Chapter 11

An Introduction to Pointers

In this chapter, we introduce the final remaining primitive data type, the pointer, which is the address of a data object. We show how to use pointer literals and variables, explain how they are represented in the computer, and present the three pointer operators: `&`, `*`, and `=`. We explain how, using pointer parameters, more than one result can be returned from a function. Pointers are covered here only at an introductory level and will be considered in greater depth in later chapters.

11.1 A First Look at Pointers

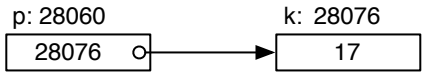
A **pointer** is like a pronoun in English; it can refer to one object now and a different object later. Pointers are used in C programs for a variety of purposes:

- To return more than one value from a function (using call by value/address).
- To create and process strings.
- To manipulate the contents of arrays and structures.
- To construct data structures whose size can grow or shrink dynamically.

In this chapter we study only the first of these uses of pointers; the others will be explored in Chapters 12, 13, 16, and 17.

11.1.1 Pointer Values Are Addresses

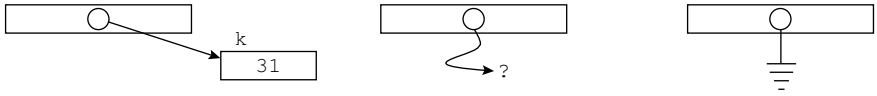
A pointer value (also called a *reference*) is the address (i.e., a specific memory location) of an object. A pointer variable can store different references at different times. If `p` is a pointer variable and the address of `k` is stored in `p`, then we say `p` *points at* `k` or `p` contains *is a reference to* `k` or `p` *refers indirectly to* `k`. In the other direction, we say that `k` *is pointed at by* `p` or `k` *is the* **referent** *of* `p`. In object diagrams, we represent



`p` is a pointer variable stored at address 28060.
`k` is an integer variable stored at address 28076.
The contents of `k` is the integer 17.
The contents of `p` is the address of `k`, which is 28076.
The arrow is a reference to `k`
We say that `p` points at `k` or `p` refers to `k`
We say that `k` is the referent of `p`.

Figure 11.1. A pointer and its referent.

We diagram a pointer variable as a box containing an arrow (a pointer). Here, we diagram three pointer variables. The first points at an integer, the second is uninitialized, and the third contains the `NULL` pointer.



Integer pointer (points to variable `k`)

Uninitialized integer pointer (points to a random location)

`NULL` pointer (points to location `0`)

Figure 11.2. Pointer diagrams.

pointer values as arrows and pointer variables as boxes from which arrows originate. The tail of each pointer arrow is a small circle that can be "stored" in a pointer variable; the head of the arrow points at its referent. Figure 11.1 shows a pointer variable named `p` that points at the integer variable `k`, which itself has a value of 17. Figure 11.1 shows a simplified way to diagram pointer variables, without the explicit memory addresses.

Pointer variables. To declare a **pointer variable**, we start with the **base type of the pointer**; that is, the type of object it can reference. After the type comes a list of pointer variable names, each preceded by an asterisk, as shown in Figure 11.3. A common mistake is to omit the asterisk in front of `p2`. This makes it a simple integer rather than a pointer. The asterisk must be written before each name, not just appended to the base type. A pointer can refer only to objects of its base type. Although a given pointer can refer to different objects at different times, we cannot use it to refer to an `int` at one moment and a `double` later. Therefore, both `p1` and `p2` in the diagram can be used to refer to an integer variable `k`, but neither could refer to a `char` or a `double`. When we use pointers in expressions, the base type of the pointer lets `C` know the actual type of the values that can be referenced, so that it can compile appropriate operations and conversions.

Pointer initialization. When a pointer is declared without an initializer (as an **unitialized pointer**), memory is allocated for it but no address is stored there, so any value previously stored in that memory location remains. Therefore, a pointer always points at *something*, even when that thing is not meaningful

Two uninitialized pointer variables are declared.

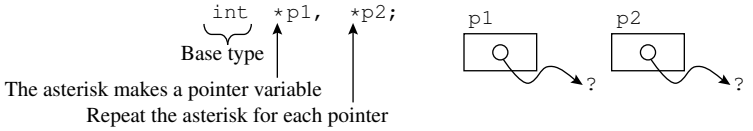


Figure 11.3. Declaring a pointer variable.

An integer pointer variable is declared and initialized to `NULL`.

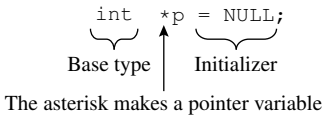


Figure 11.4. Initializing a pointer variable.

to the current program. It could be an actual object in the program, a random memory address in the middle of the code, or an illegal address that does not even correspond to a memory location the program is allowed to access. Most C compilers will not detect or give error comments about uninitialized pointers. If a program unintentionally uses one, the consequences will not be discovered until run time and can be quite unpredictable, depending on the random contents of memory when the program begins execution. Anything can happen, from apparently correct operation to strange output results to an immediate program crash. To emphasize the unknown consequences of using such pointers, we diagram an uninitialized pointer as a wavy arrow that ends at a question mark, as in the middle diagram in Figure 11.2.

The `NULL` pointer. Many data types include a literal value that means "nothing": for type `double` this value is 0.0, for `int` it is 0, and for `char` it is `\0`. There also is a "zero" value for pointer types, the **`NULL` pointer**, and it is defined in `stdio.h`. We store the value `NULL` in a pointer variable as a sign that it points to nothing. `NULL` is represented in the computer as a series of 0 bits and, technically, is a pointer to memory location 0 (which contains part of the operating system). In diagrams, we represent `NULL` using the electric "ground" symbol, as shown in the rightmost part of Figure 11.2, or an arrow that loops around, crosses itself, and ends in midair. One of the basic uses of `NULL` is to initialize a pointer, to avoid pointing at random memory locations. We often initialize pointers to `NULL` (see Figure 11.4). This indicates that the pointer refers to nothing, as opposed to something undefined.

Which nothing is correct? A pertinent question is, What is the difference between 0.0, 0, `\0`, and `NULL`? First, even though all are composed entirely of 0 bits, they are of different lengths. A `double` 0.0 often is 8 bytes long, but the character `\0` is only 1 byte long. The `NULL` pointer is the length of pointers on the local

```
int *pt = NULL;      /* An int pointer variable, initialized to NULL.   */
int *p;              /* A pointer variable that can point at any int.   */
int k = 17;          /* An integer variable initialized to 17.          */
int m;               /* An uninitialized integer variable.              */
```

Storage for these variables might be laid out in the computer's memory as follows:

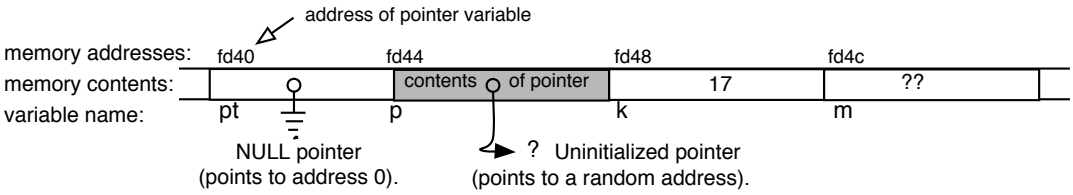


Figure 11.5. Pointers in memory.

system, which, in turn, is determined by the number of bytes required to store a memory address on that system. Second, these zero values have different types and a C compiler treats them like other values of their type when it produces compile-time error comments. For example, if the value 3.1 would be legal in some context, then the value 0.0 also would be legal. The value 0 would be acceptable and require conversion, while the value `NULL` would be inappropriate and cause a compile-time error.

