't yet specified what.

The most common use for type void pointers is in declaring function parameters. You might want to create a function that can handle different types of arguments. You can pass it a type int one time, a type float the next time,

and so on. By declaring that the function takes a void pointer as an argument, you don't restrict it to accepting only a single data type. If you declare the function to take a void pointer as an argument, you can pass the function a pointer to anything.

Here's a simple example: You want a function that accepts a numeric variable as an argument and divides it by 2, returning the answer in the argument variable. Thus, if the variable x holds the value 4, after a call to half(x) the variable x is equal to 2. Because you want to modify the argument, you pass it by reference. Because you want to use the function with any of C's numeric data types, you declare the function to take a void pointer:

```
void half(void *x);
```

Now you can call the function, passing it any pointer as an argument. There's one more thing you need, however. Although you can pass a void pointer without knowing what data type it points to, you can't dereference the pointer. Before the code in the function can do anything with the pointer, it must know the data type. You do this with a *typecast,* which is nothing more than a way of telling the program to treat this void pointer as a pointer to type. If x is a void pointer, you typecast it as follows:

```
(type *)x
```

Here, type is the appropriate data type. To tell the program that x is a pointer to type int, write

```
(int *)x
```

To dereference the pointer--that is, to access the int that x points to--write

```
*(int *)x
```

Typecasts are covered in more detail on Day 20, "Working with Memory." Getting back to the original topic (passing a void pointer to a function), you can see that, to use the pointer, the function must know the data type to which it points. In the case of the function you're writing to divide its argument by two, there are four possibilities for type: int, long, float, and double. In addition to the void pointer to the variable to be divided by two, you must tell the function the type of variable to which the void pointer points. You can modify the function definition as follows:

```
void half(void *x, char type);
```

Based on the argument type, the function casts the void pointer x to the appropriate type. Then the pointer can be dereferenced, and the value of the pointed-to variable can be used. The final version of the half() function is shown in Listing 18.2.

Listing 18.2. Using a void pointer to pass different data types to a function.

```
1: /* Using type void pointers. */
2:
3: #include <stdio.h>
4:
5: void half(void *x, char type);
6:
7: main()
8: {
9:     /* Initialize one variable of each type. */
10:
11:     int i = 20;
12:     long l = 100000;
13:     float f = 12.456;
```

```
14:     double d = 123.044444;
15:
16:     /* Display their initial values. */
17:
18:     printf("\n%d", i);
19:     printf("\n%ld", l);
20:     printf("\n%f", f);
21:     printf("\n%lf\n\n", d);
22:
23:     /* Call half() for each variable. */
24:
25:     half(&i, `i');
26:     half(&l, `l');
27:     half(&d, `d');
28:     half(&f, `f');
29:
30:     /* Display their new values. */
31:     printf("\n%d", i);
32:     printf("\n%ld", l);
33:     printf("\n%f", f);
34:     printf("\n%lf\n", d);
35:     return(0);
36: }
37:
38: void half(void *x, char type)
39: {
40:     /* Depending on the value of type, cast the */
41:     /* pointer x appropriately and divide by 2. */
42:
43:     switch (type)
44:     {
45:         case `i':
46:             {
47:             *((int *)x) /= 2;
48:             break;
49:             }
50:         case `l':
51:             {
52:             *((long *)x) /= 2;
53:             break;
54:             }
55:         case `f':
56:             {
57:             *((float *)x) /= 2;
58:             break;
59:             }
60:         case `d':
61:             {
62:             *((double *)x) /= 2;
63:             break;
64:             }
```

```
65:     }
66: }
20
100000
12.456000
123.044444
10
50000
6.228000
61.522222
```

ANNALYSIS: As implemented, the function half() on lines 38 through 66 includes no error checking (for example, if an invalid type argument is passed). This is because in a real program you wouldn't use a function to perform a task as simple as dividing a value by 2. This is an illustrative example only.

You might think that the need to pass the type of the pointed-to variable would make the function less flexible. The function would be more general if it didn't need to know the type of the pointed-to data object, but that's not the way C works. You must always cast a void pointer to a specific type before you dereference it. By taking this approach, you write only one function. If you don't make use of a void pointer, you need to write four separate functions--one for each data type.

