Chapter 8

Program Structure

This chapter is concerned with the structure of programs consisting of more than one function. We will first consider the distinction between "local" variables that are recognized only within a single function, and "global" variables that are recognized in two or more functions. We will see how global variables are defined and utilized in this chapter.

We will also consider the issue of static vs. dynamic retention of information by a local variable. That is, a local variable normally does not retain its value once control has been transferred out of its defining function. In some circumstances, however, it may be desirable to have certain local variables retain their values, so that the function can be reentered at a later time and the computation resumed.

And finally, it may be desirable to develop a large, multifunction program in terms of several independent files, with a small number of functions (perhaps only one) defined within each file. In such programs the individual functions can be defined and accessed locally within a single file, or globally within multiple files. This is similar to the definition and use of local vs. global variables in a multifunction, single-file program.

8.1 STORAGE CLASSES

We have already mentioned that there are two different ways to characterize variables: by *data type*, and by *storage class* (see Sec. 2.6). Data type refers to the type of information represented by a variable, e.g., integer number, floating-point number, character, etc. Storage class refers to the permanence of a variable, and its *scope* within the program, i.e., the portion of the program over which the variable is recognized.

There are four different storage-class specifications in C: *automatic*, *external*, *static* and *register*. They are identified by the keywords `auto`, `extern`, `static`, and `register`, respectively. We will discuss the *automatic*, *external* and *static* storage classes within this chapter. The *register* storage class will be discussed in Sec. 13.1.

The storage class associated with a variable can sometimes be established simply by the location of the variable declaration within the program. In other situations, however, the keyword that specifies a particular storage class must be placed at the beginning of the variable declaration.

EXAMPLE 8.1 Shown below are several typical variable declarations that include the specification of a storage class.

```
auto int a, b, c;

extern float root1, root2;

static int count = 0;

extern char star;
```

The first declaration states that `a`, `b` and `c` are automatic integer variables, and the second declaration establishes `root1` and `root2` as external floating-point variables. The third declaration states that `count` is a static integer variable whose initial value is 0, and the last declaration establishes `star` as an external character-type variable.

The exact procedure for establishing a storage class for a variable depends upon the particular storage class, and the manner in which the program is organized (i.e., single file vs. multiple file). We will consider these rules in the next few sections of this chapter.

8.2 AUTOMATIC VARIABLES

Automatic variables are always declared within a function and are local to the function in which they are declared; that is, their scope is confined to that function. Automatic variables defined in different functions will therefore be independent of one another, even though they may have the same name.

Any variable declared within a function is interpreted as an automatic variable unless a different storage-class specification is shown within the declaration. This includes formal argument declarations. All of the variables in the programming examples encountered earlier in this book have been automatic variables.

Since the location of the variable declarations within the program determines the automatic storage class, the keyword `auto` is not required at the beginning of each variable declaration. There is no harm in including an `auto` specification within a declaration, though this is normally not done.

EXAMPLE 8.2 Calculating Factorials Consider once again the program for calculating factorials, originally shown in Example 7.10. Within `main`, `n` is an automatic variable. Within `factorial`, `i` and `prod`, as well as the formal argument `n`, are automatic variables.

The storage-class designation `auto` could have been included explicitly in the variable declarations if we had wished. Thus, the program could have been written as follows.

```
#/* calculate the factorial of an integer quantity */

include <stdio.h>

long int factorial(int n);

main()

{
    auto int n;

    /* read in the integer quantity */

    printf("\nn = ");
    scanf("%d", &n);

    /* calculate and display the factorial */

    printf("\n!n = %ld", factorial(n));
}

long int factorial(auto int n)        /* calculate the factorial */

{
    auto int i;
    auto long int prod = 1;

    if (n > 1)
       for (i = 2; i <= n; ++i)
           prod *= i;
    return(prod);
}
```

Either method is acceptable. As a rule, however, the `auto` designation is not included in variable or formal argument declarations, since this is the default storage class. Thus, the program shown in Example 7.10 represents a more common programming style.

Automatic variables can be assigned initial values by including appropriate expressions within the variable declarations, as in the above example, or by explicit assignment expressions elsewhere in the

function. Such values will be reassigned each time the function is reentered. If an automatic variable is not initialized in some manner, however, its initial value will be unpredictable, and probably unintelligible.

An automatic variable does not retain its value once control is transferred out of its defining function. Therefore, any value assigned to an automatic variable within a function will be lost once the function is exited. If the program logic requires that an automatic variable be assigned a particular value each time the function is executed, that value will have to be reset whenever the function is reentered (i.e., whenever the function is accessed).

EXAMPLE 8.3 Average Length of Several Lines of Text Let us now write a C program that will read several lines of text and determine the average number of characters (including punctuation and blank spaces) in each line. We will structure the program in such a manner that it continues to read additional lines of text until an empty line (i.e., a line whose first character is `\n`) is encountered.

We will utilize a function (`linecount`) that reads a single line of text and counts the number of characters, excluding the newline character (`\n`) that signifies the end of the line. The calling routine (`main`) will maintain a cumulative sum, as well as a running total of the number of lines that have been read. The function will be called repeatedly (thus reading a new line each time), until an empty line is encountered. The program will then divide the cumulative number of characters by the total number of lines to obtain an average.

Here is the entire program.

```
/* read several lines of text and determine the average number of characters per line */

#include <stdio.h>

int linecount(void);

main()

{
    int n;                  /* number of chars in given line */
    int count = 0;          /* number of lines */
    int sum = 0;            /* total number of characters */
    float avg;              /* average number of chars per line */

    printf("Enter the text below\n");

    /* read a line of text and update the cumulative counters */

    while ((n = linecount()) > 0)   {
          sum += n;
          ++count;
    }

    avg = (float) sum / count;
    printf("\nAverage number of characters per line: %5.2f", avg);
}

int linecount(void)

/* read a line of text and count the number of characters */

{
    char line[80];
    int count = 0;

    while ((line[count] = getchar()) != '\n')
          ++count;
    return (count);
}
```

We see that `main` contains four automatic variables: `n`, `count`, `sum` and `avg`, whereas `linecount` contains two: `line` and `count`. (Notice that `line` is an 80-element character array, representing the contents of one line of text.) Three of these automatic variables are assigned initial values of zero.

Also, note that `count` has different meanings within each function. Within `linecount`, `count` represents the number of characters in a single line, whereas within `main`, `count` represents the total number of lines that have been read. Moreover, `count` is reset to zero within `linecount` whenever the function is accessed. This does not affect the value of `count` within `main`, since the variables are independent of one another. It would have been clearer if we had named these variables differently, e.g., `count` and `lines`, or perhaps `chars` and `lines`. We have used the same name for both variables to illustrate the independence of automatic variables within different functions.)

A sample interactive session, resulting from execution of this program, is shown below. As usual, the user's responses are underlined.

```
Enter the text below
Now is the time for all good men
to come to the aid of their country.

Average number of characters per line: 34.00
```

The scope of an automatic variable can be smaller than an entire function if we wish. In fact, automatic variables can be declared within a single compound statement. With small, simple programs there is usually no advantage in doing this, but it may be desirable in larger programs.

8.3 EXTERNAL (GLOBAL) VARIABLES

External variables, in contrast to automatic variables, are not confined to single functions. Their scope extends from the point of definition through the remainder of the program. Hence, they usually span two or more functions, and often an entire program. They are often referred to as *global variables*.

Since external variables are recognized globally, they can be accessed from any function that falls within their scope. They retain their assigned values within this scope. Therefore an external variable can be assigned a value within one function, and this value can be used (by accessing the external variable) within another function.

The use of external variables provides a convenient mechanism for transferring information back and forth between functions. In particular, we can transfer information into a function without using arguments. This is especially convenient when a function requires numerous input data items. Moreover, we now have a way to transfer multiple data items out of a function, since the `return` statement can return only one data item. (We will see another way to transfer information back and forth between functions in Chap. 10, where we discuss pointers.)

When working with external variables, we must distinguish between external variable *definitions* and external variable *declarations*. An external variable *definition* is written in the same manner as an ordinary variable declaration. It must appear outside of, and usually before, the functions that access the external variables. An external variable definition will automatically allocate the required storage space for the external variables within the computer's memory. The assignment of initial values can be included within an external variable definition if desired (more about this later).

The storage-class specifier `extern` is not required in an external variable definition, since the external variables will be identified by the location of their definition within the program. In fact, many C compilers forbid the use of `extern` within an external variable definition. We will follow this convention within this book.

