Chapter 5

Using Functions and Libraries

In this chapter, we introduce the most important tool C provides for writing manageable, debuggable programs: the function. In modern programming practice, programs are written as a collection of functions connected through well-defined interfaces. We show how to use standard library functions, functions from a personal library, and the programmer's own (programmer-defined) functions.

Functions are important because they provide a way to modularize code so that a large complex program can be written by combining many smaller parts. A **function** is a named block of code that performs a specified task when called. Many functions require one or more arguments. Each **argument** is an object or piece of information that the function can use while carrying out its specified task. Four functions were introduced in Chapter 3: `puts()`, `printf()`, `scanf()`, and `main()`. The first two perform an output task, the third performs input, while the fourth exists in every program and indicates where to begin execution.

Building a program is like building a computer. Today's computer is built by connecting boards. Each board is a group of connected chips, which consist of an integrated group of circuit components constructed by connecting logic elements.

A large program is constructed similarly. At the top level, the program includes several modules, where each module is developed separately (and stored in a separate file.) Each module is composed of object declarations and functions. These functions, in turn, call other functions.

In a well-designed program, the purpose, or task, of each function is clear and easy to describe. All its actions hang together and work at the same level of detail. No function is very long or very complex; each is short enough to comprehend in its entirety. Complexity is avoided by creating and calling other functions to do subtasks. In this way, a highly complex job can be broken into short units that interact with each other in controlled ways. This allows the whole program to be constructed and verified much more easily than a similar program written as one massive unit. Each function, then each module, is developed and debugged before it is inserted into the final, large program. This is how professional programmers have been able to develop the large, sophisticated systems we use today.

One function is special. In C, the main program is a special function. In most ways, it is like any function, but `main()` is different in three significant ways:

- Every program must have a `main()` function.
- `main()` is the only function with two standard prototypes:

```
int main( void ); /* appropriate for simple programs */
int main( int argc, char* argv[] ); /* used by some advanced programs */
```

- Both prototypes of `main()` are known to the compiler; they do not need to be declared.

5.1 Libraries

We use functions from a variety of sources. Many come to us as part of a library, which is a collection of related functions that can be incorporated into your own code. The **standard libraries** are defined by the C language standard and are part of every C compiler. In addition, software manufacturers often create proprietary libraries that are distributed with the C translator and provide facilities not covered by the standard. Often, a group of computer users shares a collection of functions that become a **personal library**. An individual programmer might use functions from all of these sources and normally also defines his or her own functions that are tailored to the tasks of a particular program.

Using library functions lets a programmer take advantage of the skill and knowledge of experts. In modern programming practice, code libraries are used extensively to increase the reliability and quality of programs and decrease the time it takes to write them. A good programmer does not reinvent the wheel.

5.1.1 Standard Libraries

C has about a dozen standard libraries; we use six of them in this text. The first one encountered by a beginning C programmer is the standard input/output library (`stdio`), which contains the functions `scanf()` and `printf()` that we have been using since Chapter 3. This library also contains functions for the input and output of specific data types, which will be explained as those types are introduced, as well as functions for file handling, which will be explained in Chapter 14.

The second most commonly used library is the mathematics library (`math`). This library contains implementations of mathematical functions such as `sin()`, `cos()` and `log`. The program examples in this chapter illustrate the use of several of these functions; a complete list is given in Figure 5.6.

Another important library is the standard library (`stdlib`). It contains functions for generating random numbers; a definition for `abs()`, the absolute value function for integers; and a variety of general utility functions. Several of these will be introduced as the need arises in later chapters. Other libraries that we use are the time library (`time`), the string library (`string`), and the character-handling library (`ctype`).

5.1.2 Other Libraries

The standard C libraries contain many useful functions for input and output, string handling, mathematical computations, and systems programming tasks. These libraries provide expert solutions for common needs, but they cannot cover every possible need. The standard libraries contain only a fraction of the useful functions that might be written. The library functions are general-purpose utilities; many were designed for the convenience of programmers creating the UNIX operating system. They are not tailored to the needs of students or scientists and engineers who write C programs in the course of their work.

Many C implementations have additional libraries; for example, a graphics library for building screen displays. Special-purpose libraries are often included with hardware that will be connected to a computer. For example, a mobile-robotics kit such as `Lego Mindstorm` includes a library of functions that are used to communicate with the robotics hardware. Finally, many companies that create software have libraries of code relating to their products; by sharing these libraries, employees can become more efficient and products become more uniform and predictable.

Programmers define their own functions to meet their needs. Some are special-purpose functions written for one application and not relevant to other jobs. However, every programmer builds a collection of function definitions that are useful again and again. These usually are simple functions that save a little writing, simplify a repetitive task, or make a programmer's job easier. Often, such a collection is shared with coworkers and becomes a personal library. In this chapter, we suggest several functions that you may wish to put into your own personal library because they will be useful again and again. We call this library "mytools"; it is discussed in Section 5.9.2.

5.1.3 Using Libraries

Prototypes Every data object has a type. The type of a literal is evident from its form; the type of a variable is declared along with the variable name. Similarly, every function has a type, called a **prototype**, which must be known before the function can be used. The prototype defines the function's **interface**; that is, how it is supposed to interact with other functions. It declares the number and types of arguments that must be provided in every call and the kind of answer that the function computes and returns (if any). This information allows the compiler to check for syntax errors in the function calls.

Header files. Each standard library has a corresponding header file whose name ends in `.h`. The **header file** for a library contains the prototype declarations for all of the functions in that library. It may also define related types and constants. For example, the header `time.h` defines a type named `time_t`, which can be used to store the current date and time. This type is related to type `int`, and is chosen to be appropriate for the local computer hardware and software. A useful constant, `INT_MAX` (the largest representable integer) is defined in `limits.h`. Also, the mathematical constant `PI` is defined in `math.h` by many C implementations.

Nine tenths of an iceberg floats under the surface of the water; we see only a small part on top. Similarly, each C library has two parts: a small public header file that declares the interface for the library and a large, concealed code file that supports the public interface.

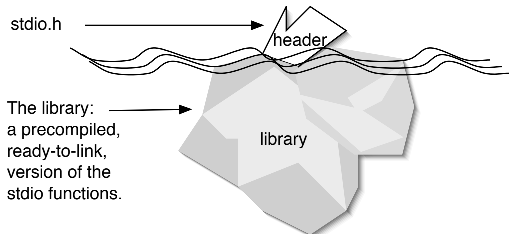

stdio.h
header
The library: a precompiled, ready-to-link, version of the stdio functions.
library

Figure 5.1. A library is like an iceberg: only a small part is visible.

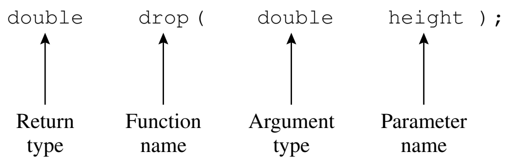


Figure 5.2. Form of a simple function prototype.

The header files for the libraries we have mentioned so far are

Standard input/output library: `<stdio.h>`	Standard library: `<stdlib.h>`
Mathematics library: `<math.h>`	Time library: `<time.h>`
Character handling: `<ctype.h>`	String library: `<string.h>`
Biggest and smallest integers `<limits.h>`	Personal library: `"mytools.h"`

To use one of the library functions in a program, you must include the corresponding header file in your program. This can be done explicitly, by writing an `#include` command for that library, or you can make your own header file that includes the header files for the standard libraries that will be used. Suppose you have such a file called `mytools.h`, that includes `stdio.h`. Then if you write the command `#include "mytools.h"` in your program, there is no need to write `#include <stdio.h>` separately.

In an `#include` command, angle brackets `<...>` around the name of the header file indicate that the header and its corresponding library are installed in the compiler's standard library area on the hard disk[1]. Use quotation marks instead of angle brackets for personal libraries like `mytools` that are stored in the programmer's own disk directory rather than in the standard system directory.

[1]This file must be stored where your compiler can find it. It always works to put it in the same directory as your program code, or in any directory that is on the compiler's "search path". To find out about the search path, ask your system administrator to help you.

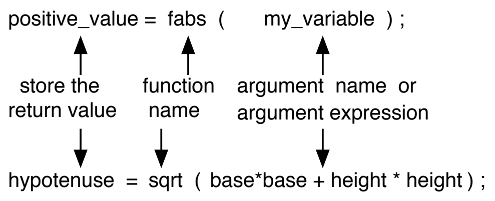

positive_value = fabs (my_variable) ;
store the return value
function name
argument name or argument expression
hypotenuse = sqrt (base*base + height * height) ;

Figure 5.3. Form of a simple function call.

5.2 Function Calls

A **function call** causes the function's code to be executed. The normal sequential **flow of control** is **interrupted**, and control is transferred to the beginning of the function. At the end of the function, control returns to the point of interruption. To call a function, write the name of the function followed by a pair of parentheses enclosing a list of zero or more **arguments**. In the following discussion, we refer to the function that contains the call as the **caller** and to the called function as the **subprogram** or, if there is no ambiguity, simply the **function**. Copies of the argument values are made by the caller and sent to the subprogram, which uses these arguments in its calculations. At the end of function execution, control returns to the caller; a **function result** also may be returned.

The function call must supply an argument value for each parameter defined by the function's prototype. The form of a simple prototype declaration is shown in Figure 5.2. It starts with the type of the answer returned by the function. This is followed by the function name and a pair of parentheses. Within the parentheses are zero or more **parameter declaration** units, consisting of a type and an identifier. The type tells us what kind of argument is expected whenever the function is called. The identifier is optional in a prototype (but required in a function definition).

The C compiler checks every function call to ensure that the correct number of arguments has been provided and that every argument is an appropriate type for the function, according to the function's prototype. It also checks that the function's result is used in an appropriate context. Generally, if a mismatch is found between the function's prototype and the function call, the compiler generates an error comment and does not produce a translated version of the code. Some type mismatches are legal according to the type rules of C but they may not be meaningful in the context of the program. In such cases, the compiler generates a warning comment, continues the translation, and produces an executable program. However, the programmer should never ignore warnings; most warnings are clues about logic errors in the program.

Calling library functions. The program example in Figure 5.4 demonstrates how to include the library header files in your code and how to call the library functions. It uses standard I/O functions, the `sqrt()` function from the mathematics library, and a function from the `stdlib` library to abort execution after an input error.

```
/* ---------------------------------------------------------------------
// Grapefruits and Gravity again, with terminal velocity, using sqrt().
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define GRAVITY 9.8

int main( void )
{
    double height;                  /* height of fall (m) */
    double time;                    /* time of fall (s) */
    double velo;                    /* terminal velocity (m/s) */

    printf( " Grapefruits and Gravity with Functions\n\n"
            " Calculate the time it would take for a grapefruit\n"
            " to fall from a helicopter at a given height.\n"
            " Enter height of helicopter (meters):  " );
    scanf( "%lg", &height );        /* keyboard input for height */

    if (height < 0) {               /* exit gracefully after error */
        printf( " Error: height must be >= 0.  You entered %g\n", height );
        exit( EXIT_FAILURE );       /* abort execution */
    }

    time = sqrt( 2 * height / GRAVITY );  /* calculate the time of fall */
    velo = GRAVITY*time;                  /* terminal velocity of fruit */
    printf( "     Time of fall = %g seconds\n", time );
    printf( "     Velocity of the object = %g m/s\n", velo );

    return EXIT_SUCCESS;
}
```

Figure 5.4. Calling library functions.

Notes on Figure 5.4. Calling library functions.

First box: the* `#include` *commands.

- We `#include <stdio.h>` so that we can use `printf()` and `scanf()`.
- We `#include <math.h>` for `sqrt()`, the square root function,
- We `#include <stdlib.h>` for the `exit()` function, which is discussed in Section 5.2.3.

Second box: calling* `exit()`*.

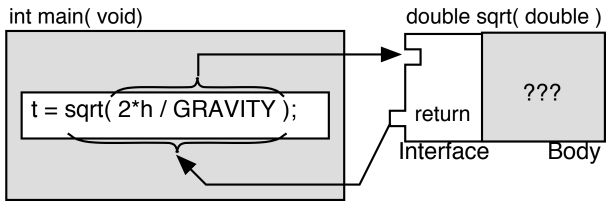
- The function `exit()` is defined in `stdlib`; its prototype is `void exit( int );`. It can be used to exit from the middle of a program after an error that makes continuation meaningless.
- The symbol `EXIT_FAILURE` is defined as 1. We use the symbolic name here, rather than a literal 1, as a form of program documentation.
- We use a simple `if` statement to test for an input error. If one is found, we display an error message, then call `exit(EXIT_FAILURE)` or `exit( 1 )`. When control returns to the system, it will display a termination comment with the exit code that written in the call on `exit()`.

```
 Grapefruits and Gravity with Functions

 Calculate the time it would take for a grapefruit
 to fall from a helicopter at a given height.
 Enter height of helicopter (meters): -2
 Error: height must be >= 0.  You entered -2

Grapefruits has exited with status 1.
```

- No `else` statement is needed because `exit()` takes control immediately and directly to the end of the program. In the flow diagram (Figure 5.17), it is diagrammed as a bolt of lightning because it "short-circuits" all the normal control structures.

Third box: calling* `sqrt()`*.

- The `sqrt()` function computes the square root of its argument. To call this function, we write the argument in parentheses after the function name. In this call, the argument is the value of the expression `2*h/GRAVITY`. The multiplication and division will be done, and the result will be sent to the `sqrt()` function and become the value of the `sqrt()`'s parameter. Then the square root will be calculated and returned to the caller. When a function returns a result, the caller must do something with it. In this case, it is stored in the variable named `t`, then used in the next two lines of `main()`.
- A function prototype describes the function's interface; that is, the argument(s) that must be brought into the function and the type of result that is sent back. In this case, one argument is brought in and one result is returned. We can diagram the passage of information to and from the function like this:



- In general, spacing makes no difference to the compiler. We could have written

  `time=by the standard to be an ( 2*height/GRAVITY);` or
  `time =sqrt (2 *height / GRAVITY);` or
  `time= sqrt (2* height/GRAVITY);`

  However, spacing makes an important difference in the readability of a program. You should use spacing selectively to make formulas as readable as possible. Current style guidelines call for spaces after the opening parenthesis and before the closing parenthesis.

