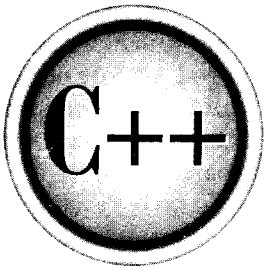
The Complete Reference



Chapter 6

Functions

Functions are the building blocks of C and C++ and the place where all program activity occurs. This chapter examines their C-like features, including passing arguments, returning values, prototypes, and recursion. Part Two discusses the C++-specific features of functions, such as function overloading and reference parameters.

The General Form of a Function

The general form of a function is

```
ret-type function-name(parameter list)
{
  body of the function
}
```

The *ret-type* specifies the type of data that the function returns. A function may return any type of data except an array. The *parameter list* is a comma-separated list of variable names and their associated types that receive the values of the arguments when the function is called. A function may be without parameters, in which case the parameter list is empty. However, even if there are no parameters, the parentheses are still required.

In variable declarations, you can declare many variables to be of a common type by using a comma-separated list of variable names. In contrast, all function parameters must be declared individually, each including both the type and name. That is, the parameter declaration list for a function takes this general form:

```
f(type varname1, type varname2, . . . , type varnameN)
```

For example, here are correct and incorrect function parameter declarations:

```
f(int i, int k, int j) /* correct */
f(int i, k, float j)   /* incorrect *
```

Scope Rules of Functions

The *scope rules* of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data.

Each function is a discrete block of code. A function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function. (For instance, you cannot use **goto** to jump into the middle of another function.) The code that constitutes the body of a function is hidden from the rest of the program and, unless it uses global variables or data, it can neither affect nor be affected

by other parts of the program. Stated another way, the code and data that are defined within one function cannot interact with the code or data defined in another function because the two functions have a different scope.

Variables that are defined within a function are called *local* variables. A local variable comes into existence when the function is entered and is destroyed upon exit. That is, local variables cannot hold their value between function calls. The only exception to this rule is when the variable is declared with the **static** storage class specifier. This causes the compiler to treat the variable as if it were a global variable for storage purposes, but limits its scope to within the function. (Chapter 2 covers global and local variables in depth.)

In C (and C++) you cannot define a function within a function. This is why neither C nor C++ are technically block-structured languages.

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the *formal parameters* of the function. They behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. As shown in the following function, the parameter declarations occur after the function name:

```
/* Return 1 if c is part of string s; 0 otherwise. */
int is_in(char *s,  char c)
{
  while(*s)
    if(*s==c) return 1;
    else s++;
  return 0;
}
```

The function **is_in( )** has two parameters: **s** and **c**. This function returns 1 if the character **c** is part of the string **s**; otherwise, it returns 0.

As with local variables, you may make assignments to a function's formal parameters or use them in an expression. Even though these variables perform the special task of receiving the value of the arguments passed to the function, you can use them as you do any other local variable.

Call by Value, Call by Reference

In a computer language, there are two ways that arguments can be passed to a subroutine. The first is known as *call by value*. This method copies the *value of* an

argument into the formal parameter of the subroutine. In this case, changes made to the parameter have no effect on the argument.

Call by reference is the second way of passing arguments to a subroutine. In this method, the *address* of an argument is copied into the parameter. Inside the subroutine, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C/C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the following program:

```
#include <stdio.h>

int sqr(int x);

int main(void)
{
  int t=10;

  printf("%d %d", sqr(t), t);

  return 0;
}

int sqr(int x)
{
  x = x*x;
  return(x);
}
```

In this example, the value of the argument to **sqr( )**, 10, is copied into the parameter **x**. When the assignment **x = x*x** takes place, only the local variable **x** is modified. The variable **t**, used to call **sqr( )**, still has the value 10. Hence, the output is **100 10**.

Remember that it is a copy of the value of the argument that is passed into the function. What occurs inside the function has no effect on the variable used in the call.

Creating a Call by Reference

Even though C/C++ uses call by value for passing parameters, you can create a call by reference by passing a pointer to an argument, instead of the argument itself. Since the address of the argument is passed to the function, code within the function can change the value of the argument outside the function.

