Chapter 9

Program Design

In Chapter 5, we introduced the concepts of functions and function calls and defined some basic terminology. In this chapter, we review that terminology, extend the rules for defining and using functions, and formalize many aspects of functions presented in preceding chapters: prototypes, function definitions, function calls, how these elements must correspond, and how the necessary communication actually happens. The concepts of local, global, and external names are presented.

We discusses modular organization and the ways that parts of a modular program must relate to each other. The process of designing a modular program is described and illustrated with a programming example.

9.1 Modular Programs

When a program has only 20 to 50 lines, a programmer can keep the entire program structure in mind at once. Many programs, though, have thousands of lines of code. To deal with this complexity, it is necessary to divide the code into relatively independent modules and consider each module in isolation from the others, usually with a main program in one module and groups of closely related functions in others. Each module is composed of functions and declarations that relate to one identifiable phase of the overall project. This is the way professionals have been able to develop the large, sophisticated systems that we use today.

A **module** is a pair of files (a `.c` file and its matching `.h` file) containing programmer-defined types, data object declarations, and function definitions. The order of these parts within the module is quite flexible; the only constraint is that *everything must be declared before it is used*. The modules themselves and the functions within them serve several purposes: they make it easy to use code written by someone else or reuse your own code in a new context. Far more important, though, is that they permit us to break a large program into manageable pieces in such a way that the interface among pieces is fixed and controllable. Functions, their prototypes, and header files make this possible in C; class definitions make it easier in C++.

Each **function** is a block of code that can be invoked, or called, from another function and will perform a specific, defined task. All its actions should be related and work at the same level of detail. For example, some functions "specialize" in input or output. Others calculate mathematical formulas or do error handling.

No function should be very long or very complex. Complexity is avoided by calling other functions to do subtasks.

One guideline for good style is to keep each function short enough that its parameters, local variable declarations, and code will fit on the video screen at the same time. This limits functions to about 20 lines of code on many computers. Following this guideline, any moderate-sized program will have many functions that are organized into several code modules or classes, with each module or class in a separate source file. The question then arises of where various objects should be declared: in which module and where within the module. This level of design is touched on in Chapter 20; we consider only the organization of a single module here.

9.2 Communication Between Functions

Before beginning this section, you should review the overview of functions and module organization that is given in Chapter 5.

One function in every program must be named `main()`; it is often called the *main program*. The **main program** can call other functions and those functions can call each other, but none can call `main()`. We use the term **caller** to refer to the function that makes the call and **subprogram** to refer to the called function. For example, in Figures 5.24 and 5.25, `main()` is the caller and `cyl_vol()` is a subprogram that is called from the second box in `main()`.

9.2.1 The Function Interface

Each function has a defined **interface**, which is declared by writing a prototype for the function. The prototype lists the return type and the type and name of one or more parameters. Either the parameter list or the return type may be replaced by `void`.

Each parameter listed in the prototype becomes a separate communication path by which the caller can send information into the function. The function's result or return value (if not `void`) is a communication path from the subprogram back to the caller, as is each address parameter. Information also can be passed between a caller and a subprogram through global variables, which will be discussed in Section 9.4. However, this practice is discouraged and should be avoided wherever possible. Extensive use of global variables can make a program difficult to debug because it vastly increases the number of possible interactions among program modules and introduces the possibility for unintended and undocumented interactions. Thus, modern programming style dictates that all information passed between caller and subprogram should go through the declared interface.

In Figure 9.1, the body of each function is shaded, indicating that it may appear as a "black box" to the programmer. The inner workings of a library function frequently are hidden from the programmer, like those of `exp()` in this case. You need not know the details of what is inside a function to be able to use it.

In contrast, interfaces are white. This symbolizes that the programmer can (and must) know the details of the interface. This information is supplied in a header file and by the documentation that normally

The arrows show the ways that information flows to and from functions in Figure 9.3.

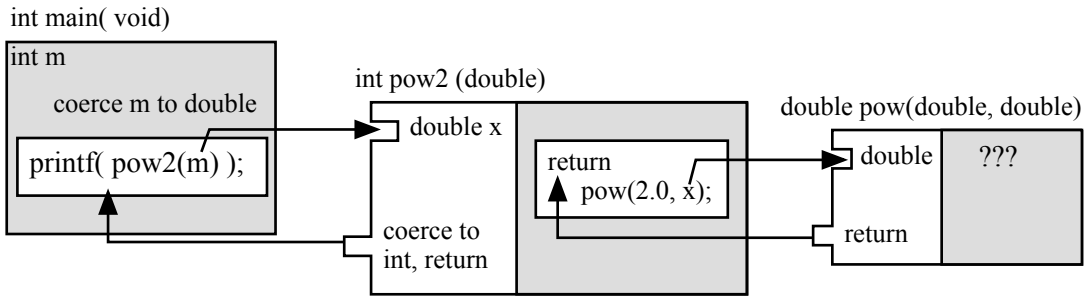


Figure 9.1. Information flow in `pow2()`.

accompanies a software library. **Header files** are used to keep the declarations consistent from module to module and to permit functions in one module to call functions in another.

The passage of information into the subprogram is represented by right-facing arrows. Function execution begins at the entry point and, when complete, control returns to the caller along a left-facing arrow, which ends at the return address in the caller. The caller continues processing from that point. The function also may return a result along the left-facing arrow.

A **function call** consists of the name of the function followed by a list of arguments in parentheses. Some functions have no parameters, in which case the parentheses in the function call still must be written but with nothing between them. During the calling process, two kinds of information are sent from the caller into the subprogram:

- One argument value for each declared parameter.
- The **return address**, which is the address of the first instruction in the *caller* after the function call. This address is passed by the caller to the subprogram on every call so that the function knows where to go when its execution is finished.

During a function call, the C run-time system allocates a stack frame, stores the argument values there, and stores the return address in the adjacent locations. Control is then transferred to the function, which allocates (an possibly initializes) storage for local variables. Control then jumps to the **entry point** of the function, which is the first line of code in its body. Execution of the subprogram begins and continues until the last line of code is completed, the program aborts by calling `exit()`, or control reaches a `return` statement, which returns control to the caller at the return address, taking along any return value produced.

The **return type** declared in the prototype is the type of value that will be returned. If the `return` statement returns an answer of some other type, C will convert it to the declared type and return the converted value. If such a conversion is not possible, the compiler will issue an error comment. This is discussed more fully in Section 9.3.

Missing prototypes. The general rule in C is that everything must be declared before it is used. There are two ways to "declare" a function: either supply a prototype or give the complete function definition. To guarantee correctness, one or the other must occur in your program before any call on that function. The C compiler[1] must know the prototype of a function to check whether a call is legal. Sometimes, however, a programmer forgets to either `#include` a necessary header file or write a prototype for a locally defined function. Sometimes the prototype is in the file but in the wrong place, coming after the first call on the function.

In any of these cases, the compiler does *not* just give an error comment about a missing prototype and quit. The first time it encounters a call on a nonprototyped function it simply *makes up* a prototype and continues compiling. The compiler will use the types of the actual arguments in the call to construct a prototype that matches, but all such created prototypes have the return type `int`. Sometimes this prototype is exactly what the programmer intended; other times it is wrong because the call depends on an automatic type conversion or contains an error. In any case, the constructed prototype becomes the official prototype for the function and is used throughout the rest of the program. If it has an incorrect parameter or return type, the compiler will compile too many or too few type coercions for each function call.

If a misplaced prototype is found later in the file and it is the same as the prototype constructed by the compiler, there is no problem. However, if it is different, the compiler will give an error comment such as *type conflict in function declaration* or *illegal redefinition of function type*. This can be an astonishing error comment if the programmer does not realize that the problem is *where* the prototype was written, not *what* was in it. If the prototype really is missing, not just misplaced, a similar error comment may be produced when the compiler reaches the actual function definition. If you see such an error comment, check that all functions have correct prototypes and that they are at the top of the program.

9.2.2 Parameter Passing

When each function is called, a new and separate memory area, called a *stack frame*, is allocated for it. The function's parameters are allocated in its stack frame and argument values are copied into the parameter during the function call. A function's local variables are also in the frame, as is the information necessary to return from the function. Figure 9.2 is a diagram of the run time memory allocated for the program in Figure 5.24. As each function is called, a new frame for it is placed on the stack. First `main()` was called, then it called `cyl_vol()`, which called `pow()`. Parts of the `pow()` frame are gray because we have no idea how this library function is implemented or what its parameters are named.

When a function returns, its stack frame is deleted. Later a frame for another function might be stored in the same addresses. In the cylinder program, storage for the `cyl_surf()` function will end up in the memory locations that `cyl_vol()` had previously occupied, and the stack frame for `pow()` will be at a slightly different address because the frame for `cyl_surf()` is bigger than the frame for `cyl_vol()`.

[1] ISO C and older C implementations differ extensively on the rules for function prototypes, definitions, and type checking. In this text, we speak only of standard ISO C.

We see the memory allocated for the program in Figure 5.24 at two moments during run time. The diagram on the left shows memory just after the `pow()` function is called by `cyl_vol()`. The diagram on the right is a later moment, during execution of `cyl_surf()`.