If a function requires an external variable that has been defined earlier in the program, then the function may access the external variable freely, without any special declaration within the function. (Remember, however, that *any alteration to the value an external variable within a function will be recognized within the entire scope of the external variable.*) On the other hand, if the function definition *precedes* the external variable definition, then the function must include a *declaration* for that external variable. The function

definitions within a large program often include external variable declarations, whether they are needed or not, as a matter of good programming practice.

An external variable *declaration* must begin with the storage-class specifier `extern`. The name of the external variable and its data type must agree with the corresponding external variable definition that appears outside of the function. Storage space for external variables will *not* be allocated as a result of an external variable declaration. Moreover, an external variable declaration *cannot* include the assignment of initial values. These distinctions between an external variable *definition* and an external variable *declaration* are very important.

EXAMPLE 8.4 Search for a Maximum Suppose we wish to find the particular value of x that causes the function

$$y = x \cos(x)$$

to be maximized within the interval bounded by $x = 0$ on the left and $x = \pi$ on the right. We will require that the maximizing value of x be known very accurately. We will also require that the search scheme be relatively efficient in the sense that the function $y = x \cos(x)$ should be evaluated as few times as possible.

One obvious way to solve this problem would be to generate a large number of closely spaced trial functions (that is, evaluate the function at $x = 0$, $x = 0.0001$, $x = 0.0002$, . . . , $x = 3.1415$, and $x = 3.1416$) and determine the largest of these by visual inspection. This would not be very efficient, however, and it would require human intervention to obtain the final result. Instead let us use the following *elimination scheme*, which is a highly efficient computational procedure for all functions that have only one maximum (i.e., only one "peak") within the search interval.

The computation will be carried out as follows. We begin with two search points at the center of the search interval, located a very small distance from each other, as shown in Fig. 8.1.

The following notation is used.

`a` = left end of the search interval
`xl` = left-hand interior search point
`xr` = right-hand interior search point
`b` = right end of the search interval
`sep` = distance between xl and xr.

If `a`, `b` and `sep` are known, then the interior points can be calculated as

```
xl = a + .5 * (b - a - sep)

xr = a + .5 * (b - a + sep) = xl + sep
```

Let us evaluate the function `y = x cos(x)` at `xl` and at `xr`. We will call these values `yl` and `yr`, respectively. Suppose `yl` turns out to be greater than `yr`. Then the maximum will lie somewhere between `a` and `xr`. Hence we retain only that portion of the search interval which ranges from `x = a` to `x = xr`. We will now refer to the old point `xr` as `b`, since it is now the right end of the new search interval, and generate two *new* search points, `xl` and `xr`. These points will be located at the center of the new search interval, a distance `sep` apart, as shown in Fig. 8.2.

On the other hand, suppose now that in our *original* search interval the value of `yr` turned out to be greater than `yl`. This would indicate that our new search interval should lie between `xl` and `b`. Hence we rename the point which was originally called `xl` to be `a` and we generate two *new* search points, `xl` and `xr`, at the center of the new search interval, as shown in Fig. 8.3.

We continue to generate a new pair of search points at the center of each new interval, compare the respective values of `y`, and eliminate a portion of the search interval until the new search interval becomes smaller than `3 * sep`. Once this happens we can no longer distinguish the interior points from the boundaries. Hence the search is ended.

Each time we make a comparison between `yl` and `yr`, we eliminate that portion of the search interval that contains the smaller value of `y`. If both interior values of `y` should happen to be identical (which can happen, though it is unusual), then the search procedure stops, and the maximum is assumed to occur at the center of the last two interior points.

Once the search has ended, either because the search interval has become sufficiently small or because the two interior points yield identical values of `y`, we can calculate the approximate location of the maximum as

```
xmax = 0.5 * (xl + xr)
```

The corresponding maximum value of the function can then be obtained as `xmax cos(xmax)`.

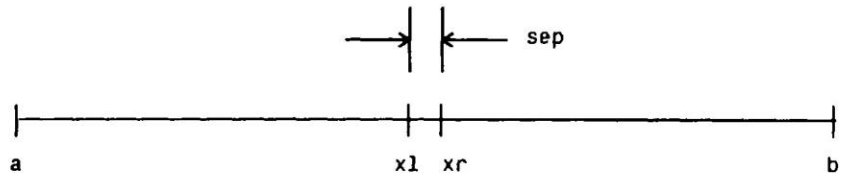

Fig. 8.1

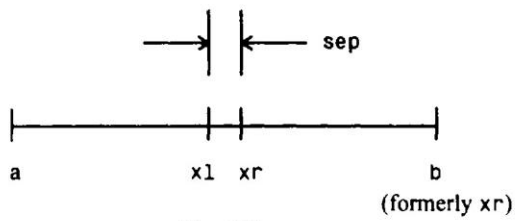

Fig. 8.2

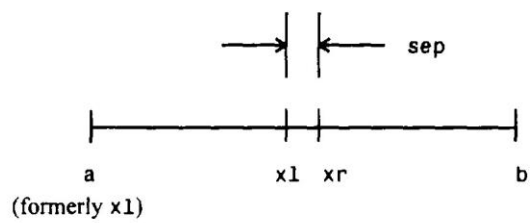

Fig. 8.3

Let us consider a program outline for the general case where a and b are input quantities but sep has a fixed value of 0.0001.

1. Assign a value of `sep = 0.0001`.
2. Read in the values of a and b.
3. Repeat the following until either `yl` becomes equal to `yr` (the desired maximum will be at the midpoint), or the most recent value of `(b - a)` becomes less than or equal to `(3 * sep)`:
   (*a*) Generate the two interior points, `xl` and `xr`.
   (*b*) Calculate the corresponding values of `yl` and `yr`, and determine which is larger.
   (*c*) Reduce the search interval, by eliminating that portion that does not contain the larger value of y.
4. Evaluate `xmax` and `ymax`.
5. Display the values of `xmax` and `ymax`, and stop.

To translate this outline into a program, we first create a programmer-defined function to evaluate the mathematical function `y = x cos(x)`. Let us call this function `curve`. This function can easily be written as follows.

```
/*  evaluate the function y = x * cos(x)  */

double curve(double x)

{
    return (x * cos(x));
}
```

Note that `cos(x)` is a call to a C library function.

Now consider step 3 in the above outline, which carries out the interval reduction. This step can also be programmed as a function, which we will call `reduce`. Notice, however, that the values represented by the variables `a`, `b`, `xl`, `xr`, `yl` and `yr`, which change through the course of the computation, must be transferred back and forth between this function and `main`. Therefore, let these variables be external variables whose scope includes both `reduce` and `main`.

Function `reduce` can be written as

```
/* interval reduction routine */

void reduce(void)
{
    xl = a + 0.5 * (b - a - CNST);
    xr = xl + CNST;
    yl = curve(xl);
    yr = curve(xr);

    if (yl > yr)   {     /* retain left interval */
       b = xr;
       return;
    }
    if (yl < yr)         /* retain right interval */
       a = xl;
    return;
}
```

Notice that the parameter that we have referred to earlier as `sep` is now represented as the character constant `CNST`. Also, notice that this function does not include any formal arguments, and it does not return anything via the `return` statement. All of the information transfers involve external variables.

It is now quite simple to write the main portion of the program, which calls the two functions defined above. Here is the entire program.

```
/* find the maximum of a function within a specified interval */

#include <stdio.h>
#include <math.h>

#define  CNST  0.0001

double a, b, xl, yl, xr, yr;        /* global variables */

void reduce(void);                  /* function prototype */
double curve(double xl);            /* function prototype */

main()

{
    double xmax, ymax;

    /* read input data (interval end points) */

    printf("\na = ");
    scanf("%lf", &a);
    printf("b = ");
    scanf("%lf", &b);

    /* interval reduction loop */

    do
       reduce();
    while ((yl != yr) && ((b - a) > 3 * CNST));
```

```
/* calculate xmax and ymax, and display the results */

    xmax = 0.5 * (xl + xr);
    ymax = curve(xmax);
    printf("\nxmax = %8.6lf   ymax = %8.6lf", xmax, ymax);
}


/* interval reduction routine */

void reduce(void)

{
    xl = a + 0.5 * (b - a - CNST);
    xr = xl + CNST;
    yl = curve(xl);
    yr = curve(xr);

    if (yl > yr)   {     /* retain left interval */
       b = xr;
       return;
    }
    if (yl < yr)         /* retain right interval */
       a = xl;
    return;
}


/*  evaluate the function y = x * cos(x)  */

double curve(double x)

{
    return (x * cos(x));
}
```

The variables `a`, `b`, `xl`, `yl`, `xr` and `yr` are defined as external variables whose scope includes the entire program. Notice that these variables are declared before `main` begins.