Pointers are passed to functions just like any other value. Of course, you need to declare the parameters as pointer types. For example, the function **swap( )**,

which exchanges the values of the two integer variables pointed to by its arguments, shows how.

```
void swap(int *x, int *y)
{
  int temp;

  temp = *x;  /* save the value at address x */
  *x = *y;    /* put y into x */
  *y = temp;  /* put x into y */
}
```

swap( ) is able to exchange the values of the two variables pointed to by **x** and **y** because their addresses (not their values) are passed. Thus, within the function, the contents of the variables can be accessed using standard pointer operations, and the contents of the variables used to call the function are swapped.

Remember that **swap( )** (or any other function that uses pointer parameters) must be called with the *addresses of the arguments*. The following fragment shows the correct way to call **swap( )**:

```
void swap(int *x, int *y);

int main(void)
{
  int i, j;

  i = 10;
  j = 20;
  printf("%d %d", i, j);
  swap(&i, &j); /* pass the addresses of i and j */
  printf("%d %d", i, j);
  return 0;
}
```

In this example, the variable **i** is assigned the value 10 and **j** is assigned the value 20. Then **swap( )** is called with the addresses of **i** and **j**. (The unary operator **&** is used to produce the address of the variables.) Therefore, the addresses of **i** and **j**, not their values, are passed into the function **swap( )**. After **swap( )** returns, the values of **i** and **j** will be exchanged.

Note *C++ allows you to fully automate a call by reference through the use of reference parameters. This feature is described in Part Two.*

Calling Functions with Arrays

Arrays are covered in detail in Chapter 4. However, this section discusses passing arrays as arguments to functions because it is an exception to the normal call-by-value parameter passing.

When an array is used as a function argument, its address is passed to a function. This is an exception to the call-by-value parameter passing convention. In this case, the code inside the function is operating on, and potentially altering, the actual contents of the array used to call the function. For example, consider the function **print_upper( )**, which prints its string argument in uppercase:

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

void print_upper(char *string);

int main(void)
{
  char s[80];

  gets(s);
  print_upper(s);
  printf("\ns is now uppercase: %s", s);
  return 0;
}

/* Print a string in uppercase. */
void print_upper(char *string)
{
  register int t;

  for(t=0; string[t]; ++t)  {
    string[t] = toupper(string[t]);
    putchar(string[t]);
  }
}
```

After the call to **print_upper( )**, the contents of array **s** in **main( )** have also been changed to uppercase. If this is not what you want, you could write the program like this:

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

```
void print_upper(char *string);

int main(void)
{
  char s[80];

  gets(s);
  print_upper(s);
  printf("\ns is unchanged: %s", s);

  return 0;
}

void print_upper(char *string)
{
  register int t;

  for(t=0; string[t]; ++t)
    putchar(toupper(string[t]));
}
```

In this version, the contents of array **s** remain unchanged because its values are not altered inside **print_upper( )**.

The standard library function **gets( )** is a classic example of passing arrays into functions. Although the **gets( )** in your standard library is more sophisticated, the following simpler version, called **xgets( )**, will give you an idea of how it works.

```
/* A simple version of the standard
   gets() library function. */
char *xgets(char *s)
{
  char ch, *p;
  int t;

  p = s;  /* gets() returns a pointer to s */

  for(t=0; t<80; ++t){
    ch = getchar();