The output from a successful run.

```
 Grapefruits and Gravity with Functions

 Calculate the time it would take for a grapefruit
 to fall from a helicopter at a given height.
 Enter height of helicopter (meters):  30
      Time of fall = 2.47436 seconds
      Velocity of the object = 24.2487 m/s

Grapefruits has exited with status 0.
```

Fourth box: the* `return` *statement.

- In previous programs, we have written `return 0;`. Here we introduce another way to write the same thing: `return EXIT_SUCCESS`. The `stdlib.h` header file defines `EXIT_SUCCESS` to be a synonym for 0, and some programmers prefer to use the symbolic name rather than the numeric value.
- The last line of the output, above, shows the message printed by the operating system after the program terminated. The status code that is displayed is the value that was written in the return statement.

5.2.1 Call Graphs

We use flow diagrams (as in Figure 5.17) to visualize the flow of control through the statements of a single function or the transition from one function to another during execution. Another kind of diagram, a **function call graph**, is useful for showing the relationships between functions that are established by function calls. In a function call graph (see Figure 5.5), a box at the top is used to represent the function `main()`. Below it, attached to the branches of a bracket, is one box for each of the functions called by the

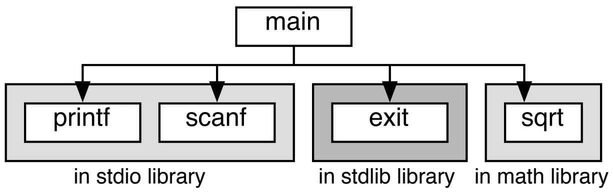


Figure 5.5. Call graph for Figure 5.4: Calling library functions.

main program.[2] As far as possible, these are listed left to right in the order in which they appear in the program. Each function has only one box; if it is called several times, there is no sign of that in the diagram. A very simple program is graphed in Figure 5.5.

The pattern of one box pointing at others is repeated when diagramming a more complex program; a box and a connection are drawn for each function called by a first-level function. In Figure 5.12, the main program calls a programmer-defined function, `banner()` (Figure 5.13), which in turn, calls functions named `time()` and `ctime()` from the `time` library. The resulting function call graph, shown in Figure 5.14, has boxes at three levels. In a large program, a function call graph will have many boxes at several levels and may have complex dependencies among the functions (arrows may point from a lower level to an upper level). We will use call graphs to visualize the relationships among functions in future program examples. The graph becomes more and more important as the number of functions increases and the interactions among functions become more complex.

In some programs it becomes beneficial to add information to the call graph concerning what is being passed between the functions. This would require an arrow for every parameter in every function call. If a program makes many function calls, this can get very complicated very quickly. Therefore, for the sake of clarity, we omit parameter and return information from the call graphs presented in this text.

5.2.2 Math Library Functions

Most of the functions in the math library are familiar to anyone who has studied high-school algebra and trigonometry. These include the trigonometric, exponential, log, and square root functions. Some of the math library functions are less familiar; these are briefly explained in the next several paragraphs, with a few important functions from other libraries.

int abs(int n). Although this is a mathematical function, it is part of the standard library (stdlib), not the math library; to use it, a program must `#include <stdlib.h>`. The result is the absolute value of n.

[2]Various elaborations of this basic scheme are in use. In one version, the function arguments are written on the arrows. We choose to introduce the concepts by using the simplest scheme.

Name	Function	Argument type(s)	Return type
`fabs(x)`	Absolute value	`double`	`double`
`ceil(x)`	Round x up	`double`	`double`
`floor(x)`	Round x down	`double`	`double`
`rint(x)`	Round x to nearest integer	`double`	`double`
`cos(x)`	Cosine of x	`double`	`double`
`sin(x)`	Sine of x	`double`	`double`
`tan(x)`	Tangent of x	`double`	`double`
`acos(x)`	Arc cosine of x	`double`	`double`
`asin(x)`	Arc sine of x	`double`	`double`
`atan(x)`	Arc tangent of x	`double`	`double`
`atan2(y, x)`	Arc tangent of y/x	`double, double`	`double`
`cosh(x)`	Hyperbolic cosine of x	`double`	`double`
`sinh(x)`	Hyperbolic sine of x	`double`	`double`
`tanh(x)`	Hyperbolic tangent of x	`double`	`double`
`exp(x)`	e^x	`double`	`double`
`log(x)`	Natural log of x	`double`	`double`
`log10(x)`	Base 10 log of x	`double`	`double`
`sqrt(x)`	Square root	`double`	`double`
`pow(x, y)`	x^y	`double, double`	`double`
`fmod(x,y)`	$x - N \times y$ for largest N such that $N \times y < x$	`double, double`	`double`

Figure 5.6. Functions in the math library.

(That is, if n is negative, the result has the same value as the argument, but with a positive sign instead of a negative sign. If the argument is zero or positive, the result is the same as the argument.)

double fabs(double x). To use this function, `#include <math.h>`. This is just like `abs()` but it works for a `double` argument instead of an `int`, and it returns a `double` result. The result is the absolute value of x. (That is, if x is negative, the result has the same value as the argument, but with a positive sign instead of a negative sign. If the argument is zero or positive, the result is the same as the argument.)

double fmod(double x, double y). To use this function, `#include <math.h>`. C has an operator, `%`, that computes the modulus function on integers; `fmod()` computes a related function for two `double`s. The result of `fmod(x, y)` is the floating-point remainder of `x/y` More precisely, `fmod(x, y) = x - k*fabs(y)` for some integer value `k`. The result has the same sign as `x` and is less than the absolute value of `y`. The function is implementation-defined if `y==0`. A few examples might make this clear:

```
fmod( 10.0, 2.0 ) = 0        and k = 5
fmod( -10.0, 2.9 ) = -1.3    and k = 3
```

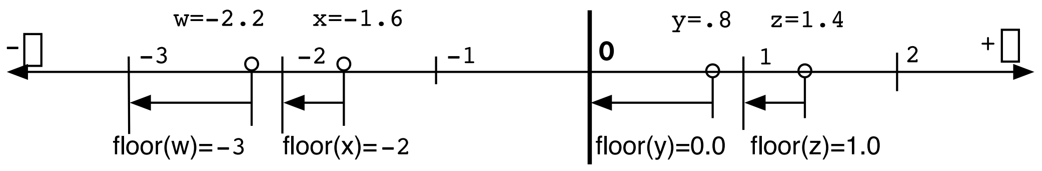


Figure 5.7. Rounding down: `floor()`.

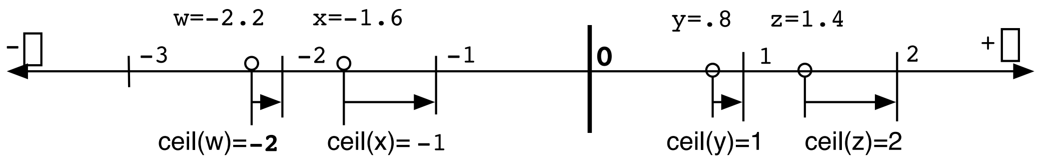


Figure 5.8. Rounding up: `ceil()`.

```
fmod( 10.5, -1.0 ) = .5      and k = 10
fmod( 10.5, 1.1 ) = .6       and k = 9
fmod( 34.5678, .01 ) = 0.0078
```

The last example, above, hints at an application for `fmod()`. Suppose a bank calculates the amount of interest due on an account, but will add only an even number of cents to the account balance. The fractional cents must be subtracted from the calculated interest before adding the interest to the account balance. This function lets us easily calculate the fractional cents.

double atan2(double x, double y). This function computes the arc tangent of `x/y`. It is explicitly defined for `y=0` is $\pi/2$ and has the same sign as `x`. This should be used in place of `atan()` for any argument expression that might have a denominator of 0.

Rounding and truncation. Suppose that we are given a `double` value and wish to eliminate the fractional part. The math library provides three functions that correspond to three of basic ways to do this job. The assignment operator provides a fourth way.

- The function `floor()` rounds down, that is, the result will be an integral value that is closer to $-\infty$ than the argument, as illustrated in Figure 5.7. This function returns a `double` value. For example, `floor(-1.6)` is `-2.0` and `floor(.9999999)` is `0.0`.
- The function `ceil()` (short for ceiling) rounds up, that is, the result will be an integral value that is closer to $+\infty$ than the argument, as illustrated in Figure 5.8. This function returns a `double` value.

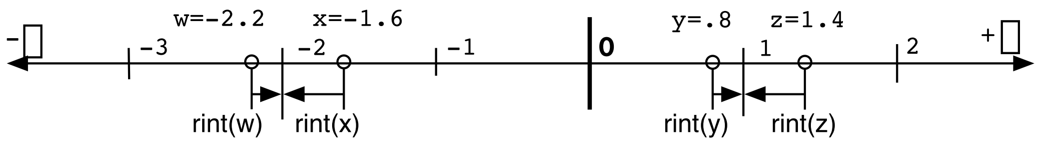


Figure 5.9. Rounding to the nearest integer: `rint()`.

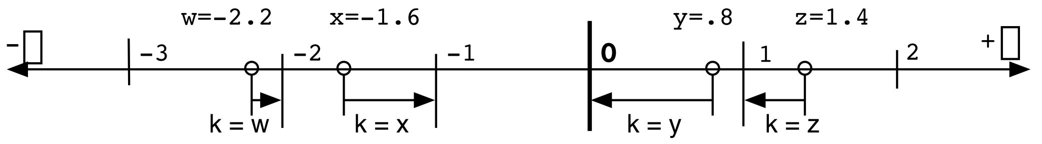


Figure 5.10. Truncation via assignment.

- The function `rint()` does what we normally refer to as "rounding". It rounds to the nearest integral value and returns it as a `double` value, as illustrated in Figure 5.9. A matching function, `lrint()`, returns the result as an integer, if it is possible to represent it as an integer. For example, `rint(-1.6)` is `-2.0` and `rint(.9999999)` is `1.0`. Compare this result to `lrint(.9999999)` = `1`
- When a double value is assigned to an integer variable, the fractional part is *truncated*, that is, the decimal places are simply discarded. This is the same as rounding positive numbers down (toward 0) and rounding negative numbers up (toward 0).

5.2.3 Other Important Library Functions

The date and time. The date and time at which a program is executed is very important for many kinds of output, including student work. Modern computers have an internal battery-operated clock. Modern systems keep that clock accurate by periodically synchronizing it to a time standard accessed over the internet. C provides a variety of functions and type definitions to help programmers use the clock[3]. Most of the time library is too difficult for this chapter, but a few items can be used simply and are presented here.

- The data type `time_t` is defined[4] by the standard to be an integer that has enough bits to hold whatever encoding of date and time is used by the local system. It might be different from system to system, but it is always exactly right to store the time. The program in Figure 5.13 shows how to declare a variable of this type and use it to store the current time.

[3]These are described in Appendix F and parts are discussed in detail in Chapter 12.

[4]Until now, we have covered only three types: `double`, `int`, and `char`. The C language actually supports many predefined types and permits the programmer to define his own. These will be introduced gradually in future chapters.

Library	Purpose	Name and Usage	Argument	Return type
stdlib	Absolute value	`k = abs( amount );`	`int`	`int`
stdlib	Abort execution after an error	`exit(1)`	`int`	returns to system
time	Data type for storing the time	`time_t now;`		creates a time variable
time	Read system clock	now = time(NULL)	NULL	the time encoded as an integer
time	Convert time code for printing	printf(ctime(&now))	integer	a printable string

Figure 5.11. A few functions in other libraries.

- The `time( )` function reads the system clock and returns an integer encoding of the time and date. It is described in Appendix F and discussed in more detail in Chapter 12. Until then, if you want to read the system clock, you should call the `time( NULL )` and store the result in a `time_t` variable.
- The `ctime( )` function is used to convert the time from the coded form to a string that can be easily printed and understood. The easiest way to use `ctime( )` is to call the function from the argument list of a `printf()` function. The argument to `ctime( )` must be the address of the `time_t` variable that was set by calling `time(NULL)`. Figure 5.13 shows how to do this.

Exception conditions. If a program encounters a serious error and cannot meaningfully continue, the appropriate action is to stop execution immediately. C++ and Java are newer languages that are built upon C; both include all the C operators and control structures, but provide an additional modern facility, called an exception handler, for managing errors and other unusual and unexpected conditions. C is an old language and it does not provide an exception handler. However, it does provide a function that aborts a program and "cleans up" the environment before returning to the system. In this text, the `exit()` function will be used when it would be appropriate to use an exception in a modern language.

- The function `exit()` is defined in `stdlib`; its prototype is `void exit( int );`. It can be used to exit from the middle of a program after an error that makes continuation meaningless.
- The constant `EXIT_FAILURE` is defined in `stdlib.h` as a synonym for the number 1 and the constant `EXIT_SUCCESS` is defined as a synonym for the number 0. These can be used as arguments to `exit()`, but a programmer can also use any integer as the argument or invent his or her own codes. The system should display the code on the screen after the program exits.

5.3 Programmer-Defined Functions

Functions serve three purposes in a program: They make it easy to use code written by someone else; they make it possible to reuse your own code in a new context; most important, though, they permit breaking a large program into small pieces in such a way that the interface between pieces is fixed and controllable. A programmer may (and generally does) modularize his or her program by dividing the entire job into smaller tasks and writing a **programmer-defined function** for each task.