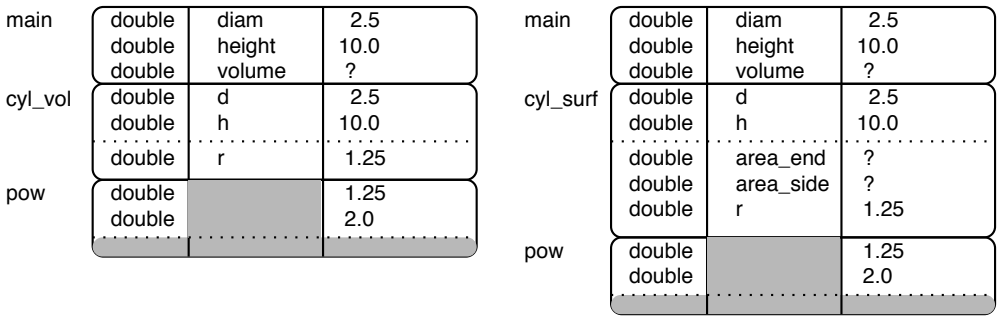


Figure 9.2. The run-time stack.

Call by value. Most arguments in C are passed from the caller to the subprogram **by value**. This means that a copy of the value of the argument is sent into the subprogram and becomes the value of the corresponding parameter. The function does not know the address of the argument, which could be a variable or the result of an expression. If the argument is a variable, the subprogram cannot change the value stored there. For example, in Figure 5.25, the subprogram `cyl_vol()` receives the value of `main()`'s variable `diam` but not the address of `diam`. The code in the body of `cyl_vol()` can change the value of its own parameter, `d`, but doing so will not change the value stored in `main()`'s `diam`. Information cannot be passed back to the caller through an ordinary parameter.

Address parameters. In contrast, some arguments are passed by passing the address of a variable (rather than its value) into the subprogram. The subprogram can both use and change the information at the argument address. An example of a function that sometimes must return more than one piece of information is `scanf()`. It uses the return value to return an error code, which we have ignored so far,[2] and it returns one or more data items through **address arguments**. When we call `scanf()`, we send it the address of each variable to be read. It reads the data, stores the input(s) in the given address(es), and returns a success or failure code. A programmer also can define such functions with address parameters; we explain how in Chapter 11.

[2]This will be explained in Chapter 14.

9.3 Parameter Type Checking

The compiler uses a prototype for two purposes: checking whether the call is legal and compiling any type conversions necessary to make the argument types match the parameter types.

Number of arguments. If the number of arguments in a function call is appropriate, the compiler considers each parameter-argument pair, one by one, comparing the parameter type declared in the prototype to the type of the argument expression. If they match exactly, code is compiled to copy the argument values into the subprogram's parameters and transfer control to its entry point. If the number of arguments supplied by a function call does not match the number declared in the prototype, the compiler prints an error comment[3].

Type coercion of mismatched arguments and parameters. If the number of arguments is the same as the number of parameters but their **types do not match** exactly, the compiler will attempt to convert each argument to the declared parameter type according to the standard type-conversion rules. We already discussed a large number of variations of the basic integer, floating-point, and character types: `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `char`, `signed char`, `unsigned char`, `float`, `double`, and `long double`. Taken as a set, these are called the **arithmetic types**. An argument of any arithmetic type can be coerced to any other arithmetic type, if needed, to make the argument's type match the type declared in the function's prototype. An instance of such **type coercion**, is shown in the Figure 9.3; the argument in the call on `pow2()` is an `int`, while the parameter is a `double`. The C compiler will include code to convert the `int` argument value to type `double` as part of the function call, and the function will receive the `double` value that it expects.

Meaningful conversions. Type conversion or coercion is possible if there is a meaningful relationship between the two types, such as both being numbers. If conversion is not possible, the compiler will issue a fatal error comment.[4] The rule in C is that any numeric type can be converted to any other numeric type. Therefore, a `short int` can be converted to a `long int` or an `unsigned int` or a `float` and vice versa. Some kinds of argument coercions are very common and compilers simply include code for the conversion and do not notify the programmer that it was necessary. For example, normally no warning would be given when a `float` value is coerced to type `double`. At other times, compilers warn the programmer that a conversion is occuring. This happens when an unusual kind of argument conversion would be required or the conversion might result in a loss of information due to a shortening of the representation, as when a `double` value is converted to type `float`. The warning you get depends on the nature of the type mismatch, the severity of the possible consequences, and your particular compiler.

[3]Some functions, such as `printf()` and `scanf()` permit the number of arguments to vary. Similar functions can be written using special argument-handling mechanisms supported by the `stdard` library.

[4]If the function has an ANSI prototype, the coercions allowed for arguments and return values are the same as those allowed for assignment statements.

```
#include <stdio.h>
#include <math.h>
int pow2 ( double x ) { return pow( 2.0, x ); }      /* 2 to the power x */

int main( void )
{   int m;

    for (m=0; m<10; ++m)  printf( "%10i\n", pow2(m) );
    return 0;
}
```

This program prints a table of the powers of 2 using a `double`→`int` function named `pow2()`. When `pow2()` is called from `main()` with an integer argument, the argument will be coerced to type `double` to match `pow2()`'s prototype. Within the function, the result of calling `pow()` will be coerced to type `int`, to match `pow2()`'s prototype, before being returned to `main()`.

Figure 9.3. The declared prototype controls all.

Type coercion of returned values. The declared return type also is compared to the actual type of the value in the `return` statement. If they are different, the value will be coerced to the declared type before it is returned. The compiler generates the conversion code automatically. For example, in Figure 9.3, the value calculated by the expression in the `return` statement is of type `double` (the math-library functions always return `double`s). However, the prototype for `pow2()` says that it returns an `int`. What happens? The C compiler will notice the type mismatch and compile code to coerce the `double` value to an `int` during the return process. The compiler may also give a warning message.

Parameter order. The order of the arguments in a function call is important. If a function's parameters are defined in one order and arguments are given in a different order in a call, the results generally will be nonsense. There is no "right" order for the parameters in a function definition; this is a design issue. However, once the definition has been written, the order of the arguments in the call must be the same. If the program has several functions that work with the same parameters, the designer should choose some order that makes sense and consistently stick to that order when defining the functions to avoid absentmindedly writing function calls with the arguments in the wrong order.

Sometimes a compiler, by performing its normal parameter type checking, can detect an error when a programmer scrambles the arguments in a function call. More often, this is not possible. For example, the function `cyl_vol()` in Figure 5.24 has two parameters, the diameter and height of a cylinder. Since the parameters are the same type, the compiler cannot tell when the programmer writes them in the wrong order. The result will be a program that compiles, runs, and produces the wrong answer. To further demonstrate the kind of nonsense that can result from mixed-up arguments, we will use a silly two-parameter function named `n_marks()` that inputs a number N and a character C and prints N copies of C. (Figure 9.4)

This program is used to show the effects of calling a function with parameters in the wrong order.

```
#include <stdio.h>
#include <math.h>
void n_marks( double n, char ch );   /* function prototype */

int main( void )
{
    char ch;     /* The character to print */
    double x;    /* How many characters to print */

    printf( "\nEnter a number and a character: " );
    scanf( "%lg %c", &x, &ch );

    printf( "\nYou entered %.2f and %c.", x, ch );
    n_marks( x, ch );            /* Call function to print x many ch's */
    return 0;
}
/* ------------------------------------------------------------------- */
void n_marks( double n, char ch )    /* Print n copies of character ch. */
{
    int k;                                  /* # of characters already printed */
    putchar( '\n' );
    for (k = 0; k<n; ++k) putchar( ch );    /* print n characters */
    printf( "\n\n" );                       /* flush output to screen */
}
```

Figure 9.4. The importance of parameter order.

The output is

```
Enter a number and a character: 45.7 !
You entered 45.70 and !.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Compare this output to the output from the erroneous call in the next paragraph.

Parameter coercion errors. When the compiler translates a function call, it either accepts the arguments as given, coerces them to the declared type of the parameter, or issues a fatal error comment. This automatic conversion can result in some weird and invalid output. For example, suppose a programmer called the

`n_marks()` function but wrote the arguments in the wrong order: `n_marks( ch, x );` The output would be

```
Enter a number and a character: 45.7 !
You entered 45.70 and !.
---------------------------------
```

This call certainly is not what a programmer would intend to write. However, according to the ISO C standard, this is a legal call. The standard dictates that the character `ch` will be converted first to an `int` and then to a `double`, while the `double x` will be converted first to an `int` and then to a `char` to match the parameter types in the prototype `void n_marks( double, char )`. Using this input, the programmer would expect to see a line of 46 `!` signs, but instead a line of 33 `-` signs is printed because of the conversions: The ASCII code for `!` is 33, which will be converted to 33.00, and the 45.7 will be converted to 45, which is the code for `-`. Some compilers at least may give a warning comment about these two "suspicious" type conversions, but all standard ISO C compilers will compile the conversion code and produce an executable program. When you try to run such a program, the results will be nonsense, as shown.

9.4 Data Modularity

A large program may contain hundreds or thousands of objects (functions, variables, constants, types, etc.). If a programmer had to remember the name, purpose, and status of this many objects, large programs would be very hard to write and harder to debug. Happily, C supports modular programming. Each module has its own independent set of objects and interaction between modules can be strictly controlled. The same name can be used twice, in different modules, to refer to different objects, so a programmer need not keep a mental catalog of the hundreds of names that might have been used. C's accessibility and visibility rules determine *where* a variable or constant may be used and *which* object a name denotes in each context. We use the program in Figures 9.5 and 9.7 to illustrate these concepts and the related concept of scope.

9.4.1 Scope and Visibility

The **scope** of a name is the portion of the program in which it exists and can be used. The three levels of scope are local, global, and external; respectively meaning that access to an object can be restricted to a single program block or function, shared by all functions in the same file or program module, or shared by parts of the program in different files or program modules. More specifically,

- Parameters and variables defined within a function are completely **local**; no other functions have access to them.
- We are permitted to declare variables outside of the function definitions, either at the top of a file or between functions. These are called **global** variables. They can be used by other modules and all the functions in the code module that occur after their definition in the file. It is possible (but not the default) to restrict the use of these symbols to the module in which they are defined.

This program calculates the pressure of a tank of CO gas using two gas models. The functions in Figure 9.7 are part of this program and should be in the same source file. A call graph is given in Figure 9.6