    switch(ch) {
```

```
      case '\n':
        s[t] = '\0'; /* terminate the string */
        return p;
      case '\b':
        if(t>0) t--;
        break;
      default:
        s[t] = ch;
    }
  }
  s[79] = '\0';
  return p;
}
```

The **xgets( )** function must be called with a character pointer. This, of course, can be the name of a character array, which by definition is a character pointer. Upon entry, **xgets( )** establishes a **for** loop from 0 to 79. This prevents larger strings from being entered at the keyboard. If more than 80 characters are entered, the function returns. (The real **gets( )** function does not have this restriction.) Because C/C++ has no built-in bounds checking, you should make sure that any array used to call **xgets( )** can accept at least 80 characters. As you type characters on the keyboard, they are placed in the string. If you type a backspace, the counter **t** is reduced by 1, effectively removing the previous character from the array. When you press ENTER, a null is placed at the end of the string, signaling its termination. Because the actual array used to call **xgets( )** is modified, upon return it contains the characters that you type.

argc and argv—Arguments to main( )

Sometimes it is useful to pass information into a program when you run it. Generally, you pass information into the **main( )** function via command line arguments. A *command line argument* is the information that follows the program's name on the command line of the operating system. For example, when you compile a program, you might type something like the following after the command prompt:

cc *program_name*

where *program_name* is a command line argument that specifies the name of the program you wish to compile.

There are two special built-in arguments, **argv** and **argc**, that are used to receive command line arguments. The **argc** parameter holds the number of arguments on the command line and is an integer. It is always at least 1 because the name of the program qualifies as the first argument. The **argv** parameter is a pointer to an array

of character pointers. Each element in this array points to a command line argument. All command line arguments are strings—any numbers will have to be converted by the program into the proper internal format. For example, this simple program prints **Hello** and your name on the screen if you type it directly after the program name.

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

int main(int argc, char *argv[])
{
  if(argc!=2) {
    printf("You forgot to type your name.\n");
    exit(1);
  }
  printf("Hello %s", argv[1]);

  return 0;
}
```

If you called this program **name** and your name were Tom, you would type **name Tom** to run the program. The output from the program would be **Hello Tom**.

In many environments, each command line argument must be separated by a space or a tab. Commas, semicolons, and the like are not considered separators. For example,

```
run Spot, run
```

is made up of three strings, while

```
Herb,Rick,Fred
```

is a single string since commas are not generally legal separators.

Some environments allow you to enclose within double quotes a string containing spaces. This causes the entire string to be treated as a single argument. Check your operating system documentation for details on the definition of command line parameters for your system.

You must declare **argv** properly. The most common method is

```
char *argv[];
```

The empty brackets indicate that the array is of undetermined length. You can now access the individual arguments by indexing **argv**. For example, **argv[0]** points to the

first string, which is always the program's name; **argv[1]** points to the first argument, and so on.

Another short example using command line arguments is the program called **countdown**, shown here. It counts down from a starting value (which is specified on the command line) and beeps when it reaches 0. Notice that the first argument containing the number is converted into an integer by the standard function **atoi( )**.If the string "display" is the second command line argument, the countdown will also be displayed on the screen.

```
/* Countdown program. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int main(int argc, char *argv[])
{
  int disp, count;

  if(argc<2) {
    printf("You must enter the length of the count\n");
    printf("on the command line.  Try again.\n");
    exit(1);
  }

  if(argc==3 && !strcmp(argv[2], "display")) disp = 1;
  else disp = 0;

  for(count=atoi(argv[1]); count; --count)
    if(disp) printf("%d\n", count);

  putchar('\a');  /* this will ring the bell */
  printf("Done");

  return 0;
}
```

Notice that if no command line arguments have been specified, an error message is printed. A program with command line arguments often issues instructions if the user attempts to run the program without entering the proper information.

To access an individual character in one of the command line arguments, add a second index to **argv**. For example, the next program displays all of the arguments with which it was called, one character at a time:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int t, i;

  for(t=0; t<argc; ++t) {
    i = 0;

    while(argv[t][i]) {
      putchar(argv[t][i]);
      ++i;
    }
    printf("\n");
  }

