16

The C Preprocessor and the C Library

You will learn about the following in this chapter:

- Preprocessor directives:

  `#define`, `#include`, `#ifdef`

  `#else`, `#endif`, `#ifndef`

  `#if`, `#elif`, `#line`, `#error`, `#pragma`
- Keywords:

  `_Generic`, `_Noreturn`, `_Static_assert`
- Functions/Macros:

  `sqrt()`, `atan()`, `atan2()`

  `exit()`, `atexit()`

  `assert()`

  `memcpy()`, `memmove()`

  `va_start()`, `va_arg()`, `va_copy()`, `va_end()`
- More capabilities of the C preprocessor
- Function-like macros and conditional compilation
- The generic selection expression
- Inline functions
- The C library in general and some of its handy functions in particular

The C language proper is built on the C keywords, expressions, and statements as well as the rules for using them. The C standard, however, goes beyond describing just the C language. It also describes how the C preprocessor should perform, establishes which functions form the standard C library, and details how these functions work. We'll explore the C preprocessor and the C library in this chapter, beginning with the preprocessor.

The preprocessor looks at your program before it is compiled (hence the term *pre*processor). Following your preprocessor directives, the preprocessor replaces the symbolic abbreviations in your program with the directions they represent. The preprocessor can include other files at your request, and it can select which code the compiler sees. The preprocessor doesn't know about C. Basically, it takes some text and converts it to other text. This description does not do justice to its true utility and value, so let's turn to examples. You've encountered examples of `#define` and `#include` all along. Now we can gather what you have learned in one place and add to it.

First Steps in Translating a Program

The compiler has to put a program through some translation phases before jumping into preprocessing. The compiler starts its work by mapping characters appearing in the source code to the source character set. This takes care of multibyte characters and trigraphs—character extensions that make the outer face of C more international. (Appendix B "Reference Section VII, Expanded Character Support," gives an overview of these extensions.)

Second, the compiler locates each instance of a backslash followed by a newline character and deletes them. That is, two physical lines such as

```
printf("That's wond\
erful!\n");
```

are converted to a single *logical line*:

```
printf("That's wonderful\n!");
```

Note that in this context, "newline character" means the character produced by pressing the Enter key to start a new line in your source code file; it doesn't mean the symbolic representation `\n`.

This feature is useful as a preparation for preprocessing because preprocessing expressions are required to be one logical line long, but that one logical line can be more than one physical line.

Next, the compiler breaks the text into a sequence of preprocessing tokens and sequences of whitespace and comments. (In basic terms, tokens are groups separated from each other by spaces, tabs, or line breaks; this chapter will look at tokens in more detail later.) One point of interest now is that each comment is replaced by one space character. So something such as

```
int/* this doesn't look like a space*/fox;
```

becomes

```
int fox;
```

Also, an implementation may choose to replace each sequence of whitespace characters (other than a newline) with a single space. Finally, the program is ready for the preprocessing phase, and the preprocessor looks for potential preprocessing directives, indicated by a # symbol at the beginning of a line.

Manifest Constants: `#define`

The `#define` preprocessor directive, like all preprocessor directives, begins with the # symbol at the beginning of a line. The ANSI and subsequent standards permit the # symbol to be preceded by spaces or tabs, and it allows for space between the # and the remainder of the directive. However, older versions of C typically require that the directive begin in the leftmost column and that there be no spaces between the # and the remainder of the directive. A directive can appear anywhere in the source file, and the definition holds from its place of appearance to the end of the file. We have used directives heavily to define symbolic, or *manifest*, constants in our programs, but they have more range than that, as we will show. Listing 16.1 illustrates some of the possibilities and properties of the `#define` directive.

Preprocessor directives run until the first newline following the #. That is, a directive is limited to one line in length. However, as mentioned earlier, the combination backslash/newline is deleted before preprocessing begins, so you can spread the directive over several physical lines. These lines, however, constitute a single logical line.

Listing 16.1 **The `preproc.c` Program**

```
/* preproc.c -- simple preprocessor examples */
#include <stdio.h>
#define TWO 2        /* you can use comments if you like   */
#define OW "Consistency is the last refuge of the unimagina\
tive. - Oscar Wilde" /* a backslash continues a definition */
                     /* to the next line                   */
#define FOUR  TWO*TWO
#define PX printf("X is %d.\

int main(void)
{
    int x = TWO;

    PX;
    x = FOUR;
    printf(FMT, x);
    printf("%s\n", OW);
    printf("TWO: OW\n");

    return 0;
}
```

Each `#define` line (logical line, that is) has three parts. The first part is the `#define` directive itself. The second part is your chosen abbreviation, known as a *macro*. Some macros, like these examples, represent values; they are called *object-like macros*. (C also has *function-like macros*, and we'll get to them later.) The macro name must have no spaces in it, and it must conform to the same naming rules that C variables follow: Only letters, digits, and the underscore (_) character can be used, and the first character cannot be a digit. The third part (the remainder of the line) is termed the *replacement list* or *body* (see Figure 16.1). When the preprocessor finds an example of one of your macros within your program, it almost always replaces it with the body. (There is one exception, as we will show you in just a moment.) This process of going from a macro to a final replacement is called *macro expansion*. Note that you can use standard C comments on a `#define` line; as mentioned earlier, each is replaced by a space before the preprocessor sees it.

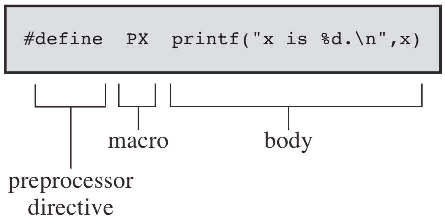

Figure 16.1 Parts of an object-like macro definition.

Let's run the example and see how it works:

```
X is 2.
X is 4.
Consistency is the last refuge of the unimaginative. - Oscar Wilde
TWO: OW
```

Here's what happened. The statement

```
int x = TWO;
```

becomes

```
int x = 2;
```

as `2` is substituted for `TWO`. Then the statement

```
PX;
```

becomes

```
printf("X is %d.\n", x);
```

as that wholesale substitution is made. This is a new wrinkle, because up to now we've used macros only to represent constants. Here you see that a macro can express any string, even a whole C expression. Note, though, that this is a constant string; `PX` will print only a variable named `x`.