```
#include <stdio.h>
#define R   8314             /* universal gas constant */
float ideal( float v );      /* prototypes */
float vander( float v );
float temp;                  /* GLOBAL variable: not inside any function */
                             /* temperature of CO gas */

int main( void )
{
    /* Local Variables ------------------------------------ */
    float m;                  /* mass of CO gas, in kilograms */
    float vol;                /* tank volume, in cubic meters */
    float vmol;               /* molar specific volume */
    float pres;               /* pressure (to be calculated) */

    printf( "\n Input the temperature of CO gas (degrees K): " );
    scanf( "%g", &temp );
    printf( "\n The mass of the gas (kg) is: " );
    scanf( "%g", &m );
    printf( "\n The tank volume (cubic m) is: " );
    scanf( "%g", &vol );
    vmol = 28.011 * vol / m;        /* molar volume of CO gas */
    pres = ideal( vmol );           /* pressure; ideal gas model */
    printf( "\n The ideal gas at %.3g K has pressure "
            "%.3f kPa \n", temp, pres );
    pres = vander( vmol );       /* pressure; Van der Waal's model */
    printf( "\n Van der Waal's gas has pressure "
            "%.3f kPa \n\n", pres );
    return 0;
}
```

Figure 9.5. Gas models before global elimination.

This is a function call graph for the gas pressure program in Figures 9.5 and 9.7.

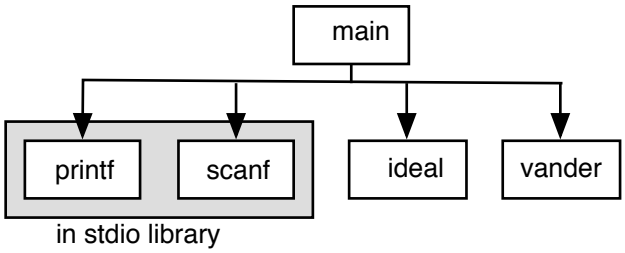


Figure 9.6. A call graph for the CO gas program.

These functions are part of the gas models program in Figure 9.5 and are found at the bottom of the same source code file.

```
/* ------------------------------------------------------------------------
**  Pressure of CO gas in a tank, using the ideal gas equation Pv = RT.
*/
float ideal( float v )
{
    float p;                    /* LOCAL variable DECLARATION */

    p = R * temp / v;           /* pressure in Pascals */
    return p / 1000.0;          /* pressure in kilo Pascals (kPa) */
}
```

```
/* ------------------------------------------------------------------------
**  Pressure of CO gas in a tank, using Van der Waal's equation,
**  P = RT/(v-b) - a/(v*v).
*/
float vander( float v )
{
    float p;                  /* LOCAL declaration, not same p as above */
    const float a = 1.474E+05;                  /* constants for CO gas */
    const float b = .0395;

    p = R * temp / (v - b) - a / (v * v);       /* pressure in Pascals */
    return p / 1000.0;                          /* kPa pressure */
}
```

Figure 9.7. Functions for gas models before global elimination.

- All global names and all the functions defined in a module default to `extern`; that is, they are **external** symbols unless declared otherwise. This means that their names are given to the system's linker and all the modules linked together in a complete program can use these variables.[5]

We say that a locally declared variable or parameter is **visible**, or **accessible**, from the point of its declaration until the block's closing `}`. This means that the statements within the block can use the declared name, but no other statements can (i.e., the scope and visibility of a local variable are the same). A global or external variable is visible everywhere in the program after its definition, *except* where another, locally declared variable bears the same name. If a function has a parameter or local variable named `x` and a statement `x = x+1`, the local `x` will be used no matter how many other objects named `x` are in the program's "universe." Therefore, the visibility of a global variable is that portion of its scope in which it is not masked or hidden by a local variable. This is a very important feature; it is what frees us from concern about conflicts with the names of all the hundreds of other objects in the program.

9.4.2 Global and Local Names

Insofar as possible, all variables should be declared locally in the function that uses them. Information used by two or more functions should be passed via parameters. The use of global variables usually is a bad idea; any variable that can be seen and used by all the functions in a program might be changed erroneously by any part of the program. When global variables are in use, no one part of the program can be fully understood or debugged without considering the entire thing.

While global variables are not encouraged, constants and types usually are declared at the top of a file because they are necessary to coordinate the actions of different parts of a program. Since the values of `const` variables and `#define`d symbols cannot be changed, their global visibility does not foster the same kind of problems as a global variable. Further, declaring constants and types at the top of a file makes them easier to locate and revise, if necessary.

A function prototype can be declared globally or locally in every function that calls it. Each style of organization has its advantages. In this text, we declare most prototypes globally because it is simpler. We illustrate some of these issues using the program in Figure 9.5, which has two subprograms (Figure 9.7) and a variety of local and global declarations (Figure 9.8).

Notes on Figures 9.5, 9.7, and 9.8. Gas models before global elimination. We use a main program, two functions, a global constant, and a global variable to examine the scope and visibility of names in C. Figure 9.8 illustrates the scope of the symbols defined in this program.

First box, Figure 9.5: global declarations and included files .

- The `#include <stdio.h>` means that everything in the file `stdio.h` will be copied into this program at this point. All the objects declared in `stdio.h`, therefore, will have a global scope in this program.
- The constant `R` and the variable `temp` are declared globally. In Figure 9.8, these names are written in the gray box, which represents the global scope. Here, they are visible and can be used by any function

[5] External linkage will be discussed in Chapters 19 and 23.

The diagram shows the scope of the functions, variables, and constants defined in the gas models program from Figures 9.5 and 9.7. Each box represents one scope; the gray box represents the global scope. Local scopes are white boxes; within each, parameters are listed above the dotted line and local variables and constants below it. Symbols defined in the gray area are visible everywhere in the program where they are not masked. Symbols in the white boxes are visible only within the scope that defines them.

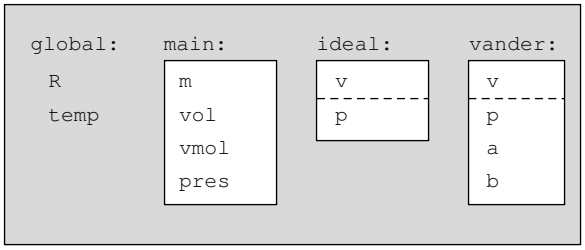


Figure 9.8. Name scoping in gas models program, before global elimination.

in this file; that is, by `main()`, `ideal()`, and `vander()`. The functions `ideal()` and `vander()` also can be called from any part of this file.

- It is considered very bad style to use a global variable such as `temp`. We do so only to demonstrate the meaning of global declarations and show how to eliminate them.
- A global constant such as `R` creates fewer problems than a global variable. It is common for a program to use global constants and is not considered bad style.

Second box, Figure 9.5: declarations for `main()`***.***

- The four variables `m`, `vol`, `vmol`, and `pres` are local to `main()` and therefore visible only within `main()`. In Figure 9.8, the leftmost white rectangle represents the scope created by `main()`. Inside it is a list of `main()`'s local variables.
- The functions `ideal()` and `vander()` cannot access the values in `m`, `vol`, `vmol`, and `pres` because the values are local within `main()`.

Third box, Figure 9.5: code for `main()`***.***

- This code can use the definitions and the global variable and constant defined in the first box as well as the variables defined in the second box.
- This code cannot use the parameters, variables, or constants defined by the two functions in Figure 9.7. If we tried to use `v`, `p`, `a`, or `b` here, it would cause an undefined-symbol error at compile time.
- The two inner boxes contain the calls on the functions `ideal()` and `vander()`. These functions will need to be modified to eliminate the use of the global variable.

Sample output from this program is:

```
Input the temperature of CO gas (degrees K): 28.5

The mass of the gas (kg) is: 1.2

The tank volume (cubic m) is: 3

The ideal gas at 28.5 K has pressure 3.385 kPa
Van der Waal's gas has pressure 3.357 kPa
```

First box, Figure 9.7: code for the function `ideal()`***.***

- Because this function is compiled at the same time as the code in Figure 9.5, it can use any object, such as `R` or `temp`, that is defined (or included) in the first box in Figure 9.5.
- In Figure 9.8, the center rectangle represents the scope created by `ideal()`. Inside it are `ideal()`'s parameter `v` and the local variable `p`, which are visible only within `ideal()` and cannot be used outside this function.

Second box, Figure 9.7: code for the function `vander()`***.***

- This function also can use objects such as `R` and `temp` that are defined (or included) in the first box in Figure 9.5.
- In Figure 9.8, the rightmost rectangle represents the scope created by `vander()`. Inside it are `vander()`'s parameter `v` and the local variables `a`, `b`, and `p` (`a` and `b` actually are constants). These are visible only within `vander()` and cannot be used outside this function.
- Each parameter or local variable declaration creates a new object. Therefore, the `p` defined here is a different variable than the `p` defined in `ideal()`, with its own memory location, even though they have the same name.
- If the local variable `p` had been named `temp`, this function would compile but not work properly because the local variable would mask the global variable. Every reference to `temp` in `vander()` would access the local variable, and the information in the global `temp` would not be accessible within the function. The next section shows how to improve the lines of communication so that these difficulties do not occur.

9.4.3 Eliminating Global Variables

We introduced a programming style in which interaction with the user is done by one function (ofen `main()`) and calculations by another. This separation of work makes a program maximally flexible and easier to modify at a later date. However, since one function reads the input data and another uses it for calculations, the data value must be communicated from the first to the second.

A beginning programmer often will be tempted to use a global variable because it provides one way to solve this communication problem. A global variable is visible to both the data input function and the calculation function, so nothing special has to be done to communicate the value from the first to the second.

This program is a revised and improved version of the gas models program in Figures 9.5 and 9.7; it solves the communication problem by using a parameter instead of a global variable. The functions in Figure 9.10 are part of the revised program and should be in the same source file.