  return 0;
}
```

Remember, the first index accesses the string, and the second index accesses the individual characters of the string.

Normally, you use **argc** and **argv** to get initial commands into your program. In theory, you can have up to 32,767 arguments, but most operating systems do not allow more than a few. You typically use these arguments to indicate a filename or an option. Using command line arguments gives your program a professional appearance and facilitates its use in batch files.

When a program does not require command line parameters, it is common practice to explicitly declare **main( )** as having no parameters. For C programs this is accomplished by using the **void** keyword in its parameter list. (This is the approach used by the programs in Part One of this book.) However, for C++ programs you may simply specify an empty parameter list. In C++, the use of **void** to indicate an empty parameter list is allowed, but redundant.

The names **argc** and **argv** are traditional but arbitrary. You may name these two parameters to **main( )** anything you like. Also, some compilers may support additional arguments to **main( )**, so be sure to check your user's manual.

The return Statement

The **return** statement itself is described in Chapter 3. As explained, it has two important uses. First, it causes an immediate exit from the function that it is in. That is, it causes program execution to return to the calling code. Second, it may be used to return a value. This section examines how the **return** statement is used.

Returning from a Function

There are two ways that a function terminates execution and returns to the caller. The first occurs when the last statement in the function has executed and, conceptually, the function's ending curly brace (}) is encountered. (Of course, the curly brace isn't actually present in the object code, but you can think of it in this way.) For example, the **pr_reverse( )** function in this program simply prints the string "I like C++" backwards on the screen and then returns.

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

void pr_reverse(char *s);

int main(void)
{
  pr_reverse("I like C++");

  return 0;
}

void pr_reverse(char *s)
{
  register int t;

  for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
}
```

Once the string has been displayed, there is nothing left for **pr_reverse( )** to do, so it returns to the place from which it was called.

Actually, not many functions use this default method of terminating their execution. Most functions rely on the **return** statement to stop execution either because a value must be returned or to make a function's code simpler and more efficient.

A function may contain several **return** statements. For example, the **find_substr( )** function in the following program returns the starting position of a substring within a string, or returns –1 if no match is found.

```
#include <stdio.h>

int find_substr(char *s1, char *s2);

int main(void)
{
```

```
  if(find_substr("C++ is fun", "is") != -1)
    printf("substring is found");

  return 0;
}

/* Return index of first match of s2 in s1. */
int find_substr(char *s1, char *s2)
{
  register int t;
  char *p, *p2;

  for(t=0; s1[t]; t++) {
    p = &s1[t];
    p2 = s2;

    while(*p2 && *p2==*p) {
      p++;
      p2++;
    }
    if(!*p2) return t; /* 1st return */
  }
   return -1; /* 2nd return */
}
```

Returning Values

All functions, except those of type **void**, return a value. This value is specified by the **return** statement. In C89, if a non-**void** function does not explicitly return a value via a **return** statement, then a garbage value is returned. In C++ (and C99), a non-**void** function *must* contain a **return** statement that returns a value. That is, in C++, if a function is specified as returning a value, any **return** statement within it must have a value associated with it. However, if execution reaches the end of a non-**void** function, then a garbage value is returned. Although this condition is not a syntax error, it is still a fundamental flaw and should be avoided.

As long as a function is not declared as **void**, you may use it as an operand in an expression. Therefore, each of the following expressions is valid:

```
x = power(y);
if(max(x,y) > 100) printf("greater");
for(ch=getchar(); isdigit(ch); ) ... ;
```

As a general rule, a function cannot be the target of an assignment. A statement such as

```
swap(x,y) = 100; /* incorrect statement */
```

is wrong. The C/C++ compiler will flag it as an error and will not compile a program that contains it. (As is discussed in Part Two, C++ allows some interesting exceptions to this general rule, enabling some types of functions to occur on the left side of an assignment.)

When you write programs, your functions generally will be of three types. The first type is simply computational. These functions are specifically designed to perform operations on their arguments and return a value based on that operation. A computational function is a "pure" function. Examples are the standard library functions **sqrt( )** and **sin( )**, which compute the square root and sine of their arguments.

The second type of function manipulates information and returns a value that simply indicates the success or failure of that manipulation. An example is the library function **fclose( )**, which is used to close a file. If the close operation is successful, the function returns 0; if the operation is unsuccessful, it returns **EOF**.

The last type of function has no explicit return value. In essence, the function is strictly procedural and produces no value. An example is **exit( )**, which terminates a program. All functions that do not return values should be declared as returning type **void**. By declaring a function as **void**, you keep it from being used in an expression, thus preventing accidental misuse.