The next line also represents something new. You might think that `FOUR` is replaced by `4`, but the actual process is this:

```
x = FOUR;
```

becomes

```
x = TWO*TWO;
```

which then becomes

```
x = 2*2;
```

The macro expansion process ends there. The actual multiplication takes place not while the preprocessor works, but during compilation, because the C compiler evaluates all constant expressions (expressions with just constants) at compile time. The preprocessor does no calculation; it just makes the suggested substitutions very literally.

Note that a macro definition can include other macros. (Some compilers do not support this nesting feature.)

In the next line

```
printf (FMT, x);
```

becomes

```
printf("X is %d.\n",x);
```

as `FMT` is replaced by the corresponding string. This approach could be handy if you had a lengthy control string that you had to use several times. Alternatively, you can do the following:

```
const char * fmt = "X is %d.\n";
```

Then you can use `fmt` as the `printf()` control string.

In the next line, `OW` is replaced by the corresponding string. The double quotation marks make the replacement string a character string constant. The compiler will store it in an array terminated with a null character. Therefore,

```
#define HAL 'Z'
```

defines a character constant, but

```
#define HAP "Z"
```

defines a character string: `Z\0`.

In the example, we used a backslash immediately before the end of the line to extend the string to the next line:

```
#define OW "Consistency is the last refuge of the unimagina\
tive. - Oscar Wilde"
```

Note that the second line is flush left. Suppose, instead, we did this:

```
#define OW "Consistency is the last refuge of the unimagina\
    tive. - Oscar Wilde"
```

Then the output would be this:

```
Consistency is the last refuge of the unimagina    tive. - Oscar Wilde
```

The space between the beginning of the line and `tive` counts as part of the string.

In general, wherever the preprocessor finds one of your macros in your program, it replaces it literally with the equivalent replacement text. If that string also contains macros, they, too, are replaced. The one exception to replacement is a macro found within double quotation marks. Therefore,

```
printf("TWO: OW");
```

prints `TWO: OW` literally instead of printing

```
2: Consistency is the last refuge of the unimaginative. - Oscar Wilde
```

To print this last line, you would use this:

```
printf("%d: %s\n", TWO, OW);
```

Here, the macros are outside the double quotation marks.

When should you use symbolic constants? You should use them for most numeric constants. If the number is some constant used in a calculation, a symbolic name makes its meaning clearer. If the number is an array size, a symbolic name makes it simpler to change the array size and loop limits later. If the number is a system code for, say, `EOF`, a symbolic representation makes your program much more portable; just change one `EOF` definition. Mnemonic value, easy alterability, portability—these features all make symbolic constants worthwhile.

It is true that the `const` keyword now supported by C allows for a more flexible way of creating constants. With `const` you can create global constants and local constants, numeric constants, array constants, and structure constants. On the other hand, macro constants can be used to specify the sizes of standard arrays and as initialization values for `const` values:

```
#define LIMIT 20
const int LIM = 50;
static int data1[LIMIT];    // valid
static int data2[LIM];      // not required to be valid
const int LIM2 = 2 * LIMIT; // valid
const int LIM3 = 2 * LIM;   // not required to be valid
```

Let's look at the "not required to be valid" comments. In C, the array size for nonautomatic arrays is supposed to be an integer constant expression, meaning that it's a combination of integer constants, such as `5`, enumeration constants, and `sizeof` expressions. This doesn't include values declared using `const`. (This is one respect in which C++ differs from C; in C++ you can use `const` values as part of constant expressions.) However, an implementation may accept other forms of constant expressions. So, for example, GCC 4.7.3 doesn't accept the declaration for `data2`, but Clang 4.6 does.

Tokens

Technically, the body of a macro is considered to be a string of *tokens* rather than a string of characters. C preprocessor tokens are the separate "words" in the body of a macro definition. They are separated from one another by whitespace. For example, the definition

```
#define FOUR 2*2
```

has one token—the sequence `2*2`—but the definition

```
#define SIX 2 * 3
```

has three tokens in it: `2`, `*`, and `3`.

Character strings and token strings differ in how multiple spaces in a body are treated. Consider this definition:

```
#define EIGHT 4    *    8
```

A preprocessor that interprets the body as a character string would replace `EIGHT` with `4  *  8`. That is, the extra spaces would be part of the replacement, but a preprocessor that interprets the body as tokens will replace `EIGHT` with three tokens separated by single spaces: `4 * 8`. In other words, the character string interpretation views the spaces as part of the body, but the token interpretation views the spaces as separators between the tokens of the body. In practice, some C compilers have viewed macro bodies as strings rather than as tokens. The difference is of practical importance only for usages more intricate than what we're attempting here.

Incidentally, the C compiler takes a more complex view of tokens than the preprocessor does. The compiler understands the rules of C and doesn't necessarily require spaces to separate tokens. For example, the C compiler would view `2*2` as three tokens because it recognizes that each `2` is a constant and that `*` is an operator.

Redefining Constants

Suppose you define `LIMIT` to be 20, and then later in the same file you define it again as 25. This process is called *redefining a constant*. Implementations differ on redefinition policy. Some consider it an error unless the new definition is the same as the old. Others allow redefinition, perhaps issuing a warning. The ANSI standard takes the first view, allowing redefinition only if the new definition duplicates the old.

Having the same definition means the bodies must have the same tokens in the same order. Therefore, these two definitions agree:

```
#define SIX 2 * 3
#define SIX 2       *    3
```

Both have the same three tokens, and the extra spaces are not part of the body. The next definition is considered different:

```
#define SIX 2*3
```

It has just one token, not three, so it doesn't match. If you want to redefine a macro, use the `#undef` directive, which we discuss later.

If you do have constants that you need to redefine, it might be easier to use the `const` keyword and scope rules to accomplish that end.

Using Arguments with `#define`

By using arguments, you can create *function-like macros* that look and act much like functions. A macro with arguments looks very similar to a function because the arguments are enclosed within parentheses. Function-like macro definitions have one or more arguments in parentheses, and these arguments then appear in the replacement portion, as shown in Figure 16.2.

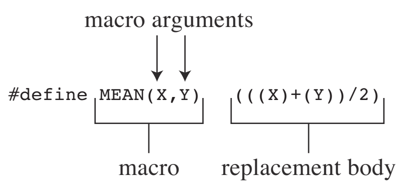

Figure 16.2 Parts of a function-like macro definition.