Execution of the program, with `a` = 0 and `b` = 3.141593, produces the following interactive session. The user's responses are underlined, as usual.

```
a = 0
b = 3.141593

xmax = 0.860394   ymax = 0.561096
```

Thus, we have obtained the location and the value of the maximum within the given original interval.

External variables can be assigned initial values as a part of the variable definitions, but the initial values must be expressed as *constants* rather than expressions. These initial values will be assigned only once, at the beginning of the program. The external variables will then retain these initial values unless they are later altered during the execution of the program.

If an initial value is not included in the definition of an external variable, the variable will automatically be assigned a value of zero. Thus, external variables are never left dangling with undefined, garbled values. Nevertheless, it is good programming practice to assign an explicit initial value of zero when required by the program logic.

EXAMPLE 8.5 Average Length of Several Lines of Text Shown below is a modification of the program previously presented in Example 8.3, for determining the average number of characters in several lines of text. The present version makes use of external variables to represent the total (cumulative) number of characters read, and the total number of lines.

```
/* read several lines of text and determine the average number of characters per line */

#include <stdio.h>

int sum = 0;                /* total number of characters */
int lines = 0;              /* total number of lines */

int linecount(void);


main()

{
    int n;                  /* number of chars in given line */
    float avg;              /* average number of chars per line */

    printf("Enter the text below\n");

    /* read a line of text and update the cumulative counters */

    while ((n = linecount()) > 0)   {
          sum += n;
          ++lines;
    }

    avg = (float) sum / lines;
    printf("\nAverage number of characters per line: %5.2f", avg);
}


/* read a line of text and count the number of characters */

int linecount(void)

{
    char line[80];
    int count = 0;

    while ((line[count] = getchar()) != '\n')
          ++count;
    return (count);
}
```

Notice that `sum` and `lines` are external variables that represent the total (cumulative) number of characters read and the total number of lines, respectively. Both of these variables are assigned initial values of zero. These values are successively modified within `main`, as additional lines of text are read.

Also, recall that the earlier version of the program used two different automatic variables, each called `count` in different parts of the program. In the present version of the program, however, the variables that represent the same quantities have different names, since one of the variables (`lines`) is now an external variable.

You should understand that `sum` and `lines` need not be assigned zero values explicitly, since external variables are always set equal to zero unless some other initial value is designated. We include the explicit zero initialization in order to clarify the program logic.

Arrays can also be declared either automatic or external, though automatic arrays cannot be initialized. We will see how initial values are assigned to array elements in Chap. 9.

Finally, it should be pointed out that there are inherent dangers in the use of external variables, since an alteration in the value of an external variable within a function will be carried over into other parts of the program. Sometimes this happens inadvertently, as a *side effect* of some other action. Thus, there is the possibility that the value of an external value will be changed unexpectedly, resulting in a subtle programming error. You should decide carefully which storage class is most appropriate for each particular programming situation.

8.4 STATIC VARIABLES

In this section and the next, we make the distinction between a *single-file* program, in which the entire program is contained within a single source file, and a *multifile* program, where the functions that make up the program are contained in separate source files. The rules governing the static storage class are different in each situation.

In a single-file program, static variables are defined within individual functions and therefore have the same scope as automatic variables; i.e., they are local to the functions in which they are defined. Unlike automatic variables, however, static variables retain their values throughout the life of the program. Thus, if a function is exited and then re-entered at a later time, the static variables defined within that function will retain their former values. This feature allows functions to retain information permanently throughout the execution of a program.

Static variables are defined within a function in the same manner as automatic variables, except that the variable declaration must begin with the `static` storage-class designation. Static variables can be utilized within the function in the same manner as other variables. They cannot, however, be accessed outside of their defining function.

It is not unusual to define automatic or static variables having the same names as external variables. In such situations the local variables will take precedence over the external variables, though the values of the external variables will be unaffected by any manipulation of the local variables. Thus the external variables maintain their independence from locally defined automatic and static variables. The same is true of local variables within one function that have the same names as local variables within another function.

EXAMPLE 8.6 Shown below is the skeletal structure of a C program that includes variables belonging to several different storage classes.

```
float a, b, c;

void dummy(void);

main()
{
    static float a;

    . . . . .
}

void dummy(void)
{
    static int a;
    int b;

    . . . . .
}
```

Within this program `a`, `b` and `c` are external, floating-point variables. However, `a` is redefined as a *static* floating-point variable within `main`. Therefore, `b` and `c` are the only external variables that will be recognized within `main`. Note that the static *local* variable `a` will be independent of the *external* variable `a`.

Similarly, a and b are redefined as integer variables within dummy. Note that a is a static variable, but b is an automatic variable. Thus, a will retain its former value whenever dummy is reentered, whereas b will lose its value whenever control is transferred out of dummy. Furthermore, c is the only *external* variable that will be recognized within dummy.

Since a and b are local to dummy, they will be independent of the external variables a, b and c, and the static variable a defined within main. The fact that a and b are declared as integer variables within dummy and floating-point variables elsewhere is therefore immaterial.

Initial values can be included in the static variable declarations. The rules associated with the assignment of these values are essentially the same as the rules associated with the initialization of external variables, even though the static variables are defined locally within a function. In particular:

1. The initial values must be expressed as constants, not expressions.
2. The initial values are assigned to their respective variables at the beginning of program execution. The variables retain these values throughout the life of the program, unless different values are assigned during the course of the computation.
3. Zeros will be assigned to all static variables whose declarations do not include explicit initial values. Hence, static variables will always have assigned values.

EXAMPLE 8.7 Generating Fibonacci Numbers The Fibonacci numbers form an interesting sequence in which each number is equal to the sum of the previous two numbers. In other words,

$$F_i = F_{i-1} + F_{i-2}$$

where F_i refers to the ith Fibonacci number. The first two Fibonacci numbers are defined to equal 1; i.e.,

$$F_1 = F_2 = 1$$

Hence

$$F_3 = F_2 + F_1 = 1 + 1 = 2$$

$$F_4 = F_3 + F_2 = 2 + 1 = 3$$

$$F_5 = F_4 + F_3 = 3 + 2 = 5$$

and so on.

Let us write a C program that generates the first n Fibonacci numbers, where n is a value specified by the user. The main portion of the program will read in a value for n, and then enter a loop that generates and writes out each of the Fibonacci numbers. A function called fibonacci will be used to calculate each Fibonacci number from its two preceding values. This function will be called once during each pass through the main loop.

When fibonacci is entered, the computation of the current Fibonacci number, f, is very simple provided the two previous values are known. These values can be retained from one function call to the next if we assign them to the static variables f1 and f2, which represent F_{i-1} and F_{i-2}, respectively. (We could, of course, have used external variables for this purpose, but it is better to use local variables, since F_{i-1} and F_{i-2} are required only within the function.) We then calculate the desired Fibonacci number as

```
f = f1 + f2
```

and update the values of f2 and f1 using the formulas

```
f2 = f1
```

and

```
f1 = f
```

Here is the complete C program.

```
/* program to calculate successive Fibonacci numbers */

#include <stdio.h>

long int fibonacci(int count);

main()

{
    int count, n;

    printf("How many Fibonacci numbers? ");
    scanf("%d", &n);
    printf("\n");

    for (count = 1; count <= n; ++count)
        printf("\ni = %2d   F = %ld", count, fibonacci(count));
}


long int fibonacci(int count)

/* calculate a Fibonacci number using the formulas

   F = 1 for i < 3, and F = F1 + F2 for i >= 3 */

{
    static long int f1 = 1, f2 = 1;
    long int f;

    f = (count < 3) ? 1 : f1 + f2;
    f2 = f1;
    f1 = f;
    return(f);
}
```

Notice that long integers are used to represent the Fibonacci numbers. Also, note that `f1` and `f2` are static variables that are each assigned an initial value of 1. These initial values are assigned only once, at the beginning of the program execution. The subsequent values are retained between successive function calls, as they are assigned. You should understand that `f1` and `f2` are strictly local variables, even though they retain their values from one function call to another.