The implementation of pointers. A pointer variable is a storage location in which a pointer can be stored. The pointer itself is the address of another variable. Thus, a *pointer variable* has an address and also contains an address; a *pointer* is the address of its referent. The dual nature of pointers can be confusing, even to experienced programmers. Sometimes it helps to understand how pointers actually are implemented in the computer at run time. The basics are illustrated in Figure 11.5. There, we declare two integer variables and two integer pointers, then diagram the variables created by the declarations. (We assume that an `int` fills 2 bytes and a pointer 4.) Hypothetical memory addresses are shown above the boxes to help explain the actions of certain pointer operations in the next section.

11.1.2 Pointer Operations

C has three basic operators[1] that deal with pointers: `&`, `*`, and `=`. In addition, `scanf()` supports a format specifier for printing pointer values. While each operation is straightforward, sometimes the use of pointers can be confusing.

[1]Pointers to aggregate types follow different syntactic rules than pointers to simple variables and use an additional operator. The differences are explained in Chapters 13 and 17.

References and indirection. The expression `&k` (the `&` **of a variable**) gives us the address of the variable `k`, also called a *reference to* `k`. The **dereference operator**, `*`, also called **indirection**, is the inverse of `&`; that is, `*p` means the referent of `p` (the object at which `p` points). Since, by definition, `&` and `*` are inverse operations, `*&k == k` and `&*p == p`.[2]

The expression `*p` stands for the referent the same way a pronoun stands for a noun. If `k` is the referent of the pointer `p`, then `m = *p` means the same thing as `m = k`. We say that we *dereference* `p` *to get* `k`. Similarly, `*p = n` means the same thing as `k = n`. Longer expressions also can be written; anywhere that you might write a variable name, you can write an asterisk and the name of a pointer that refers to the variable. For example, `m = *p + 2` adds 2 to the value of `k` (the referent of `p`) and stores the result in `m`. Since the dereference operator and the multiply operator use the same symbol, `*`, the C compiler distinguishes between them by context. If the operator has operands on both the left and right, it means "multiply." If it has only one operand, on the right, it means "dereference."

Pointer assignment. As just seen, a pointer can be involved in an assignment operation in three ways:

1. We can make an assignment directly *to* a pointer, as in `p = &k`.
2. We can access a value *through* a pointer and assign it to a variable, as in `m = *p`.
3. We can use a pointer to make an *indirect assignment* to its referent, as in `*p = m+2`.

Direct assignments are useful for string manipulation (Section 12.1). Indirect assignment and access through a pointer are used with call by value/address parameters (Section 11.2). Figure 11.6 illustrates the three kinds of pointer assignment using the variables declared in Figure 11.5.

Notes on Figure 11.6. Pointer operations. To show the actual relationship between a pointer variable, its contents, and its referent, the addresses and contents of each pointer and variable in this program have been printed and diagrammed.

First box: pointer declarations. Figure 11.5 contains a diagram of the initial contents of the two pointers and the variables declared here.

Second box: direct pointer assignments. There are two ways to assign a value to a pointer: assign either the address of a variable of the correct base type or the contents of another pointer of a matching type.

- The base type of `p` is the same as the type of `k`, so we are permitted to make `p` refer to `k` with the assignment `p = &k`. In the diagram below the output, note that the address of `k` is written in the variable `p` and that the arrow coming from `p` ends at `k`. We say that `p` *refers to* (or *points at*) `k`.
- Pointers `p` and `pt` are the same type, so we can copy the contents of `p` into `pt` with the assignment `pt = p`. This causes the contents of `p` (which is the address of `k`) to be copied into `pt`. In the diagram, you can see that both `p` and `pt` contain arrows with heads pointing at `k`.

[2]The combinations `*&k` and `&*p`, therefore, are silly and not written in a program.

This short program connects the program fragments from this section and adds output statements that show the contents and addresses of the variables.

```
#include <stdio.h>

int main( void )
{
    int * pt = NULL;    /* An int pointer variable, initialized to NULL. */
    int * p;            /* A pointer variable that can point at any int. */
    int   k = 17;       /* An integer variable initialized to 17. */
    int   m;            /* An uninitialized integer variable. */

    p = &k;             /* Use & to set a pointer to k's memory location. */
    pt = p;             /* Copy a pointer. */

    m = *p + 2;         /* Add 2 to the value of p's referent; store result in m. */
    p = m;              /* Copy the value of m into p's referent. */

    printf( "address of  p: %p  contents of  p: %p\n", &p, p );
    printf( "address of pt: %p  contents of pt: %p\n", &pt, pt );
    printf( "address of  k: %p  contents of  k: %i\n", &k, k );
    printf( "address of  m: %p  contents of  m: %i\n", &m, m );
    return 0;
}
```

When compiled and run on a system with 4-byte integers, the following output was produced:

```
address of  p: 0xbffffd44  contents of  p: 0xbffffd48
address of pt: 0xbffffd40  contents of pt: 0xbffffd48
address of  k: 0xbffffd48  contents of  k: 19
address of  m: 0xbffffd4c  contents of  m: 19
```

This corresponds to the memory layout that follows. Here, the memory addresses are shown in hexadecimal notation and only the last four digitsare shown.

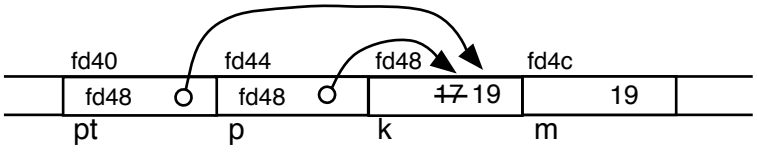


Figure 11.6. Pointer operations.

Third box: indirect pointer assignments.

- The first line dereferences `p` to get `k`, then fetches the value of `k` and adds 2 to it. The result (19) is stored in `m`; the value stored in `k` is not changed at this time. It is not necessary to use parentheses in this expression because `*` has higher precedence than `+`.
- The second line copies the value of `m` into the referent of `p`, which still is `k`. This changes the value of `k` from 17 to 19, as diagrammed.

Fourth box: Output.

The output shows how memory is laid out in our computer, which is running Gnu C[3] and uses 4-byte integers. The memory addresses are printed using the `%p` format specifier, which prints numbers in unsigned hexadecimal notation, with a leading `0x`. Compare this diagram and the output to the memory diagram in Figure 11.5.

11.2 Call by Value/Address

Most parameter values are passed from the caller to a function using **call by value**; that is, by copying the argument value from the caller's memory area into the parameter variable within the function's memory area. However, there are three situations in C in which copying the argument value is not done, is inefficient, or cannot do the job that is required:

1. When an array of any size is being passed (discussed in Chapter 10).
2. When a function needs to return more than one result (discussed in this section).
3. When a large structure is being passed (discussed in Chapter 13).