Here's a sample definition:

```
#define SQUARE(X) X*X
```

It can be used in program like this:

```
z = SQUARE(2);
```

This looks like a function call, but it doesn't necessarily behave identically. Listing 16.2 illustrates using this and a second macro. Some of the examples also point out possible pitfalls, so read them carefully.

Listing 16.2 **The `mac_arg.c` Program**

```
/* mac_arg.c -- macros with arguments */
#include <stdio.h>
#define SQUARE(X) X*X
#define PR(X)   printf("The result is %d.\n", X)
int main(void)
{
    int x = 5;
    int z;

    printf("x = %d\n", x);
    z = SQUARE(x);
    printf("Evaluating SQUARE(x): ");
    PR(z);
    z = SQUARE(2);
    printf("Evaluating SQUARE(2): ");
    PR(z);
    printf("Evaluating SQUARE(x+2): ");
    PR(SQUARE(x+2));
    printf("Evaluating 100/SQUARE(2): ");
    PR(100/SQUARE(2));
    printf("x is %d.\n", x);
    printf("Evaluating SQUARE(++x): ");
    PR(SQUARE(++x));
    printf("After incrementing, x is %x.\n", x);

    return 0;
}
```

The `SQUARE` macro has this definition:

```
#define SQUARE(X) X*X
```

Here, `SQUARE` is the macro identifier, the `X` in `SQUARE(X)` is the macro argument, and `X*X` is the replacement list. Wherever `SQUARE(x)` appears in Listing 16.2, it is replaced by `x*x`. This differs from the earlier examples in that you are free to use symbols other than `x` when you use this macro. The `x` in the macro definition is replaced by the symbol used in the macro call in the program. Therefore, `SQUARE(2)` is replaced by `2*2`, so the `x` really does act as an argument.

However, as you will soon see, a macro argument does not work exactly like a function argument. Here are the results of running the program. Note that some of the answers are different from what you might expect. Indeed, your compiler might not even give the same answer as what's shown here for the next-to-last line:

```
x = 5
Evaluating SQUARE(x): The result is 25.
Evaluating SQUARE(2): The result is 4.
```

```
Evaluating SQUARE(x+2): The result is 17.
Evaluating 100/SQUARE(2): The result is 100.
x is 5.
Evaluating SQUARE(++x): The result is 42.
After incrementing, x is 7.
```

The first two lines are predictable, but then you come to some peculiar results. Recall that `x` has the value `5`. This might lead you to expect that `SQUARE(x+2)` would be `7*7`, or `49`, but the printout says it is `17`, a prime number and certainly not a square! The simple reason for this misleading output is the one we have already stated—the preprocessor doesn't make calculations; it just substitutes character sequences. Wherever the definition shows an `x`, the preprocessor substitutes the characters `x+2`. Therefore,

```
x*x
```

becomes

```
x+2*x+2
```

The only multiplication is `2*x`. If `x` is `5`, this is the value of this expression:

```
5+2*5+2 = 5 + 10 + 2 = 17
```

This example pinpoints an important difference between a function call and a macro call. A function call passes the value of the argument to the function while the program is running. A macro call passes the argument token to the program before compilation; it's a different process at a different time. Can the definition be fixed to make `SQUARE(x+2)` yield 36? Sure. You simply need more parentheses:

```
#define SQUARE(x)  (x)*(x)
```

Now `SQUARE(x+2)` becomes `(x+2)*(x+2)`, and you get the desired multiplication as the parentheses carry over in the replacement string.

This doesn't solve all the problems, however. Consider the events leading to the next output line:

```
100/SQUARE(2)
```

becomes

```
100/2*2
```

By the laws of precedence, the expression is evaluated from left to right: `(100/2)*2` or `50*2` or `100`. This mix-up can be cured by defining `SQUARE(x)` as follows:

```
#define SQUARE(x)  (x*x)
```

This produces `100/(2*2)`, which eventually evaluates to `100/4`, or `25`.

To handle both of the previous two examples, you need this definition:

```
#define SQUARE(x)  ((x)*(x))
```

The lesson here is to use as many parentheses as necessary to ensure that operations and associations are done in the right order.

Even these precautions fail to save the final example from grief:

```
SQUARE(++x)
```

becomes

```
++x*++x
```

and `x` gets incremented twice, once before the multiplication and once afterward:

```
++x*++x = 6*7 = 42
```

Because the order of operations is left open, some compilers render the product `7*6`. Yet other compilers might increment both terms before multiplication, yielding `7*7`,or 49. Indeed, evaluating this expression results in what the standard calls undefined behavior. In all these cases, however, `x` starts with the value `5` and ends up with the value `7`, even though the code looks as though `x` was incremented just once.

The simplest remedy for this problem is to avoid using `++x` as a macro argument. In general, don't use increment or decrement operators with macros. Note that `++x` would work as a function argument because it would be evaluated to `6`, and then the value `6` would be sent to the function.

Creating Strings from Macro Arguments: The # Operator

Here's a function-like macro:

```
#define PSQR(X)  printf("The square of X is %d.\n", ((X)*(X)));
```

Suppose you used the macro like this:

```
PSQR(8);
```

Here's the output:

```
The square of X is 64.
```

Note that the `X` in the quoted string is treated as ordinary text, not as a token that can be replaced.

Suppose you do want to include the macro argument in a string. C enables you to do that. Within the replacement part of a function-like macro, the # symbol becomes a preprocessing operator that converts tokens into strings. For example, say that `x` is a macro parameter, and then `#x` is that parameter name converted to the string `"x"`. This process is called *stringizing*. Listing 16.3 illustrates how this process works.

Listing 16.3 **The `subst.c` Program**

```
/* subst.c -- substitute in string */
#include <stdio.h>
#define PSQR(x) printf("The square of " #x " is %d.\n",((x)*(x)))

int main(void)
{
    int y = 5;

    PSQR(y);
    PSQR(2 + 4);

    return 0;
}
```

Here's the output:

```
The square of y is 25.
The square of 2 + 4 is 36.
```

In the first call to the macro, `#x` was replaced by `"y"`, and in the second call `#x` was replaced by `"2 + 4"`. ANSI C string concatenation then combined these strings with the other strings in the `printf()` statement to produce the final strings that were used. For example, the first invocation becomes this:

```
printf("The square of " "y" " is %d.\n",((y)*(y)));
```

Then string concatenation converts the three adjacent strings to one string:

```
"The square of y is %d.\n"
```

Preprocessor Glue: The ## Operator

Like the # operator, the ## operator can be used in the replacement section of a function-like macro. Additionally, it can be used in the replacement section of an object-like macro. The ## operator combines two tokens into a single token. For example, you could do this:

```
#define XNAME(n) x ## n
```

Then the macro

```
XNAME(4)
```

would expand to the following:

```
x4
```

Listing 16.4 uses this and another macro using ## to do a bit of token gluing.