The output corresponding to a value of n = 30 is shown below. As usual, the user's response is underlined.

```
How many Fibonacci numbers? 30

i =  1      F = 1
i =  2      F = 1
i =  3      F = 2
i =  4      F = 3
i =  5      F = 5
i =  6      F = 8
i =  7      F = 13
i =  8      F = 21
i =  9      F = 34
i = 10      F = 55
i = 11      F = 89
i = 12      F = 144
i = 13      F = 233
```

```
i = 14      F = 377
i = 15      F = 610
i = 16      F = 987
i = 17      F = 1597
i = 18      F = 2584
i = 19      F = 4181
i = 20      F = 6765
i = 21      F = 10946
i = 22      F = 17711
i = 23      F = 28657
i = 24      F = 46368
i = 25      F = 75025
i = 26      F = 121393
i = 27      F = 196418
i = 28      F = 317811
i = 29      F = 514229
i = 30      F = 832040
```

It is possible to define and initialize static arrays as well as static single-valued variables. The use of arrays will be discussed in the next chapter.

8.5 MULTIFILE PROGRAMS

A *file* is a collection of information stored as a separate entity within the computer or on an auxiliary storage device. A file can be a collection of data, a source program, a portion of a source program, an object program, etc. In this chapter we will consider a file to be either an entire C program or a portion of a C program, i.e., one or more functions. (See Chap. 12 for a discussion of data files, and their relationship to C programs.)

Until now, we have restricted our attention to C programs that are contained entirely within a single file. Many programs, however, are composed of multiple files. This is especially true of programs that make use of lengthy functions, where each function may occupy a separate file. Or, if there are many small related functions within a program, it may be desirable to place a few functions within each of several files. The individual files will be compiled separately, and then linked together to form one executable object program (see Sec. 5.4). This facilitates the editing and debugging of the program, since each file can be maintained at a manageable size.

Multifile programs allow greater flexibility in defining the scope of both functions and variables. The rules associated with the use of storage classes become more complicated, however, because they apply to functions as well as variables, and more options are available for both external and static variables.

Functions

Let us begin by considering the rules associated with the use of functions. Within a multifile program, a function definition may be either *external* or *static*. An external function will be recognized throughout the entire program, whereas a static function will be recognized only within the file in which it is defined. In each case, the storage class is established by placing the appropriate storage-class designation (i.e., either `extern` or `static`) at the beginning of the function definition. The function is assumed to be *external* if a storage-class designation does not appear.

In general terms, the first line of a function definition can be written as

```
storage-class  data-type  name(type 1  arg 1,  type 2  arg 2, . . .,
                                                    type n  arg n)
```

where *storage-class* refers to the storage-class associated with the function, *data-type* refers to the data-type of the value returned by the function, *name* refers to the function name, *type 1, type 2, . . ., type n* refer to the formal argument types, and *arg 1, arg 2, . . ., arg n* refer to the formal arguments themselves. Remember that the storage-class, the data-type, and the formal arguments need not all be present in every function definition.

When a function is defined in one file and accessed in another, the latter file must include a function *declaration*. This declaration identifies the function as an external function whose definition appears elsewhere. Such declarations are usually placed at the beginning of the file, ahead of any function definitions.

It is good programming practice to begin the declaration with the storage-class specifier `extern`. This storage-class specifier is not absolutely necessary, however, since the function will be assumed to be external if a storage-class specifier is not present.

In general terms, a function *declaration* can be written as

```
storage-class  data-type  name(argument type 1,  argument type 2, . . .,
                                                          argument type n);
```

A function declaration can also be written using full function prototyping (see Sec. 7.4) as

```
storage-class  data-type  name(type 1  arg 1,  type 2  arg 2, . . .,
                                                          type n  arg n);
```

Remember that the storage-class, the data-type and the argument types need not all be present in every function declaration.

To execute a multifile program, each individual file must be compiled and the resulting object files linked together. To do so, we usually combine the source files within a *project*. We then *build* the project (i.e., compile all of the source files and link the resulting object files together into a single executable program). If some of the source files are later changed, we *make* another executable program (i.e., compile the new source files and link the resulting object files, with the unchanged object files, into a new executable program). The details of how this is done will vary from one version of C to another.

EXAMPLE 8.8 Here is a simple program that generates the message "Hello, there!" from within a function. The program consists of two functions: `main` and `output`. Each function appears in a separate file.

First file:

```
/* simple, multifile program to write "Hello, there!" */

#include <stdio.h>

extern void output(void);      /* function prototype */

main()
{
    output();
}
```

Second file:

```
extern void output(void)      /* external function definition */
{
    printf("Hello, there!");
    return;
}
```

Notice that `output` is assigned the storage class `extern`, since it must be accessed from a file other than the one in which it is defined; it must therefore be an external function. Hence, the keyword `extern` is included in both the function declaration (in the first file) and the function definition (in the second file). Since `extern` is a default storage class, however, we could have omitted the keyword `extern` from both the function declaration and the function definition. Thus, the program could be written as follows:

First file:

```
/* simple, multifile program to write "Hello, there!" */

#include <stdio.h>

void output(void);      /* function prototype */

main()
{
    output();
}
```

Second file:

```
void output(void)      /* external function definition */
{
    printf("Hello, there!");
    return;
}
```

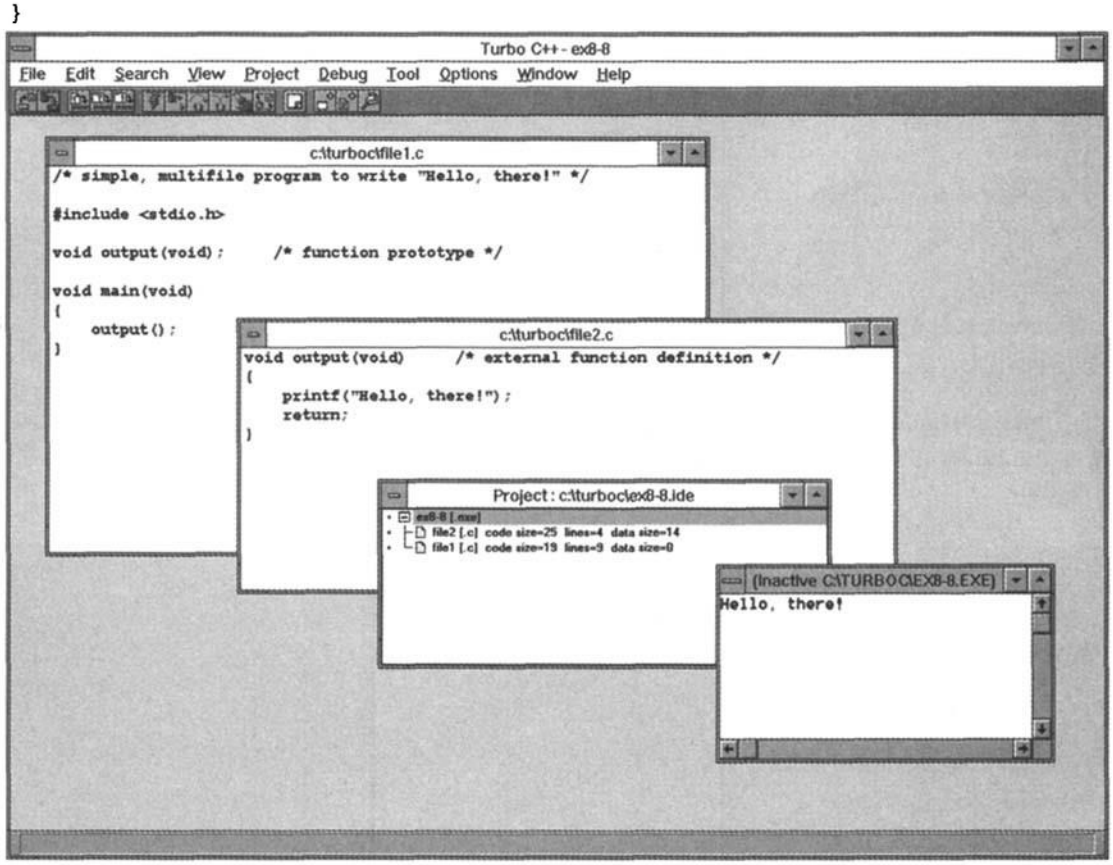

Fig. 8.4

Let us now build a Turbo C++ project corresponding to this multifile program. To do so, we first enter the source code shown in the first file, and save it in a file called `FILE1.C`. We then enter the source code shown in the second file, and save it in a file called `FILE2.C`. These two files are shown within separate windows in Fig. 8.4.