All these situations are handled in C by passing an address, not a value, to the function.

Chapter 10 discusses call by reference, which is used to pass array arguments[4]. In this method, we pass only the address of the first array slot, not the array's list of values, into the function. Within the function, the reference is transparent to the programmer. That is, references to the array, with or without subscripts, are written exactly the same way in the function as they are in the caller. This makes a large amount of data available to the function efficiently and allows the function to store information into the array. As demonstrated in Section 10.4, a program can pass an empty array into a function, which then fills it with information. When the function returns, that information is in the array and can be used by the caller.

This chapter discusses **call by value/address**. This is a special case of call by value in which the argument is the address of a variable or a pointer to a variable in the caller's memory area. This argument must be stored in a pointer parameter in the function's area. This gives the function full access to the variable that belongs to the caller. This much is exactly like call by reference. However, the similarity ends there because call by value/address is *not* transparent to the programmer; an *indirection* or *dereference* operator must be used in the function's code to access the underlying argument in the storage area of the caller. The remainder of this section explains, in detail, how this works and how to use it.

[3]This is the open-code C compiler from the Free Software Foundation.

[4]Call-by-reference is much more widely used in Java and in C++.

11.2.1 Address Arguments

When call by value is used to pass an address argument, the function receives a reference to the caller's variable and can read from and write to that variable. By reading the caller's variable, the function obtains the value placed there by the calling program. By writing (storing) into the caller's variable, the function can pass a value back to the calling program. When the function ends, the value that it wrote is still in the caller's variable.

You already are familiar with one function that requires an address argument: `scanf()`. When calling `scanf()`, we use an `&` with arguments of simple types such as `long` and `double`. This technique is not limited to `scanf()`. In any function, we can pass the address of a simple variable by writing an `&` followed by the name of the variable. The function must have a corresponding parameter of a pointer type.

Another way to pass the address of a simple variable is to write the name (with no ampersand) of a pointer variable that refers to it. In this case, the value of the pointer, which is the address of its referent, is passed.

11.2.2 Pointer Parameters

A function declares that it is expecting an address argument from the caller by declaring the corresponding parameter with a pointer type. When the argument is an array, as in the statistics program of Figures 11.12 through 10.18, the corresponding parameter can be declared either as a pointer or as an array with an unspecified dimension. For example, if the actual argument were an array of integers, a formal parameter named `ara` could be declared as either `int* ara` or `int ara[]`. The two declarations are identical in most respects, but it is cleaner style to use the latter for arrays. In either case, the parameter name can be used with subscripts within the function to refer to the array elements.

For nonarrays, a **pointer parameter** is declared with an asterisk. When the function is called, the address argument is copied into the corresponding pointer parameter. The base type of the pointer parameter must match the type of the address argument. For example, if a function expects to receive the address of an integer variable, it should declare the corresponding parameter to be of type `int*` (as in the function `f1()` in Figure 11.7). The result is an in-out parameter.

Sometimes an address argument is used to pass a large data structure efficiently[5]. In this case, we may want to have an input parameter that does not permit outward flow of information. To achieve this, the parameter is declared as a constant pointer. For example, `const int * xp` (as in the function `f3()` in Figure 11.7).

Notes on Figure 11.7. Call by value/address. Here we have two simple functions that illustrate two techniques for altering the value of a variable in the caller's memory area.

First box: prototypes. Functions `f1()` and `f2()` have the same prototype, having one parameter that is an integer pointer. The pointer permits the function to export information. C does not provide a way to restrict a parameter to output-only. In function `f2()`, the usage is output-only, but the syntax would permit two-way flow of information.

[5]This will be discussed in Chapter 13.

This program illustrates call by value/address syntax and gives examples of in, in-out and output parameters.

```
#include <stdio.h>
#include <math.h>
void f1( int * xp );           /* uses an in/out parameter */
void f2( int * xp );           /* uses an output parameter*/
double f3( const int * xp );   /* uses an in parameter */

int main( void )
{
    int k = 1;
    double answer;

    printf( "Original value of k: %i\n", k );
    f1( &k );       /* This function changes the value of k. */
    printf( "After f1(), changed value of k: %i\n", k );

    f2( &k );       /* This function changes the value of k. */
    printf( "After f2(), input is stored in k: %i\n", k );

    answer = f3( &k );     /* This function cannot change k. */
    printf( "After f3(), the square root of %i = %.3f\n", k, answer );
    return 0;
}

/* ------------------------------------------------------------------------ */
void f1( int * xp )   /* xp is an in/out parameter*/
{
    *xp = *xp + 2;   /* add 2 to the old value of xp's referent. */
}

/* ------------------------------------------------------------------------ */
void f2( int * xp )   /* xp is an output parameter. */
{
    printf( "Enter an integer: " );
    scanf( "%i", xp );
}

/* ------------------------------------------------------------------------ */
double f3( const int * xp )   /* xp is an in parameter. */
{
    return  sqrt( *xp );      /* use xp's referent (no change). */
}
```

Figure 11.7. Call by value/address.

Second and fifth boxes: indirect reference through an in-out parameter.

- We pass `&k`, the address of `k`, as the argument to the pointer parameter in `f1()`. This address will be stored in the memory location for `xp` when the call occurs, so `xp` will refer to `k`.
- We use the initial value of `k` in this function by writing `*xp`. After adding 2, we store the result back into `k` by writing `*xp =`
- We call `xp` an *input* parameter because we use the information that the caller stored in it. We call it an *output* parameter because we change that information. Thus, it is an in-out parameter.
- The value of `k` is displayed before and after the function call to show that the call both used and changed the value of `main`'s variable. (See the output, below.)

Third and sixth boxes: using an output parameter.

- We use `xp`, an output parameter here, to return a value from `scanf()` back to the caller. The new value is printed in `main()` after the function call.
- We want to pass the address of `k` to `scanf()` so that input can be stored in `k`. We could do this by writing `&*xp` as the argument, but this simplifies to just `xp`, which contains the original address of `k` as it was passed into the function. The `scanf()` call stores a value directly into `main()`'s variable `k`.

Fourth and seventh boxes: using a `const *` input parameter.

- Sometimes our data is stored in large structures (presented in Chapter 13) that occupy many bytes of memory. It is undesirable to copy the whole structure because of the time and the space that would consume. In such a situation we use call by value/address. However, we also want to protect the caller's variable against the possibility of being changed by the function or by any other function it might call. To do this, we can use a `const` *pointer* parameter, also known as a *read-only parameter*.
- In this function, we use a `const *` parameter. This lets us use but not change the value of `main`'s variable. To access that value, we write `*xp`.

Sample output.

```
Original value of k: 1
After f1(), changed value of k: 3
Enter an integer: 7
After f2(), input is stored in k: 7
After f3(), the square root of 7 = 2.646
```

11.2.3 A More Complex Example

In some situations, call by value does not provide enough information to enable a function to do its task. The simplest example consists of a function that wants to swap the values of its two parameters. In Figures 11.8 through 11.10, we examine two possible versions of this simple **swap function**. The first fails to swap the values; the second works properly.