Listing 16.4 **The `glue.c` Program**

```
// glue.c -- use the ## operator
#include <stdio.h>
#define XNAME(n) x ## n
#define PRINT_XN(n) printf("x" #n " = %d\n", x ## n);

int main(void)
{
    int XNAME(1) = 14;  // becomes int x1 = 14;
    int XNAME(2) = 20;  // becomes int x2 = 20;
    int x3 = 30;
    PRINT_XN(1);        // becomes printf("x1 = %d\n", x1);
    PRINT_XN(2);        // becomes printf("x2 = %d\n", x2);
    PRINT_XN(3);        // becomes printf("x3 = %d\n", x3);
     return 0;
}
```

Here's the output:

```
x1 = 14
x2 = 20
x3 = 30
```

Note how the `PRINT_XN()` macro uses the # operator to combine strings and the ## operator to combine tokens into a new identifier.

Variadic Macros: ... and `__VA_ARGS__`

Some functions, such as `printf()`, accept a variable number of arguments. The `stdvar.h` header file, discussed later in this chapter, provides tools for creating user-defined functions with a variable number of arguments. And C99/C11 does the same thing for macros. Although not used in the standard, the word *variadic* has come into currency to label this facility. (However, the process that has added *stringizing* and *variadic* to the C vocabulary has not yet led to labeling functions or macros with a fixed number of arguments as fixadic functions and normadic macros.)

The idea is that the final argument in an argument list for a macro definition can be ellipses (that is, three periods). If so, the predefined macro `__VA_ARGS__` can be used in the substitution part to indicate what will be substituted for the ellipses. For example, consider this definition:

```
#define PR(...) printf(__VA_ARGS__)
```

Suppose you later invoke the macro like this:

```
PR("Howdy");
PR("weight = %d, shipping = $%.2f\n", wt, sp);
```

For the first invocation, `__VA_ARGS__` expands to one argument:

```
"Howdy"
```

For the second invocation, it expands to three arguments:

```
"weight = %d, shipping = $%.2f\n", wt, sp
```

Thus, the resulting code is this:

```
printf("Howdy");
printf("weight = %d, shipping = $%.2f\n", wt, sp);
```

Listing 16.5 shows a slightly more ambitious example that uses string concatenation and the # operator:

Listing 16.5 The `variadic.c` Program

```
// variadic.c -- variadic macros
#include <stdio.h>
#include <math.h>
#define PR(X, ...) printf("Message " #X ": " __VA_ARGS__)

int main(void)
{
    double x = 48;
    double y;

    y = sqrt(x);
    PR(1, "x = %g\n", x);
    PR(2, "x = %.2f, y = %.4f\n", x, y);

    return 0;
}
```

In the first macro call, `X` has the value `1`, so `#X` becomes `"1"`. That makes the expansion look like this:

```
print("Message " "1" ": " "x = %g\n", x);
```

Then the four strings are concatenated, reducing the call to this:

```
print("Message 1: x = %g\n", x);
```

Here's the output:

```
Message 1: x = 48
Message 2: x = 48.00, y = 6.9282
```

Don't forget, the ellipses have to be the last macro argument:

```
#define WRONG(X, ..., Y)  #X #__VA_ARGS__ #y  // won't work
```

Macro or Function?

Many tasks can be done by using a macro with arguments or by using a function. Which one should you use? There is no hard-and-fast rule, but here are some considerations.

Macros are somewhat trickier to use than regular functions because they can have odd side effects if you are unwary. Some compilers limit the macro definition to one line, and it is probably best to observe that limit, even if your compiler does not.

The macro-versus-function choice represents a trade-off between time and space. A macro produces inline code; that is, you get a statement in your program. If you use the macro 20 times, you get 20 lines of code inserted into your program. If you use a function 20 times, you have just one copy of the function statements in your program, so less space is used. On the other hand, program control must shift to where the function is and then return to the calling program, and this takes longer than inline code.

Macros have an advantage in that they don't worry about variable types. (This is because they deal with character strings, not with actual values.) Therefore, the `SQUARE(x)` macro can be used equally well with `int` or `float`.

C99 provides a third alternative—inline functions. We'll look at them later in this chapter.

Programmers typically use macros for simple functions such as the following:

```
#define MAX(X,Y)    ((X) > (Y) ? (X) : (Y))
#define ABS(X)       ((X) < 0 ? -(X) : (X))
#define ISSIGN(X)   ((X) == '+' || (X) == '-' ? 1 : 0)
```

(The last macro has the value `1`, or true, if `x` is an algebraic sign character.)

Here are some points to note:

- Remember that there are no spaces in the macro name, but that spaces can appear in the replacement string. ANSI C permits spaces in the argument list.
- Use parentheses around each argument and around the definition as a whole. This ensures that the enclosed terms are grouped properly in an expression such as

  ```
  forks = 2 * MAX(guests + 3, last);
  ```

- Use capital letters for macro function names. This convention is not as widespread as that of using capitals for macro constants. However, one good reason for using capitals is to remind yourself to be alert to possible macro side effects.
- If you intend to use a macro instead of a function primarily to speed up a program, first try to determine whether it is likely to make a significant difference. A macro that is used once in a program probably won't make any noticeable improvement in running time. A macro inside a nested loop is a much better candidate for speed improvements. Many systems offer program profilers to help you pin down where a program spends the most time.

Suppose you have developed some macro functions you like. Do you have to retype them each time you write a new program? Not if you remember the #include directive, reviewed in the following section.

File Inclusion: #include

When the preprocessor spots an #include directive, it looks for the following filename and includes the contents of that file within the current file. The #include directive in your source code file is replaced with the text from the included file. It's as though you sat down and typed in the entire contents of the included file at that particular location in your source file. The #include directive comes in two varieties:

#include <stdio.h>	←Filename directive>> in (angle brackets)>)> (angle)>angle brackets
#include "mystuff.h"	←Filename in double quotation marks

On a Unix system, the angle brackets tell the preprocessor to look for the file in one or more standard system directories. The double quotation marks tell it to first look in your current directory (or some other directory that you have specified in the filename) and then look in the standard places:

#include <stdio.h>	←Searches directive>> system directories
#include "hot.h"	←Searches your current working directory
#include "/usr/biff/p.h"	←Searches the /usr/biff directory

Integrated development environments (IDEs) also have a standard location or locations for the system header files. Many provide menu choices for specifying additional locations to be searched when angle brackets are used. As with Unix, using double quotes means to search a local directory first, but the exact directory searched depends on the compiler. Some search the same directory as that holding the source code; some search the current working directory; and some search the same directory as that holding the project file.