Sometimes, functions that really don't produce an interesting result return something anyway. For example, **printf( )** returns the number of characters written. Yet it would be unusual to find a program that actually checked this. In other words, although all functions, except those of type **void**, return values, you don't have to use the return value for anything. A common question concerning function return values is, "Don't I have to assign this value to some variable since a value is being returned?" The answer is no. If there is no assignment specified, the return value is simply discarded. Consider the following program, which uses the function **mul( )**:

```
#include <stdio.h>

int mul(int a, int b);

int main(void)
{
  int x, y, z;

  x = 10;   y = 20;
```

```
  z = mul(x, y);            /* 1 */
  printf("%d", mul(x,y));   /* 2 */
  mul(x, y);                /* 3 */

  return 0;
}

int mul(int a, int b)
{
  return a*b;
}
```

In line 1, the return value of **mul( )** is assigned to **z**. In line 2, the return value is not actually assigned, but it is used by the **printf( )** function. Finally, in line 3, the return value is lost because it is neither assigned to another variable nor used as part of an expression.

Returning Pointers

Although functions that return pointers are handled just like any other type of function, a few important concepts need to be discussed.

Pointers to variables are neither integers nor unsigned integers. They are the memory addresses of a certain type of data. The reason for this distinction is because pointer arithmetic is relative to the base type. For example, if an integer pointer is incremented, it will contain a value that is 4 greater than its previous value (assuming 4-byte integers). In general, each time a pointer is incremented (or decremented), it points to the next (or previous) item of its type. Since the length of different data types may differ, the compiler must know what type of data the pointer is pointing to. For this reason, a function that returns a pointer must declare explicitly what type of pointer it is returning. For example, you should not use a return type of **int** * to return a **char** * pointer!

To return a pointer, a function must be declared as having a pointer return type. For example, this function returns a pointer to the first occurrence of the character **c** in string **s**:

```
/* Return pointer of first occurrence of c in s. */
char *match(char c, char *s)
{
  while(c!=*s && *s) s++;
  return(s);
}
```

If no match is found, a pointer to the null terminator is returned. Here is a short program that uses **match( )**:

```
#include <stdio.h>

char *match(char c, char *s);  /* prototype */

int main(void)
{
  char s[80], *p, ch;

  gets(s);
  ch = getchar();
  p = match(ch, s);

  if(*p)  /* there is a match */
    printf("%s ", p);
  else
    printf("No match found.");

  return 0;
}
```

This program reads a string and then a character. If the character is in the string, the program prints the string from the point of match. Otherwise, it prints **No match found**.

Functions of Type void

One of **void**'s uses is to explicitly declare functions that do not return values. This prevents their use in any expression and helps avert accidental misuse. For example, the function **print_vertical( )** prints its string argument vertically down the side of the screen. Since it returns no value, it is declared as **void**.

```
void print_vertical(char *str)
{
  while(*str)
    printf("%c\n", *str++);
}
```

Here is an example that uses **print_vertical( )**.

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

```
void print_vertical(char *str);  /* prototype */

int main(int argc, char *argv[])
{
  if(argc > 1) print_vertical(argv[1]);

  return 0;
}

void print_vertical(char *str)
{
  while(*str)
    printf("%c\n", *str++);
}
```

One last point: Early versions of C did not define the **void** keyword. Thus, in early C programs, functions that did not return values simply defaulted to type **int**. Therefore, don't be surprised to see many examples of this in older code.

What Does main( ) Return?

The **main( )** function returns an integer to the calling process, which is generally the operating system. Returning a value from **main( )** is the equivalent of calling **exit( )** with the same value. If **main( )** does not explicitly return a value, the value passed to the calling process is technically undefined. In practice, most C/C++ compilers automatically return 0, but do not rely on this if portability is a concern.

Recursion

In C/C++, a function can call itself. A function is said to be *recursive* if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself, and is sometimes called *circular definition*.