```
#include <stdio.h>
#define R  8314           /* universal gas constant */

float ideal( float v, float temp );
float vander( float v, float temp );

int main( void )
{
    /* Local Variables ------------------------------------ */
    float temp;           /* Temperature of CO gas */
    float m;              /* mass of CO gas, in kilograms */
    float vol;            /* tank volume, in cubic meters */
    float vmol;           /* molar specific volume */
    float pres;           /* pressure (to be calculated) */

    printf( "\n Input the temperature of CO gas (degrees K): " );
    scanf( "%g", &temp );
    printf( "\n The mass of the gas (kg) is: " );
    scanf( "%g", &m );
    printf( "\n The tank volume (cubic m) is: " );
    scanf( "%g", &vol );

    vmol = 28.011 * vol / m;                    /* molar volume of CO gas */
    pres = ideal( vmol, temp );    /* pressure; ideal gas model */
    printf( "\n The ideal gas at %.3g K has pressure "
            "%.3f kPa \n", temp, pres );
    pres = vander( vmol, temp );   /* pressure; Van der Waal's model */
    printf( " Van der Waal's gas has pressure "
            "%.3f kPa \n\n", pres );

    return 0;
}
```

Figure 9.9. Eliminating the global variable.

These functions illustrate how to eliminate a global variable from Figure 9.7. The boxes highlight the changes necessary to replace the global variable by adding a parameter to each function.

```
/* ------------------------------------------------------------------------
**  Pressure of CO gas in a tank, using the ideal gas equation Pv = RT.
*/
float ideal( float v, float temp )
{
    float p;                  /* LOCAL variable DECLARATION */

    p = R * temp / v;         /* pressure in Pascals */
    return p / 1000.0;        /* pressure in kilo Pascals (kPa) */
}

/* ------------------------------------------------------------------------
**  Pressure of CO gas in a tank, using Van der Waal's equation,
**  P = RT/(v-b) - a/(v*v).
*/
float vander( float v, float temp )
{
    float p;                  /* LOCAL declaration, not same p as above */
    const float a = 1.474E+05;
    const float b = .0395;                  /* constants for CO gas */

    p = R * temp / (v - b) - a / (v * v);   /* pressure in Pascals */
    return p / 1000.0;                      /* kPa pressure */
}
```

Figure 9.10. Functions for gas models after global elimination.

In a small program, using global variables might seem easy and communicating through parameters might seem to be a nuisance. However, global variables almost always are a mistake,[6] because they allow unintended interactions between distant parts of the program. They make it hard to follow the flow of data through the process, and they make it harder to modify and extend the program. In a large program, global variables become a debugging nightmare. It is hard to know what parts of the program change them and under what conditions. Therefore, it is important, from the beginning, to avoid global variables and learn to use parameters effectively.

We will use the program in Figures 9.5 and 9.7 to demonstrate the technique for eliminating global variables. The result is shown in Figures 9.9 and 9.10. Parameters are the right way to solve the communication

[6]The program in Figure **??** shows an instance where the use of global variables is acceptable.

problem. They make the sharing of data explicit and they prevent unintended sharing with unrelated functions. In general, global variables should be replaced by parameters. The transformation works for globals used to send information into a function; a variant of this technique[7] is needed if the function also uses the global variables to send information back out.

Notes on Figures 9.9 and 9.10. Eliminating the global variable. We start with the main program in Figure 9.5 and the two functions in Figure 9.7, which communicate through a global variable. To eliminate this variable and replace it by a parameter, we need to change five lines in Figure 9.5 and two lines in Figure 9.7.

First box of Figure 9.9. The prototypes for the functions `ideal()` and `vander()` found in the first box of Figure 9.5 need to be changed to include an additional parameter of the same type as the global variable. In both cases, we add the new parameter second. It is important to be consistent about parameter order when adding parameters, to avoid the problems mentioned earlier.

Second box of Figure 9.9. The declaration of `temp` must be moved out of the global area where it was defined in Figure 9.5 and into `main()`'s local area.

Third and fourth boxes of Figure 9.9. The function calls in these boxes must be modified to include an additional argument, the value of the former global variable, which is now a local variable in `main()`.

First box of Figure 9.10. A new parameter was added to the parameter list in the prototype for `ideal()`. We also must add the parameter to the function header.

Second box of Figure 9.10. We must add the new parameter to the list for `vander()` as well.

9.5 Program Design and Construction

In the gas pressure example, we started with a program and its two functions and analyzed it section by section. This is a good way to understand how a given program works, but it gives little perspective on how a program with functions is developed. In this section, we reverse the process and show how to develop a program and its subprograms from the specification. Section 9.5.1 lists the steps and describes them briefly. Section 9.5.2 explains these steps more fully and applies them to a real problem.

9.5.1 The Process

The DNA: a complete specification. The first step in designing a program is to decide what you want the program to do and specify it as precisely as possible. If there is any doubt about the specifications or any missing information, this must be cleared up before proceeding.

[7] This variation uses pointer parameters, which will be covered in Chapter 11.

Start with the "skin" of `main()`**.** Write the routine portions of `main()`. If you wish to test or run the program on several data sets, write a work loop in the body of main.

Define the skeleton of the work to be done. Start by listing the major phases in processing a single data set. Generally these phases are input, calculation, and output; but one of these phases might not be needed and the calculation phase may have multiple steps. Write the code to perform each phase if it is only one or two lines long. Otherwise, invent a name for a function that will do each task. These statements will be in a `work()` function if you have one, otherwise in `main()`.

The circulatory system: declarations and prototypes. Go back to the top of `main()` and write whatever declarations and prototypes you will need to support your skeleton code. These do not have to be perfect. Write a comment for each one.

Health check: compile. It is much easier to find compiler errors when you compile and check the code a little bit at a time. However, a program will not compile if it calls functions that have not yet been written.

One solution to this is to write a *stub* for each function that has been named but not yet defined. A stub is a function that does nothing except print its own greeting message and return some value of the correct type (or void). The return value does not need to be meaningful; any definition is OK, as long as its parameter types and return value type match the prototype. Compile the program initially as soon as the stubs are written. Then later, after every function is fleshed-out, compile the program again.

An alternative to writing function stubs is to temporarily **amputate** calls on functions that have not yet been written by enclosing them in comment marks. Sometimes entire lines are amputated. At other times, a function call is commented out and replaced by a literal constant. The comments can easily be removed when the function is written, later.

The brains: developing the functions. Tackle the functions one at a time, in any convenient order. For each, write a comment block that describes the purpose of the function and any preconditions. Then go through the same steps as for `main()`: routine parts, skeleton, declarations, and possibly more prototypes and stubs. As you learn more about your functions, you may need to add parameters to the prototypes you have previously written.

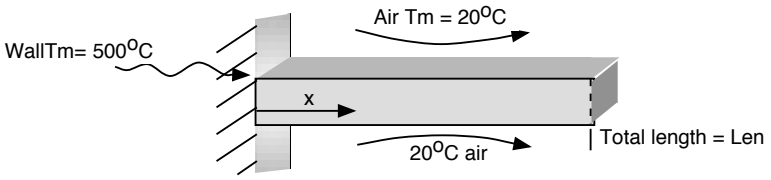
Integration and tesing. Combine and test all of the program parts according to your plan.

9.5.2 Applying the Process: Stepwise Development of a Program

We now apply the design steps from Section 9.5.1 to develop a program for a real problem: calculating the temperature of a cooling fin. This problem arose from a real application: designing the cooling apparatus for a piece of machinery. Figure 9.11 is a diagram of the cooling fin and specifications for a program to analyze its temperature gradient. The problem specification comes directly from the set of engineering principles and formulas in use by the designer.[8]

[8] F. Incropera and D. DeWitt, *Fundamentals of Heat and Mass Transfer*, 4th ed. (New York: Wiley, 1996).

Problem scope: A long, slender cooling fin of rectangular cross section extends out from a wall, as shown. Print a table of temperatures every 0.005 m along the fin.



Formulas: The temperature of the fin at a distance x from the wall is:

$$\text{Temperature}(x) = AirTm + (WallTm - AirTm) \times \frac{\cosh[FinC \times (Len - x)]}{\cosh(FinC \times Length)}$$

Constants:
Temperature of the air, $AirTm = 20°\text{C}$
Temperature of the wall at the base of the fin, $WallTm = 500°\text{C}$
Fin constant, $FinC = 26.5$

Input: Length of the fin, *Len*, in meters, which must be greater than zero.

Output required: Print column headings `Distance from base (m)` and `Temperature (C)`. Beneath the headings print a table of temperatures starting with the temperature at the wall ($x = 0$). Print one line for each increment of 0.005 m up to the length of the fin.

Computational requirements: Print the distance from the wall to three decimal places and the temperature to one.

Test plan: Using the numbers from the diagram, for a wall that is .6 meters long, the temperature at the wall should be $500°C$ and $.01m$from the wall, the temperature should be $398.535°C$.

Figure 9.11. Problem specification and test plan: fin temperature.

A fin is slender if its length is an order of magnitude greater than the height or width of its rectangular cross section. The cooling properties of a fin depend on temperatures of the wall and the air and a fin constant, *FinC*, which is a function of the film coefficient of the air and the thermal conductivity and cross-sectional area of the fin.

Steps in writing `main()`.

- *The skin.* We start by writing the `#include` commands (`stdio`, as usual, and `math` because this is

a numeric application.) and the first and last few lines of `main()`, the greeting message and return statement