When you need a function that can deal with different data types, you often can write a macro to take the place of the function. The example just presented--in which the task performed by the function is relatively simple--would be a good candidate for a macro. (Day 21, "Advanced Compiler Use," covers macros.)

DO cast a void pointer when you use the value it points to.

DON'T try to increment or decrement a void pointer.

18-3. Functions That Have Variable Numbers of Arguments

You have used several library functions, such as printf() and scanf(), that take a variable number of arguments. You can write your own functions that take a variable argument list. Programs that have functions with variable argument lists must include the header file STDARG.H.

When you declare a function that takes a variable argument list, you first list the fixed parameters--those that are always present (there must be at least one fixed parameter). You then include an ellipsis (...) at the end of the parameter list to indicate that zero or more additional arguments are passed to the function. During this discussion, please remember the distinction between a parameter and an argument, as explained on Day 5, "Functions: The Basics."

How does the function know how many arguments have been passed to it on a specific call? You tell it. One of the fixed parameters informs the function of the total number of arguments. For example, when you use the printf() function, the number of conversion specifiers in the format string tells the function how many additional arguments to expect. More directly, one of the function's fixed arguments can be the number of additional arguments. The example you'll see in a moment uses this approach, but first you need to look at the tools that C provides for dealing with a variable argument list.

The function must also know the type of each argument in the variable list. In the case of printf(), the conversion specifiers indicate the type of each argument. In other cases, such as the following example, all arguments in the variable list are of the same type, so there's no problem. To create a function that accepts different types in the

variable argument list, you must devise a method of passing information about the argument types. For example, you could use a character code, as was done in the function half() in Listing 18.2.

The tools for using a variable argument list are defined in STDARG.H. These tools are used within the function to retrieve the arguments in the variable list. They are as follows:

va_list	A pointer data type.
va_start()	A macro used to initialize the argument list.
va_arg()	A macro used to retrieve each argument, in turn, from the variable list.
va_end()	A macro used to "clean up" when all arguments have been retrieved.

I've outlined how these macros are used in a function, and then I've included an example. When the function is called, the code in the function must follow these steps to access its arguments:

1. Declare a pointer variable of type va_list. This pointer is used to access the individual arguments. It is common practice, although certainly not required, to call this variable arg_ptr.
2. Call the macro va_start(), passing it the pointer arg_ptr as well as the name of the last fixed argument. The macro va_start() has no return value; it initializes the pointer arg_ptr to point at the first argument in the variable list.
3. To retrieve each argument, call va_arg(), passing it the pointer arg_ptr and the data type of the next argument. The return value of va_arg() is the value of the next argument. If the function has received n arguments in the variable list, call va_arg() n times to retrieve the arguments in the order listed in the function call.
4. When all the arguments in the variable list have been retrieved, call va_end(), passing it the pointer arg_ptr. In some implementations, this macro performs no action, but in others, it performs necessary clean-up actions. You should get in the habit of calling va_end() in case you use a C implementation that requires it.

Now for that example. The function average() in Listing 18.3 calculates the arithmetic average of a list of integers. This program passes the function a single fixed argument, indicating the number of additional arguments followed by the list of numbers.

Listing 18.3. Using a variable-size argument list.

```
1: /* Functions with a variable argument list. */
2:
3: #include <stdio.h>
4: #include <stdarg.h>
5:
6: float average(int num, ...);
7:
8: main()
9: {
10:     float x;
11:
12:     x = average(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
13:     printf("\nThe first average is %f.", x);
14:     x = average(5, 121, 206, 76, 31, 5);
15:     printf("\nThe second average is %f.\n", x);
16:     return(0);
17: }
18:
19: float average(int num, ...)
```

```
20: {
21:     /* Declare a variable of type va_list. */
22:
23:     va_list arg_ptr;
24:     int count, total = 0;
25:
26:     /* Initialize the argument pointer. */
27:
28:     va_start(arg_ptr, num);
29:
30:     /* Retrieve each argument in the variable list. */
31:
32:     for (count = 0; count < num; count++)
33:         total += va_arg( arg_ptr, int );
34:
35:     /* Perform clean up. */
36:
37:     va_end(arg_ptr);
38:
39:     /* Divide the total by the number of values to get the */
40:     /* average. Cast the total to type float so the value */
41:     /* returned is type float. */
42:
43:     return ((float)total/num);
44: }
The first average is 5.500000.
The second average is 87.800000.
```