Functions can take no arguments or many, of any combination of types, and can return or not return values. The type of a function is a composite of the type of value it returns and the set of types of its arguments.

Function	Prototype
`main()`	`int main( void );`
`exit()`	`void exit( int );`
`sqrt ()`	`double sqrt( double );`

We begin the study of programmer-defined functions by creating functions of two types: double→double and void→void.

Double→double functions. Some functions calculate and return values when they are called. For example, in Figure 5.4, the function `sqrt()` calculates a mathematical function and returns the result to `main()`, which stores it in a variable and later prints it. The functions `sqrt()`, `log()`, `sin()`, and `cos()` all accept an argument of type `double` and return an answer of type `double`. We say, informally, that these are **double→double functions** because their prototypes are of the form `double funcname(double)`. A double→double function must be called in some context where a value of type `double` makes sense. Often, these functions are called from the right side of an assignment statement or from a `printf()` statement. Examples of calls on double→double functions follow:

```
time = sqrt( 2 * height / GRAVITY );
printf( "The natural log of %g is %g\n", x, log( x ) );
```

Void functions. Some functions return no value to the calling program. Their purpose is to cause a side effect; that is, perform an input or output operation or change the value of some memory variable. These functions are called *void functions* because their prototypes start with the keyword `void` instead of a return type. *Void* means "nothing"; it is not a type name, but we need it as a placeholder to fill the space a type name normally would occupy in a prototype. We need some keyword because, if the return-type field in a function header is left blank, the return type defaults to `int`.[5]

Some `void` functions, such as `exit()`, require arguments; others, do not. The latter are called **void→void functions** and have prototypes of the form `void funcname( void )`. To call a void→void function, write the function name followed by empty parentheses and a semicolon. Often, the function call will stand by itself on a line.

[5]This default makes no sense in ISO C; it is an unfortunate holdover from pre-ISO days, when C did not even have a type `void`. The default to type `int` was kept in ISO C to maintain compatibility with old versions of C.

5.3.1 Function syntax.

A function has two parts: a **prototype** and a **definition**. When writing a program, prototypes for all the functions are normally written at the top, between the preprocessor commands and the beginning of `main()`. Function definitions are written at the bottom of the file, following the end of `main()`.

A function definition, in turn, has two parts: a **function header** (which must correspond to the prototype) and a **function body**, which is a block of code enclosed in curly brackets. The body starts with a series of (optional) declarations. These create local variables and constants for use by the function. The local declarations are followed by a series of program statements that use the local variables and the function's arguments (if any) to compute a value or perform a task. The body may contain one or more `return` statements, which return a value to the calling program.

The complete set of rules for creating and using functions in C is extensive and complex; it is presented in some detail in Chapter 9. In this section, we begin by writing the two types of functions discussed above. We illustrate how to define these function types, write prototypes for them, call them, and draw flowcharts (using barred boxes) to show the flow of control. We give a few examples and some brief guidelines so that the student may begin using functions in his or her own programs. The next figures illustrate, in context, how to write the parts of void→void and double→double functions.

5.3.2 Defining Void→Void Functions

We show how to write void→void functions first, using Figure 5.12 to illustrate the discussion. This program uses a void→void function to print user instructions and error comments. We ask the user to enter the number of passengers in a car. If the input number is greater than 5, we print an error message and beep four times.

Notes on Figures 5.12 and 5.13. Calling void→void functions.

First box: the standard header files.

- The prototypes for the time library and the standard I/O library are brought into the program by these `#include` statements.
- The header `<time.h>` is included because one of the programmer-defined functions will call functions from the time library. When we include prototypes at the top of a program, either explicitly or by including a header file, the corresponding functions can be called anywhere in the program.

Second box: the prototypes.

- Either a prototype declaration or the actual function definition should occur in a program before any calls on the function.[6]
- The include statements bring in prototypes for the standard functions. However, we must write prototypes for the three functions defined at the bottom of this program, `beep()`, `instructions()`, and `banner()`.

[6]If a function is called before it is declared, the C compiler will construct a prototype for that function that may or may not work properly.

The banner function is shown in Figure 5.13.

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

/* Prototype declarations for the programmer-defined functions. ---------- */
void banner( void );
void instructions( void ) ;
void beep( void );

int main( void )
{
    int n_pass;                        /* Number of passengers in the car. */

    banner();                         /* Display output headings. */
    instructions();                   /* Display instructions for the user. */

    scanf( "%i", &n_pass );

    if (n_pass > 5)  beep();     /* Error message for n>5 */
    else printf( " OK! \n" );   /* Success message for good entry */
    return 0;
}

/* ---------------------------------------------------------------------- */
void instructions( void )                          /* function definition */
{
    printf( " This is a legal-passenger-load tester for 6-seat sedans.\n"
            " Please input the number of passengers you will transport: " );
}

/* ---------------------------------------------------------------------- */
void beep( void )                                  /* function definition */
{
      printf( " Bad data! \n\a\a\a\a" );        /* error message and beeps */
}
```

Figure 5.12. Using void→void functions.

This function is part of the program in Figure 5.12, and belongs at the bottom of the same source-code file.

```
/* ----------------------------------------------------------------------- */
void banner( void )                /* Print a neat header for the output.  */
{
   time_t now = time(NULL);
   printf( "\n-------------------------------------------------------\n" );
   printf( "    Patience S. Goodenough\n    CS 110\n    ");
   printf( ctime( &now ) );
   printf(  "-------------------------------------------------------\n" );
}
```

Figure 5.13. Printing a banner on your output.

Library functions are surrounded by shaded boxes; programmer-defined functions are shown with no surrounding box.

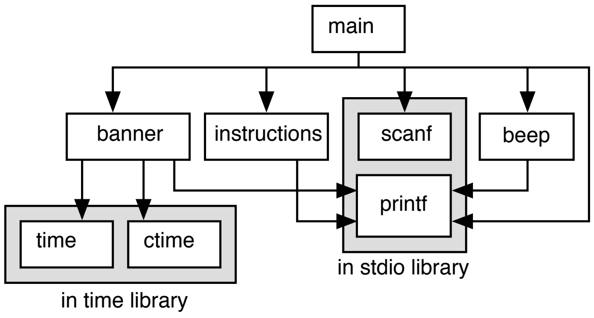


Figure 5.14. A call graph for the beep program.

- The prototype for every void→void function follows the simple pattern shown here: the word `void` followed by the name of the function, followed by the word `void` again, in parentheses. Every prototype ends in a semicolon.

Third and fourth boxes: the function calls.

- These lines all call a void→void function. The call consists of the function name followed by empty parentheses. Often, as with the calls on `instructions()` and `banner()`, a void→void function call will stand alone on a line.
- Often a void→void function is used to give general instructions or feedback to the user, as in the second

box. Sometimes, one is used to inform the user that an error has happened, as is the case with the call on `beep()` in the third box. Such calls often form one clause of an `if` statement.

Fourth and fifth boxes: two easy function definitions.

- We define two void→void functions: `instructions()` and `beep()`. We use a comment line of dashes to mark the beginning of each function so that it is easy to locate on a video screen or printout.
- Each function definition starts with a header line that is the same as the prototype except that *it does not end in a semicolon.* Following each header line is a block of code enclosed in curly brackets that defines the actions of the function.
- Most void→void functions, like these two, perform input or output operations. The symbol `\a`, used in the `beep()` function, is the ASCII code for a beeping noise. Many systems will emit an audible beep when this symbol is "printed" as part of the output. Other systems may print a small box instead of emitting a sound. Some systems give no audible or visible output.

The output

Here is the output from two test runs (the second banner has been omitted):

```
-------------------------------------------------------
    Patience S. Goodenough
    CS 110
    Sat Aug  9 18:21:17 2003
-------------------------------------------------------
 This is a legal-passenger-load tester for 6-seat sedans.
 Please input the number of passengers you will transport: 2
 OK!

Tester has exited with status 0.
---------------------------------------------------
 This is a legal-passenger-load tester for 6-seat sedans.
 Please input the number of passengers you will transport: 10
 Bad data!

Tester has exited with status 0.
```

Figure 5.13 a longer function definition.

- We define a void→void function named `banner()` that prints a neat and visible output header containing the programmer's name and the date and time when the program was executed. The date and time are produced by calling functions from the `time` library.
- The first line of this function declares a variable named `now` of type `time_t` and initializes it to the current time by calling `time( NULL )`.
- Then we call `ctime()` to convert the coded date and time into a string that can be printed. The third call on `printf()` shows how do use this function. Be sure to write the ampersand when you copy this code.
- An output header, similar to the one this function produces should be part of the output produced by every student program. You should use it with everything you hand in to the teacher. One way to do

this is to start your own personal file, say `mytools.c` containing reuseable code. Make a matching file called `mytools.h` containing the prototypes for the functions in `mytools.c`. Include `mytools.h` in every main program you write, and add both `mytools` files to the project you create for your program.

5.3.3 Returning Results from a Function

A function that returns a value is fundamentally different from a `void` function. A `void` function simply causes some side effect, such as output, like `banner()` in Figure 5.19. The call on such a function forms a separate statement in the code. In contrast, a function that returns a value interacts with the rest of the program by creating information for further processing. A `return` statement is used to send a result from a function back to the caller. It is represented in the diagram in Figure 5.21 as a tab sticking out of the function's interface. A `return` statement can be placed anywhere in the function definition, and more than one `return` statement can be used in the same function.[7]

A function that returns a value can be called anywhere in a C statement that a variable name or literal of the same type would be permitted. Often, as in the call on `f()` in Figure 5.19, a function is called in an assignment statement. The return address for this call is in the middle of the statement, just before the assignment happens. When the value is returned from the call, `main()` will resume execution by assigning the returned value to the variable `z`.

If a function is called in the middle of an expression, the result of the function comes back to the calling program in that spot and is used to compute the value of the rest of the expression. The call on `exp()` in Figure 5.19 illustrates this. The function is called from the middle of a `return` statement: `return y * exp(y)`. The return address for this call is in the middle of the statement, just before the multiplication happens. After a value is returned by `exp()`, it will be multiplied by the value of `y` and the result returned to `main()`.

Finally, as in the call on `g()` in Figure 5.19, a function can be called from the argument list of another function. It is quite common to nest function calls in this way.

5.3.4 Arguments and Parameters

Function parameters introduce variability into the behavior of a function. A void→void function without parameters always does the same thing in the same way.[8] In contrast, introducing even one parameter permits the actions of a function and its results to depend on the data being processed. By parameterizing a piece of code, we can make it useful under a much more general set of circumstances.

Formal parameters are part of a function definition and specify a set of unknowns; **arguments** are part of a function call and supply values for those unknowns. In Figure 5.21, parameters are represented by notches along the left edge of each function's interface. Right-facing arrows connect each argument to the notch of the corresponding parameter; these arrows represent the direction in which information flows from the caller to the subprogram.

[7]However, we strongly recommend using a single `return` statement at the end of the function.

[8]An exception to this occurs if the function uses global variables or user input.

The function `f()` has one parameter, a `double` value named `y`. Even if other objects in the program have the same name, the parameter `y` in `f()` will be a distinct object, occupying a separate memory location. It is quite common to have two objects with the same name defined in different functions.

Looking at the list of library functions in Figure 5.6, we see that the `exp()` function has one parameter of type `double`. The name of this parameter is not known because `exp()` is a library function and its details have been concealed from us. During the calling process, the argument value is stored in the parameter variable, making a complete object.

A function can be `void` or have one or more parameters. Its prototype defines the correct **calling sequence**; that is, the number, order, and types of the arguments that must be written in a call. The call must supply one argument expression per parameter;[9] if the number of arguments does not match the number of parameters, the program will not compile. When a function call is executed, each argument expression in the call will be evaluated and its value will be passed from the caller to the subprogram, where it will be stored in the corresponding parameter. For example, the argument in the call on `f()` is the value of the variable named `x` in the main program. This value is a `double`, so it can be stored in the `double` parameter with no conversion.

An ISO C prototype states the name of a function, the types of its parameters, and the return type. The parameters also may be named in the prototype, but such names are optional and often omitted. A function header states the same information, except that parameter names *are required* in the function header.

Inside a function, the parameter names are used to refer to the argument values; the first parameter name in the function header refers to the first argument in the function call, and so on. In Figure 5.21, when `main()` makes the call `f(x)`, the value of `x` is copied into the parameter named `y`. Within the body of `f()`, the value stored in this parameter will be used wherever the code refers to the name `y`. During execution of `f()`, this value is further copied and stored in the parameter variable of `exp()`.

Formal Parameter Names The function header (which is the first line of the function definition) the name of a function, the types of its parameters, and the return type These names provide a way for the programmer to refer to the parameters in the function's code.

Any legal name may be given to a formal parameter. It may be the same as or different from the name of a variable used in the function call, and both can be the same as or different from the optional name in the function's prototype.[10] The names chosen do not affect the meaning of the program because argument values are matched up with parameters by position, not by name. For example, the main program in Figure 5.15 uses a variable named `h` in the call on the `drop()` function (Figure 5.18). Within `drop()`, however, the parameter is named `height`, so the value of `main's h` will be stored in `drop's height`.

[9]Some functions accept a variable number of arguments; `scanf()` is an example. However, the details of how this is accomplished are beyond the scope of this text.

[10]However, it is good style to use the same name in the prototype and the function header.

5.3.5 Defining a Double→Double Function

We use Figures 5.15 through 5.18 to illustrate the construction and call of a double→double function, which is somewhat more complicated than a void→void function because argument information must be passed into the function and a result must be returned to the caller.

Notes on Figure 5.15. The grapefruit returns.

First box: the prototypes.

- A prototype for a double→double function gives the function name and specifies that it requires one `double` parameter and returns a `double` result. The general form is

  ```
  double function_name( double parameter_name );
  ```

  where the parameter name is optional.
- This box contains a prototype declaration for the function named `drop()`. It states that `drop()` requires one argument of type `double` and returns a `double` result. This information permits the C compiler to check whether a call on `drop()` is written correctly.
- There is also a prototype declaration for the void→void function named `title()`, which is similar to the `instructions()` function in the previous example.

Second box: the void→void function call. The prototype for `title()` must precede this function call. The definition of the function is at the bottom of the program, in the fourth box.

A function call interrupts the normal sequence of execution. Look at the flow diagram in Figure 5.17. The first statement is the call on `title()`. When that call is executed, control leaves the main sequence of boxes and travels along the dotted line to the beginning of the `title()` function. Then control proceeds, in sequence, to the return statement, and finally returns to where it came from along the second dotted arrow.

Third box: the double→double function call. This box calls `drop()`. The prototype for `drop()` was given in the first box and the function is defined in Figure 5.18, but would be placed in the same source file, below the definition of `title()`.

The form of any function call must follow the form of the prototype. When we call a double→double function, we must supply one argument expression of type `double`. In this call, the argument expression is a simple variable name, `h`. When we call `drop()`, we send a copy of the value of `h` to the function to be used in its calculations.

A function call interrupts sequential execution and sends control into the function. Figure 5.17 depicts this interruption as a dotted arrow going from the function call to the beginning of the function. From there, control flows through the function to the return statement, which sends control back to the point of interruption, as shown by the lower dotted arrow. When a double→double function returns, it brings back a `double` value, which should be either used or stored. In this example, we store the result in the `double` variable `t`.

A grapefruit is dropped from a helicopter hovering at height `h`. This continues development of the program in Figure 5.4. The `drop()` function is shown in Figure 5.18.