```
#include <stdio.h>
#include <math.h>
int main( void )
{
    puts( " Temperature Along a Cooling Fin" );
    /* Program code will go here. */
    return 0;
}
```

- *Multiple data sets.* We will write this program for only one data set, so we do not need a processing loop or a `work()` function. Therefore, the skeleton of the program becomes the body of `main()`.
- *The skeleton.* Next, prepare the **function skeleton**; We start by listing the two major phases in creating a table for a single fin situation. We need two basic steps:

  a. Input and validate a fin length.
  b. Print a temperature table for that length.

  We write the code for step (a) directly because it requires only a few lines of code.

```
do {
     printf( " Please enter length of the fin (> 0.0 m): " );
     scanf( "%g", &Length );
} while (Length <= 0);
```

- *The circulatory system.* We finish step (a) by writing the declaration for `Length`.

```
float Length;       /* Length of the cooling fin. */
```

- Step (b) is more complex, however, so we invent a function to perform the task and name it `print_table()`. We write a first draft of the prototype for `print_table()`, giving it the parameters and return type we think it will need. This information comes from the formula given in the specification: all values are constant except the fin length, so the length is the only parameter needed. We set the return type to `void` because most printing functions are `void`.

```
void print_table( float Length );
```

  Only the prototype is written at this stage; the code itself will be written later. This is just a first guess at the proper prototype: if there are too many parameters or too few, or the types are wrong, that will be corrected later. Now we return our attention to `main()` and write a call on the new function.

```
print_table( Length );
```

  If (unlike this example) the new function returns a value, we must store the result in a variable and we may need to write a declaration for that variable.

This code was written piece by piece on the preceding few pages. The corresponding call graph follows.

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

void print_table( float Length );

int main( void )
{
    float Length;     /* Input:  the length of the cooling fin. */

    puts( " Temperature Along a Cooling Fin" );
    do {
        printf( " Please enter length of the fin (> 0.0 m): " );
        scanf( "%g", &Length );
    } while (Length <= 0);

    print_table( Length );
    return 0;
}
/* Stub:  ------------------------------------------------------------------
** Print a table of temperatures for a fin of given length, in meters.
*/
void print_table( float Length )
{
    printf( " >>> Entering print_table with parameter %g", Length );
}
```

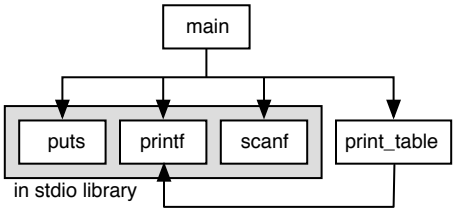


Figure 9.12. First draft of `main()` for the fin program, with a function stub.

- *Health check.* The first draft of the main program is now complete; it is shown in Figure 9.12 with the corresponding call graph. We would like to use the C compiler to check the work so far but we cannot compile the program as it stands because the definition of `print_table()` is missing. So we create a stub for `print_table()`. This consists of an identifying comment, a function header that matches the prototype, and a single output statement to let the programmer track the program's progress. Since this function has a parameter, we use the stub to print its value, giving us confidence that the data is being correctly communicated to the function. We put this stub at the end of the program.

The stub served its purpose when we compiled this file. There were a few typographical errors, but the program is so short that they were easy to find and fix. The corrected code is shown in Figure 9.12. It ran successfully, producing this output:

```
Temperature Along a Cooling Fin
Please enter length of the fin (m): .06
>>> Entering print_table with parameter 0.06
```

From this, we can see that the program is starting and ending properly and receiving its input correctly.

- *The brains.* We now can start work on the function. If a program has two or more functions, we code them one at a time, in any convenient order. For each, we work on the skin, the skeleton, the circulation, and the brains.

  Sometiimes this leads to inventing another function; sometimes it means writing statements that compute the formulas given in the specification.

Steps in writing `print_table()`.

- *The DNA and skin.* We have written a prototype and a function stub for `print_table()`. Now we need to think carefully about the function itself. One part of that process is to write a set of *preconditions* for the function, that is, things that must be true when the function is called. This function has one parameter, the length of the fin, which must (obviously) be a positive number, so we add a line to the comment block to document this precondition. This precondition announces that it is the job of the caller `main()` to validate the input – it will not be validated here. We now have this skeleton:

```
/* Stub:  ----------------------------------------------------------------
** Print a table of temperatures for a fin of given length, in meters.
** The length must be greater than 0.
*/
void print_table( float Length )
{
    printf( " >>> Entering print_table with parameter %g", Length );
}
```

- *The skeleton.*
  To print a table, we must

  1. print table headings,
  2. print several lines of detail, and
  3. print table footings.

  Steps (1) and (3) are simple enough to write directly. We do that and leave space to insert step (2) later.

```
printf( "\n Distance from base (m)    Temperature (C) \n" );
printf( " ----------------------    --------------- \n" );
        /* STEP 2 WILL GO HERE */
printf( " ----------------------    --------------- \n" );
```

- *The circulatory system.*
  We need a constant in the `print_table()` function: the step size for the loop. The specification says that the temperature should be calculated every 0.005 meters, but it always is unwise to bury constants like that in the code. We define the step size as a local constant so that it will be easy to modify in the future. It is obvious that we also need at least one variable, `distance`, to hold the current distance from the wall and another, `temp`, to hold the result of the temperature calculation:

  ```
  float dist;               /* Distance from wall. */
  float temp;               /* Temperature at that distance. */
  const float step = .005;  /* Step size for loop. */
  ```

  For step 2, we need a loop that will produce one line of output on each iteration. For each line, we must compute and print the temperature at the current distance, x, from the wall. We invent a function named `compute_temp()` to compute the temperature. It needs a parameter, x, which we declare as type `float`, like all the other variables in this program. The function result is a temperature, so that will also be type `float`. The prototype becomes:

  ```
  float compute_temp( float x );
  ```

  The revised function call graph is shown in Figure 9.13.

- *The brains.*
  Printing the detailed lines of the table requires a loop that starts at distance 0.0 from the wall and increases to the length of the fin in increments of the defined step size. Since the loop variable, `dist`, is not an integer, we need an epsilon value for the floating comparison that ends the loop. This epsilon must be smaller than the step size; we arbitrarily set it to half the step size:

  ```
  float epsilon = step/2.0;       /* Tolerance */
  ```

  We are ready to code the loop. We start with the loop skeleton and defer the computation and printout:

  ```
  for (dist = 0.0; dist < Length+epsilon; dist += step) {/*Defer*/}
  ```

  Now we approach the loop body. The loop prints the lines of the table one at a time. For each one, we must compute the current distance, x, from the wall and the temperature at distance x. The distance computation is handled by the loop control; the remaining tasks are done by `compute_temp()` and `printf()`:

  ```
  temp = compute_temp( dist );
  printf( "%12.3f %24.1f \n", dist, temp );
  ```

```
/* -------------------------------------------------------------
** Print a table of temperatures for a fin "Length" meters long.
** Length must be greater than 0.
*/
void print_table( float Length )
{
    float dist;                   /* Distance from wall. */
    float temp;                   /* Temperature at that distance. */
    const float step = .005;      /* Step size for loop. */
    float epsilon = step/2.0;     /* Tolerance */

    printf( "\n Distance from base (m)    Temperature (C) \n"
    printf( " ----------------------    --------------- \n" );
    for (dist = 0.0; dist < Length + epsilon; dist += step) {
        temp = 1.1;      /* compute_temp( dist ); */
        printf( "%12.3f %24.1f \n", dist, temp );
    }
    printf( " ----------------------    --------------- \n" );
}
```

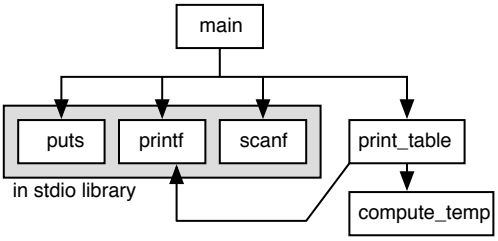


Figure 9.13. Fin program: First draft of `print_table()` function.