ANALYSIS: The function average() is first called on line 19. The first argument passed, the only fixed argument, specifies the number of values in the variable argument list. In the function, as each argument in the variable list is retrieved on lines 32 through 33, it is added to the variable total. After all arguments have been retrieved, line 43 casts total as type float and then divides total by num to obtain the average.

Two other things should be pointed out in this listing. Line 28 calls va_start() to initialize the argument list. This must be done before the values are retrieved. Line 37 calls va_end() to "clean up," because the function is done with the values. You should use both of these functions in your programs whenever you write a function with a variable number of arguments.

Strictly speaking, a function that accepts a variable number of arguments doesn't need to have a fixed parameter informing it of the number of arguments being passed. For example, you could mark the end of the argument list with a special value not used elsewhere. This method places limitations on the arguments that can be passed, however, so it's best avoided.

18-4. Functions That Return a Pointer

In previous chapters you have seen several functions from the C standard library whose return value is a pointer. You can write your own functions that return a pointer. As you might expect, the indirection operator (*) is used in both the function declaration and the function definition. The general form of the declaration is

```
type *func(parameter_list);
```

This statement declares a function func() that returns a pointer to type. Here are two concrete examples:

```
double *func1(parameter_list);
struct address *func2(parameter_list);
```

The first line declares a function that returns a pointer to type double. The second line declares a function that returns a pointer to type address (which you assume is a user-defined structure).

Don't confuse a function that returns a pointer with a pointer to a function. If you include an additional pair of parentheses in the declaration, you declare a pointer to a function, as shown in these two examples:

```
double (*func)(...);    /* Pointer to a function that returns a
double. */
double *func(...);      /* Function that returns a pointer to a
double. */
```

Now that you have the declaration format straight, how do you use a function that returns a pointer? There's nothing special about such functions--you use them just like any other function, assigning their return value to a variable of the appropriate type (in this case, a pointer). Because the function call is a C expression, you can use it anywhere you would use a pointer of that type.

Listing 18.4 presents a simple example, a function that is passed two arguments and determines which is larger. The listing shows two ways of doing this: one function returns an int, and the other returns a pointer to int.

Listing 18.4. Returning a pointer from a function.

```
1: /* Function that returns a pointer. */
2:
3: #include <stdio.h>
4:
5: int larger1(int x, int y);
6: int *larger2(int *x, int *y);
7:
8: main()
9: {
10:     int a, b, bigger1, *bigger2;
11:
12:     printf("Enter two integer values: ");
13:     scanf("%d %d", &a, &b);
14:
15:     bigger1 = larger1(a, b);
16:     printf("\nThe larger value is %d.", bigger1);
17:     bigger2 = larger2(&a, &b);
18:     printf("\nThe larger value is %d.\n", *bigger2);
19:     return(0);
20: }
21:
22: int larger1(int x, int y)
23: {
24:     if (y > x)
25:         return y;
26:     return x;
27: }
28:
29: int *larger2(int *x, int *y)
```

```
30: {
31:     if (*y > *x)
32:         return y;
33:
34:     return x;
35: }
Enter two integer values: 1111 3000
The larger value is 3000.
The larger value is 3000.
```

ANALYSIS: This is a relatively easy program to follow. Lines 5 and 6 contain the prototypes for the two functions. The first, larger1(), receives two int variables and returns an int. The second, larger2(), receives two pointers to int variables and returns a pointer to an int. The main() function on lines 8 through 20 is straightforward. Line 10 declares four variables. a and b hold the two variables to be compared. bigger1 and bigger2 hold the return values from the larger1() and larger2() functions, respectively. Notice that bigger2 is a pointer to an int, and bigger1 is just an int.