ANSI C doesn't demand adherence to the directory model for files because not all computer systems are organized similarly. In general, the method used to name files is system dependent, but the use of the angle brackets and double quotation marks is not.

Why include files? Because they have information the compiler needs. The stdio.h file, for example, typically includes definitions of EOF, NULL, getchar(), and putchar(). The last two are defined as macro functions. It also contains function prototypes for the C I/O functions.

The .h suffix is conventionally used for *header files*—files with information that are placed at the head of your program. Header files often contain preprocessor statements. Some, such as stdio.h, come with the system, but you are free to create your own.

Including a large header file doesn't necessarily add much to the size of your program. The content of header files, for the most part, is information used by the compiler to generate the final code, not material to be added to the final code.

Header Files: An Example

Suppose you developed a structure for holding a person's name and also wrote some functions for using the structure. You could gather together the various declarations in a header file. Listing 16.6 shows an example of this.

Listing 16.6 **The `names_st.h` Header File**

```
// names_st.h -- names_st structure header file
// constants
#include <string.h>
#define SLEN 32

// structure declarations
struct names_st
{
    char first[SLEN];
    char last[SLEN];
};

// typedefs
typedef struct names_st names;

// function prototypes
void get_names(names *);
void show_names(const names *);
char * s_gets(char * st, int n);
```

This header file includes many of the kinds of things commonly found in header files: #define directives, structure declarations, `typedef` statements, and function prototypes. Note that none of these things are executable code; rather, they are information that the compiler uses when it creates executable code.

This particular header file is a bit naïve. Normally, you should use `#ifndef` and `#define` to protect against multiple inclusions of a header file. We'll return to that technique later.

Executable code normally goes into a source code file, not a header file. For example, Listing 16.7 shows the function definitions for those functions prototyped in the header file. It includes the header file so that the compiler will know about `names` type.

Listing 16.7 **The `name_st.c` Source File**

```
// names_st.c -- define names_st functions
#include <stdio.h>
#include "names_st.h"     // include the header file

// function definitions
void get_names(names * pn)
{
    printf("Please enter your first name: ");
    s_gets(pn->first, SLEN);

    printf("Please enter your last name: ");
    s_gets(pn->last, SLEN);
 }

void show_names(const names * pn)
{
    printf("%s %s", pn->first, pn->last);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // look for newline
        if (find)                  // if the address is not NULL,
            *find = '\0';          // place a null character there
        else
            while (getchar() != '\n')
                continue;          // dispose of rest of line
    }
    return ret_val;
}
```

The `get_names()` function uses `fgets()` (via `s_gets()`) so as not to overflow the destination arrays. Listing 16.8 is an example of a program that uses this header and source code file.

Listing 16.8 **The `useheader.c` Program**

```
// useheader.c -- use the names_st structure
#include <stdio.h>
#include "names_st.h"
```

```
// remember to link with names_st.c

int main(void)
{
    names candidate;

    get_names(&candidate);
    printf("Let's welcome ");
    show_names(&candidate);
    printf(" to this program!\n");
    return 0;
}
```

Here is a sample run:

```
Please enter your first name: Ian
Please enter your last name: Smersh
Let's welcome Ian Smersh to this program!
```

Note the following points about this program:

- Both source code files use the `names_st` structure, so both have to include the `names_st.h` header file.
- You need to compile and link the `names_st.c` and the `useheader.c` source code files.
- Declarations and the like go into the `names_st.h` header file; function definitions go into the `names_st.c` source code file.

Uses for Header Files

A look through any of the standard header files can give you a good idea of the sort of information found in them. The most common forms of header contents include the following:

- **Manifest constants**—A typical `stdio.h` file, for instance, defines `EOF`, `NULL`, and `BUFSIZ` (the size of the standard I/O buffer).
- **Macro functions**—For example, `getchar()` is usually defined as `getc(stdin)`, `getc()` is usually defined as a rather complex macro, and the `ctype.h` header typically contains macro definitions for the `ctype` functions.
- **Function declarations**—The `string.h` header (`strings.h` on some older systems), for example, contains function declarations for the family of string functions. Under ANSI C and later, the declarations are in function prototype form.
- **Structure template definitions**—The standard I/O functions make use of a `FILE` structure containing information about a file and its associated buffer. The `stdio.h` file holds the declaration for this structure.

- **Type definitions**—You might recall that the standard I/O functions use a pointer-to-`FILE` argument. Typically, `stdio.h` uses a `#define` or a `typedef` to make `FILE` represent a pointer to a structure. Similarly, the `size_t` and `time_t` types are defined in header files.

Many programmers develop their own standard header files to use with their programs. This is particularly valuable if you develop a family of related functions and/or structures.

Also, you can use header files to declare external variables to be shared by several files. This makes sense, for example, if you've developed a family of functions that share a variable for reporting a status of some kind, such as an error condition. In that case, you could define a file-scope, external-linkage variable in the source code file containing the function declarations:

```
int status = 0;    // file scope, source code file
```

Then, in the header file associated with the source code file, you could place a reference declaration:

```
extern int status;  // in header file
```

This code would then appear in any file in which you included the header file, making the variable available to those files that use that family of functions. This declaration also would appear, through inclusion, in the function source code file, but it's okay to have both a defining declaration and a reference declaration in the same file, as long as the declarations agree in type.

Another candidate for inclusion in a header file is a variable or array with file scope, internal linkage, and `const` qualification. The `const` part protects against accidental changes, and the `static` part means that each file including the header gets its own copy of the constants so that there isn't the problem of needing one file with a defining declaration and the rest with reference declarations.

The `#include` and `#define` directives are the most heavily used C preprocessor features. We'll look at the other directives in less detail.