In the format, we use `%f` conversion specifiers because we want a neat table with the same number of decimal places printed for every line (3 for `dist` and 1 for `temp`, written with `%12.3f` and `%24.1f`, respectively). We supply a field width so that the output will appear in neat columns. To find the correct field width, we either count the letters in the headings or guess; a guess that is too big or too small can be adjusted after we see the output. We now have completed the first draft of the code for `print_table()`; it is shown in Figure 9.13.

- *Health check.*
  While coding the brains of a function, a programmer sometimes discovers that the function needs more information than its parameters supply. If that information is not supplied by global constants, the function header, prototype, and call(s) must be edited to supply the added data.

In the process of writing `print_table()`, we did not need to change the parameter list, so the call on `print_table()` inside `main()` still is correct and we do not need to modify `main()` at this time. (However, we might need to do this later.)

We have now completed a main program and one function with a tentative call on a second function that has not been written. It is time to compile again. We cannot compile the program as it stands because of the call on `compute_temp()`. Although we could write another function stub, in this case we choose to temporarily "amputate" the call on `compute_temp()` by enclosing it in a comment and setting the variable `temp` to an arbitrary (but recognizable) constant value:

```
temp = 1.1;  /* compute_temp( dist ); */
```

Now we compile the program, fix any compilation errors, and run it. Our program ran successfully, producing the output that follows (only the first and last few lines are shown).

```
Temperature Along a Cooling Fin
Please enter length of the fin (m): .06

Distance from base (m)    Temperature (C)
----------------------    ---------------
      0.000                      1.1
      0.005                      1.1
      0.010                      1.1
       ...                       ...
      0.050                      1.1
      0.055                      1.1
      0.060                      1.1
```

Looking at this table, we see that the number of rows printed is correct and the numbers are adequately centered under the headings. The numbers in the first column are correct and the numbers in the second column are the constant value we used when we amputated the call on `compute_temp()`.

Steps in writing `compute_temp()`.

- *The DNA and the skin.*
  The `compute_temp()` function must calculate the temperature of the fin at a given distance from the wall. Tentatively, we have given it one parameter, a `float` named x, which will range between 0 meters and the length of the fin. The function must return a `float` value to `print_table()`. We write a comment block and the shell of the function:

```
/* ------------------------------------------------
** Compute and return the temperature at distance x from the wall.
** x must be between 0.0 meters and the length of the fin.
*/
float compute_temp( float x ){/* Code goes here. */}
```

- *The skeleton and the brains.*
  We compute the temperature using the formula in the specifications (see Figure 9.11) and return the answer. This is a simple task that need not be broken down further. Here is the resulting statement:

```
return AirTm + (WallTm-AirTm) * cosh( FinCnow*(Length-x) ) / cosh( FinC*Length );
```

Looking at this formula, we see that it involves several variables (wall temperature, air temperature, fin constant, and length of the fin), not just the distance x from the wall. However, the first three values are given as constant numbers in the problem specification. We could write constants for these quantities in our formula, but the program would be much more useful if these values could be varied. Therefore, we choose to define all three constants in `main()` and add them to the parameter lists of both `print_table()` and `compute_temp()`. By placing all these constants in `main()`, we make them easy to find and modify. It also would be easy to change them into input variables if that were desired. Last, the fin length is a missing parameter; it is read in `main()` and we must pass it from `main()` through `print_table()` to this function. Similarly, we must pass the constants from `main()` to `compute_temp()`. To do so, we need to change the prototypes and headers of both functions and correct two function calls.

- *Circulation.*
  We now have three constants and one input variable that must be passed from `main()` to `compute_temp()` as parameters. Although the order of the new parameters is not crucial, we want an order that makes sense and can be remembered, so we put all the properties of the fin together. Our new prototypes are

  ```
  void print_table( float Length, float FinC, float WallTm, float AirTm );
  float compute_temp( float x, float Length, float FinC, float WallTm, float AirTm )
  ```

  Next, we add three constant definitions to `main()` and change the call on `print_table()` to use them:

  ```
  const float FinC = 26.5;    /* Fin constant. */
  const float WallTm = 500.0; /* Wall temperature, C */
  const float AirTm = 20.0;   /* Air temperature, C */
  print_table( Length, FinC, WallTm, AirTm );
  ```

  Last, the function call in `print_table()` must also be changed:

  ```
  temp = compute_temp( dist, Length, FinC, WallTm, AirTm );
  ```

- *Health check.* The completed program is shown in Figure 9.14. The first and last few lines of the output are shown below. The final development step is to compare these results to the ones in the test plan, to verify that we are fully finished.

```
Temperature Along a Cooling Fin
Please enter length of the fin (m): .06

Distance from base (m)    Temperature (C)
----------------------    ---------------
      0.000                     500.0
      0.005                     445.5
      0.010                     398.5
       ....                     .....
      0.055                     209.6
      0.060                     208.0
----------------------    ---------------
```

The problem specifications are given in Figure 9.11.

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

void print_table ( float Length, float FinC, float WallTm, float AirTm );
float compute_temp( float x, float Length, float FinC, float WallTm, float AirTm );

int main( void )
{
    float Length;                   /* Length of the cooling fin, in meters. */
    const float FinC = 26.5;        /* Fin constant. */
    const float WallTm = 500.0;     /* Wall temperature, C. */
    const float AirTm = 20.0;       /* Air temperature, C. */

    puts( " Temperature Along a Cooling Fin" );
    do {
        printf( " Please enter length of the fin (> 0.0 m):  " );
        scanf( "%g", &Length );
    } while (Length <= 0);
    print_table( Length, FinC, WallTm, AirTm );
    return 0;
}
/* ---------------------------------------------------------------
** Print a table of temperatures for a fin that is "Length" meters long.
** Length must be greater than 0.
*/
void
print_table( float Length, float FinC, float WallTm, float AirTm )
{
    float dist;                     /* Distance from wall. */
    float temp;                     /* Temperature at x. */
    const float step = .005;        /* Step size for loop. */
    const float epsilon = step/2.0; /* Tolerance. */

    printf( "\n Distance from base (m)    Temperature (C) \n" );
    printf( " ----------------------    --------------- \n" );
    for (dist = 0.0; dist < Length + epsilon; dist += step) {
        temp = compute_temp( dist, Length, FinC, WallTm, AirTm );
        printf( "%12.3f %24.1f \n", dist, temp);
    }
    printf( " ----------------------    --------------- \n" );
}