This version of swap does not work because call by value is used to pass the parameters.

```
 #include <stdio.h>
void badswap( double f1, double f2 );
int main( void )
{
    double x = 10.2, y = 7;
    printf( "Before badswap: x=%5.1g  y=%5.1g\n", x, y );
    badswap( x, y );
    printf( "After badswap:  x=%5.1g  y=%5.1g\n", x, y );
}
```

```
void badswap( double f1, double f2 )
{
    double swapper = f1;
    f1 = f2;
    f2 = swapper;
}
```

Figure 11.8. A swap function with an error.

Notes on Figures 11.8, 11.9, and 11.10. Seeing the difference in the swap functions.

First and second boxes, Figure 11.8: the first version of the swap. The first box declares two parameters as type `double`, not `double*`, so the arguments will be passed by value. When `main()` calls `badswap(x,y)` (second box), the current values of `x` and `y` are copied into `badswap()`'s parameters `f1` and `f2`. The function receives these values, not the addresses of the variables. This is shown in the diagram for `badswap()` on the left side of Figure 11.10.

Third box, Figure 11.8: the bad swap. When `badswap()` swaps the values, it swaps the copies stored in the parameters, not the originals. In the diagram, note that the assignments in `badswap()` cause no changes to the variables of `main()`. The program output is

```
Before badswap: x= 10.2  y=  7.0
After badswap:  x= 10.2  y=  7.0
```

First and second boxes, Figure 11.9: the good version of swap. In contrast, this version uses call by value/address (first box). The arguments are two addresses (second box), which are stored in the `swap()` parameters `fp1` and `fp2`. In the diagram, you can see that the parameters of `swap()` are pointers to the variables of `main()`.

This version of swap works because call by value/address is used to pass the parameters.

```
 #include <stdio.h>

void swap( double * fp1, double * fp2 );

int main( void )
{
    double x = 10.2, y = 7;
    printf( "Before swap: x=%5.1g  y=%5.1g\n", x, y );
    swap( &x, &y );
    printf( "After swap:  x=%5.1g  y=%5.1g\n", x, y );
}

/* ------------------------------------------------- */

void swap( double * fp1, double * fp2 )
{
    double swapper = *fp1;
    *fp1 = *fp2;
    *fp2 = swapper;
}
```

Figure 11.9. A swap function that works.

The memory use for Figure 11.8 is diagrammed on the left. Each function has its own memory area, and values of the arguments are copied from variables of `main()` into the parameters of `badswap()`. Assignments change only the values in the parameters.

The memory use for Figure 11.9 is diagrammed on the right. The arguments here are pointers to variables of `main()`. Indirect assignments made through these pointers change the underlying variables.

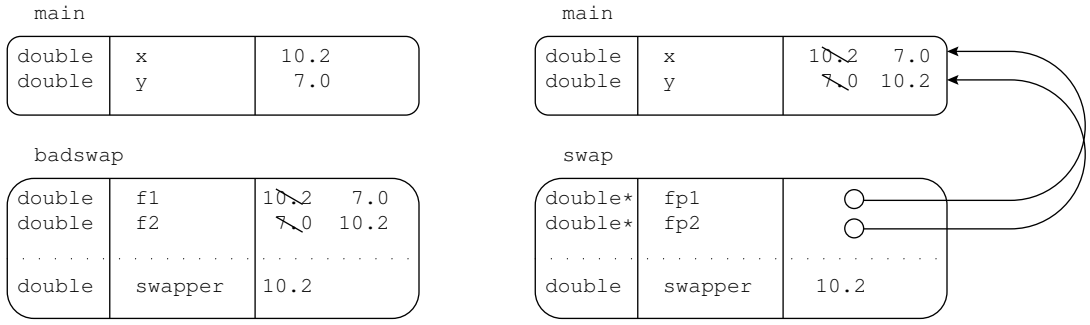


Figure 11.10. Seeing the difference.

Problem scope: Calculate the arithmetic mean, variance, and standard deviation of N experimentally determined data values.

Input: The user will specify the number of data values, N, then N data values will be typed in. These will be real numbers.

Restrictions: No more that 50 data values will be processed.

Formulas:

$$\begin{aligned} Mean &= \frac{\sum_{k=1}^{N} x_k}{N} \\ Variance &= \frac{\sum_{k=1}^{N} (x_k - mean)^2}{N-1} \qquad \text{for } N < 20 \\ Variance &= \frac{\sum_{k=1}^{N} (x_k - mean)^2}{N} \qquad \text{for } N \geq 20 \\ Standard\ deviation &= \sqrt{Variance} \end{aligned}$$

Output required: The mean, variance, and standard deviation of the N points, accurate to at least two decimal places.

Figure 11.11. Problem specifications: Statistics.

Third box, Figure 11.9: the good swap. The function receives pointers to the variables, not copies of their values. When `swap()` says `swapper = *fp1`, it copies the value of the referent of `fp1` into `swapper`. When it executes `*fp1 = *fp2;`, it copies the value of the referent of `fp2` into the referent of `fp1`. This changes the value of the corresponding variable in `main()`, not the pointer in the `swap()` parameter. We say that `*fp1 = *fp2;` fetches the value *indirectly through* `fp2` and stores it *indirectly through* `fp1`. The final output from this program is

```
Before swap: x= 10.2  y=  7.0
After swap:  x=  7.0  y= 10.2
```

11.3 Application: Statistical Measures

In many experiments, the measured values of the experimental variable are distributed about the mean value in a bell-shaped curve centered on the **mean value of the array**; that is, the arithmetic average of the data values. Such a distribution is called a *normal*, or *Gaussian*, *distribution.*

The **variance** and **standard deviation** of a set of data are measures of how significantly the measured values differ from the true mean of the distribution. To compute these measures accurately, we need at least 20 data values. For fewer than 20 data points, we use the slightly different formulas, shown in Figure 11.11, to estimate the variance and standard deviation. In these equations, $x_1, x_2, x_3, \ldots, x_N$ are the data values and $N - 1$ is called the *degree of freedom* of the data.

In the next program example, we introduce a method for computing these statistical measures for N data values: $x_1, x_2, x_3, \ldots, x_N$.[6] The program specification is given in Figure 11.11 and the main program is shown in Figure 11.12. To keep the flow of logic in all parts of the program simple and uniform, major phases of the computation have been written as separate functions that work with a data array. We call `get_data()` to read the data into an array, then pass that array to the `stats()` function, and finally, print the answers. A call graph is given in Figure 11.13 and the two array-processing functions are found in Figure 11.14.

Notes on Figure 11.12. Mean and standard deviation.

First and third boxes: the data array and the definition of N.

Every part of this program uses the value of N to define the number of data values that are to be read, processed, or output. We easily can increase or decrease the length by changing only the `#define` at the top of the program. This is one of the most important ideas in computing: By using loops and arrays, we can process a virtually unlimited number of data items. Files that store large amounts of data are the final element needed in this application. We will revisit this example in Chapter 14 to show how such data can be read from a file.

It is rare that the maximum number of data values is used, since the value of N usually is set to a comfortably large value. Therefore, the user needs to specify the size of the current data set. This value, rather than N, will serve as the processing limit in each of the array functions.

Second box: the function prototypes.