A simple example of a recursive function is **factr( )**, which computes the factorial of an integer. The factorial of a number **n** is the product of all the whole numbers between 1 and **n**. For example, 3 factorial is 1 x 2 x 3, or 6. Both **factr( )** and its iterative equivalent are shown here:

```
/* recursive */
int factr(int n) {
  int answer;

  if(n==1) return(1);
```

```
  answer = factr(n-1)*n; /* recursive call */
  return(answer);
}

/* non-recursive */
int fact(int n) {
  int t, answer;

  answer = 1;

  for(t=1; t<=n; t++)
    answer=answer*(t);

  return(answer);
}
```

The nonrecursive version of **fact( )** should be clear. It uses a loop that runs from 1 to **n** and progressively multiplies each number by the moving product.

The operation of the recursive **factr( )** is a little more complex. When **factr( )** is called with an argument of 1, the function returns 1. Otherwise, it returns the product of **factr(n–1)*n**. To evaluate this expression, **factr( )** is called with **n–1**. This happens until **n** equals 1 and the calls to the function begin returning.

Computing the factorial of 2, the first call to **factr( )** causes a second, recursive call with the argument of 1. This call returns 1, which is then multiplied by 2 (the original **n** value). The answer is then 2. Try working through the computation of 3 factorial on your own. (You might want to insert **printf( )** statements into **factr( )** to see the level of each call and what the intermediate answers are.)

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function. Recursive functions could be said to "telescope" out and back.

Often, recursive routines do not significantly reduce code size or improve memory utilization over their iterative counterparts. Also, the recursive versions of most routines may execute a bit slower than their iterative equivalents because of the overhead of the repeated function calls. In fact, many recursive calls to a function could cause a stack overrun. Because storage for function parameters and local variables is on the stack and each new call creates a new copy of these variables, the stack could be exhausted. However, you probably will not have to worry about this unless a recursive function runs wild.

The main advantage to recursive functions is that you can use them to create clearer and simpler versions of several algorithms. For example, the Quicksort algorithm is difficult to implement in an iterative way. Also, some problems, especially ones related to artificial intelligence, lend themselves to recursive solutions. Finally, some people seem to think recursively more easily than iteratively.

When writing recursive functions, you must have a conditional statement, such as an **if**, somewhere to force the function to return without the recursive call being executed. If you don't, the function will never return once you call it. Omitting the conditional statement is a common error when writing recursive functions. Use **printf( )** liberally during program development so that you can watch what is going on and abort execution if you see a mistake.

Function Prototypes

In C++ all functions must be declared before they are used. This is normally accomplished using a *function prototype*. Function prototypes were not part of the original C language. They were, however, added when C was standardized. While prototypes are not technically required by Standard C, their use is strongly encouraged. Prototypes have always been *required* by C++. In this book, all examples include full function prototypes. Prototypes enable both C and C++ to provide stronger type checking, somewhat like that provided by languages such as Pascal. When you use prototypes, the compiler can find and report any illegal type conversions between the type of arguments used to call a function and the type definition of its parameters. The compiler will also catch differences between the number of arguments used to call a function and the number of parameters in the function.

The general form of a function prototype is

type func_name(type parm_name1, type parm_name2,. . .,
type parm_nameN);

The use of parameter names is optional. However, they enable the compiler to identify any type mismatches by name when an error occurs, so it is a good idea to include them.

The following program illustrates the value of function prototypes. It produces an error message because it contains an attempt to call **sqr_it( )** with an integer argument instead of the integer pointer required. (It is illegal to convert an integer into a pointer.)

```
/* This program uses a function prototype to
   enforce strong type checking. */

void sqr_it(int *i); /* prototype */

int main(void)
{
```

```
  int x;

  x = 10;
  sqr_it(x);  /* type mismatch */

  return 0;
}

void sqr_it(int *i)
{
  *i = *i * *i;
}
```

A function's definition can also serve as its prototype if the definition occurs prior to the function's first use in the program. For example, this is a valid program.

```
#include <stdio.h>

/* This definition will also serve
   as a prototype within this program. */
void f(int a, int b)
{
  printf("%d ", a % b);
}

int main(void)
{
  f(10,3);

  return 0;
}
```

In this example, since **f( )** is defined prior to its use in **main( ),** no separate prototype is required. While it is possible for a function's definition to serve as its prototype in small programs, it is seldom possible in large ones—especially when several files are used. The programs in this book include a separate prototype for each function because that is the way C/C++ code is normally written in practice.