Next, we select `New` from the `Project` menu, and specify `EX8-8.IDE` as the project name. This will result in the `Project` window being opened, as shown near the center of Fig. 8.4. Within this window, we see that the project will result in an executable program called `EX8-8.EXE`. This executable program will be obtained from the previous two source files, `FILE1.C` and `FILE2.C`.

The program can now be executed by selecting `Run` from the `Debug` menu, as explained in Chap. 5 (see Example 5.4). The resulting message, `Hello, there!`, is displayed in the output window, as shown in the lower right portion of Fig. 8.4.

If a file contains a static function, it may be necessary to include the storage class `static` within the function declaration or the function prototype.

EXAMPLE 8.9 Simulation of a Game of Chance (Shooting Craps) Here is another version of the craps game simulation, originally presented in Example 7.11. In this version the program consists of two separate files. The first file contains `main`, whereas the second file contains the functions `play` and `throw`.

First file:

```
/* simulation of a craps game */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define  SEED   12345

extern void play(void);     /* function prototype */

main()
{
    char answer = 'Y';

    printf("Welcome to the Game of CRAPS\n\n");
    printf("To throw the dice, press RETURN\n\n");

    srand(SEED);     /* initialize the random number generator */

    /* main loop */

    while (toupper(answer) != 'N')   {
        play();
        printf("\nDo you want to play again? (Y/N) ");
        scanf(" %c", &answer);
        printf("\n");
    }
    printf("Bye, have a nice day");
}
```

Second file:

```
#include <stdio.h>
#include <stdlib.h>

static int throw(void);          /* function prototype */
extern void play(void)           /* external function definition */
```

```
/* simulate one complete game */

{
    int score1, score2;
    char dummy;

    printf("\nPlease throw the dice . . .");
    scanf("%c", &dummy);
    printf("\n");
    score1 = throw();
    printf("\n%2d", score1);

    switch (score1)   {

    case 7:   /* win on first throw */
    case 11:

        printf(" - Congratulations!  You WIN on the first throw\n");
        break;

    case 2:    /* lose on first throw */
    case 3:
    case 12:

        printf(" - Sorry, you LOSE on the first throw\n");
        break;

    case 4:    /* additional throws are required */
    case 5:
    case 6:
    case 8:
    case 9:
    case 10:

        do   {
            printf(" - Throw the dice again . . .");
            scanf("%c", &dummy);
            score2 = throw();
            printf("\n%2d", score2);
        }  while (score2 != score1 && score2 != 7);

        if (score2 == score1)
           printf(" - You WIN by matching your first score\n");
        else
           printf(" - You LOSE by failing to match your first score\n");
        break;
    }

    return;
}


/* simulate one throw of a pair of dice */

static int throw(void)     /* static function definition */

{
    float x1, x2;          /* random floating-point numbers between 0 and 1 */
    int n1, n2;            /* random integers between 1 and 6 */
```

```
    x1 = rand() / 32768.0;
    x2 = rand() / 32768.0;

    n1 = 1 + (int) (6 * x1);      /* simulate first die */
    n2 = 1 + (int) (6 * x2);      /* simulate second die */

    return(n1 + n2);              /* score is sum of two dice */
}
```

Notice that `play` is defined as an external function, so it can be accessed from `main` (because `main` and `play` are defined in separate files). Therefore, `play` is declared an external function within the first file. On the other hand, `throw` is accessed only by `play`. Both `throw` and `play` are defined in the second file. Hence `throw` need not be recognized in the first file. We can therefore define `throw` to be a static function, confining its scope to the second file.

Also, notice that each file has a separate set of `#include` statements for the header files `stdio.h` and `stdlib.h`. This ensures that the necessary declarations for the library functions are included in each file.

When the individual files are compiled and linked, and the resulting executable program is run, the program generates a dialog identical to that shown in Example 7.11, as expected.

Variables

Within a multifile program, external (global) variables can be defined in one file and accessed in another. We again emphasize the distinction between the *definition* of an external variable and its *declarations*. An external variable *definition* can appear in only one file. Its location within the file must be external to any function definition. Usually, it will appear at the beginning of the file, ahead of the first function definition.

External variable definitions may include initial values. Any external variable that is not assigned an initial value will automatically be initialized to zero. The storage-class specifier `extern` is not required within the definition; in fact, many versions of C specifically forbid the appearance of this storage-class specifier in external variable *definitions*. Thus, external variable definitions are recognized by their location within the defining files and by their appearance. We will follow this convention in this book.

In order to access an external variable in another file, the variable must first be *declared* within that file. This declaration may appear anywhere within the file. Usually, however, it will be placed at the beginning of the file, ahead of the first function definition. The declaration *must* begin with the storage-class specifier `extern`. Initial values *cannot* be included in external variable declarations.

The value assigned to an external variable may be altered within any file in which the variable is recognized. Such changes *will* be recognized in all other files that fall within the scope of the variable. Thus, external variables provide a convenient means of transferring information between files.

EXAMPLE 8.10 Shown below is a skeletal outline of a two-file C program that makes use of external variables.

First file:

```
int a = 1, b = 2, c = 3;      /* external variable DEFINITION */

extern void funct1(void);     /* external function DECLARATION */

main()                        /* function DEFINITION */
{
    . . . . .
}
```

Second file:

```
extern int a, b, c          /* external variable DECLARATION */

extern void funct1(void)    /* external function DEFINITION */
{
    . . . . .
}
```

The variables a, b and c are defined as external variables within the first file, and assigned the initial values 1, 2 and 3, respectively. The first file also contains a *definition* of the function `main`, and a *declaration* for the external function `funct1`, which is defined elsewhere. Within the second file we see the *definition* of `funct1`, and a *declaration* for the external variables a, b and c.

Notice that the storage-class specifier `extern` appears in both the *definition* and the *declaration* of the external function `funct1`. This storage-class specifier is also present in the *declaration* of the external *variables* (in the second file), but it does *not* appear in the *definition* of the external variables (in the first file).

The scope of a, b and c is the entire program. Therefore these variables can be accessed, and their values altered, in either file, i.e., in either `main` or `funct1`.

EXAMPLE 8.11 Search for a Maximum In Example 8.4 we presented a C program that determines the value of x which causes the function

$$y = x \cos(x)$$

to be maximized within a specified interval. We now present another version of this program, in which each of the three required functions is placed in a separate file.

First file:

```
/* find the maximum of a function within a specified interval */

#include <stdio.h>

double a, b, xl, yl, xr, yr, cnst = 0.0001;     /* external variable definition */

extern void reduce(void);                       /* external function prototype */
extern double curve(double xl);                 /* external function prototype */

main()                                          /* function definition */

{
    double xmax, ymax;

    /* read input data (interval end points) */

    printf("\na = ");
    scanf("%lf", &a);
    printf("b = ");
    scanf("%lf", &b);

    /* interval reduction loop */

    do
       reduce();
    while ((yl != yr) && ((b - a) > 3 * cnst));
```

```
    /* calculate xmax and ymax, and display the results */

    xmax = 0.5 * (xl + xr);
    ymax = curve(xmax);
    printf("\nxmax = %8.6lf   ymax = %8.6lf", xmax, ymax);
}
```

Second file:

```
/* interval reduction routine */

extern double a, b, xl, yl, xr, yr, cnst;       /* external variable declaration */

extern double curve(double xl);                 /* external function prototype */

extern void reduce(void)                        /* external function definition */

{
    xl = a + 0.5 * (b - a - cnst);
    xr = xl + cnst;
    yl = curve(xl);
    yr = curve(xr);

    if (yl > yr)   {     /* retain left interval */
       b = xr;
       return;
    }
    if (yl < yr)         /* retain right interval */
       a = xl;
    return;
}
```

Third file:

```
/*  evaluate the function y = x * cos(x)  */

#include <math.h>

extern double curve(double x)                   /* external function definition */

{
    return (x * cos(x));
}
```

The external function `reduce`, which is defined in the second file, is declared in the first file. Therefore its scope is the first two files. Similarly, the external function `curve`, which is defined in the third file, is declared in the first and second files. Hence, its scope is the entire program. Notice that the storage-class specifier `extern` appears in both the function definitions and the function prototypes.