Line 15 calls larger1() with the two ints, a and b. The value returned from the function is assigned to bigger1, which is printed on line 16. Line 17 calls larger2() with the address of the two ints. The value returned from larger2(), a pointer, is assigned to bigger2, also a pointer. This value is dereferenced and printed on the following line.

The two comparison functions are very similar. They both compare the two values. The larger value is returned. The difference between the functions is in larger2(). In this function, the values pointed to are compared on line 31. Then the pointer for the larger value's variable is returned. Notice that the dereference operator is used in the comparisons, but not in the return statements on lines 32 and 34.

In many cases, as in Listing 18.4, it is equally feasible to write a function to return a value or a pointer. Which one you select depends on the specifics of your program--mainly on how you intend to use the return value.

DO use all the elements described in this chapter when writing functions that have variable arguments. This is true even if your compiler doesn't require all the elements. The elements are va_list, va_start(), va_arg(), and va_end().

DON'T confuse pointers to functions with functions that return pointers.

18-5. Summary

In this chapter, you learned some additional things your C programs can do with functions. You learned the difference between passing arguments by value and by reference, and how the latter technique allows a function to "return" more than one value to the calling program. You also saw how the void type can be used to create a generic pointer that can point to any type of C data object. Type void pointers are most commonly used with functions that can be passed arguments that aren't restricted to a single data type. Remember that a type void pointer must be cast to a specific type before you can dereference it.

This chapter also showed you how to use the macros defined in STDARG.H to write a function that accepts a variable number of arguments. Such functions provide considerable programming flexibility. Finally, you saw how to write a function that returns a pointer.

Q&A

Q Is passing pointers as function arguments a common practice in C programming?

A Definitely! In many instances, a function needs to change the value of multiple variables, and there are two ways this can be accomplished. The first is to declare and use global variables. The second is to pass pointers so that the function can modify the data directly. The first option is advisable only if nearly every function will use the variable; otherwise, you should avoid it. (See Day 12, "Understanding Variable Scope.")

Q Is it better to modify a variable by assigning a function's return value to it or by passing a pointer to the variable to the function?

A When you need to modify only one variable with a function, usually it's best to return the value from the function rather than pass a pointer to the function. The logic behind this is simple. By not passing a pointer, you don't run the risk of changing any data that you didn't intend to change, and you keep the function independent of the rest of the code.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. When passing arguments to a function, what's the difference between passing by value and passing by reference?
2. What is a type void pointer?
3. What is one reason you would use a void pointer?
4. When using a void pointer, what is meant by a typecast, and when must you use it?
5. Can you write a function that takes a variable argument list only, with no fixed arguments?
6. What macros should be used when you write functions with variable argument lists?
7. What value is added to a void pointer when it's incremented?
8. Can a function return a pointer?

Exercises

1. Write the prototype for a function that returns an integer. It should take a pointer to a character array as its argument.
2. Write a prototype for a function called numbers that takes three integer arguments. The integers should be passed by reference.
3. Show how you would call the numbers function in exercise 2 with the three integers int1, int2, and int3.
4. BUG BUSTER: Is anything wrong with the following?

```
void squared(void *nbr)
{
*nbr *= *nbr;
}
```

5. BUG BUSTER: Is anything wrong with the following?

```
float total( int num, ...)
{
int count, total = 0;
for ( count = 0; count < num; count++ )
total += va_arg( arg_ptr, int );
return ( total );
}
```

Because of the many possible solutions, answers are not provided for the following exercises.
6. Write a function that (a) is passed a variable number of strings as arguments, (b) concatenates the strings, in order, into one longer string, and (c) returns a pointer to the new string to the calling program.
7. Write a function that (a) is passed an array of any numeric data type as an argument, (b) finds the largest and smallest values in the array, and (c) returns pointers to these values to the calling program. (Hint: You need some way to tell the function how many elements are in the array.)
8. Write a function that accepts a string and a character. The function should look for the first occurrence of the character in the string and return a pointer to that location.