These are the prototypes for the functions in Figure 11.14. Both of these functions have an array parameter and an integer parameter that gives the size of the data set. In `stats()`, we restrict the array to input-only by writing `const` as part of the parameter type. We do this because `stats()` needs to use, but not to modify, the array values. In addition, `stats()` has two output parameters. Note that the square brackets must be used for an array in the parameter declaration, but they are omitted in the function call.

Fifth box: the function calls.

- The function calls must correspond to the prototypes. Each call must have the same number of arguments (of the same type and in the same order) as the parameters declared by the prototype. The pairs are
  - In `get_data()`, `x` is an array parameter which is passed by reference into a `double[]` parameter. Because `x` is an array, this is a a reference parameter, and therefore is in/out. However, it will be used in an output-only style, to carry the data back to `main()`.
  - In `stats()`, `x` is an array parameter which is passed by reference into a `const double[]` parameter. This is an input parameter. Because it is a reference parameter, the function will be able to read input from the array. Because of the `const`, the function will not be able to change the data in the array.

[6] J. P. Holman, *Experimental Methods for Engineers*, 7th ed. (New York: McGraw-Hill, 2001).

The specification for this program is in Figure 11.11 and the call graph is in Figure 11.13. The three programmer-defined functions are in Figures **??** and **??**.

```
#include <stdio.h>
#include <math.h>           /* For sqrt() */

#define N   50              /* Maximum number of data values.   */

void get_data( double x[], int n );
void stats( const double x[], int n, double * meanp, double * variancep );

int main( void )
{
    double x[N];   /* An array for the N data values. */
    int num;       /* Actual number of data values. */

    double mean;   /* The mean of the values in array x. */
    double var;    /* Variance of the data in array x. */
    double stdev;   /* Standard deviation of the data in array x. */

    puts( "\n Computing statistics on a set of numbers." );
    printf( "\n Enter number of values in data set (2..%i): ", N );
    for (;;) {
        scanf( "%i", &num );
        if (num > 1 && num <= N) break;
        printf( "Error: %i is out of legal range, try again: \n", num );
    }
    printf( " Computing statistics on %i data values.\n", num );

    get_data ( x, num );
    stats( x, num, &mean, &var );

    stdev = sqrt(var);

    printf( "\n\n The mean of the %i data values is = %.2f \n", num, mean );
    printf( " The variance is = %.2f\n", var );
    printf( " The standard deviation is = %.2f \n", stdev );
    return 0;
}
```

Figure 11.12. Mean and standard deviation.

This graph is for the program found in Figures 11.12 and 11.14.

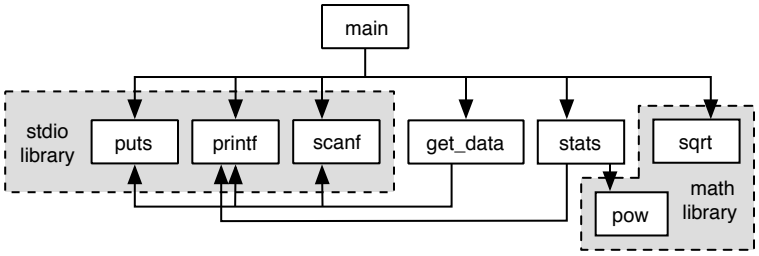


Figure 11.13. Call graph for the mean and standard deviation program.

- `num`, in both functions, the length of the array, which is passed by value into an `int` parameter. This is an input parameter; call by value with an argument of a simple type does not support outward flow of information.
- `meanp` and `variancep` in `stats()` are used as output parameters. In the call, we write `& mean` and `& variance` to pass references into function, where they are stored in the parameters. The parameter `meanp` is type `double*`, which is appropriate for storing the address of a `double` variable.

Sixth box: Using the returned value.

After returning from `stats`, the variables `mean` and `var` contain meaningful values. We now call `sqrt(var)` to compute the standard deviation, then print the statistics.

Output.

The `main()` function prints headings, reads the number of data items, calls the four functions, and then prints the answers. Following is sample output:

```
Computing statistics on a set of numbers.

Enter number of values in data set (2..50): 5
Computing statistics on 5 data values.
Please enter data values when prompted.
x[0] = 77.89
x[1] = 76.55
x[2] = 76.32
x[3] = 79.43
x[4] = 75.12

   x[0] = 77.89
   x[1] = 76.55
   x[2] = 76.32
   x[3] = 79.43
   x[4] = 75.12

The mean of the 5 data values is = 77.06
The variance is = 2.72
The standard deviation is = 1.65
```

These functions are called from the main program in Figure 11.12. After reading a set of `n` experimentally determined data values into an array, `x`, we use summing loops to calculate the mean and variance of those values.

```
/* ------------------------------------------------------------------ */
/* Given an empty array of length n, input data values to fill it.  */
void get_data( double x[], int n )
{
    int k;                          /* Loop counter and subscript. */
    puts( "Please enter data values when prompted." );
    for (k = 0; k < n; ++k) {
        printf( "x[%i] = ", k );    /* Prompt for kth value. */
        scanf( "%lg", &x[k] );      /* Read into array slot k. */
    }
}
/* ------------------------------------------------------------------ */
void stats( const double x[], int n, double * meanp, double * variancep )
{
    int k;                /* Counter and array subscript for both loops. */
    double sum;           /* Accumulator for both loops. */
    double divisor;       /* From the definition of variance. */
    double local_mean;    /* Local copy of first answer. */

    for (sum = k = 0; k < n; ++k) {
        printf( "\n    x[%d] = %.2f", k, x[k] );
        sum += x[k];
    }
    *mean = local_mean = sum / n;   /* Store average locally, also return it. */

    for (sum = k = 0; k < n; ++k) {
        sum += pow( (x[k] - local_mean), 2 );
    }
    if (n < 20) divisor = n-1;
    else divisor = n;
    *variance = sum / divisor;    /* Return the variance. */
}
```

Figure 11.14. The `get_data()` and `stats()` functions.

Notes on Figure 11.14. The `get_data()` and `stats()` functions.

The* `get_data()` *function.

The array parameter, `x`, is an output parameter that is passed by reference, When control enters this function, the parameter array is empty. After the last data value has been read, control returns to the caller and the caller can use the values stored in the array by the function.

First box: variables for the* `stats()` *function.

The third parameter will be used to send one answer (the average) back to the caller. But we also define a local variable for the average to make it easier and more efficient to use in later calculations.

Second box: the average.

This is the usual loop for computing the average of an array. Once all `n` values have been summed, we can calculate the average and return it to `main()`. The division that computes the mean is after the loop, rather than within it, because there is no need to perform a division on every repetition. We need not worry about division by 0 because `n` has been validated in `main()`.

The average is first stored in a local variable, then returned to the caller in the same statement by writing `*meanp = local_mean`. We keep a local copy of the average because we will use it again, repeatedly, in the next loop.

Third box: the* `pow()` *function.

The `pow()` function is in the `math` library. It raises a `double` value to a `double` power and returns a `double` result. In this call, the integer 2 will be coerced to type `double` before it is passed to the function. The result will be the square of the first argument.

Fourth box: returning the second answer.

Once all `n` squares have been summed, we set the divisor to `n` or `n-1`, according to the specifications, then perform the division and return the result to `main()` by assigning it to `*variancep`.

Unlike the average, we do not store the variance in a local variable after we compute it because we do not need it again in this function.