Now consider the external variables `a`, `b`, `xl`, `yl`, `xr`, `yr` and `cnst`, which are defined in the first file. Observe that `cnst` is assigned an initial value within the definition. These variables are utilized, and hence declared, in the second file, but not in the third file. Note that the variable *declaration* (in the second file) includes the storage-class specifier `extern`, but the variable *definition* (in the first file) does not include a storage-class specifier.

Finally, notice the `#include <math.h>` statement at the beginning of the third file. This statement causes the header file `math.h` to be included in the third source file, in support of the `cos` library function.

Execution of this program results in output that is identical to that shown in Example 8.4.

Within a file, external variables can be defined as static. To do so, the storage-class specifier `static` is placed at the beginning of the definition. The scope of a static external variable will be the remainder of the file in which it is defined. It will not be recognized elsewhere in the program (i.e, in other files). Thus, the use of static external variables within a file permits a group of variables to be "hidden" from the remainder of a program. Other external variables having the same names can be defined in the remaining files. (Usually, however, it is not be a good idea to use identical variable names. Such identically named variables may cause confusion in understanding the program logic, even though they will not conflict with one another syntactically.)

EXAMPLE 8.12 Generating Fibonacci Numbers Let us return to the problem of calculating Fibonacci numbers, which we originally considered in Example 8.7. If we rewrite the program as a two-file program employing static external variables, we obtain the following complete program.

First file:

```
/* program to calculate successive Fibonacci numbers */

#include <stdio.h>

extern long int fibonacci(int count);     /* external function prototype */

main()                                    /* function definition */
{
    int count, n;

    printf("How many Fibonacci numbers? ");
    scanf("%d", &n);
    printf("\n");

    for (count = 1; count <= n; ++count)
        printf("\ni = %2d   F = %ld", count, fibonacci(count));
}
```

Second file:

```
/* calculate a Fibonacci number (F = 1 for i < 3, and F = F1 + F2 for i >= 3) */

static long int f1 = 1, f2 = 1;           /* static external variable definition */

long int fibonacci(int count)             /* external function definition */
{
    long int f;

    f = (count < 3) ? 1 : f1 + f2;
    f2 = f1;
    f1 = f;
    return(f);
}
```

In this program the function `fibonacci` is defined in the second file and declared in the first file, so that its scope is the entire program. On the other hand, the variables `f1` and `f2` are defined as static external variables in the second file. Their scope is therefore confined to the second file. Note that the variable definition in the second file includes the assignment of initial values.

Execution of this program results in output that is identical to that shown in Example 8.7.

8.6 MORE ABOUT LIBRARY FUNCTIONS

Our discussion of multifile programs can provide additional insight into the use of library functions. Recall that library functions are prewritten routines that carry out various commonly used operations or calculations (see Sec. 3.6). They are contained within one or more library files that accompany each C compiler.

During the process of converting a C source program into an executable object program, the compiled source program may be linked with one or more *library files* to produce the final executable program. Thus, the final program may be assembled from two or more separate files, even though the original source program may have been contained within a single file. The source program must therefore include declarations for the library functions, just as it would for programmer-defined functions that are placed in separate files.

One way to provide the necessary library-function declarations is to write them explicitly, as in the multifile programs presented in the last section. This can become tedious, however, since a small program may make use of several library functions. We wish to simplify the use of library functions to the greatest extent possible. C offers us a clever way to do this, by placing the required library-function declarations in special source files, called *header files*.

Most C compilers include several header files, each of which contains declarations that are functionally related (see Appendix H). For example, `stdio.h` is a header file containing declarations for input/output routines; `math.h` contains declarations for certain mathematical functions; and so on. The header files also contain other information related to the use of the library functions, such as symbolic constant definitions.

The required header files must be merged with the source program during the compilation process. This is accomplished by placing one or more `#include` statements at the beginning of the source program (or at the beginning of the individual program files). We have been following this procedure in all of the programming examples presented in this book.

EXAMPLE 8.13 Compound Interest Example 5.2 originally presented the following C program for carrying out simple compound interest calculations.

```
/* simple compound interest problem */

#include <stdio.h>
#include <math.h>

main()

{
    float p,r,n,i,f;

    /* read input data (including prompts) */

    printf("Please enter a value for the principal (P): ");
    scanf("%f", &p);
    printf("Please enter a value for the interest rate (r): ");
    scanf("%f", &r);
    printf("Please enter a value for the number of years (n): ");
    scanf("%f", &n);

    /* calculate i, then f */

    i = r / 100;
    f = p * pow((1 + i),n);

    /* display the output */

    printf("\nThe final value (F) is: %.2f\n", f);
}
```

This program makes use of two header files, `stdio.h` and `math.h`. The first header file contains declarations for the `printf` and `scanf` functions, whereas the second header file contains a declaration for the power function, `pow`.

We can rewrite the program if we wish, removing the `#include` statements and adding our own function declarations, as follows.

```
/* simple compound interest problem */

extern int printf();                    /* library function declaration */
extern int scanf();                     /* library function declaration */
extern double pow(double, double);      /* library function declaration */

main()

{
    float p,r,n,i,f;

    /* read input data (including prompts) */

    printf("Please enter a value for the principal (P): ");
    scanf("%f", &p);
    printf("Please enter a value for the interest rate (r): ");
    scanf("%f", &r);
    printf("Please enter a value for the number of years (n): ");
    scanf("%f", &n);

    /* calculate i, then f */

    i = r / 100;
    f = p * pow((1 + i),n);

    /* display the output */

    printf("\nThe final value (F) is: %.2f\n", f);
}
```

This version of the program is compiled in the same way as the earlier version, and it will generate the same output when executed. In practice the use of such programmer-supplied function declarations is not done, however, as it is more complicated and it provides additional sources of error. Moreover, the error checking that occurs during the compilation process will be less complete, because the argument types are not specified for the `printf` and `scanf` function. (Note that the number of arguments in `printf` and `scanf` can vary from one function call to another. The manner in which argument types are specified under these conditions is beyond the scope of our present discussion.)

Platform independence (i.e., *machine independence*) is a significant advantage in this approach to the use of library functions and header files. Thus, machine-dependent features can be provided as library functions, or as character constants or *macros* (see Sec. 14.4) that are included within the header files. A typical C program will therefore run on many different kinds of computers without alteration, provided the appropriate library functions and header files are utilized. The portability resulting from this approach is a major contributor to the popularity of C.

Review Questions

8.1 What is meant by the storage class of a variable?

8.2 Name the four storage-class specifications included in C.

8.3 What is meant by the scope of a variable within a program?

8.4 What is the purpose of an automatic variable? What is its scope?

8.5 How is an automatic variable defined? How is it initialized? What happens if an automatic variable is not explicitly initialized within a function?

8.6 Does an automatic variable retain its value once control is transferred out of its defining function?

8.7 What is the purpose of an external variable? What is its scope?

8.8 Summarize the distinction between an external variable definition and an external variable declaration.

8.9 How is an external variable defined? How is it initialized? What happens if an external variable definition does not include the assignment of an initial value? Compare your answers with those for automatic variables.

8.10 Suppose an external variable is defined outside of function A and accessed within the function. Does it matter whether the external variable is defined before or after the function? Explain.

8.11 In what way is the initialization of an external variable more restricted than the initialization of an automatic variable?

8.12 What is meant by side effects?

8.13 What inherent dangers are there in the use of external variables?

8.14 What is the purpose of a static variable in a single-file program? What is its scope?

8.15 How is a static variable defined in a single-file program? How is a static variable initialized? Compare with automatic variables.

8.16 Under what circumstances might it be desirable to have a program composed of several different files?

8.17 Compare the definition of functions within a multifile program with the definition of functions within a single-file program. What additional options are available in the multifile case?

8.18 In a multifile program, what is the default storage class for a function if a storage class is not explicitly included in the function definition?

8.19 What is the purpose of a static function in a multifile program?

8.20 Compare the definition of external variables within a multifile program with the definition of external variables within a single-file program. What additional options are available in the multifile case?

8.21 Compare external variable definitions with external variable declarations in a multifile program. What is the purpose of each? Can an external variable declaration include the assignment of an initial value?

8.22 Under what circumstances can an external variable be defined to be static? What advantage might there be in doing this?