```
/* ---------------------------------------------------------------------
// Modify the Grapefruits and Gravity program by using a double→double
// function to compute the travel time of the grapefruit.
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define GRAVITY 9.81

void title( void );
double drop( double height );   /* Prototype declaration of drop. */

int main( void )
{
    double h;                   /* height of fall (m) */
    double t;                   /* time of fall (s) */
    double v;                   /* terminal velocity (m/s) */

    title();                    /* Call function to print titles. */
    printf( " Enter height of helicopter (meters): " );
    scanf( "%lg", &h );         /* keyboard input for height */

    t = drop( h );              /* Call drop.  Send it the argument h. */
    v = GRAVITY * t;            /* velocity of grapefruit at this time */

    printf( "    Time of fall = %g seconds\n", t );
    printf( "    Velocity of the object = %g m/s\n", v );
    return 0;
}

/* ------------------------------------------------------------- */
void title( void ) {
    printf(" Grapefruits and Gravity with a Drop Function\n\n"
           " Calculate the time it would take for a grapefruit\n"
           " to fall from a helicopter at a given height.\n" );
}
```

Figure 5.15. The grapefruit returns.

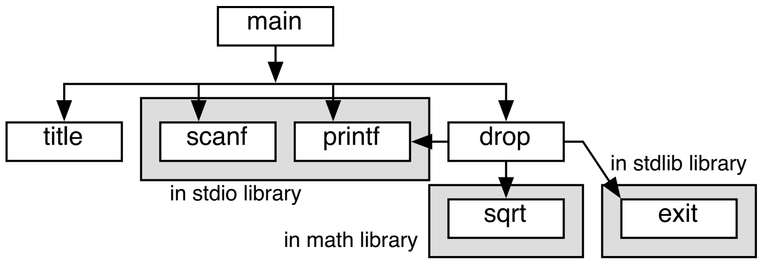


Figure 5.16. Call graph for the grapefruit returns.

Program output. Here is one set of output from the grapefruit program (the banners and termination messages have been omitted):

```
Grapefruits and Gravity with a Drop Function

Calculate the time it would take for a grapefruit
to fall from a helicopter at a given height.
Enter height of helicopter (meters):  872
   Time of fall = 13.3401 seconds
   Velocity of the object = 130.733 m/s
```

Notes on Figure 5.18. Definition of the `drop()` function. We show how to write the definition of a double→double function.

The comment block and the function header.

- Every function should start with a block of comment lines, the **function comment block**, that separate the definition from other parts of the program and describe the purpose of the function. Comment marks (`/*...*/`) begin the first line and end the last line of the block comment. These comments provide a neat and visible heading for the function.
- The first line of a function definition is called the *function header*. It must be like the prototype except that the parameter name is required (not optional) in the header and the header does not end with a semicolon.
- A parameter can be given any convenient name, which need not be the same as the name of the argument that will appear in future function calls. A new variable is created for the parameter and can be used only by the function itself.
- When a function is called, the expression in parentheses is evaluated and its value passed into the function and used to initialize the parameter variable. This value is called the *actual argument*. Within

This is a flow diagram of the program in Figures 5.15 and 5.18. Function calls are depicted using a box with an extra bar on the bottom to indicate a **transfer of control**. The dotted lines show how the control sequence is interrupted when the function call sends control to the beginning of the `drop()` function. Control flows through the function then returns via the lower dotted line to the box from which it was called. The call on `fatal()` (after an error) ends execution immediately and returns control to the operating system.

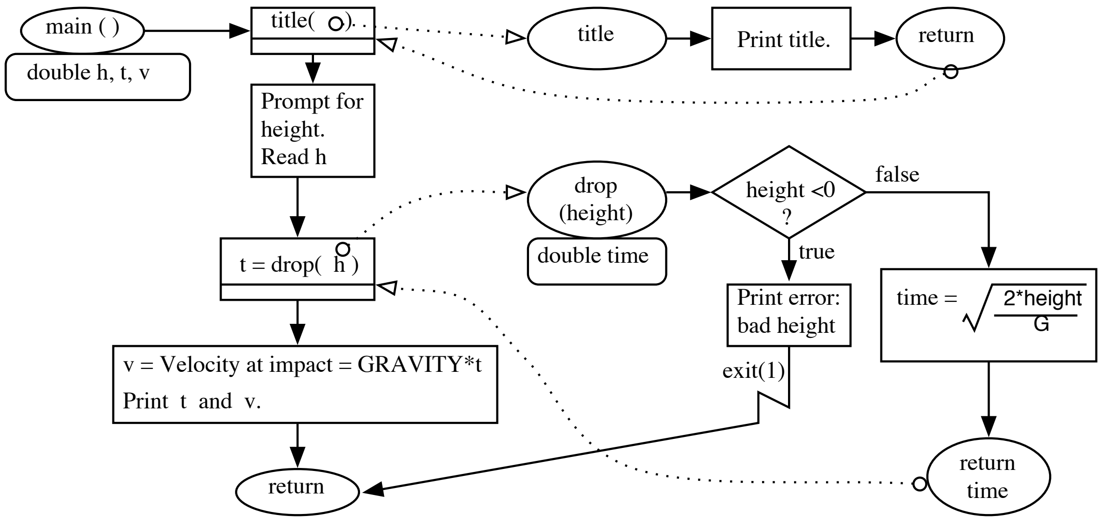


Figure 5.17. Flow diagram for the grapefruit returns.

the function, the parameter name is used to refer to the argument value. In the function call, the argument was `main()`'s variable `h`, but the `drop()` function's parameter is named `height`. This is no error. *A double→double function can be called with any* `double` *argument value.* The argument can be the result of an expression, a variable with the same name, or a variable with a different name. Within the function, the parameter name (`height`) is used to refer to the argument value.

First box: a local variable declaration. The code block of a function can and usually does start with declarations for **local variables**. Memory for these variables will be allocated when the function is called and deallocated when the function returns. These variables are for use only by the code in the body of the function; no other function can use them. In the example program, we declare a local variable named `time`.

Second box: the function code. Statements within the function body may use the parameter variable and any local variables that were declared. References also may be made to constants defined globally (at the top of the program). The use of global variables is legal but strongly discouraged in `C`. The statements

This function is called from Figure 5.15.

```
/* ------------------------------------------------------------------ ---
// Time taken for an object dropped from height meters to hit the ground.  */
double drop( double height )
{
    double time;                           /* create a local variable */

    if (height < 0) {                      /* exit gracefully after error. */
        printf( " Error: height must be >= 0.  You entered %g\n", height );
        exit( 1 );                         /* abort execution */
    }
    time = sqrt( 2 * height / GRAVITY ); /* calculate the time of fall */

    return time;
}
```

Figure 5.18. Definition of the `drop()` function.

in this function are like the corresponding lines of Figure 5.4 except that they use the parameter name and local variable name instead of the names of the variables in the main program.

Inner box: calling another function. An error has been detected, so we print an appropriate comment and call `exit( 1 )` to terminate the program immediately.

Last box: the* `return` *statement. On completion, a double→double function must return a result of type `double` to its caller. This is done using the `return` statement. A `return` statement does two things: it sends control immediately back to the calling program and it tells what the return value (the result) of the function should be. It does not need parentheses. In Figure 5.18, the result from `sqrt()` is stored in the local variable `time`. To make that answer available to the caller, we return the value of `time`. On executing the `return` statement, the value of `time` is passed back to the caller and control is returned to the caller at the statement containing the function call, as depicted by the dotted line in Figure 5.17.

5.4 Organization of a Module

Generally, a program has a `main()` function that calls several other library and programmer-defined functions. To compile `main()`, the compiler needs to know the **prototype** of every function called. One way this information can be supplied is by putting `main()` at the bottom of the module, while the definitions of the other functions come before it. However, many programmers dislike having the main program at the end of the file and it is customary, in C, to put `main()` at the top. When this is done, prototypes for all the functions that `main()` calls must be written above[11] it. This pattern has been followed in every program example given so far. There is one major exception to this organizational guideline: When a function is so simple that its entire definition can fit on one line, the function itself often is written at the top of the file in place of a prototype.

When you call a function from one of the C libraries, you use code that is already compiled and ready to link to your own code (see Chapter 5). Header files such as `stdio.h` and `math.h` contain prototypes (not C source code) for the precompiled library functions. When your code module uses a **library** function you `#include` the library header file at the top. This causes the preprocessor to insert all the prototypes for the library functions into your module, making it possible for the compiler to properly check your calls on the library functions. The order of parts, from the top of the source file to the bottom follows. These principles lead to the following layout for the parts in a simple program:

- `#include` commands for header files.
- Constant definitions and type declarations.[12]
- Prototypes (function declarations) and one-line functions.
- `main()`, which contains function calls.
- Function definitions, possibly containing more calls.
- Figure 5.19 illustrates the principles with a complete program and two functions.

Notes on Figure 5.19. Functions, prototypes, and calls.

First box: things that precede `main()` ***in a code module.***

- When the C compiler reaches the `#include` command, it puts a copy of the `tools.h` file into this program. This file contains prototypes for the functions in the `tools` library, including `banner()` and `bye()`, called in this program.
- Included files often contain other `#include` commands. For example, the `tools.h` file contains `#include` commands for the library header files, `stdio.h`, `math.h`, `string.h`, `time.h`, and `ctype.h`. If we include `tools.h` in a program, we need not write separate include commands for these other library header files.
- This program uses two constants, representing the minimum and maximum values acceptable for input. These constants are defined after the `#include` command and before the prototypes.
- We need a prototype for a function if a call on it comes before its definition in the file. Function `f()` is called (second box) from `main()` and defined after `main()` (fourth box). Therefore, `f()` needs a prototype, which is given on the fourth line of this box.

[11]The prototypes also may be written inside the calling function. However, we wish to discourage this practice.

[12]Type declarations will be discussed in Chapters 11, 12 and 13.

A function's prototype may be given first, then the call, and finally the full definition of the function, like function `f()` here. Alternatively, the function may be fully defined before it is called; for example, function `g()` is defined before `main()`, which calls it.