Other Directives

Programmers may have to prepare C programs or C library packages that have to work in a variety of environments. The choices of types of code can vary from one environment to another. The preprocessor provides several directives that help the programmer produce code that can be moved from one system to another by changing the values of some `#define` macros. The `#undef` directive cancels an earlier `#define` definition. The `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` directives allow you to specify different alternatives for which code is compiled. The `#line` directive lets you reset line and file information, the `#error` directive lets you issue error messages, and the `#pragma` directive lets you give instructions to the compiler.

The #undef Directive

The #undef directive "undefines" a given #define. That is, suppose you have this definition:

```
#define LIMIT 400
```

Then the directive

```
#undef LIMIT
```

removes that definition. Now, if you like, you can redefine LIMIT so that it has a new value. Even if LIMIT is not defined in the first place, it is still valid to undefine it. If you want to use a particular name and you are unsure whether it has been used previously, you can undefine it to be on the safe side.

Being Defined—The C Preprocessor Perspective

The preprocessor follows the same rules as C about what constitutes an identifier: An identifier can consist only of uppercase letters, lowercase letters, digits, and underscore characters, and a digit cannot be the first character. When the preprocessor encounters an identifier in a preprocessor directive, it considers it to be either defined or undefined. Here, *defined* means defined by the preprocessor. If the identifier is a macro name created by a prior #define directive in the same file and it hasn't been turned off by an #undef directive, it's defined. If the identifier is not a macro but is, say, a file-scope C variable, it's not defined as far as the preprocessor is concerned.

A defined macro can be an object-like macro, including an empty macro, or a function-like macro:

```
#define LIMIT 1000         // LIMIT is defined
#define GOOD               // GOOD is defined
#define A(X)  ((-(X))*(X)) // A is defined
int q;                     // q not a macro, hence not defined
#undef GOOD                // GOOD not defined
```

Note that the scope of a #define macro extends from the point it is declared in a file until it is the subject of an #undef directive or until the end of the file, whichever comes first. Also note that the position of the #define in a file will depend on the position of an #include directive if the macro is brought in via a header file.

A few predefined macros, such as __DATE__ and __FILE__ (discussed later this chapter), are always considered defined and cannot be undefined.

Conditional Compilation

You can use the other directives mentioned to set up conditional compilations. That is, you can use them to tell the compiler to accept or ignore blocks of information or code according to conditions at the time of compilation.

The `#ifdef`, `#else`, and `#endif` Directives

A short example will clarify what conditional compilation does. Consider the following:

```
#ifdef MAVIS
     #include "horse.h"  // gets done if MAVIS is #defined
     #define  STABLES     5
#else
     #include "cow.h"    // gets done if MAVIS isn't #defined
     #define  STABLES   15
#endif
```

Here we've used the indentation allowed by newer implementations and by the ANSI standard. If you have an older implementation, you might have to move all the directives, or at least the # symbols (see the next example), to flush left:

```
#ifdef MAVIS
#   include "horse.h"  /* gets done if MAVIS is #defined    */
#   define  STABLES     5
#else
#   include "cow.h"    /* gets done if MAVIS isn't #defined */
#   define  STABLES   15
#endif
```

The `#ifdef` directive says that if the following identifier (`MAVIS`) has been defined by the preprocessor, follow all the directives and compile all the C code up to the next `#else` or `#endif`, whichever comes first. If there is an `#else`, everything from the `#else` to the `#endif` is done if the identifier isn't defined.

The form `#ifdef #else` is much like that of the C `if else`. The main difference is that the preprocessor doesn't recognize the braces (`{}`) method of marking a block, so it uses the `#else` (if any) and the `#endif` (which must be present) to mark blocks of directives. These conditional structures can be nested. You can use these directives to mark blocks of C statements, too, as Listing 16.9 illustrates.

Listing 16.9 **The `ifdef.c` Program**

```
/* ifdef.c -- uses conditional compilation */
#include <stdio.h>
#define JUST_CHECKING
#define LIMIT 4

int main(void)
{
    int i;
    int total = 0;

    for (i = 1; i <= LIMIT; i++)
```

```
    {
        total += 2*i*i + 1;
#ifdef JUST_CHECKING
        printf("i=%d, running total = %d\n", i, total);
#endif
    }
    printf("Grand total = %d\n", total);

    return 0;
}
```

Compiling and running the program as shown produces this output:

```
i=1, running total = 3
i=2, running total = 12
i=3, running total = 31
i=4, running total = 64
Grand total = 64
```

If you omit the `JUST_CHECKING` definition (or enclose it inside a C comment, or use `#undef` to undefine it) and recompile the program, only the final line is displayed. You can use this approach, for example, to help in program debugging. Define `JUST_CHECKING` and use a judicious selection of `#ifdef`s, and the compiler will include program code for printing intermediate values for debugging. After everything is working, you can remove the definition and recompile. If, later, you find that you need the information again, you can reinsert the definition and avoid having to retype all the extra print statements. Another possibility is using `#ifdef` to select among alternative chunks of codes suited for different C implementations.

The `#ifndef` Directive

The `#ifndef` directive can be used with `#else` and `#endif` in the same way that `#ifdef` is. The `#ifndef` asks whether the following identifier is *not* defined; `#ifndef` is the negative of `#ifdef`. This directive is often used to define a constant if it is not already defined. Here's an example:

```
/* arrays.h  */
#ifndef SIZE
   #define SIZE 100
#endif
```

(Older implementations might not permit indenting the `#define` directive.)

Typically, this idiom is used to prevent multiple definitions of the same macro when you include several header files, each of which may contain a definition. In this case, the definition in the first header file included becomes the active definition and subsequent definitions in other header files are ignored.

Here's another use. Suppose we place the line

```
#include "arrays.h"
```

at the head of a file. This results in `SIZE` being defined as 100. But placing

```
#define SIZE 10
#include "arrays.h"
```

at the head sets `SIZE` to 10. Here, `SIZE` is defined by the time the lines in `arrays.h` are processed, so the `#define SIZE 100` line is skipped. You might do this, for example, to test a program using a smaller array size. When it works to your satisfaction, you can remove the `#define SIZE 10` statement and recompile. That way, you never have to worry about modifying the header array itself.

The `#ifndef` directive is commonly used to prevent multiple inclusions of a file. That is, header files usually are set up along the following lines:

```
/* things.h */
#ifndef THINGS_H_
     #define THINGS_H_
     /* rest of include file */
#endif
```

Suppose this file somehow got included several times. The first time the preprocessor encounters this include file, `THINGS_H_` is undefined, so the program proceeds to define `THINGS_H_` and to process the rest of the file. The next time the preprocessor encounters this file, `THINGS_H_` is defined, so the preprocessor skips the rest of the file.