8.23 What is the scope of a static external variable?

8.24 What is the purpose of a header file? Is the use of a header file absolutely necessary?

Problems

8.25 Describe the output generated by each of the following programs.

(*a*)

```
#include <stdio.h>

int funct1(int count);

main()
{
    int a, count;

    for (count = 1; count <= 5; ++count)   {
        a = funct1(count);
        printf("%d  ", a);
    }
}
```

```
funct1(int x)
{
    int y = 0;

    y += x;
    return(y);
}
```

(*b*)

```
#include <stdio.h>

int funct1(int count);

main()
{
    int a, count;

    for (count = 1; count <= 5; ++count)   {
        a = funct1(count);
        printf("%d  ", a);
    }
}

funct1(int x)
{
    static int y = 0;

    y += x;
    return(y);
}
```

(*c*)

```
#include <stdio.h>

int funct1(int a);
int funct2(int a);

main()
{
    int a = 0, b = 1, count;

    for (count = 1; count <= 5; ++count)   {
        b += funct1(a) + funct2(a);
        printf("%d  ", b);
    }
}

funct1(int a)
{
    int b;

    b = funct2(a);
    return(b);
}

funct2(int a)
{
    static int b = 1;

    b += 1;
    return(b + a);
}
```

8.26 Write the first line of the function definition for each of the situations described below.

(*a*) The second file of a two-file program contains a function called `solver` which accepts two floating-point quantities and returns a floating-point argument. The function will be called by other functions which are defined in both files.

(*b*) The second file of a two-file program contains a function called `solver` which accepts two floating-point quantities and returns a floating-point argument, as in the preceding problem. Recognition of this function is to remain local within the second file.

8.27 Add the required (or suggested) function declarations for each of the skeletal outlines shown below.

(*a*) This is a two-file program.

First file:

```
main()
{
    double x, y, z;

    . . . . .

z = funct1(x, y);

    . . . . .
}
```

Second file:

```
double funct1(double a, double b)
{
    . . . . .
}
```

(*b*) This is a two-file program.

First file:

```
main()
{
    double x, y, z;

    . . . . .

    z = funct1(x, y);

    . . . . .
}
```

Second file:

```
double funct1(double a, double b)
{
    double c;

    c = funct2(a, b);

    . . . . .
}
```

```
static double funct2(double a, double b)
{
    . . . . .
}
```

8.28 Describe the output generated by each of the following programs.

(*a*)
```
#include <stdio.h>

int a = 3;

int funct1(int count);

main()
{
    int count;

    for (count = 1; count <= 5; ++count)   {
        a = funct1(count);
        printf("%d  ", a);
    }
}

funct1(int x)
{
    a += x;
    return(a);
}
```

(*b*)
```
#include <stdio.h>

int a = 100, b = 200;

int funct1(int a, int b);

main()
{
    int count, c, d;

    for (count = 1; count <= 5; ++count)   {
        c = 20 * (count - 1);
        d = 4 * count * count;
        printf("%d  %d  ", funct1(a, c), funct1(b, d));
    }
}

funct1(int x, int y)
{
    return(x - y);
}
```

(*c*)
```
#include <stdio.h>

int a = 100, b = 200;

int funct1(int c);
```

```
main()
{
    int count, c;

    for (count = 1; count <= 5; ++count)   {
        c = 4 * count * count;
        printf("%d  ", funct1(c));
    }
}

funct1(int x)
{
    int c;

    c = (x < 50) ? (a + x) : (b - x);
    return(c);
}
```

(*d*)

```
#include <stdio.h>

int a = 100, b = 200;

int funct1(int count);
int funct2(int c);

main()
{
    int count;

    for (count = 1; count <= 5; ++count)
        printf("%d  ", funct1(count));
}

funct1(int x)
{
    int c, d;

    c = funct2(x);
    d = (c < 100) ? (a + c) : b;
    return(d);
}

funct2(int x)
{
    static int prod = 1;

    prod *= x;
    return(prod);
}
```

(*e*)

```
#include <stdio.h>

int funct1(int a);
int funct2(int b);
```

```
main()
{
    int a = 0, b = 1, count;

    for (count = 1; count <= 5; ++count)   {
        b += funct1(a + 1) + 1;
        printf("%d  ", b);
    }
}

funct1(int a)
{
    int b;

    b = funct2(a + 1) + 1;
    return(b);
}

funct2(int a)
{
    return(a + 1);
}
```

(*f*)

```
#include <stdio.h>

int a = 0, b = 1;

int funct1(int a);
int funct2(int b);

main()
{
    int count;

    for (count = 1; count <= 5; ++count)   {
        b += funct1(a + 1) + 1;
        printf("%d  ", b);
    }
}

funct1(int a)
{
    int b;

    b = funct2(a + 1) + 1;
    return(b);
}

funct2(int a)
{
    return(a + 1);
}
```

(g)
```
#include <stdio.h>

int a = 0, b = 1;

int funct1(int a);
int funct2(int b);

main()
{
    int count;

    for (count = 1; count <= 5; ++count)   {
        b += funct1(a + 1) + 1;
        printf("%d  ", b);
    }
}

funct1(int a)
{
    b = funct2(a + 1) + 1;
    return(b);
}

funct2(int a)
{
    return(b + a);
}
```

(*h*)
```
#include <stdio.h>

int count = 0;

void funct1(void);

main()
{
    printf("Please enter a line of text below\n");
    funct1();
    printf("%d", count);
}

void funct1(void)
{
    char c;

    if ((c = getchar()) != '\n')   {
       ++count;
       funct1();
    }
    return;
}
```

Programming Problems

8.29 The program given in Example 8.4 can easily be modified to *minimize* a function of x. This minimization procedure can provide us with a highly effective technique for calculating the roots of a nonlinear algebraic

equation. For example, suppose we want to find the particular value of x that causes some function $f(x)$ to equal zero. A typical function of this nature might be

$$f(x) = x + \cos(x) - 1 - \sin(x).$$

If we let $y(x) = f(x)^2$, then the function $y(x)$ will always be positive, except for those values of x that are roots of the given function [i.e., for which $f(x)$, and hence $y(x)$, will equal zero]. Therefore, any value of x that causes $y(x)$ to be minimized will also be a root of the equation $f(x) = 0$.

Modify the program shown in Example 8.4 to minimize a given function. Use the program to obtain the roots of the following equations:

(*a*) $x + \cos(x) = 1 + \sin(x), \qquad \pi/2 < x < \pi$

(*b*) $x^5 + 3x^2 + 10, \qquad 0 <= x <= 3$ (see Example 6.21)

8.30 Modify the program shown in Example 7.11 so that a sequence of craps games will be simulated automatically, in a noninteractive manner. Enter the total number of games as an input variable. Include within the program a counter that will determine the total number of wins. Use the program to simulate a large number of games (e.g., 1000). Estimate the probability of coming out ahead when playing multiple games of craps. This value, expressed as a decimal, is equal to the number of wins divided by the total number of games played. If the probability exceeds 0.500, it favors the player; otherwise it favors the house.

8.31 Rewrite each of the following programs so that it includes at least one programmer-defined function, in addition to the main function. Be careful with your choice of arguments and (if necessary) external variables.

(*a*) Calculate the weighted average of a list of numbers [see Prob. 6.69(*a*)].

(*b*) Calculate the cumulative product of a list of numbers [see Prob. 6.69(*b*)].

(*c*) Calculate the geometric average of a list of numbers [see Prob. 6.69(*c*)].

(*d*) Calculate and tabulate a list of prime numbers [see Prob. 6.69(*f*)].

(*e*) Compute the sine of x, using the method described in Prob. 6.69(*i*).

(*f*) Compute the repayments on a loan [see Prob. 6.69(*j*)].

(*g*) Determine the average exam score for each student in a class, as described in Prob. 6.69(*k*).

8.32 Write a complete C program to solve each of the problems described below. Utilize programmer-defined functions wherever appropriate. Compile and execute each program using the data given in the problem description.

(*a*) Suppose you place a given sum of money, A, into a savings account at the beginning of each year for n years. If the account earns interest at the rate of i percent annually, then the amount of money that will have accumulated after n years, F, is given by

$$F = A\,[(1 + i/100) + (1 + i/100)^2 + (1 + i/100)^3 + \cdots + (1 + i/100)^n]$$

Write a conversational-style C program to determine the following.

(*i*) How much money will accumulate after 30 years if \$1000 is deposited at the beginning of each year and the interest rate is 6 percent per year, compounded annually?

(*ii*) How much money must be deposited at the beginning of each year in order to accumulate \$100,000 after 30 years, again assuming that the interest rate is 6 percent per year, with annual compounding?