```
#include <stdio.h>
#include <math.h>
#define MIN   10.5
#define MAX   87.0
double f( double y );         /* Prototype for function f, defined below. */
double g( double y ) { return( y * y + y ); }  /* Definition of function g. */

int main( void )
{
    double x, z;
    banner();
    printf( "\n Enter a value of x between %.2f and %.2f: ", MIN, MAX );
    scanf( "%g", &x );
    z = f( x ) ;
    printf( "\n The value of x * exp(x) is: %g \n", z );
    printf( "\n The value of x * x + x  is: %g \n", g( x ) );
    return 0;
}

/* ---------------------------------------------------------------- */
/* Definition of function to calculate f = y * e to the power y. ----- */
double
f( double y )
{
    return y * exp( y ) ;
}
```

Figure 5.19. Functions, prototypes, and calls.

This is a function call graph for the functions, prototypes, and calls of the program in Figure 5.19. Shading around a boxes indicates that the function is in a standard library.

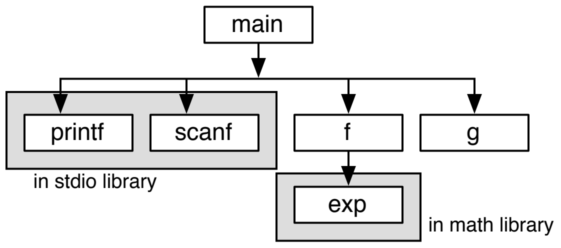


Figure 5.20. Function call graph with two levels of calls.

- The actual definition of function `g()` is given here, rather than a prototype. When a function definition comes before any use of that function, no prototype is needed. This often is done when a function is so simple that all its work is done in the `return` statement and so short that it can be written on one line.

Second and third boxes: Calls on the programmer-defined functions.

- We create `f()` and `g()` as two functions separate from `main()`, so that it is easy to change them when we need to do some other calculation. A good modular design keeps the calculation portion of a program separate from the user-interaction portion.
- In the second box, we set `z` equal to the result of calling function `f()` with the value of the variable `x`. Function `f()` was only prototyped before `main()`, so when the compiler reaches this box, the full definition of `f()` will not be known to it. However, the prototype for `f()` already was supplied, so the compiler knows that a call on `f()` should have one `double` argument and return a `double` result. This information is necessary to translate the call properly.
- In the third box, the function `g()` is called, and its return value is then passed directly to `printf()`. Since `g()` already was fully defined, the compiler has full knowledge of `g()` when it reaches this line and, therefore, is able to compile this call correctly.
- A sample output from this program, excluding the banner and closing comment, is

```
Enter a value of x between 10.50 and 87.00: 13.2

The value of x * exp(x) is: 7.13281e+06

The value of x * x + x  is: 187.44
```

This illustrates the function calls in Figure 5.19.

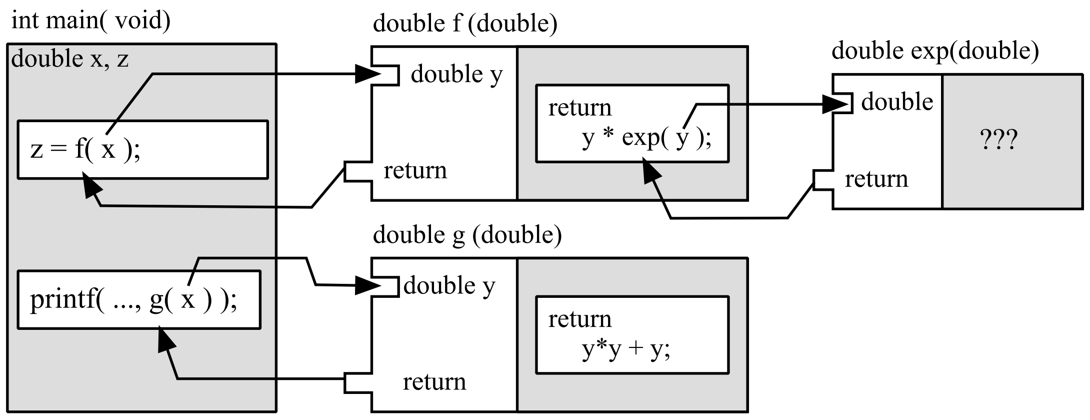


Figure 5.21. A function is a black box.

***Fourth box: Definition of programmer's function* `f()`.**

- Here we define `f()`. The return type and parameter list in the function definition must agree with the prototype given earlier.
- Function definitions should start with a blank line and a comment describing the action or purpose of the function. Discipline yourself to do this. The dashed line provides a visual separation and helps the programmer find the beginning of the function definition. This is extremely useful in a long program; make it a habit in your work.
- Compare this definition to the previous one-line definition of `g()`. The definition of `f()` begins with a descriptive header. The code itself is spread out vertically, with only one program element on each line, according to the accepted guidelines for good style. The definition of `g()` is written compactly on one line; that style is used only for very simple functions.

Inner box: A call on a library function.

- We call the function `exp()`, which is in the `math` library. We can do this because the header file, `math.h`, was included by `tools.h`, which was included in this file.
- The variable `y` is a `double`. The prototype for `exp()` says that its parameter is a `double`. The type mismatch here is not a problem. The compiler will note the mismatch and automatically compile code to convert the `double` value to a `double` format during the calling process.

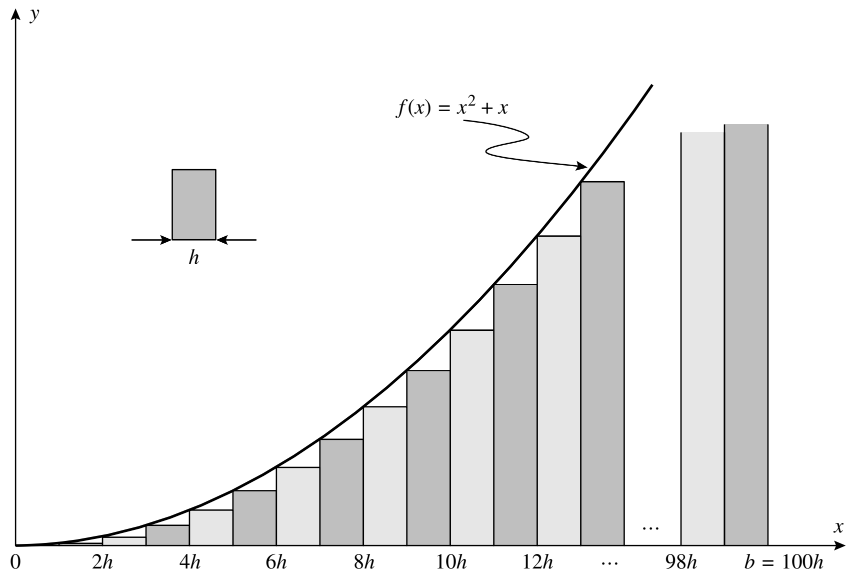


Figure 5.22. The area under a curve.

5.5 Application: Numerical Integration by Summing Rectangles

We introduce the topic of integration with a simple integration program where the function to be integrated is coded as part of the main program. The definite integral of a function can be interpreted as the area under the graph of that function over the interval of integration on the x-axis. If a function is continuous, we can approximate its integral by covering the area under the curve with a series of boxes and adding up the areas of these boxes. Several methods for numerical integration are based on a version of this idea:

- Divide the interval of integration into a series of subintervals.
- For each subinterval, approximate the area under the curve in that interval by a shape such as a rectangle or trapezoid whose area is easy to calculate.
- Calculate and add up the areas of all these shapes.

The simplest way to approximate the integral of a function is to use rectangles to approximate the area under the curve in each subinterval; the diagram in Figure 5.22 and the program in Figure 5.23 illustrate this approach. For each subinterval, we place a rectangle between the curve and the x-axis such that the upper-left corner of the rectangle touches the curve and the bottom of the rectangle lies on the x-axis. In this example, we calculate the integral of the function $f(x)$:

$$f(x) = x^2 + x, \qquad \text{where } 0 \le x \le b$$

by summing 100 rectangular areas, each of width $h = b/100$, as shown by Figure 5.22. For example, the area of the rectangle between $2h$ and $3h$ is $h \times f(2h)$. Generalizing this formula, we get

$$\text{Area}_k = h \times f(k \times h)$$

Now, summing over 100 rectangles, we get

$$\text{Area } = \sum_{k=0}^{99} h \times f(k \times h)$$

Notes on Figures 5.22 and 5.23: Integration by Rectangles.

First box: the function definition. This is the entire definition of the function we will integrate. Short functions can be written on one line.

Second box: declarations with initializations.

- In this initial example, we integrate a single fixed function over a fixed interval using a fixed number of rectangles. Later, we present other integration programs that do the job in more general and more accurate ways.
- We integrate the function over the interval $0 \ldots 1$.
- We divide this interval into 100 parts of length `h = .01`.

Third box: initializations for the loop.

- Although we could combine these initializations with the declarations, that is bad style. For trouble-free development of complex programs, all loop initializations should be placed immediately before the loop.
- To prepare for the loop, we initialize the `sum` that will accumulate the areas.
- The loop variable for this `while` loop is `k`. The first rectangle lies between a and $a+1 \times h$, so we initialize `k` to 0. Since $a = 0.0$, the first rectangle we sum starts at $x = 0.0 + 0 \times h = 0.0$.

Fourth box: the loop.

- We execute the loop from `k = 0` until (but not when) `k = 100`. Therefore, we do it 100 times.
- The last rectangle summed starts at $x = a + 99h$ and goes to $x = a + 100h$.
- We use the `++` operator to increment `k` each time around the loop.
- Note how convenient the `+=` operator is for adding a new term to a sum.

Fifth box: the output.

- The *exact* area under this curve for the interval $0 \ldots 1$ is .833333. Since we are calculating an approximate answer, it will be slightly different.
- The actual output is

```
Integrate x*x + x from 0 to 1.
The area is 0.82335
```

```
#include <stdio.h>
#define N 100
double f( double x ){ return x*x + x; }

int main( void )
{
    double a = 0.0;       /* Lower limit of integration. */
    double b = 1.0;       /* Upper limit of integration. */
    double h = (b - a)/N; /* Width of one of the N rectangles summed. */

    double x;             /* Current function argument; a <= x < b. */
    double sum;           /* Total area of rectangles. */
    int k;                /* Loop counter. */

    printf( " Integrate x*x + x from %g to %g. \n", a, b );

    sum = 0.0;
    k = 0;

    while(k < N) {            /* Add up N rectangles. */
        x = a + k * h;        /* Lower left corner of kth rectangle. */
        sum += h * f( x );    /* Area of rectangle at x. */
        ++k;
    }

    printf( " The area is %g\n", sum );
    return 0;
}
```

Figure 5.23. Integration by summing rectangles.

5.6 Functions with More Than One Parameter

5.6.1 Function Syntax: Two Parameters

We have studied a variety of `void` and non-`void` functions having either no parameters or one parameter and discussed the essentials of prototypes, parameters, and calls. Most functions, though, have more than one parameter, so we need to study the few remaining facets of the syntax for defining and calling functions with a more complex interface. The next few paragraphs summarize the syntax for functions with two parameters and a return value. The forms for three or more parameters follow the same pattern, with additional clauses added to the parameter list. We will use the following terminology in this discussion:

- `f-name` means any function name;
- `p-type` means the type of a parameter, `vp-name` means the name of a parameter,
- `r-type` means the type returned by the function,
- `var` means the name of a variable,
- `exp` means any expression of the correct type, possibly a simple variable name.

Prototypes. The fundamental form for the 2-argument prototype is:

```
r-type f-name( p1-type p1-name, p2-type p2-name );
```

However, parameter names are optional in a prototype, so we could also write the prototype like this:

```
r-type f-name( p1-type, p2-type );
```

For example, suppose we have a function named `cyl_vol` that takes two double parameters. Its prototype could be written in one of these two ways

```
double cyl_vol( double d, double h );
double cyl_vol( double, double );
```

In mathematical notation, this is called a double×double→double function.

Call syntax. A function call imitates the simpler form of the prototype; argument expressions of the appropriate type must be written, but parameter names are always omitted. The general syntax and a sample call on `cyl_vol` are:

```
General syntax:  var = f-name( exp1, exp2 );
Example:  volume = cyl_vol ( diameter, height );
```

Definition syntax. A function definition imitates the longerform of the prototype; argument types and names must both be given. This is the general syntax and function header of `cyl_vol`:

```
General syntax:  r-type f-name( p1-type p1-name, p2-type p2-name ){...
Example:  double cyl_vol ( double d, double h ){...
```

We illustrate these rules for the format of a two-parameter function using the program in Figures 5.24 and 5.25

Notes on Figures 5.24 and 5.25: Two parameters and a return value.

First box: the prototypes. We will use two double×double→double functions to calculate two properties of a cylinder: volume and surface area. The prototype for the first, `cyl_vol()`, is written with the optional parameter names and the prototype for the second function, `cyl_surf()` is written without. As always, either prototype could be written either way.

This program illustrates the syntax for calling two-parameter non-`void` functions. The two functions called here are defined in Figure 5.25.

```
/* ----------------------------------------------------------------------
// Calculate the volume of a cylinder.
*/
#include <stdio.h>
#include <math.h>
#define PI  3.1415927
double cyl_vol( double d, double h );          /* long form of prototype  */
double cyl_surf( double, double );             /* short form of prototype */

int main( void )
{
    double diam, height;           /* Inputs: dimensions of the cylinder. */
    double volume;                 /* Output: its volume.                 */

    printf( "\n Calculate the Volume of a Cylinder\n\n"
            " Enter its diameter and height: " );
    scanf( "%lg%lg", &diam, &height );

    volume = cyl_vol( diam, height );
    printf( "\t The volume of this cylinder = %g\n", volume );

    printf ( "\t Its surface area = %g \tn", cyl_surf( diam, height ) );

    return 0;
}
```

Figure 5.24. Using functions with two parameters and return values.

Second box: the first function call and output.

- Since `cyl_vol()` returns a value, calls will be in the context of an assignment statement or in the argument list of another function call. This call is part of an assignment.
- First, we call the function and save the answer in `volume`. Then, on the next line, we send the value of `volume` to `printf()`. The calculation could be combined with the output and condensed into one statement by putting the call on `cyl_vol()` directly into the argument list for `printf()`, thus:

```
printf( "\t The volume of this cylinder = %.2f \n",
        cyl_vol( diam, height ) );
```

We illustrate the syntax for definition of two-parameter non-`void` functions. These functions are called from Figure 5.24.

```
/* ------------------------------------------------------------------ */
double                             /* Calculate the volume of a cylinder  */
cyl_vol( double d, double h ) {    /* with diameter=d and height=h;       */
    double r = d / 2;              /* r is the radius of the cylinder.    */
    return PI * pow( r, 2 ) * h;
}
```

```
/* ------------------------------------------------------------------ */
double                             /* Calculate surface area of cylinder  */
cyl_surf( double d, double h ) {   /* with diameter d and height h;       */
    double area_end, area_side;
    double r= d / 2;               /* r is the radius of the cylinder.    */
    area_end = PI * pow( r, 2 )    /* each end is a circle.               */
    area_side = h * 2 * PI * r;    /* length of side = circle perimiter   */
    return area_side + 2 * area_end;     /* surface is the side + 2 ends */
}
```

Figure 5.25. Functions with two parameters and return values.

- It is a matter of personal style which way you write this code. The one-line version is more concise and takes slightly less time and space. The two-line version, however, is easier to modify, works better with an on-line debugger, and enhances seeing and understanding the technical calculation.

Third box: the second function call and output. The calculation and output can be combined and condensed into one statement by putting the call directly into the argument list for `printf()`, as shown here.

Figure 5.25: definitions of `cyl_vol` ***and*** `cyl_surf`.

- Following modern guidelines for style, we prefer to write the return type, alone, on the first line of the function definition. The second line starts with the function name on the left. Using this style makes it somewhat easier to find the function names when you scan a long program and allows writing a comment about the return value. Of course, these two lines also could be combined, thus:

  ```
  double cyl_vol( double d, double h ) { ...
  ```

- The code in the body of this function uses only three variable names: `d, h`, and `r`. The first two are parameters to the function, the third is defined at the top of the function's block of code. All three objects are local to the function; that is, these variables are defined by the function and only this function can access them. Every properly designed function follows this pattern and confines its code to use only locally defined names. All interaction between the function and outside variables happens through the parameter list or the function's return value.

Inner box: raising a number to a power. Very few functions in the `math` library require two arguments; the most commonly used is the function `pow()`, which is used to raise a number to a power. In this box, we are calculating the square of `r`, the radius.

This function's prototype is `double pow( double, double)`. The first argument is the number to be raised, the second argument is the power to which to raise it. Both may be any `double` number. For example, `pow( 100, .5 )` raises 100 to the power .5, which is the same as calculating its square root.

Output.