Why would you include a file more than once? The most common reason is that many include files include other files, so you may include a file explicitly that another include file has already included. Why is this a problem? Some items that appear in include files, such as declarations of structure types, can appear only once in a file. The standard C header files use the `#ifndef` technique to avoid multiple inclusions. One problem is to make sure the identifier you are testing hasn't been defined elsewhere. Vendors typically solve this by using the filename as the identifier, using uppercase, replacing periods with an underscore, and using an underscore (or, perhaps, two underscores) as a prefix and a suffix. If you check your `stdio.h` header file, for example, you'll probably find something similar to this:

```
#ifndef _STDIO_H
#define _STDIO_H
// contents of file
#endif
```

You can do something similar. However, you should avoid using the underscore as a prefix because the standard says such usage is reserved. You wouldn't want to accidentally define a macro that conflicts with something in the standard header files. Listing 16.10 uses `#ifndef` to provide multiple-inclusion protection for the header file from Listing 16.6.

Listing 16.10 **The `names.h` Header File**

```
// names.h --revised with include protection

#ifndef NAMES_H_
#define NAMES_H_

// constants
#define SLEN 32

// structure declarations
struct names_st
{
    char first[SLEN];
    char last[SLEN];
};

// typedefs
typedef struct names_st names;

// function prototypes
void get_names(names *);
void show_names(const names *);
char * s_gets(char * st, int n);

#endif
```

You can test this header file with the program shown in Listing 16.11. This program should work correctly when using the header file shown in Listing 16.10, and it should fail to compile if you remove the `#ifndef` protection from Listing 16.10.

Listing 16.11 **The `doubincl.c` Program**

```
// doubincl.c -- include header twice
#include <stdio.h>
#include "names.h"
#include "names.h"   // accidental second inclusion

int main()
{
    names winner = {"Less", "Ismoor"};
    printf("The winner is %s %s.\n", winner.first,
            winner.last);
    return 0;
}
```

The `#if` and `#elif` Directives

The `#if` directive is more like the regular C `if`. It is followed by a constant integer expression that is considered true if nonzero, and you can use C's relational and logical operators with it:

```
#if SYS == 1
#include "ibm.h"
#endif
```

You can use the `#elif` directive (not available in some older implementations) to extend an `if-else` sequence. For example, you could do this:

```
#if SYS == 1
     #include "ibmpc.h"
#elif SYS == 2
     #include "vax.h"
#elif SYS == 3
     #include "mac.h"
#else
     #include "general.h"
#endif
```

Newer implementations offer a second way to test whether a name is defined. Instead of using

```
#ifdef VAX
```

you can use this form:

```
#if defined (VAX)
```

Here, `defined` is a preprocessor operator that returns `1` if its argument is `#defined` and `0` otherwise. The advantage of this newer form is that it can be used with `#elif`. Using it, you can rewrite the previous example this way:

```
#if defined (IBMPC)
     #include "ibmpc.h"
#elif defined (VAX)
     #include "vax.h"
#elif defined (MAC)
     #include "mac.h"
#else
     #include "general.h"
#endif
```

If you were using these lines on, say, a VAX, you would have defined `VAX` somewhere earlier in the file with this line:

```
#define VAX
```

One use for these conditional compilation features is to make a program more portable. By changing a few key definitions at the beginning of a file, you can set up different values and include different files for different systems.

Predefined Macros

The C standard specifies several predefined macros, which Table 16.1 lists.

Table 16.1 **Predefined Macros**

Macro	Meaning
`__DATE__`	A character string literal in the form "Mmm dd yyyy" representing the date of preprocessing, as in Nov 23 2013
`__FILE__`	A character string literal representing the name of the current source code file
`__LINE__`	An integer constant representing the line number in the current source code file
`__STDC__`	Set to 1 to indicate the implementation conforms to the C Standard
`__STDC_HOSTED__`	Set to 1 for a hosted environment; 0 otherwise
`__STDC_VERSION__`	`For C99, set to 199901L; for C11, set to 201112L`
`__TIME__`	The time of translation in the form "hh:mm:ss"

While we're discussing predefined identifiers, the C99 standard provides for one called `__func__`. It expands to a string representing the name of the function containing the identifier. For this reason, the identifier has to have function scope, whereas macros essentially have file scope. Therefore, `__func__` is a C language predefined identifier rather than a predefined macro.

Listing 16.12 shows several of these predefined identifiers in use. Note that some of them are C99 additions, so a pre-C99 compiler might not accept them. For GCC you may have to use the `-std=c99` or the `-std=c11` flag.

Listing 16.12 **The `predef.c` Program**

```
// predef.c -- predefined identifiers
#include <stdio.h>
void why_me();

int main()
{
    printf("The file is %s.\n", __FILE__);
```

```
    printf("The date is %s.\n", __DATE__);
    printf("The time is %s.\n", __TIME__);
    printf("The version is %ld.\n", 3TDC_VERSION__);
    printf("This is line %d.\n", __LINE__);
    printf("This function is %s\n", __func__);
    why_me();

    return 0;
}


void why_me()
{
    printf("This function is %s\n", __func__);
    printf("This is line %d.\n", __LINE__);
}
```

Here's a sample run:

```
The file is predef.c.
The date is Sep 23 2013.
The time is 22:01:09.
The version is 201112.
This is line 11.
This function is main
This function is why_me
This is line 21.
```

`#line` and `#error`

The `#line` directive lets you reset the line numbering and the filename as reported by the `__LINE__` and `__FILE__` macros. You can use `#line` like this:

```
#line 1000     // reset current line number to 1000
#line 10  "cool.c"  // reset line number to 10, file name to cool.c
```

The `#error` directive causes the preprocessor to issue an error message that includes any text in the directive. If possible, the compilation process should halt. You could use the directive like this:

```
#if __STDC_VERSION__ != 201112L
#error Not C11

#endif
```

Attempting to compile the program could then produce results like this:

```
$ gcc newish.c
newish.c:14:2: error: #error Not C11
$ gcc -std=c11 newish.c
$
```

The compilation process failed when the compiler used an older standard and succeeded when it used the C11 standard.