11.3.1 Summary: Returning Results from a Function

We have now demonstrated three ways to return a result from a function:

1. By using the `return` statement.
2. By using a pointer parameter.
3. By storing the results in an array parameter.

The first method is simple and it supports a total separation between the "territory" of the calling program and the actions of the function. The function need not receive an address from the caller to use

`return`. This kind of separation is a highly important debugging tool and should be used wherever possible. Unfortunately, the `return` statement is limited to one result.[7]

The second method can be used for multiple results, as in the previous program, but it involves the use of the address-of operator (`&`) in the function call, the dereference operator (`*`) in the function body, and a pointer declaration in the function's prototype and header. Using these operators is somewhat awkward and can become confusing. Sometimes the required stars and ampersands are omitted accidentally. More significant, though, is that each pointer parameter supplies the function with an address in the memory area that belongs to its caller. Each such address can be a source of unintended damaging interaction between the two code units. Therefore, we prefer to pass parameters by value wherever possible. Even so, call by value/address is quite important in C and frequently must be used. [8]

Finally, returning large numbers of results of the same type is possible by using an array. It generally is convenient, efficient, and not confusing. However, the array parameter still is an address of storage that belongs to the caller. Therefore, an array parameter (like a pointer parameter) reduces the isolation of one part of the program from the other, potentially making debugging harder.[9]

11.4 What You Should Remember

11.4.1 Major Concepts

- A pointer is an address. If `p` is a pointer, then the base type of `p` determines how the compiler will interpret the data stored in the referent of `p`. For example, if a `float` pointer is dereferenced, the result is treated like a `float`.
- There are two major pointer operators, `&` and `*`, which are the inverses of one another. The address operator, `&`, is used to refer to the memory location of its operand. The dereferencing operator, `*`, uses the address in the pointer operand to either retrieve or store a value at that location.
- Pointer variables, like all others, can have garbage in them if they are not initialized. Sometimes this garbage could be an accessible memory location and other times not. The value `NULL` often is used to initialize pointers. Any attempt to reference this location on a properly protected system will cause an immediate and consistent run-time error that can be tracked down more easily than the intermittent errors caused by using a random address.
- When a parameter is a pointer variable, the corresponding argument must be an array, a pointer, or an address. The called function then can store data at that address and, by doing so, change the value in a variable that belongs to the caller. We call this method of parameter passing *call by value/address*.
- When a parameter is a `const` pointer variable, the corresponding argument must be an array, a pointer, or an address. The called function then can use the data at that address but cannot change it.

[7] This one result however can be a complex object, such as a structure that contains multiple components. Structures will be discussed in Chapter 13.

[8] This is a defect of C. It is corrected in C++, which supports a third form of parameter passing, termed *call by reference*.

[9] Clearly, none of these three mechanisms is an ideal solution to the problem. For this reason, the reference parameter, a fourth method of returning results, was implemented in C++.

- Using pointer parameters, one can return multiple results from a single function. An array parameter is translated as a pointer and lets us pass a large amount of data to or from a function efficiently.

11.4.2 Programming Style

- In this text, pointer variable declarations place the `*` next to the base type (as in `float* f`) or let it float (as in `float * f`) between the type and the name. However, it is quite common, for various historical reasons, for programmers to use the style `float *f`, in which the asterisk is attached to the variable name. We prefer the style `float* f` because it clarifies that the data type of the variable is a pointer type.
- To avoid confusion about which variables are pointers and which are not, it is best to declare only one pointer per line. This lets you maintain our preceding style convention and provides space for a comment. If you declare more than one pointer on the same line, be sure to use an `*` for each one.
- Use the correct zero literal. For pointers, use `NULL`. Reserve `0` for use as an integer.
- It is good practice to initialize pointers to `NULL`, which makes pointer usage errors easier to find.
- Do not use the operator combinations `&*` and `*&`. Since the operators cancel each other out, there is no need for either.
- In a function definition, put the parameters that bring information into the function first and the call-by-value/address parameters that carry information out of the function last. Any in-out parameters can be placed in between.
- Minimizing the use of call by value/address increases the separation between caller and subprogram, which is helpful when debugging. However, call by value/address is an important mechanism. Learn to use it wisely.
- Do not use a global variable to pass information into or out of a function. In all other cases, pass the information as a parameter. The one exception to this rule is when the global variable is used with a piece of pre-existing code that you cannot change,
- When using both the return value and pointer parameters to return results from a function, use the return value for a result that *always* is meaningful, such as a status code. Use parameters for results that may or may not be meaningful, depending on circumstances.
- If a function, `f()`, is not called by `main()` and is called by only a single other function, then the prototype for `f()` may be written inside the function that calls it. This properly limits its accessibility to the scope the programmer intended.

11.4.3 Sticky Points and Common Errors

- The most common pointer error is an attempt to use a pointer that has not been set to refer to anything. Sometimes such pointers have a `NULL` value; sometimes they contain garbage. In both cases, the attempt to use such a pointer is an error. On some systems, this causes an immediate crash.

On others, execution may continue indefinitely before anything unexpected happens. Pointers are like pronouns; until they are initialized to point at specific variables, they must not be used. Be careful with your pointers and check them first when a program that uses pointers malfunctions.

- There can be confusion between the multiply and dereference operators. It should be clear from the context which is being used. If, by accident, an operand is omitted in an expression, the compiler might interpret the `*` as multiplication (when dereference was intended) without generating an error message. Using redundant parentheses can help the compiler interpret your code as you intended, but excess parentheses can clutter the expression, potentially causing other kinds of errors. Some compromise in style is needed.
- Do not reverse the use of `&` and `*`. Using the wrong operator always leads to trouble.
- When using call by value/address, a common oversight is to omit some of the required asterisks in the function or the ampersand in the call. This can produce a variety of compile-time error comments that may warn you about a type mismatch between argument and parameter but not tell you exactly what is wrong. For example, if the omission is in the parameter declaration, the error comment actually will be on the first line in the function where that parameter is used. Sometimes a beginner "corrects" the thing that was not wrong, which leads to different errors, and so on, until the code is a mess. When you have a type mismatch error, think carefully about why the error happened and fix the line that actually is wrong. Sometimes drawing a memory diagram can help clear up the confusion.
- Even if you have no type mismatches between arguments and parameters, you may not have the kind of communication you want. If you want call by value/address, you must declare things properly; otherwise, you get functions like `badswap()` in Figure 11.8. Still other times you might omit the `*` where it is needed inside the function, and the address in a pointer parameter will be changed rather than the contents of the other memory location. The compiler may not complain about this, but it certainly will affect the logic of your program. Also, forgetting the `&` in front of an argument for a pointer parameter may not generate a compiler error, but the value of the parameter during execution will be nonsense and usually cause the program to crash. Some compilers give warnings about errors like this.

11.4.4 Where to Find More Information

- Strings pointers and ragged arrays are covered in Chapter 12.
- Array processing using pointers is explained in Chapter 17.
- Pointers to functions are covered in Chapter 17.
- Pointers to dynamic storage allocation areas are found in Chapter 16.