```
 Calculate the Volume of a Cylinder

 Enter its diameter and height: 2.0  10.0
 The volume of this cylinder = 31.4159
 Its surface area = 69.115

Cylinder has exited with status 0.
```

5.7 Application: Generating "Random" Numbers

5.7.1 Pseudo-Random Numbers

Many computer applications (experiments, games, simulations) require the computer to make some sort of random choice. To serve this need, programs called *pseudo-random number generators* have been devised. These start with some arbitrary initial value (or values), called the **seed**, and apply an algorithm to generate another value that seems unrelated to the first. Then this first result will be used as the seed for the next value and so on, as long as the user wishes to keep generating values.

The numbers generated by these algorithms are called **pseudo-random numbers** because they are not really random but the output of an algorithm and an initial value. If the same algorithm is run again with the same starting point, the same series of "random" numbers will be produced. The goal, therefore, is to find an algorithm and a seed that will produce a long series of numbers with no detectable pattern and without duplicating the seed. Repeating a seed would cause the series to enter a cycle.

The `C` function `rand()` generates pseudo-random integers, which might be 2 or 4 bytes long, depending on the local definition of type `int`. (The actual range of values is 0 ... `RAND_MAX`, which is commonly the same as `INT_MAX`.) This function does not implement the best known algorithm but is good enough for many purposes. The function is found in the `standard` library; to use it you must `#include <stdlib.h>`. Before calling `rand()` the first time, you must call another function, `srand()` to supply an initial seed value. This seed could be any integer value, such as a literal constant or a number entered by the user. However, in general, the user should not be bothered with selecting a seed, and a constant seed is undesirable because it always will result in the same pseudo-random series. (A constant seed can be useful during the debugging process so that error conditions can be repeated.) What therefore is needed for most applications is a handy source of numbers that are constantly changing and nonrepetitive. One such source is attached to most computers: the real-time clock. Therefore, it is quite common to read the clock to get an initial random seed. This technique is illustrated in Figure 5.26.

5.7.2 How Good Is the Standard Random Number Generator?

The next program generates a large quantity of random integers and counts the occurrences of 0. According to probability theory, if we generate numbers in the range $0 \ldots n-1$, approximately $1/n$ of the values should be counted. No single program run can confirm whether the generator is fair. However, repeated trials or larger sample sizes will give some feeling for the quality of the random number generator being used. If the results are close to the expected value most of the time, the generator is performing well; otherwise, its behavior is questionable.

Notes on Figure 5.26. Generating random numbers.

First box: initializing the random number generator.

1. `C` provides a function in the `time` library that permits a program to read the system's real-time clock (if there is one). The return value of `time()` is an integer encoding of the time that has type `time_t` (an integer of some system-dependent length defined in `time.h`).
2. The argument in the call to `time()` normally is the address of a variable where we want the time stored. The function `time()` stores the current time into the given address in the same way that `scanf()` stores an input value into a variable whose address is given to it.[13] However, this function also returns the same time value through the normal function return mechanism. Since we only need this information once, we use a special constant value, `NULL`, as the argument; this is legal and tells the function that we don't want a second copy of the information stored anywhere in memory.
3. Our purpose here is not to know the actual time. Rather, we use the clock as a convenient source of a seed for the random-number generator. A good seed is an unpredictable number that never is the same twice, and the time of day suits this purpose very well.
4. The type cast operator, `(unsigned)`, in front of the call on the `time()` function is used to convert the `time_t` value returned by the `time()` function into the `unsigned int` form expected by `srand()`. Using an explicit cast instead of the standard automatic coercion eliminates a compiler warning message on some systems.

Second box: data input and validation. We eliminate divisors less than 2 because they are meaningless. We also set an arbitrary upper limit on the range of numbers that will be generated. Since the limit is relatively small, we can use a short integer to store it and there is space for printing many columns of numbers. Here is an example of the error handling:

[13]How this actually is done, using call by address, is discussed in Chapter 11.

We generate a series of pseudo-random numbers and print them in neat columns. When finished, we also print the number of zeros generated and the number expected, based on probability theory.

```
#include "mytools.h"
#define HOW_MANY 500 /* Generate HOW_MANY random numbers */
#define NCOL 10      /* Number of columns in which to print the output. */
#define MAX 100      /* Upper limit on size of random numbers generated */

int main( void )
{
    long num;        /* a randomly generated integer */
    short select;    /* input:  upper limit on range of random numbers */
    short n;         /* # of random numbers generated */
    int count;       /* # of zeros generated */

    banner();

    srand( (unsigned) time( NULL ) );/* seed random number generator. */

    printf( " Generate %i random numbers in the range 0..n-1. \n"
            " Please choose n between 2 and %i: ", HOW_MANY, MAX );
    scanf( "%hi", &select );
    if (select < 2 || select > MAX) fatal( " Number is out of range." );

    /* Generate random numbers and test for zeros. -------------------- */

    count = 0;                       /* Count zeros generated. */
    for (n = 0; n < HOW_MANY; ) {    /* Generate HOW_MANY random numbers. */

        num = rand();                /* Generate a random long integer. */
        num %= select;               /* Scale to range 0..select-1. */

        ++n;                         /* Count the trials and... */
        printf( "%5li", num );       /* ...print all numbers generated. */
        if (n % NCOL == 0) puts( "" ); /* End line every NCOL outputs. */

        if (num == 0) ++count;       /* ..count the zeros. */
    }

    if (count % NCOL != 0) printf( "\n" );  /* End last line of output. */
    printf( "\n %5i   zeros were generated.", count );
    printf( "\n %7.1f are expected on average.\n", HOW_MANY/(double)select );
    return 0;
}
```

Figure 5.26. Generating random numbers.

```
Please choose n between 2 and 100: 1
Number is out of range.

Error exit; press '.' and 'Enter' to continue
```

A validation loop could be used here. We take the simpler approach of using `fatal()` because little effort has been invested so far in running this program and little is wasted by restarting it after an error.

Large outer box: generating and testing the numbers. This loop calls `rand()` many times and collects some information about the results. With the constant definitions given, we will generate 500 integers in the range $0 \ldots n-1$, where $n <= 100$. These numbers will be printed in 10 columns. Occurrences of 0 will be counted.

First inner box: generating the numbers.

1. The function `rand()` returns a number between 0 and `RAND_MAX`. According to the standard, this number may vary, but it is at least 32,767. We must scale this number to the desired range `0..n-1`.
2. The modulus operator is exactly what we want for a scaling operation, since its result is between 0 and the modulus -1. We compute `num % select` and store the result back in `num`.

Second inner box: lines of output.

1. The counter `n` keeps track of the total number of random numbers produced so far.
2. We want the output printed in columns, so we use a fixed field width in the conversion specifier: `%7li`. Ten columns, each seven characters wide, will fit conveniently onto the usual 80-column line.
3. We want to print a `'\n'` after every group of `NCOL` numbers but not after every number. To do this, we count the output items as they are produced and print a newline character every time the counter is an even multiple of `NCOL`; that is, `n % NCOL == 0`.

Third inner box: counting the zeros. To assess the “fairness” of the random-number generator, we can count the number of times a particular result shows up. If numbers in the range $0 \ldots n-1$ are being generated, then each individual number should occur `HOW_MANY / n` times. In this program, we expect each number in the possible range $1 \ldots$ `select`-1 to occur approximately 500/`select` times. The following are the first and last lines of output from three runs. Note that the number of zeros generated on two of three trials differs substantially from the number expected. This is an indication that the `rand()` function does not produce a very even distribution of numbers on our computer.

```
Generate 500 random numbers in the range 0..n-1.
Please choose n between 2 and 100: 25
    7   19   24    8   18   20    8   10    4    5
    5   18   14   19    5   11   24   17   11    5
   ...
    2    0    5   21   22    3   21    7   23   18

    18   zeros were generated.
    20.0 are expected on average.
```

```
---------------------------------------------------
Please choose n between 2 and 100: 33
   ...
   12   zeros were generated.
   15.2 are expected on average.
---------------------------------------------------
Please choose n between 2 and 100: 33
   ...
   20   zeros were generated.
   15.2 are expected on average.
```

5.8 Application: A Guessing Game

In a classic game, one player thinks of a number and a second player is given a limited number of tries to guess it. The first player must say whether the guess is too small, correct, or too large. We illustrated a simple example of this game in Figure 6.30. Now, we implement the full game in the next program example, with the computer taking the part of the first player.

5.8.1 Strategy

Even if a person has never seen this game, it does not take long to figure out an optimal strategy for the second player:

- Keep track of the smallest and largest remaining possible value.
- On each trial, guess the number midway between them.

The computer's response to each guess will eliminate half the remaining values, allowing the human player to close relentlessly in on the hidden number.

Figure 5.27 illustrates a game in which the range is 1 . . . 1,000 and the hidden number is 458. The player makes an optimal sequence of guesses, halving the range of remaining values each time: 500, 250, 375, 437, 468, 453, 461, 457, 459, 458. In this example, 10 guesses are required to home in on the hidden number. In fact, with this strategy, this also is the maximum number of trials required to find any number in the given range. Half the time the player will be lucky and it will take fewer guesses. The code for the program that implements this game is given in Figures 5.28 (`main()`) and 5.29 (`one_game()`, which handles the sequence of guesses).

5.8.2 Playing the Game

Choosing and scaling the number. We can call `rand()`, as was done in Figure 5.26, to get a number in the range `0..INT_MAX`. In this game, however, we require a random number between 1 and 1,000, so the number returned by `rand()` must be scaled and adjusted to fall within the desired range. To do this, we use the `%` (modulus) operator. For instance, `rand() % TOP` gives us a random number in the range

In this example, the total range of possible values is 1 ... 1,000. The hidden number, *num* = 458, is represented by a dashed line. The solid vertical lines represent the guesses of a player using an optimal strategy; only the first five guesses are shown.

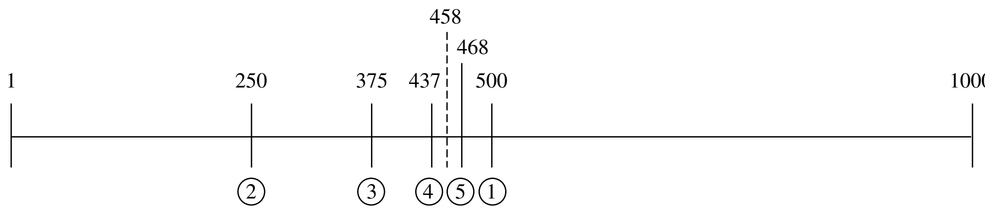


Figure 5.27. Halving the range.

(`0..TOP-1`). We then adjust it to the desired range simply by adding 1. This formula is used in the first box of Figure 5.29.[14]

Setting a limit on the number of guesses. The strategy shown in Figure 5.27 is an example of an important algorithm called **binary search**,[15] because we search for a target value by dividing the remaining set of values in half. If N values were possible in the beginning, the second player always could discover the number in T trials or fewer, where $N \leq 2^T$. Another way to say this is that

$$T = \lceil \log_2 N \rceil$$

where $\lceil \ldots \rceil$ means that we round to the next higher integer. Paraphrased, the maximum number of trials required will be the base-2 logarithm of the number of possibilities, rounded up to the next larger integer. In our case, this is

$$\lceil \log_2 1000 \rceil = \lceil 9.96578 \rceil = 10$$

To introduce an element of luck into the game, set the maximum number of guesses to something smaller than T. This, in fact, is what we do in Figure 5.28; we round *down* instead of *up*, allowing too few guesses about half of the time. This "stacks" the game in favor of the computer.

Calculating a base-2 logarithm. The C math library provides two logarithm functions: one calculates the base-10 log, the other the natural log (base e). Neither of these is what we want, but you can use the natural log function to compute the log of any other base, B, by the following formula:

$$\log_B(x) = \frac{\log_e(x)}{\log_e(B)}$$

[14]For reasons too complex to explain here, this formula has a slight bias toward lower numbers in the range. However, if the range is small compared to `RAND_MAX`, the bias is insignificant.

[15]Other examples of binary search are given in Chapters 11 and 19.

This main program calls the function in Figure 5.29. It repeats the game as many times as the player wishes.

```
#include "mytools.h"
void one_game( int tries );

#define TOP  1000           /* Top of guessing range. */

int main( void )
{
    int do_it_again;                           /* repeat-or-stop switch */

    const int tries = log( TOP ) / log( 2 );    /* One too few. */

    banner();
    puts( "\n This is a guessing game.\n I will "
          "think of a number and you must guess it.\n" );

    srand( (unsigned)time( NULL ) );      /* seed number generator. */

    do {  one_game( tries );
          printf( "\n\n Enter 1 to continue, 0 to quit: " );
          scanf( "%i", &do_it_again );
    } while (do_it_again != 0);

    return 0;
}
```

Figure 5.28. Can you guess my number?

In C, the natural log function is named `log()`, so to calculate the base-2 log of 1,000, we write

`log( 1000 ) / log( 2 )`

This formula is used in the second box of Figure 5.28.

Notes on Figure 5.28. Can you guess my number?

First box: guessing range.

In this game, the player will try to guess a number between 1 and `TOP`.

This function is called from Figure 5.28. It plays the number-guessing game once.