/* ---------------------------------------------------------------
** Compute the temperature at distance x from wall; 0<=x<=Length
*/
float
compute_temp( float x, float Length, float FinC, float WallTm, float AirTm )
{
    return AirTm + (WallTm - AirTm) * cosh( FinC*(Length-x) ) / cosh( FinC*N );
}
```

Figure 9.14. Temperature along a cooling fin.

9.6 What You Should Remember

9.6.1 Major Concepts

Modular design. Large programs are organized into modules, which contain related sets of functions and constants. It is the compiler's job to properly compile and link together the files containing the modules. The organization of a module follows a stylistic pattern. Modular design is a skill worth learning if you intend to write programs of any substantial size.

Parameter passing. Most parameters are passed by value; that is, the value of the argument is copied into the parameter variable. This isolates the subprogram from the caller, making it impossible for the subprogram to change the values of the caller's variables. In Chapter 11, another parameter-passing mechanism (call by value/address) will be introduced that can support two-way communication between caller and subprogram.

Parameter and return value type conversion. The type of a function argument in a call should match the type of the corresponding formal parameter in the prototype. If they are not identical, the argument will be coerced (automatically converted) to the parameter's type, if that is possible. Any numeric type (including `char`) can be converted to any other numeric type. Similarly, the value returned by a function will be converted to the declared return type. Normally, this type coercion causes no trouble. However, converting from a longer representation to a shorter one can cause overflow or loss of precision and converting from `char` to anything except `int` usually is wrong.

Parameter names. The name of a parameter is arbitrary. It identifies the parameter value within the function and therefore should be meaningful, but it has no connection to anything outside the function, even if other objects have the same name. The order in which arguments are given in the call, not their names, determines which parameter receives each argument value.

Parameter order. The order of parameters for a function is arbitrary. However, related parameters should be grouped together, and when several functions are defined with similar parameters, the order should be consistent. The number and order of arguments are not arbitrary. Parameters and arguments are paired according to the order in which they are written (not by name or type).

Call graphs. A function call graph is a diagram that shows the caller-subprogram relationships of all the functions in a program. More sophisticated forms may include descriptions of the information being sent into and out of the subprogram.

Scope and visibility. The scope of an object is the portion of a program in which it exists. The visibility of an object is the portion of its scope in which it can be accessed. An object can have external, global, or local scope, meaning that it is available to the entire program, restricted to a single module, or restricted to a single function, respectively.

Stub testing. When a program is long and has many functions, stub testing often is the best way to construct and debug it. In this technique, the main program is written first, along with a stub for each function it calls. The stub is a function header, some comments, and only enough code to print the parameter values. If not a `void` function, it also must return some fixed and arbitrary answer. After the main function compiles and runs correctly by calling the stubs, the stubs are filled in, one at a time, with real code. This code, in turn, may require the construction of new stubs. After one or a few stubs have been fleshed out, the code is compiled and tested again. This is repeated until all stubs have been replaced by complete functions. This technique is important because it forces the programmer to work in a structured way; it ensures that all parts of the program are kept consistent with each other as they are developed. Also, compiler errors are easier to find and fix because there is never a large amount of new code being compiled for the first time.

9.6.2 Programming Style

Monolithic vs. modular. If you cannot look at a function on your computer screen and understand what it is doing, it is too long, too complex, or both. Keep function definitions short enough to see the beginning and the end at the same time. Generally, it is good to keep code for user interaction and code for mathematical computation in separate functions. Modular development also aids the compiling and debugging process.

Placement of prototypes in the file. The easiest way to be sure that the compiler uses the correct prototype to translate every function call is to write all the `#include` commands and all your prototypes at the top of each code module. Although other arrangements may be legal, according to C's rules, putting the prototypes at the top is the easiest way to avoid errors caused by misplaced prototypes, missing arguments, and incorrect argument order in function calls.

Global vs. local. Define variables locally, not globally, wherever possible. This tactic makes logical errors easier to locate and fix because it limits unintentional interaction between parts of a program. In general, parameters should be used for interfunction communication. Global definitions should be used only for new type definitions and constants shared by several parts of a program.

Function documentation. Every function should start with a highly visible comment line, such as a row of dashes. This line is very useful during debugging because it helps you find the functions quickly. You should be able to give a succinct description of the purpose of each function. This description should start on the second line of the comment at the top of the function definition. This comment also must make clear the purpose of each parameter and include a discussion of any preconditions.

Parameter and argument consistency. If more than one function uses the same set of parameters, reduce confusion by being consistent in their order and naming.

Function layout. The parts of a function definition can be arranged in many ways. The layout recommended here is the easiest to read and extends best to advanced programming in C++.

- Every function definition should start with a comment block, as described previously. The first line of code should be the return type and a comment, if needed, that describes the meaning of the return value (nothing else).
- The second line of code should start with the name of the function, followed by a left parenthesis. If all parameters will fit on one line, they should come next, ending with a right parenthesis. Otherwise, put each parameter on a separate line, with a comment. Put the closing right parenthesis on a line by itself, aligned directly under the matching left parenthesis.
- On the next line, write the left curly bracket in the first column.
- Next come the declarations with their comments; indent them two to four columns. Then leave a blank line and write the function body, indented similarly. Increase the indentation for each conditional or loop statement.
- Write the right curly bracket that ends the function in the leftmost column on a separate line.

9.6.3 Sticky Points and Common Errors

Parameter order. The arguments in a function call must be written in the same order as the matching parameters in the definition. If the order is scrambled, the code often will compile and run but produce incorrect results. Using an incorrect number of arguments can lead to confusing errors at compile time.

The declaration must precede the call. Either a function prototype or the complete function definition must precede all calls on the function. If this is not done, the compiler will construct a prototype automatically, which often will be wrong. This normally results in an error comment about an *illegal function redefinition* when the function definition is translated.

Call vs. prototype vs. definition. Beginning programmers often confuse the syntax of a function call with the syntax of the corresponding prototype and header. These have parallel but different forms. The function call supplies a list of argument values; although each value has a type, the type names are not written in the call. In contrast, the function prototype and header do not know what values eventually will be supplied by future calls, so they list the types of the parameters. In addition, the function header must give parameter names, so that the code can refer to the parameters and use them. Last, a semicolon is found at the end of a prototype but not at the end of a function header.

A global vs. local mix-up. Having global variables leads to trouble for a variety of reasons. One scenario is the following: A local variable declaration is omitted, and it happens to have the same name as a global variable. The compiler cannot detect the omission and will use the global variable. Any assignments to this variable will change the global value, causing unexpected side effects in other parts of the program. Such errors are hard to track down because they are not in the part of the program that seems to be wrong and produces incorrect results.

9.6.4 New and Revisited Vocabulary

A large number of terms relating to functions and function calls have been introduced in this chapter. This list is provided as a review of the new concepts covered.

main program	function call	return type
subprogram	calling sequence	return value
function	caller	scope
module	call graph	visibility
library	argument	accessibility
header file	call by value	local
prototype	type coercion	global
function definition	arithmetic types	stack frame
interface	call by address	external
formal parameter	address argument	preconditions
parameter list	entry point	function skeleton
parameter order	return address	function stub
type matching	`return` statement	stub testing
type conflict	missing prototype	testing by amputation

9.7 Exercises

9.7.1 Self-Test Exercises

1. Local and Global. Look on the web site at the program for Newton's method. Fill in the following chart listing the symbols *defined* (not used) in each program scope. List the global symbols on the first line and add one line below that for each function in the program; remember that every function definition creates a new scope. The line for `main()` has been started for you.

Scope	Parameters	Variables	Constants
`global`	—		
`main()`	—		

2. Call graph. The main program, which follows, calls two of the functions declared at the top. Which standard libraries would need to be included to compile this program? Draw a function call graph for this program.

```
#define MAX 7
void pattern( int p );
void stars( int n );
void space( int m, int n );
```

```
int odd( int n ) { return n % 2; }

int main( void ) /*-------------------------------------------------*/
{
    int k;
    for (k = 0; k < MAX; ++k) {
        if (k < 2 || k >= MAX-2) stars( MAX );
        else pattern( k );
        printf( "\n" );
    }
}

void stars( int n ) /*-------------------------------------------------*/
{
    int k;
    for (k = 0; k < n; ++k) printf( "*" );
}

void space( int m, int n ) /*-------------------------------------------*/
{
    int k;
    for (k = 0; k <= n; k += 2) printf( " %i", m );
}

void pattern( int p ) /*-------------------------------------------------*/
{
    stars( 1 );
    space( p, MAX-4 );
    if (odd(MAX)==1) printf( " " );
    stars( 1 );
}
```

3. Tracing calls. Trace the execution of the code in problem 2. Make a list of the function calls, one per line, listing the name, arguments, and return value for each. Show the program's output as well.
4. Local names. List all the local symbols defined in the main program in Figure 9.4. List the parameters and local variables in the function named `n_marks()`.
5. Visibility. List all the variable names used in the function named `cyl_vol()` in Figure 5.24. For each, say whether it is a parameter or a local variable.
6. Control flow. Draw a flow diagram for the fin temperature program in Figure 9.14.
7. Prototypes and calls. Given the prototypes and declarations that follow, say whether each of the lettered function calls is legal and meaningful. If the call would cause a type conversion of either an argument or the return value, say so. If the call has an error, fix it.

```
void   squawk( int );
int    triple( int );
double power( double, int );
```

```
int j, k;
float f;
double x, y;
```

(a) `squawk( 3 );`
(b) `squawk( f );`
(c) `triple( 3 );`
(d) `f = triple( k );`
(e) `j = squawk( k );`
(f) `y = power( 3, x );`
(g) `y = power( x, 3 );`
(h) `x = power( double y, int k );`
(i) `y = power( triple( k ), x );`
(j) `printf( "%i %i", k, triple( k ) );`

9.7.2 Using Pencil and Paper

1. Control flow. Draw a flow diagram for the Newton's method program on the text web site.
2. Call graph. Draw a call graph for the `n_marks` program in Figure 9.4.
3. Prototypes and calls. Given the prototypes and declarations that follow, say whether each of the lettered function calls is legal and meaningful. If the call would cause a type conversion of either an argument or a return value, say so. If the call has an error, fix it.

```
double rand_dub( void );
int half( double );
int series( int, int, double );
```

```
int j, k;
float f;
double x, y;
```

(a) `half( 5 );`
(b) `rand_dub( y );`
(c) `x = rand_dub();`
(d) `j = half();`
(e) `f = half( x );`
(f) `j = series( x, 5 );`
(g) `j = series( 5, (int)x, y );`
(h) `y = series( j, k, rand_dub() );`
(i) `printf( "%g %g", x, half( x ) );`
(j) `printf( "%i %g", k, rand_dub( k ) );`

4. Call graph. The main program that follows calls the three functions defined at the top. Which standard libraries would need to be included to compile this program? Draw a function call graph for this program.

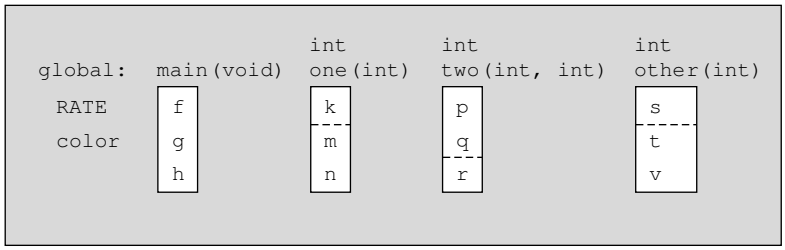
```
double f( double x ) { return x / 2.0; }
double g( double x ) { return 1.0 + x; }
double h( double x ) { return x * 3.0; }

int main( void )
{
    double x = 1;
    double sum = 0.0;