11.4.5 New and Revisited Vocabulary

These are the most important terms, concepts, and keywords presented in this chapter.

pointer	memory address	call by reference
pointer variable	`&` of a variable	call by value/address
base type of pointer	`*` operator	address argument
`NULL` pointer	indirection	pointer parameter
uninitialized pointer	output parameter	swap function
referent	input parameter	mean value of an array
reference	in-out parameter	variance
dereference	call by value	standard deviation

11.5 Exercises

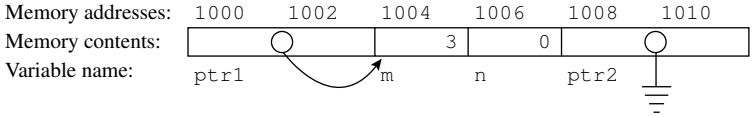
11.5.1 Self-Test Exercises

1. Complete each of the following C statements by adding an asterisk, ampersand, or subscript wherever needed to make the statement do the job described by the comment. Use these declarations:

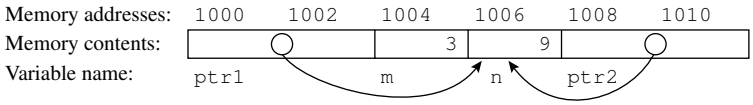
```
float x, y;
float s[4] = {62.3, 65.5, 41.2, 73.0};
float * fp;
```

   (a) `fp = y;  /* Make fp refer to y. */`
   (b) `fp = s;  /* Point fp at slot containing 65.5. */`
   (c) `x = fp;  /* Copy fp's referent into x. */`
   (d) `fp = y;  /* Copy y's data into fp's referent. */`
   (e) `fp = s[];  /* Copy 73.0 into fp's referent. */`
   (f) `scanf( "%g", x );  /* Read data into x. */`
   (g) `scanf( "%g", fp );  /* Read data into fp's referent. */`
   (h) `printf( "%g", s );  /* Print value of second slot. */`

2. (a) Given the following diagram of four variables, write code that declares and initializes them, as pictured. On this system, an `int` occupies two bytes.



   (b) Now write two or three direct or indirect pointer assignment statements to change the memory values to the configuration shown here:



3. Consider the function prototype that follows. Write a short function definition that matches the prototype and the description above it. Then write a main program that declares any necessary variables, makes a meaningful call on the function, and prints the answer.

```
/* Return true via out1 if in1 == in2, false otherwise. */
void same( int in1, int in2, int* out1 );
```

4. All the questions that follow refer to the given partially finished program.

```
#include <stdio.h>
void freeze( int temperatures[], int n );
void show( int temperatures[], int n );

int main( void )
{
    int max = 6;
    int degrees[6] = { 34, 29, 31, 36, 37, 33 };
    freeze( degrees, max );
    printf( "\n After freeze: max = %i\n", max );
    ........................
}
/* ------------------------------------------- */
void freeze( int temperatures[], int n )
{   int k;
    for(k = n-1; k >= 0; --k)
        if (temperatures[k] >= 32) --n;
}
```

   (a) The second parameter of the `freeze()` function is supposed to be a call-by-value/address parameter. However, the programmer forgot to write the necessary ampersands and asterisks in the prototype, call, function header, and function code. Add these characters where needed so that changes made to the parameter `n` actually change the underlying variable `max` in the main program.

   (b) Write a function named `show()`, according to the prototype given, that will display all the numbers in the array on the computer screen. Write a call on this function on the dotted line in `main()`.

   (c) Draw a storage diagram similar to the one in Figure 11.10 and use it to trace execution of the call on `freeze()` and the following `printf()` statement. On your diagram, show every value that is changed.

   (d) What is the output from this program after the additions?

5. (Advanced question) Trace the execution of the program that follows. Make a memory diagram following the example of Figure 11.10 and showing each program scope in a separate box. On your diagram, show the value given to each parameter when a function is called, how the values of its variables change as execution proceeds, and the value(s) returned by the function. Show the output produced on a separate part of your page.

```
#include <stdio.h>
int y = 2, z = 3;          /* Global variables! */

int func1( int* x, int* y );
void func2( int* x ){  *x = y;  y = z;  z = *x;  }
```

```
int main( void )
{
    int x = func1( &y, &x );
    printf( "X = %i  Y = %i  Z = %i\n", x, y, z );
}

int func1( int* x, int* y )
{
    *y = z+1;
    *x = *y;
    func2( y );
    z = *x+2;
    return *y;
}
```

11.5.2 Using Pencil and Paper

1. Complete each of the following C statements by adding an asterisk, ampersand, or subscript wherever needed to make the statement do the job described by the comment. Use these declarations:

```
short s, t;
short age[] = { 30, 65, 41, 23 };
short * agep, * maxp;
```

a. `agep = age; /* Make agep refer to first age in array */`

(b) `s = agep; /* Copy value of agep's referent into s. */`

(c) `agep = age[]; /* Copy 65 into agep's referent. */`

(d) `maxp = agep; /* Make maxp refer to agep's referent. */`

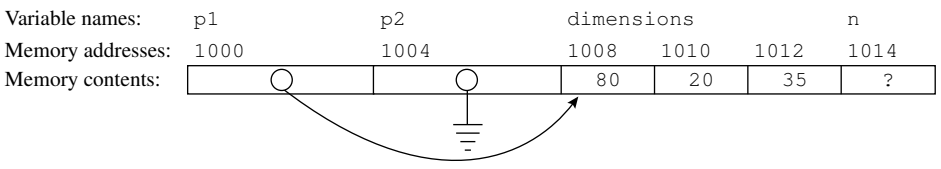
e.
```
agep = (age[]+age[])/2;  /* Store mean of 2nd and last
                                ages in agep's referent. */
```

(f) `scanf( "%hi", age[] );  /* Read into third array slot. */`

(g) `scanf( "%hi", agep );  /* Read into agep's referent. */`

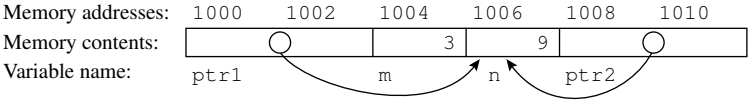
(h) `printf( "%hi", agep );  /* Print agep's referent. */`

2. (a) Given the diagram of three variables and an array, write code that declares the variables and initializes the array and the pointers, as pictured.



(b) Write a function that takes the `dimensions` array as its parameter. Use a nested `if...else` statement to identify the smallest dimension and return the subscript of its slot.

(c) Now call the function and store the result in `n`, then use `n` to set `p2` to point at the smallest dimension.



3. All the questions that follow refer to the given partially finished program.

```
#include <stdio.h>
#define MAX 10
void count( int ages[], int adults, int teens );

int main( void )
{
    int family[MAX] = {44, 43, 21, 18, 15, 13, 11, 9, 7};
    int Nadults=0, Nteens=0, Nkids=0;

    puts( "\nFamily Structure" );
    count( family, Nadults, Nteens );
    Nkids = MAX - (Nadults + Nteens);
    printf( "Family has %i adults, %i teens, %i kids\n",
            Nadults, Nteens, Nkids );
    puts( "----------------------------------------\n" );
}
void count( int ages[], int adults, int teens )
{
    int k;
    for (k = 0; k < MAX; ++k) {
        if ( ages[k] >= 18 ) ++adults;
        else if ( ages[k] >= 13 ) ++teens;
    }
}
```

(a) The second and third parameters of the `count()` function need to be pointer parameters. However, the programmer forgot to write the necessary ampersands and asterisks in the prototype, call, function header, and function code. Add these characters where needed so that changes made to the parameters `adults` and `teens` actually change the underlying variables in the main program.