```
void
one_game( int tries )
{
    int k;                                   /* Loop counter. */
    int guess;                               /* User's input. */

    const int num = 1 + rand() % TOP;        /* The hidden value. */

    printf( " My number is between 1 and %i;"
            " I will let you guess %i times.\n"
            " Please enter a guess at each prompt.\n", TOP, tries );

    for (k = 1; k <= tries; ++k) {
        printf( "\n Try %i: ", k );
        scanf( "%i", &guess );
        if (guess == num) break;
        if (guess > num) printf( " No, that is too high.\n" );
        else printf( " No, that is too low.\n" );
    }
    if (guess == num) printf( " YES!!  That is just right.  You win!  \n" );
    else printf( " Too bad --- I win again!\n" );
}
```

Figure 5.29. Guessing a number.

Second box: the number of trials.

- We calculate `tries`, the maximum number of trials the user is allowed. It is defined as a **constant** because it depends only on `TOP` and does not change from one game to another. We use a `const` variable, rather than `#define`, because the definition is not just a simple number.[16]
- The result of the division operation is a `double` value that is coerced to type `int` when stored in `tries`.
- `C` permits us to use a formula to define the value of a constant. Such formulas can use literal constants (such as 2) and globally defined symbols (such as `TOP`). Inside a function definition, the parameter values also can be used in a constant expression.
- We calculate the base-2 logarithm of the number of possible hidden values and use this to set the maximum number of guesses. As described, we set this maximum so that the player will succeed about half the

[16]For reasons beyond the scope of this book, this is more efficient.

time. The rest of the time, the player will be one guess short of success, even if he or she is using the optimal search strategy.

Third box: initializing the random number generator. As in Figure 5.26, we initialize C's random number generator with the current time of day.

Notes on Figure 5.29. Guessing a number. The `one_game()` function is called from `main()` in Figure 5.28 for each round of the game that the player wishes to play.

First box: the hidden number. The first thing this function does is choose a hidden value. The value is defined as a constant because it does not change from the beginning of a round to the end of that round. In this case, the constant expression involves the value of a global constant. To calculate a random hidden number, we use the random number generator `rand()`, scaling and adjusting its value to the required range, as discussed earlier in this section.

Second outer box: playing the game. The code in this box is almost identical to the earlier version in Figure 6.30. Correct guesses are handled by an `if...break` in the inner box. When the loop exits, the program prints a failure message if the most recent guess was wrong.

Here is some sample output (omitting output of the query loop). The first two lines were printed by the main program, the rest of the dialog was printed by the `one_game()` function. In this sample game, the player was lucky and guessed the hidden number in only five tries. We expect this to happen only once in every 32 games.

```
This is a guessing game. I will think of a number and you must guess it.

My number is between 1 and 1000; I will let you guess 9 times.
Please enter a guess at each prompt.

Try 1: 500
No, that is too high.

Try 2: 250
No, that is too high.

Try 3: 125
No, that is too high.

Try 4: 62
No, that is too low.

Try 5: 93
YES!!  That is just right.  You win!
```

5.9 What You Should Remember

5.9.1 Major Concepts

- Functions can be used to break up programs into modules of manageable complexity. In this way, a highly complex job can be broken into short units that interact with each other in controlled ways. These modules should have the following properties:
  - The purpose of each function should be clear and simple.
  - All its actions should hang together and work at the same level of detail.
  - A function should be short enough to comprehend in its entirety.
  - The length and complexity of a function can be minimized by calling other functions to do subtasks.
- Function definitions can come from the standard `C` libraries such as the `stdio` and `math` libraries or from a personal library. Functions also can be defined by the programmer.
- Some functions compute and return values; others do not. A function with no return value is called a `void` function. Such functions normally perform input or output of some sort. This chapter presented programmer-defined functions with zero, one, and two parameters (void→void, double→double, and double×double→double respectively). Additional types of library functions and programmer-defined functions are introduced in subsequent chapters as additional data types are discussed.
- A call to a function that returns a value normally is found in an assignment statement, an expression, or an output statement.
- The basic components of a function are the prototype and the function definition, which consists of a header and a body.
- Arguments are the means by which a calling program communicates data to a function. Parameters are the means by which a function receives the communicated data. When a function is called, the actual arguments in the function call are passed into the function and become the values of the function's formal parameters.
- After passing the parameter values, control is passed into the function. Computation starts at the top and proceeds through the function until it is finished, at which time the result is returned and control is transferred back to the calling program. A more detailed look at functions will be given in Chapter 9.
- The name of the parameter in a function does *not* need to be the same as the name of a variable used in calling that function.
- Functions allow an application program to be constructed and verified much more easily than a similar program written as one massive unit. Each function, then each module, is developed and debugged before it is inserted into the final, large program. This is how professional programmers have been able to develop the large, sophisticated systems that we use today.
- Here is a (questionable) rhyme to help you remember the four ways to extract an integer from a `double` value:

`ceil()` goes up and `floor()` goes down; `rint()` goes nearby but assign doesn't round.

- Random numbers. Applications such as games, experiments, and quiz programs require a program to make a series of randomized selections from a preset list of numbered options. To do this, we use an algorithm called a *pseudo-random number generator*, which generates a series of integers with no apparent pattern. The random number then is scaled to be in the proper range if it does not fall within the range of selection numbers.

  The `standard` library provides the functions `srand()` and `rand()`, which together implement a pseudo-random number generator.

5.9.2 Local Libraries and Header Files

At various places in this chapter, suggestions were made about building a personal library called "mytools". In this section, we gather together the various parts that could be included in that library and show how they would be organized into a header file `mytools.h` and a code file `mytools.c`, and how a client program would use such a library.

The header file for mytools.

```
/* ------------------------------------------------------------------------
// Alice Fischer's personal tools library.                    File:  mytools.h
// Last updated on Tue Sep 16 2003.
*/

/* Standard library headers that I need. --------------------------------- */
#include <stdio.h>      /* for puts(), printf(), and scanf() */
#include <stdlib.h>     /* for exit() */
#include <time.h>       /* for time_t, time() and ctime() */

#define PI  3.1415927   /* defined on some systems but not on mine. */
#define GRAVITY 9.8  /* needed for several programs. */

/* Prototypes for my own tool functions. --------------------------------- */
void banner( void );              /* Print a neat header for the output.  */
```

The code file for mytools.

```
/* ----------------------------------------------------------------------
// Alice Fischer's personal tools library.                     File:  mytools.c
// Last updated on Tue Sep 16 2003.
*/
#include "mytools.h"
/* ------------------------------------------------------------------- */
void banner( void )               /* Print a neat header for the output.  */
{
    time_t now = time(NULL);
    printf( "\n-----------------------------------------------------\n" );
    printf( "    Alice E. Fischer\n    CS 110\n    ");
    printf( ctime( &now ) );
    printf(  "-----------------------------------------------------\n" );
}
```

A program that uses mytools.

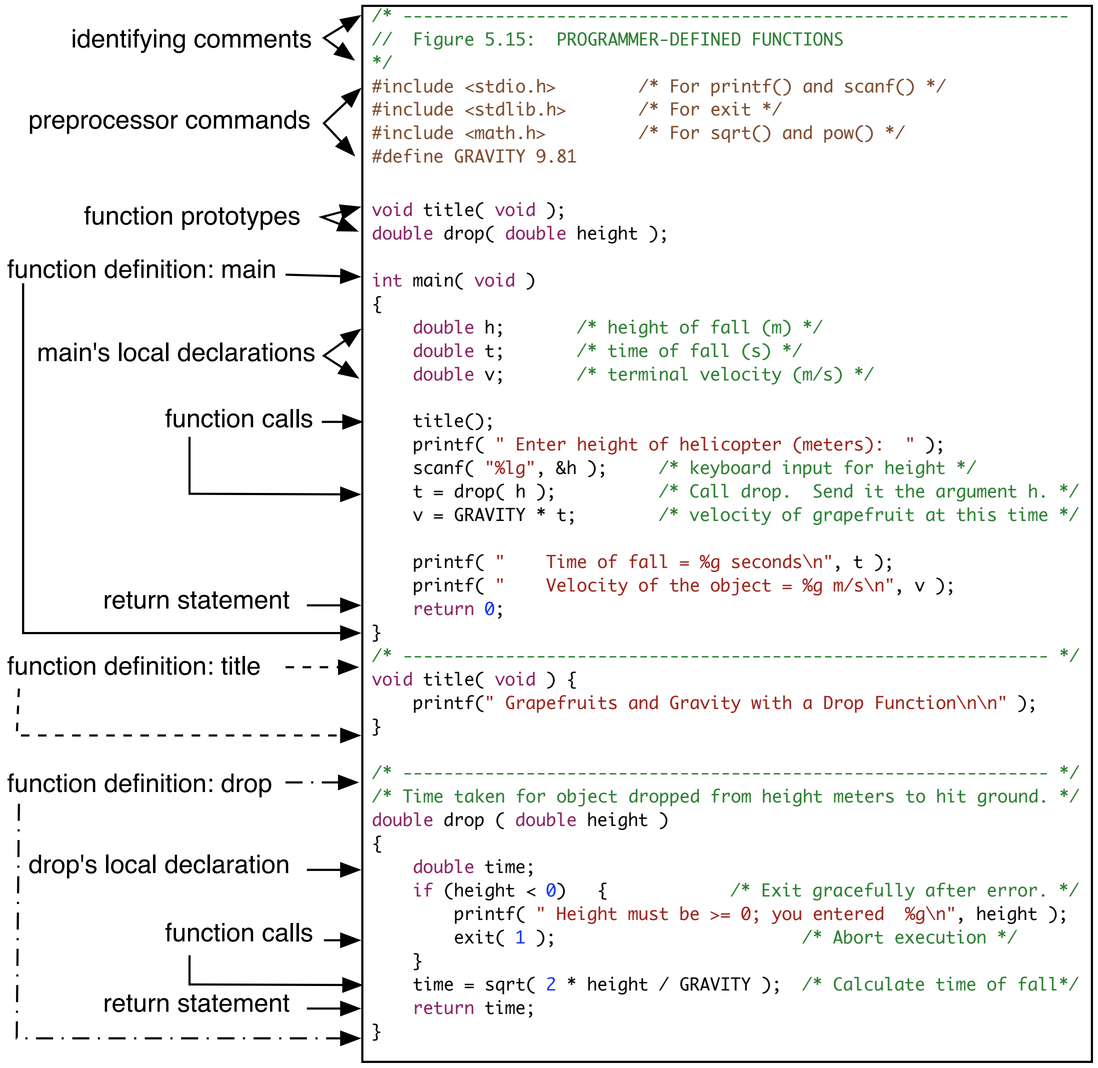
```
/* ------------------------------------------------------------------
// Demo program for using a personal library.  Both files mytools.h and
// mytools.c must be added to the project for this program.
// ------------------------------------------------------------------
*/
#include "mytools.h"
int main( void )
{
    int count; /* Number of people in the room. */
    banner();
    printf( " Demonstrating the use of a personal library.\n"
            " How many people are watching?  " );
    scanf( "%i", &count );
    if (count < 2)  puts( " Not enough.\n" );
    else puts( " Hello you-all!\n" );
    return EXIT_SUCCESS;
}
```

Sample output from this demo.

```
-----------------------------------------------------
    Alice E. Fischer
    CS 110
    Tue Sep 16 12:26:45 2003
-----------------------------------------------------
 Demonstrating the use of a personal library.
 How many people are watching?  3
 Hello you-all!
```

5.9.3 The Order of the Parts of a Program

The following diagram summarizes the order in which it is customary to write prototypes, function definitions, and other elements in a source code file. The prototypes are normally written first, followed by the `main()` function then all other functions, in any convenient order.