In each case, first determine the unknown amount of money. Then create a table showing the total amount of money that will have accumulated at the end of each year. Use the function written for Prob. 7.43 to carry out the exponentiation.

(*b*) Modify the above program to accommodate quarterly rather than annual compounding of interest. Compare the calculated results obtained for both problems. *Hint*: The proper formula is

$$F = A\,[(1 + i/100m)^m + (1 + i/100m)^{2m} + (1 + i/100m)^{3m} + \cdots + (1 + i/100m)^{nm}]$$

where m represents the number of interest periods per year.

(*c*) Home mortgage costs are determined in such a manner that the borrower pays the same amount of money to the lending institution each month throughout the life of the mortgage. The fraction of the total monthly payment that is required as an interest payment on the outstanding balance of the loan varies, however, from month to month. Early in the life of the mortgage most of the monthly payment is required to pay interest, and only a small fraction of the total monthly payment is applied toward reducing the amount of the loan. Gradually, the outstanding balance becomes smaller, which causes the monthly interest payment to decrease, and the amount available to reduce the outstanding balance therefore increases. Hence the balance of the loan is reduced at an accelerated rate.

Typically, prospective home buyers know how much money they must borrow and the time required for repayment. They then ask a lending institution how much their monthly payment will be at the prevailing interest rate. They should also be concerned with how much of each monthly payment is charged to interest, how much total interest they have paid since they first borrowed the money, and how much money they still owe the lending institution at the end of each month.

Write a C program that can be used by a lending institution to provide a potential customer with this information. Assume that the amount of the loan, the annual interest rate and the duration of the loan are specified. The amount of the monthly payment is calculated as

$$A = iP(1 + i)^n / [(1 + i)^n - 1]$$

where A = monthly payment, dollars

P = total amount of the loan, dollars

i = monthly interest rate, expressed as a decimal (e.g., 1/2 percent would be written 0.005)

n = total number of monthly payments

The monthly interest payment can then be calculated from the formula

$$I = iB$$

where I = monthly interest payment, dollars

B = current outstanding balance, dollars

The current outstanding balance is simply equal to the original amount of the loan, less the sum of the previous payments toward principal. The monthly payment toward principal (i.e., the amount which is used to reduce the outstanding balance) is simply

$$T = A - I$$

where T = monthly payment toward principal.

Use the program to calculate the cost of a 25-year, $50,000 mortgage at an annual interest rate of 8 percent. Then repeat the calculations for an annual interest rate of 8.5 percent. Make use of the function written for Prob. 7.43 to carry out the exponentiation. How significant is the additional 0.5 percent in the interest rate over the entire life of the mortgage?

(*d*) The method used to calculate the cost of a home mortgage in the previous problem is known as a *constant payment* method, since each monthly payment is the same. Suppose instead that the monthly payments were computed by the method of simple interest. That is, suppose that the same amount is applied toward reducing the loan each month. Hence

$$T = P / n$$

However, the monthly interest will depend on the amount of the outstanding balance; that is,

$$I = iB$$

Thus the total monthly payment, $A = T + I$, will decrease each month as the outstanding balance diminishes.

Write a C program to calculate the cost of a home mortgage using this method of repayment. Label the output clearly. Use the program to calculate the cost of a 25-year, $50,000 loan at 8 percent annual interest. Compare the results with those obtained in part (*c*) above.

(*e*) Suppose we are given a number of discrete points (x_1, y_1), (x_2, y_2), . . . , (x_n, y_n) read from a curve $y = f(x)$, where x is bounded between x_1 and x_n. We wish to approximate the area under the curve by breaking up the curve into a number of small rectangles and calculating the area of these rectangles. (This is known as the *trapezoidal rule*.) The appropriate formula is

$$A = (y_1 + y_2)(x_2 - x_1)/2 + (y_2 + y_3)(x_3 - x_2)/2 + \cdots + (y_{n-1} + y_n)(x_n - x_{n-1})/2$$

Notice that the average height of each rectangle is given by $(y_i + y_{i+1})/2$ and the width of each rectangle is equal to $(x_{i+1} - x_i)$; $i = 1, 2, \ldots, n - 1$.

Write a C program to implement this strategy, using a function to evaluate the formula $y = f(x)$. Use the program to calculate the area under the curve $y = x^3$ between the limits $x = 1$ and $x = 4$. Solve this problem first with 16 evenly spaced points, then with 61 points, and finally with 301 points. Note that the accuracy of the solution will improve as the number of points increases. (The exact answer to this problem is 63.75.)

(*f*) Part (*e*) above describes a method known as the *trapezoidal rule* for calculating the area under a curve $y(x)$, where a set of tabulated values (x_1, y_1), (x_2, y_2), . . . , (x_n, y_n) is used to describe the curve. If the tabulated values of x are equally spaced, then the equation given in the preceding problem can be simplified to read

$$A = (y_1 + 2y_2 + 2y_3 + 2y_4 + \cdots + 2y_{n-1} + y_n)h/2$$

where h is the distance between successive values of x.

Another technique that applies when there is an even number of equally spaced intervals (i.e., an odd number of data points) is *Simpson's rule*. The computational equation for implementing Simpson's rule is

$$A = (y_1 + 4y_2 + 2y_3 + 4y_4 + 2y_5 + \cdots + 4y_{n-1} + y_n)h/3$$

For a given value of h, this method will yield a more accurate result than the trapezoidal rule. (Note that the method requires about the same amount of computational complexity as the trapezoidal rule.)

Write a C program for calculating the area under a curve using either of the above techniques, assuming an odd number of equally spaced data points. Implement each method with a separate function, and utilize another independent function to evaluate $y(x)$.

Use the program to calculate the area under the curve

$$y = e^{-x}$$

where x ranges from 0 to 1. Calculate the area using each method, and compare the results with the correct answer of $A = 0.7468241$.

(*g*) Still another technique for calculating the area under a curve is to employ the *Monte Carlo* method, which makes use of randomly generated numbers. Suppose that the curve $y = f(x)$ is positive for any value of x between the specified lower and upper limits $x = a$ and $x = b$. Let the largest value of y within these limits be y^*. The Monte Carlo method proceeds as follows.

(*i*) Begin with a counter set equal to zero.

(*ii*) Generate a random number, r_x, whose value lies between a and b.

(*iii*) Evaluate $y(r_x)$.

(*iv*) Generate a second random number, r_y, whose value lies between 0 and y^*.

(*v*) Compare r_y with $y(r_x)$. If r_y is less than or equal to $y(r_x)$, then this point will fall on or under the given curve. Hence the counter is incremented by 1.

(*vi*) Repeat steps (*ii*) through (*v*) a large number of times. Each time will be called a *cycle*.

(*vii*) When a specified number of cycles has been completed, the fraction of points that fell on or under the curve, F, is computed as the value of the counter divided by the total number of cycles. The area under the curve is then obtained as

$$A = Fy^*(b - a).$$

Write a C program to implement this strategy. Use this program to find the area under the curve $y = e^{-x}$ between the limits $a = 0$ and $b = 1$. Determine how many cycles are required to obtain an answer that is accurate to three significant figures. Compare the computer time required for this problem with the time required for the preceding problem. Which method is better?

(*h*) A normally distributed random variate x, with mean μ and standard deviation σ, can be generated from the formula

$$x = \mu + \sigma \frac{\sum_{i=1}^{N} r_i - N/2}{\sqrt{N/12}}$$

where r_i is a uniformly distributed random number whose value lies between 0 and 1. A value of $N = 12$ is frequently selected when using this formula. The underlying basis for the formula is the *central limit theorem*, which states that a set of mean values of uniformly distributed random variates will tend to be normally distributed.

Write a C program that will generate a specified number of normally distributed random variates with a given mean and a given standard deviation. Let the number of random variates, the mean and the standard deviation be input quantities to the program. Generate each random variate within a function that accepts the mean and standard deviation as arguments.

(*i*) Write a C program that will allow a person to play a game of tic-tac-toe against the computer. Write the program in such a manner that the computer can be either the first or the second player. If the computer is the first player, let the first move be generated randomly. Write out the complete status of the game after each move. Have the computer acknowledge a win by either player when it occurs.

(*j*) Write a complete C program that includes a recursive function to determine the value of the nth Fibonacci number, F_n, where $F_n = F_{n-1} + F_{n-2}$ and $F_1 = F_2 = 1$ (see Example 8.7). Let the value of n be an input quantity.