(b) Draw a storage diagram similar to the one in Figure 11.10 and use it to trace execution of the corrected program. On the diagram, show every value changed and show the output on a separate part of the page.

4. Draw a call graph for the bisection program on the text website. Include all programmer-defined and library functions.

5. Consider the function prototype that follows. Write a short function definition that matches the given prototype and description. Write a main program that declares necessary variables, makes a meaningful call on the function, and prints the answer.

```
/* Set the referent of dp to its own absolute value. */
/* Return +1 if it was positive, 0 if it was zero, */
/* and -1 otherwise. */
int signum( double* dp );
```

6. Look at the main program and functions for the bisection program on the text website. Fill in the following table, listing the symbols *defined* (not used) in each program scope. List global symbols on the first line. Allow one line below that for each function in the program (`main()` has been started for you).

Scope	Input Parameters	Output Parameters	Variables	Constants
`global`	—	—		
`main()`	—	—		
...				

11.5.3 Using the Computer

1. Pointer and referent.

   Write a program that creates the integer array `int ara[] = {11, 13, 17, 19, 23, 29, 31}`. Also create an integer pointer `pt` and make it point at the beginning of the array. Write `printf()` statements that will print the address and contents of both `ara[0]` and `pt`. Use this format:

   ```
   printf( "address of pt: \t %p   contents:\t %p\n", &pt, pt );
   ```

   Then write 10 similar `printf()` statements following the same format to print the address and contents of the slots designated by the following expressions: `(*pt+3)`, `*pt`, `(pt[3])`, `*&pt`, `*pt[3]`, `&*pt`, `*(pt+3)`, `(*pt++)`, `*(pt++)`, `(*pt)++`. Some of these will cause compile-time errors when printing the address field, the contents field, or both; in such cases, delete the illegal expression and print dashes instead of its value. When the program finally compiles and runs, use the output to complete the following table, grouping together items that have the same memory address:

	Address	Contents
pt		
ara		
...		

   Finally, make four lists: (a) illegal pointer expressions, (b) expressions that have identical meanings, (c) expressions that change pointer values, and (d) expressions that change integer values. It will require careful reasoning to get the last two lists correct.

2. Exam grades.
   Start with the program in Figures 11.12 through 11.14; modify it as follows:

   (a) In the main program, declare an array to store exam scores for a class of 15 students. Print out appropriate headings and instructions for the user. Call the appropriate functions to read in the exam scores and calculate their mean and standard deviation. Print the mean and standard deviation. Then call the `grades()` function described here to assign grades to the students' scores and print them.

(b) Modify the `average()` function so that it does not print the individual exam scores during its processing.

(c) Write a new function, named `grades()`, with three parameters: the array of student scores, the mean, and the standard deviation. This function will go through the array of exam scores again and assign a letter grade to each student according to the following criteria. Using one line of output per student, print the array subscript, the score, and the grade in columns. The grading criteria are

i. A, if the score is greater than or equal to the mean plus the standard deviation.
ii. B, if the score is between the mean and the mean plus the standard deviation.
iii. C, if the score is between the mean and the mean minus the standard deviation.
iv. D, if the score is between the mean minus the standard deviation and the mean minus twice the standard deviation.
v. F, if the score is less than the mean minus twice the standard deviation.

If a score is exactly equal to one of these boundary limits, give the student the higher grade.

3. Positive and negative.

Write a function, named `sums()`, that has two input parameters; an array, `a`, of `floats`; and an integer, `n`, which is the number of values stored in the array. Compute the sum of the positive values in the array and the sum of the negative values. Also count the number of values in each category. Return these four answers through output parameters. Write a main program that reads no more than 10 real numbers and stores them in an array. Stop reading numbers when a 0 is entered. Call the `sums()` function and print the answers it returns. Also compute and print the average values of the positive and negative sets.

4. More stats.

Start with the statistics program in Figures 11.12 through 11.14. Add a parallel array for the student ID numbers, which should be read as input. Add functions to do these three tasks:

(a) Find the maximum score and return the score and its array index through pointer parameters.
(b) Find the minimum score and return the score and its array index through pointer parameters.
(c) Find the score that is closest to the average and return the score and its array index through pointer parameters.

In `main()`, call your three functions and print the student ID number and score for the best, closest to average, and weakest student.

5. Sorting.

Write a `void` function, named `order()`, that has three integer parameters: a, b, and c. Compare the parameter values and arrange them in numerical order so that $a < b < c$. Use call by value/address so the calling program receives the values back in order. In addition, the function `order()` should start by printing the addresses and contents of its parameters, as well as the contents of the locations to which they point. Write a main program that enters three integers, prints their values and addresses, orders them by calling the function `order()`, and prints their values again after the call. Add a query loop to allow testing several sets of integers.

6. Compound interest.

(a) Write a function to compute and return the amount of money, A, that you will have in n years if you invest P dollars now at annual interest rate i. Take n, i, and P as parameters. The formula is

$$A = P(1 + i)^n$$

(b) Write a function to compute and return the amount of money, P, that you would need to invest now at annual interest rate i in order to have A dollars in n years. Take n, i, and A as parameters. The formula is:

$$P = \frac{A}{(1+i)^n}$$

(c) Write a function that will read and validate the inputs for this program. Using call by value/address, return an enumerated constant for the choice of formulas, a number of years, an interest rate, and an amount of money, in dollars. All three numbers must be greater than 0.0.

(d) Write a `main` program that will call the input routine to gather the data. Then, depending on the user's choice, it should call the appropriate calculation function and print the results of the calculation.

7. Sorting boxes.
A set of boxes are on the floor. We want to put them in two piles, those larger than the average box and those smaller than or equal to the average box. Write a program to label the boxes in the following manner:

(a) Rewrite the `get_data()` function in Figure 11.14 so that you can enter data into three arrays rather than one. These arrays should hold the length, width, and height of the boxes, respectively.

(b) Write a function, named `volume()`, to compute and store the volume of each box in a fourth parallel array. The volume of a box is the product of its length, width, and height.

(c) Simplify the `stats()` function in Figure 11.14 so that it computes only the average, not the variance. Rename it `average()`. Then write a function, `print_boxes()`, that will print the data for each box in a nice table, including a column containing the appropriate label, `big` or `small`, depending on whether the volume of the box is larger or smaller than the average volume.

8. Class average and more.

An instructor has a set of exam scores stored in a data file. Not only does he want a report containing the average and standard deviation for the exam, he wants lots of other statistics. These include the high score, the low score, the median score, and the coefficient of variation, cv. The *median score* is defined to be the middle one in the array of scores, if that array is sorted. This *coefficient of variation* relates the "error" measured by the standard deviation to the "actual" value measured by the arithmetic mean as $cv = \text{stdev}/\text{mean}$. Write a program that will read, at most, 100 exam scores from a user-specified file and print out the indicated statistics. Use portions of the statistics programs in this chapter, as appropriate.