The only function that does not require a prototype is **main( ),** since it is the first function called when your program begins.

Because of the need for compatibility with the original version of C, there is a small but important difference between how C and C++ handle the prototyping of

a function that has no parameters. In C++, an empty parameter list is simply indicated in the prototype by the absence of any parameters. For example,

```
int f(); /* C++ prototype for a function with no parameters */
```

However, in C this prototype means something different. For historical reasons, an empty parameter list simply says that *no parameter information* is given. As far as the compiler is concerned, the function could have several parameters or no parameters. In C, when a function has no parameters, its prototype uses **void** inside the parameter list. For example, here is **f( )**'s prototype as it would appear in a C program.

```
float f(void);
```

This tells the compiler that the function has no parameters, and any call to that function that has parameters is an error. In C++, the use of **void** inside an empty parameter list is still allowed, but is redundant.

Remember *In C++, **f( )** and **f(void)** are equivalent.*

Function prototypes help you trap bugs before they occur. In addition, they help verify that your program is working correctly by not allowing functions to be called with mismatched arguments.

One last point: Since early versions of C did not support the full prototype syntax, prototypes are technically optional in C. This is necessary to support pre-prototype C code. If you are porting older C code to C++, you may need to add full function prototypes before it will compile. Remember: Although prototypes are optional in C, they are required by C++. This means that every function in a C++ program must be fully prototyped.

Standard Library Function Prototypes

Any standard library function used by your program must be prototyped. To accomplish this, you must include the appropriate *header* for each library function. All necessary headers are provided by the C/C++ compiler. In C, the library headers are (usually) files that use the .H extension. In C++, headers may be either separate files or built into the compiler itself. In either case, a header contains two main elements: any definitions used by the library functions and the prototypes for the library functions. For example, **stdio.h** is included in almost all programs in this part of the book because it contains the prototype for **printf( )**. The headers for the standard library are described in Part Three.

Declaring Variable-Length Parameter Lists

You can specify a function that has a variable number of parameters. The most common example is **printf( )**. To tell the compiler that an unknown number of arguments may be passed to a function, you must end the declaration of its parameters using three periods. For example, this prototype specifies that **func( )** will have at least two integer parameters and an unknown number (including 0) of parameters after that.

```
int func(int a, int b, ...);
```

This form of declaration is also used by a function's definition.

Any function that uses a variable number of parameters must have at least one actual parameter. For example, this is incorrect:

```
int func(...); /* illegal */
```

Old-Style Versus Modern Function Parameter Declarations

Early versions of C used a different parameter declaration method than does either Standard C or Standard C++. This early approach is sometimes called the *classic* form. This book uses a declaration approach called the *modern* form. Standard C supports both forms, but strongly recommends the modern form. Standard C++ only supports the modern parameter declaration method. However, you should know the old-style form because many older C programs still use it.

The old-style function parameter declaration consists of two parts: a parameter list, which goes inside the parentheses that follow the function name, and the actual parameter declarations, which go between the closing parentheses and the function's opening curly brace. The general form of the old-style parameter definition is

```
type func_name(parm1, parm2, . . .parmN)
type parm1;
type parm2;
.
.
.
type parmN;
{
function code
}
```

For example, this modern declaration:

```
float f(int a, int b, char ch)
{
  /* ... */
}
```

will look like this in its old-style form:

```
float f(a, b, ch)
int a, b;
char ch;
{
  /* ... */
}
```

Notice that the old-style form allows the declaration of more than one parameter in a list after the type name.

Remember *The old-style form of parameter declaration is designated as obsolete by the C language and is not supported by C++.*