    while (sum < 100) {
        x = h( x );
        printf( " %6.2f \t", x );
        sum += x;
        if (fmod( x, 2.0 ) == 0) x = f( x );
        else x = g( x );
        printf( " %6.2f \n", x );
    }
    printf( " ------\n %6.2f \n", sum );
}
```

5. Tracing calls. Trace the execution of the program in exercise 4, above. Make a list of the function calls, one per line, listing the name, arguments, and return value for each. Show the program's output as well.

6. Local and Global. Look at the program for fin temperatures in Figure 9.14. Fill in the following table, listing the symbols that are *defined* (not used) in each program scope. List the global symbols on the first line and add one line below that for each function in the program (the line for `main()` has been started for you).

Scope	Parameters	Variables	Constants
`global`	—		
`main()`	—		

7. Visibility. The diagram that follows depicts a main program with three functions: `one()`, `two()`, and `other()`. Within the box for each function, parameters are shown above the dashed line, local variables below it. All functions return an `int` result. All variables and parameters are type `int`. The global constant, `RATE`, and global variable, `color`, also are type `int`.



For each function call shown that follows, say whether it is legal or illegal. Fix any illegal statements.

(a) In `main()`: `f = one( RATE );`.
(b) In `main()`, after calling `one()`: `f = two( g, k );`.
(c) In `one()`: `n = two( m );`.
(d) In `one()`: `n = two( color, m );`.
(e) In `one()`: `n = other( k );`.
(f) In `two()`, after being called from `one()`: `r = other( k );`.

9.7.3 Using the Computer

1. A function.

   Write a function to compute the formula

   $$f(x) = (3x + 1)^{\frac{1}{2}}$$

   Write a main program that will sum $f(x)$ for the values $x = 0$ to $x = 1000$ in steps of 50. Print out the value of $f(x)$ and the current sum at every step in a nice neat table.

2. Tables.

   Write a program that contains two function definitions:

   $$f1(n, x) = e^{\sqrt{nx}} \times sin(nx)$$
   $$f2(n, x) = e^{\sqrt{nx}} \times cos(nx)$$

   where n is an integer and x is a `double`. In the main program, input a value for x and restrict it to the range $0.1 \leq x \leq 2.5$. Print a neat table showing the values of $f1(n, x)$ and $f2(n, x)$ as n goes from 0 to 30. Print column headings above the first line. Show all numbers to three decimal places.

3. Bubbles.

   Modify your program from computer exercise 3 in Chapter 7; change the `bubble()` function so that it takes both σ and r as parameters. Then change the `main()` function to ask the user to enter a value for σ as well. Define an error function that prints a message "Input out of range." Use it to screen out values of σ less than 0.001 or greater than 0.003 lb/ft and values of r less than 0.0002 or greater than 0.015 ft. Call this function from `main()`.

4. An arithmetic series.

   Each term of an arithmetic series is a constant amount greater than the term before it. Suppose the first term in a series is a and the difference between two adjacent terms is d. Then the kth term is $t_k = a + (k-1)d$. Write a function `term()` with three parameters, a, d, and k, that will return the kth term of the series defined by a and d. Use type `long int` for all variables. Write a program that prompts the user for a and d, then prints the first 100 terms of the series, with 5 terms on each line of output, arranged neatly in columns. Quit early if integer overflow occurs.

5. A geometric series.

   A *geometric progression* is a series of numbers in which each term is a constant times the preceding term. The constant, R, is called the *common ratio*. If the first term in the series is a, then succeeding terms will be aR, aR^2, aR^3, ... The kth term of the series will be $t_k = aR^{k-1}$. Write a function `term()` with three parameters (a, R, and k) that will return the kth term of the series defined by a and R. Use type `double` for all variables, since the terms grow large rapidly when R is greater than 1. Write a main program that prompts the user for a and R, then prints the terms of the series they define until either 50 terms have been printed, 5 terms per line, or floating-point overflow or underflow occurs.

6. Torque.

   Given a solid body, the net torque T (Nm) about its axis of rotation is related to the moment of inertia I and the angular acceleration acc (rad/s^2) according to the formula for the conservation of angular momentum:

   $$T = I \cdot acc$$

   The moment of inertia depends on the shape of the body; for a disk with radius r, it is

   $$I = 0.5mr^2$$

   (a) Write a function named `moment()` to calculate and return the moment of inertia of a solid disk. It should take two parameters: the radius and mass of the disk.

   (b) Write a `work()` function that will prompt the user to enter the radius, mass, and angular acceleration of a disk. Limit the inputs to be within these ranges:

   $$0.093 \leq r \leq 0.207$$
   $$0.088 < m \leq 11$$

   Compute and print the torque of the disk, calling `moment()` to compute I first.

   (c) Write a main program that will allow the user to compute several torques.

7. An AC circuit.

   An AC circuit that you have designed is operating at a voltage V (rms volts) alternating at a frequency f (cyc/s). It is constructed of a capacitor C (farads), inductor L (henrys), and a resistor R (ohms) in

series. Some important properties of this circuit are

Impedance:	$Z = \left[R^2 + \left(2\pi fL - \frac{1}{2\pi fC}\right)^2\right]^{0.5}$	(ohms)
Current:	$I = \frac{V}{Z}$	(rms amps)
Power used:	Power $= \frac{V \times I \times R}{Z}$	(watts)

Write a function to compute an answer for each formula. Assume that f is a constant 120 cyc/s and $C = 0.00000001$ farads. Then write a main program that will input values for V, L, and R and output the impedance, current, and power used. Validate the inputs and ensure that they are within the following ranges:

$$60 \leq V \leq 200$$
$$0.1 \leq L \leq 10$$
$$100 \leq R \leq 1000$$

8. Ice cream cones.
Suppose you are a professional party planner. Given the number of guests expected, you must plan a menu and deliver enough food to serve the crowd. In this problem, you will write a program to calculate how many packages of ice cream must be bought to fill one ice cream cone for each guest. The guest will select the size of the cone (diameter, height). Your suppliers sell ice cream in containers of various sizes and shapes.

   Write a function named `cone()` that will prompt for and read the diameter, d, and the height, h, of the cone to be used, then calculate and return the volume of ice cream needed to fill the cone. Assume that the cone part will be filled entirely and there will be a hemisphere of ice cream on top. The formulas are:

$$\text{Volume of cone} = \frac{\pi \times d^2 \times h}{12}$$

$$\text{Volume of hemisphere} = \frac{\pi \times d^3}{12}$$

   Ice cream comes in cartons that are either the shape of a barrel or a box. Write a function named `carton()` that will prompt for an alphabetic code, R for "barrel" or X for "box". Use a switch statement to execute the appropriate input and computation instructions, then return the volume of the selected carton. For a barrel, input the diameter and height; for a box, input the length, width, and height. Use one of these formulas:

$$\text{Volume of barrel} = \frac{\pi \times diameter^2 \times height}{4}$$

$$\text{Volume of box} = length \times width \times height$$

   In your main program, prompt for and input the number of guests, then call your `cone()` function and your `carton()` function. Finally, compute and display the number of cartons of ice cream you must buy to fill all those cones. Use the `ceil()` function to round up to a whole number of cartons.

9. Wedding cake.
   Another common party food is wedding cake. Given the number of guests expected at the wedding, write a program to calculate how many layers your cake must have to serve everyone, and how much that cake will cost.

   The cake will be square and have two or more tiers. The top tier will be 6" wide and will not be eaten at the wedding reception. Each other tier will be 2" thick and 4" wider than the tier above it. Guests will be served from layers 2, 3, 4...; each serving will be 2" square.

   Write the following one-line functions:

   - `width()`: calculate the width of a tier given the layer number (layer 1=6", layer 2 = (6+4)", layer 3 = (6+4+4)", etc.).
   - `servings()`: calculate the number of servings a tier will provide, given the layer number. Call `width()`.
   - `price()`: the top tier and decorations cost $40.00; the other tiers cost $1.00 per serving. (This will be more than $1.00 per guest because part of the bottom tier will be left over.)

   Write a function named `layers()` that will calculate how many tiers are needed. Hint: start with 1 tier, which serves 0 people. Use a loop to call `servings()` and add tiers until you have accumulated enough portions to serve the party. In general, you will end up with more than enough servings, since part of the last tier will be left over.

   In your main program, prompt for and read the number of guests, then call the `layers()` and `price()` functions to calculate the size and cost of the cake. Print out these answers.

10. Scheduled time of arrival.
    Airline travelers often want to know what time a flight will arrive at its destination in the local time of the destination. This can be calculated given the following data:

    - The scheduled takeoff time, in hours and minutes on a 24-hour clock. (Valid hours are 0...23, valid minutes are 0...59)
    - The scheduled duration of the flight, in hours and minutes.
    - The number of time zone boundaries the flight will cross. This number should be negative if traveling from East to West, positive if going West to East.
    - Whether the international date line will be crossed. This number should be +1 if it is crossed traveling from East to West, −1 if crossed while going West to East, and 0 if it is not crossed. Legal values are in the range -23...23.

    Using top-down design, write a program with functions that will make this calculation for a series of flights. For each flight, your program must input these data values from the user and print the scheduled time at which the flight should arrive at its destination. This time is calculated as follows:

    - Starting with the takeoff time, add or subtract an hour for each time-zone change.
    - Then add the duration of the flight to this time.
    - Finally, adjust the time by adding or subtracting a day if the flight crossed the international date line.
    - Use integer division and the modulus operator to convert minutes to hours + minutes, and hours to days + hours.

    Print the local time of arrival using a 24-hour clock. Also print `-1 day` if the flight will land the day before it took off or `+1 day` if it will land the day after it took off (both are possible).

    Suggestion: Write a function that will input, validate, and return one integer. It should have two integer parameters: the minimum and maximum acceptable values for the input.