```
/* ------------------------------------------------------------------
//  Figure 5.15:  PROGRAMMER-DEFINED FUNCTIONS
*/
#include <stdio.h>        /* For printf() and scanf() */
#include <stdlib.h>       /* For exit */
#include <math.h>         /* For sqrt() and pow() */
#define GRAVITY 9.81

void title( void );
double drop( double height );

int main( void )
{
    double h;       /* height of fall (m) */
    double t;       /* time of fall (s) */
    double v;       /* terminal velocity (m/s) */

    title();
    printf( " Enter height of helicopter (meters):  " );
    scanf( "%lg", &h );     /* keyboard input for height */
    t = drop( h );          /* Call drop.  Send it the argument h. */
    v = GRAVITY * t;        /* velocity of grapefruit at this time */

    printf( "    Time of fall = %g seconds\n", t );
    printf( "    Velocity of the object = %g m/s\n", v );
    return 0;
}
/* ---------------------------------------------------------------- */
void title( void ) {
    printf(" Grapefruits and Gravity with a Drop Function\n\n" );
}

/* ---------------------------------------------------------------- */
/* Time taken for object dropped from height meters to hit ground. */
double drop ( double height )
{
    double time;
    if (height < 0)   {            /* Exit gracefully after error. */
        printf( " Height must be >= 0; you entered  %g\n", height );
        exit( 1 );                            /* Abort execution */
    }
    time = sqrt( 2 * height / GRAVITY );  /* Calculate time of fall*/
    return time;
}
```

5.9.4 Programming Style

- Keep the main program and all function definitions short. An entire function should fit on one video screen, so that the programmer can see the declarations and the code at the same time. When a function begins to get long, this often is a sign that it should be broken into two or more functions.
- Every function definition should start with a distinctive and highly visible comment. We use a dashed line followed by a brief description of the purpose of the function. This comment block helps you to find each portion of your program quickly and easily, both on screen and on paper.
- Function names should be descriptive, distinctive, and not overly long.
- All the object names used in a function should be either parameter names or defined locally as variables. This provides maximum isolation of each function from all others, which substantially aids debugging.

5.9.5 Sticky Points and Common Errors

- The prototype must match the function header. If there is a mismatch, the program will not compile correctly.
- Prototypes end in semicolons. If the semicolon is missing, the compiler "thinks" that the prototype is the function header. This will cause many meaningless error comments.
- Definitions must *not* have a semicolon after the function header. If a semicolon is written there, the compiler "thinks" the header is a prototype and will give an error comment on the next line.
- If the prototype is missing or comes after the first function call, the compiler will construct a prototype using the types of the parameters given in the first function call. The return type always will be `int`, which may or may not be correct. If the constructed prototype is wrong, it will cause the compiler to give error comments on lines that are correct.

5.9.6 Where to Find More Information

- Chapter 9 presents the full process of top-down programming and stub testing. This discussion has been deferred until the basic mechanics of functions and function calls are better understood.
- Chapter 9 has a more complete discussion of functions and prototypes.
- Chapter 9 explains local and non-local variables and how they are implemented using activation records on the run-time stack. Figure 9.2 shows how storage might be managed at run time for the cylinder program in Figure 5.24.
- Functions and type definitions to help programmers use the clock are described in Appendix F and parts are discussed in detail in Chapter 12.
- Chapter 12 shows a revised version of the `banner()` function.
- The website for this chapter introduces a function named `fatal()` that combines the actions of `printf()` and `exit()`, allowing error handling to be done in one statement.

- The results of a subprogram must be passed back to the caller. Depending on the function's purpose, there may be no, one, or more results. but only one result can be returned through a `return` statement. There are several ways around this difficulty:
  - Chapter 10 discusses array parameters, which can be sent, empty, into a function, then filled in the function and returned containing a potentially large amount of data.
  - Chapter 11 shows how pointer arguments may be used to return information from a function to the caller.
  - Chapter 13 discusses compound objects (structures) that can contain and return many pieces of information packed into a single object.

5.9.7 New and Revisited Vocabulary

These are the most important terms and concepts that were introduced or discussed in this chapter:

function	function comment block	call graph
standard library	function body	pseudo-random numbers
personal library	function return	seed
header file	programmer-defined function	binary search
function call	prototype declaration	constant expression
caller	function definition	void→int function
argument	function interface	void→void function
subprogram	function header	int→void function
transfer of control	parameter	double→double function
interrupted flow	local variable	double×double→double function
exception		

The following C keywords, header files, library functions, constants, and types were discussed in this chapter:

`return` statement	`<stdlib.h>`	`time_t` (from `time.h`)
`<stdio.h>`	`EXIT_FAILURE` (from `stdlib.h`)	`time(NULL)` (from `time.h`)
`scanf()` (from `stdio`)	`EXIT_SUCCESS` (from `stdlib.h`)	`ctime()` (from `time.h`)
`printf()` (from `stdio`)	`exit()` (from `stdlib`)	`<math.h>`
`puts()` (from `stdio`)	`abs()` (from `stdlib`)	`sqrt()` (from `math`)
`<limits.h>`	`srand()`(from `stdlib`)	`floor()` (from `math`)
`INT_MAX` (from `limits.h`)	`rand()`(from `stdlib`)	`log()` (from `math`)
`RAND_MAX` (from `stdlib.h`	`<time.h>`	`pow()` (from `math`)

5.10 Exercises

5.10.1 Self-Test Exercises

1. For each part, write a double→double function that computes the formula and returns the result:

   (a) Tangent of angle $x = \dfrac{\sin(x)}{\cos(x)}$
   (b) Surface area of a sphere with radius $r = 4 \times \pi \times r^2$

2. For each part, write a double$\times$double$\rightarrow$double function that computes the formula and returns the result:

   (a) Hypotenuse of a right triangle: length $= \sqrt{\text{base}^2 + \text{height}^2}$
   (b) Polar to rectangular coordinates: $y = r \times \cos(\text{theta})$
   (c) Rectangular to polar coordinates: theta $=$ arc tangent(y/x) for $x \neq 0$

3. Write the prototype that corresponds to each function definition:

   (a) `double cube( double x ) { return x*x*x; }`
   (b) `void three_beeps() { beep(); beep(); beep(); }`
   (c) `int surprise( void ) { return 17; }`
   (d) `int ratio( double a, double b) { return a/b; }`

4. Explain the difference between

   (a) An argument and a parameter
   (b) A prototype and a function header
   (c) A header file and a source code file
   (d) A personal library and a standard library
   (e) A function declaration and a function call
   (f) A function declaration and a function definition

5. What happens on your compiler when a prototype for a programmer-defined function is omitted? To find out, start with the code from Figures 5.15 and 5.18 and delete the prototype above the main program.

6. Prototypes and calls. Given the prototypes and declarations that follow, fix the errors in each of the lettered function calls.

```
void   squawk( void );
int    half( int );
double area( double, double);

int j, k;
double x, y, z;
```

   (a) `k = squawk();`
   (b) `squawk( 3 );`
   (c) `j = half( 5, k );`
   (d) `j = half( int k );`
   (e) `y = area( double 3.0, x );`

7. Find the error here. Using the declarations given in problem 6, find the one function call below that has an error and explain what the error is. (The other three are correct.)
   (a) `y = area( x+2, 3 );`
   (b) `y = half( half( k ) );`
   (c) `printf( "%g %g", x, half( x ) );`
   (d) `y = area( sin( x ), z );`

5.10.2 Using Pencil and Paper

1. List everything you must write in your program when you want to *use* (not define) each of the following:
   (a) A programmer-defined void→void function named `help()`
   (b) The `fatal()` function
   (c) The `sqrt()` function
2. Write a prototype that could correspond to each function call below:
   (a) `do_it();`
   (b) `x = calculate( y, z);`
   (c) `k = get_Integer();`
   (d) `x = cubeRoot( y );`
3. For each part, write a double→double function that computes the formula and returns the result:
   (a) Sine: $\sin(x) = \sqrt{1 - \cos^2(x)}$
   (b) Tangent: $\tan(2x) = \dfrac{2\tan(x)}{1 - \tan^2(x)}$
   (c) Diagonal of a square with side $s = \sqrt{2 \times s^2}$
   (d) Volume of a sphere with radius $r = \dfrac{4\pi}{3} \times r^3$
4. Write a double×double→double function that computes the formula and returns the result:

   Sum of angles: $\sin(x + y) = \sin(x)\cos(y) + \cos(x)\sin(y)$
5. What happens on your compiler when a required `#include` command is omitted? Will a program compile? Will it work correctly?
6. Prototypes and calls. Given the prototypes and declarations that follow, say whether each of the lettered function calls is legal and makes sense. If the call has an error, fix it.

```
void   squawk( void );
int    half( int );
double area( double, double);

int j, k;
double x, y, z;
```

(a) `j = squawk();`
(b) `half( k );`
(c) `j = half( 5 * k );`
(d) `y = area( double 3.0, double x );`
(e) `y = area( z );`
(f) `y = area( pow( x, 2 ), z );`
(g) `scanf( "%i %i", &k, &half( k ) );`
(h) `y = area( x, y, z );`

7. Write the prototype that corresponds to each function definition. Then define appropriate variables and write a legal call on the function.

   (a) `double inches( double cm ) { return cm / 2.54; }`
   (b)
```
void beeps( int n )
{   int k = 0;
     while (k < n) {
         beep();
         ++k;
     }
}
```

5.10.3 Using the Computer

1. Geometric mean.

   A *geometric progression* is a series such as 1, 2, 4, 8, ... or 1, 5, 25, 125, ... such that each number in the series is multiplied by a constant to get the next number. More formally, the constant, R, is called the **common ratio**. If the first term in the series is a, then succeeding terms will be $a*R$, $a*R^2$, $a*R^3$, ... For instance, if $a = 1$ and $R = 2$, we get 1, 2, 4, 8, ... and if $a = 1$ and $R = 5$, we get 1, 5, 25, 125, ...

   Given terms $k-1$ and $k+1$ in a geometric progression, write a main program and at least one function to compute term k and the common ratio, R. Term k is called the *geometric mean* between the other two terms. The formulas for the kth term and common ratio are

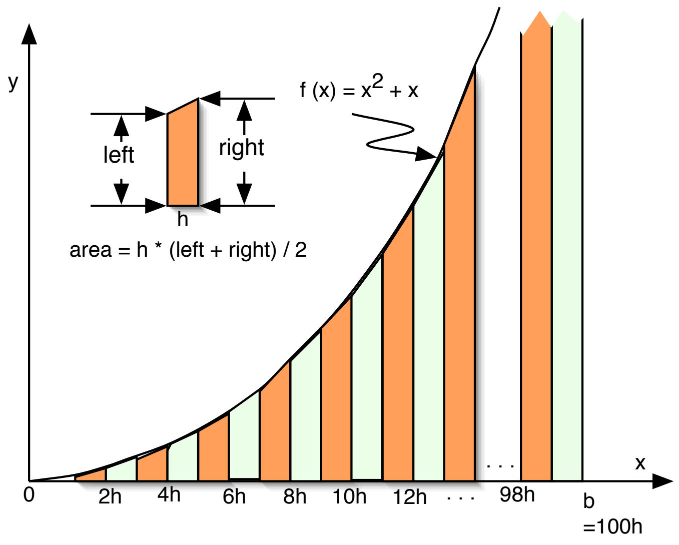
$$R = \sqrt{\frac{t_{k+1}}{t_{k-1}}} \quad \text{and} \quad t_k = R \times t_{k-1}$$

2. Numerical integration.

   The program in Figure 5.23 integrates the function $f(x) = x^2 + x$ for $0 \leq x \leq 1.0$. Modify this program to integrate the function $f(x) = x^2 + 2x + 2$ for $-1.0 \leq x \leq 1.0$. If you have studied symbolic integration, compare your result to the exact analytical answer.

3. Integration using trapezoids.

   Modify the program in figure 5.23 to use trapezoids, rather than rectangles, to approximate the area under the curve, according to this sketch of the trapezoid method:



Compare your answers to those from Figure 5.23. What can you say about their accuracy?

4. Precision of numerical computations.

   Figure 5.23 gives a program that integrates a function by rectangles. Keep the function `f()` and modify the main program so that the integration process will be repeated using 10, 20, 40, 80, 160, and 320 rectangles. Print a neat table of the number of rectangles used and the results computed each time. Compare the answers. What can you say about their accuracy?

5. Sales tax.

   Write a double→double function whose parameter is a purchase price. Calculate and return T, the total price, including sales tax. Define the sales tax rate as a `const` R whose value is 6%. Write a main program that will read P, the before-tax amount of a purchase, call your function to calculate the after-tax price, and print out the answer. Both prices will be in units of dollars and cents. What are the appropriate types for P, R, and T? Why? Use the program in Figure 5.15 as a guide.

6. Fence me in.
   A farmer has several rectangular fields to fence. The fences will be made of three strands of barbed wire, with fence posts no more than 6 feet apart and a stronger post on every corner. Write a complete specification, with diagram, for a program that will input the length and width of a field, in feet. Make sure that each input is within a meaningful range. If so, calculate the area of the field and the total length of barbed wire required to go around the field. Also calculate the number of fence posts needed (use the `ceil()` function from the `math` library). Now write a program that will perform the calculations and display the results.

7. A spherical study.

   Write four functions to compute the following properties of a sphere, given a diameter, d, which is greater than or equal to 0.0:

   (a) Radius $r = d/2$
   (b) Surface area $= 4 \times \pi \times r^2$
   (c) Circumference $= \pi d$
   (d) Volume $= \frac{4\pi}{3} \times r^3$

   Write a main program that will input the diameter of a sphere, call all four functions, and print out the four results. Do not accept inputs less than 0.0.

8. Take-home pay.

   Write a double→double function whose parameter is an employee's gross pay for one month. Compute and return the take-home pay, given the following constants:

   - Medical plan deduction = \$75.65
   - Social security tax rate = 7.51%
   - Federal income tax rate = 16.5%
   - State income tax rate = 4.5%
   - United Fund deduction = \$15.00

   The medical deduction must be subtracted from the gross pay before the tax amounts are computed. Then the taxes should be computed and subtracted from the gross. As each one is computed, print the amount. Finally, subtract the United Fund contribution and return the remaining amount. Your main program should print the final pay amount.

9. Hourly Employee.

   Write a double×double→double function whose parameters are an employee's hourly pay rate and number of hours worked per week. Compute and return the gross pay.

   Your main program should call the hourly-pay function, then use its result to call the take-home pay function described in the previous problem. Print the take-home pay, as before.

10. Compound interest.

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

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

    (b) Write a main program that will permit the user to enter P, i, and n. Call your function to compute A. Your main program should echo the inputs and print the final dollar amount.

11. Probability.

    A statistician needs to evaluate the probability, p, of the value x occurring in a sample set with a known normal distribution. The mean of the distribution is $\mu = 10.71$ and the standard deviation is $\sigma = 1.14$.

(a) Write a double→double function with parameter x that computes the value of the probability formula for a normal distribution, which follows. To compute e^x, use the `exp()` function from the math library; its prototype is `double exp( double x )`.

$$p = \frac{1}{\sigma \times \sqrt{2\pi}} \times e^{-d}, \qquad \text{where } d = \frac{[(x-\mu)/\sigma]^2}{2}$$

(b) Write a main program to input the value for x, call your probability function, and print the results.