#pragma

Modern compilers have several settings that can be modified by command-line arguments or by using an IDE menu. The #pragma lets you place compiler instructions in the source code. For example, while C99 was being developed, it was referred to as C9X, and one compiler used the following pragma to turn on C9X support:

```
#pragma c9x on
```

Generally, each compiler has its own set of pragmas. They might be used, for example, to control the amount of memory set aside for automatic variables or to set the strictness of error checking or to enable nonstandard language features. The C99 standard does provide for three standard pragmas of rather technical nature that we won’t discuss here.

C99 also provides the _Pragma preprocessor operator. It converts a string into a regular pragma. For example,

```
_Pragma("nonstandardtreatmenttypeB on")
```

is equivalent to the following:

```
#pragma nonstandardtreatmenttypeB on
```

Because the operator doesn’t use the # symbol, you can use it as part of a macro expansion:

```
#define PRAGMA(X) _Pragma(#X)
#define LIMRG(X) PRAGMA(STDC CX_LIMITED_RANGE  X)
```

Then you can use code like this:

```
LIMRG ( ON )
```

Incidentally, the following definition doesn’t work, although it looks as if it might:

```
#define LIMRG(X) _Pragma(STDC CX_LIMITED_RANGE  #X)
```

The problem is that it relies on string concatenation, but the compiler doesn’t concatenate strings until after preprocessing is complete.

The _Pragma operator does a complete job of “destringizing”; that is, escape sequences in a string are converted to the character represented. Thus,

```
_Pragma("use_bool \"true \"false")
```

becomes

```
#pragma use_bool "true "false
```

Generic Selection (C11)

In programming, the term *generic programming* indicates code that is not specific to a particular type but which, once a type is specified, can be translated into code for that type. C++, for example, lets you create generic algorithms in the form of templates that the compiler can then use to instantiate code automatically for a specified type. C doesn't have anything quite like that. However, C11 adds a new sort of expression, called a *generic selection expression*, that can be used to select a value on the basis of the type of an expression, that is, on whether the expression type is `int`, `double`, or some other type. The generic selection expression is not a preprocessor statement, but its usual use is a part of a `#define` macro definition that has some aspects of generic programming.

A generic selection expression looks like this:

```
_Generic(x, int: 0, float: 1, double: 2, default: 3)
```

Here `_Generic` is a new C11 keyword. The parentheses following `_Generic` contain several comma-separated terms. The first term is an expression, and each remaining item is a type followed by a colon followed by a value, such as `float: 1`. The type of the first term is matched to one of the labels, and the value of the whole expression is the value following the matched label. For example, suppose `x` in the preceding expression is a type `int` variable. Then the type of `x` matches the `int:` label, making `0` the value of the whole expression. If the type doesn't match a label, the value associated with the `default:` label is used for the whole expression. A generic selection statement is a little like a `switch` statement, except that the type of an expression rather than the value of an expression is matched to a label.

Let's look at an example combining a generic selection statement with a macro definition:

```
#define MYTYPE(X) _Generic((X),\
    int: "int",\
    float : "float",\
    double: "double",\
    default: "other"\
)
```

Recall that a macro has to be defined on one logical line, but you can use a `\` to break the one logical line into multiple physical lines. In this case, the generic selection expression evaluates to a string. For example, the macro invocation `MYTYPE(5)` evaluates to the string `"int"` because the type for the value `5` matches the `int:` label. Listing 16.13 illustrates this macro further.

Listing 16.13 **The `predef.c` Program**

```
//  mytype.c

#include <stdio.h>

#define MYTYPE(X) _Generic((X),\
    int: "int",\
    float : "float",\
    double: "double",\
    default: "other"\
)

int main(void)
{
    int d = 5;

    printf("%s\n", MYTYPE(d));     // d is type int
    printf("%s\n", MYTYPE(2.0*d)); // 2.0* d is type double
    printf("%s\n", MYTYPE(3L));    // 3L is type long
    printf("%s\n", MYTYPE(&d));    // &d is type int *
    return 0;
 }
```

Here is the output:

```
int
double
other
other
```

The final two instances of `MYTYPE()` use types without matching labels, so the default string is used. We could have used more type labels to extend the capabilities of the macro, but the example serves to illustrate how `_Generic`-based macros work.

When evaluating a generic selection expression, the program does not evaluate the first term; it only determines the type. And the only expression it does evaluate is the one with the matching label.

You can use `_Generic` to define macros that act like type-independent ("generic") functions. The section later in this chapter about the `math` library provides an example.

Inline Functions (C99)

Normally, a function call has overhead. That means it takes execution time to set up the call, pass arguments, jump to the function code, and return. As you've seen, you can use a macro to place code inline, thus avoiding that overhead. C99, borrowing from C++ (but not always exactly), added another approach, *inline functions*. From the name, you might expect that an inline function replaces a function call with inline code, but you would be misled. What the C99 and C11 standards actually say is this: "Making a function an inline function suggests that calls to the function be as fast as possible. The extent to which such suggestions are effective is implementation-defined." So making a function an inline function may cause the compiler to replace the function call with inline code and/or perform some other sorts of optimizations, or it may have no effect.

There are different ways to create inline function definitions. The standard says that a function with internal linkage can be made inline and that the definition for the inline function must be in the same file in which the function is used. So a simple approach is to use the `inline` function specifier along with the `static` storage-class specifier. Usually, inline functions are defined before the first use in a file, so the definition also acts as a prototype. That is, the code would look like this:

```
#include <stdio.h>
inline static void eatline()   // inline definition/prototype
{
    while (getchar() != '\n')
         continue;
}

int main()
{
...
    eatline();                 // function call
...
}
```

Seeing the inline declaration, the compiler could choose, for example, to replace the `eatline()` function call with the function body. That is, the effect could end up the same as if you had written this code instead:

```
#include <stdio.h>
inline static void eatline()    // inline definition/prototype
{
    while (getchar() != '\n')
         continue;
}

int main()
{
...
    while (getchar() != '\n')   // function call replaced
         continue;
...
}
```

Because an inline function doesn't have a separate block of code set aside for it, you can't take its address. (Actually, you can take the address, but then the compiler will generate a non-inline function.) Also, an inline function may not show up in a debugger.

An inline function should be short. For a long function, the time consumed in calling the function is short compared to the time spent executing the body of the function, so there is no great savings in time using an inline version.

For the compiler to make inline optimizations, it has to know the contents of the function definition. This means the inline function definition has to be in the same file as the function call. For this reason, an inline function ordinarily has internal linkage. Therefore, if you have a multifile program, you need an inline definition in each file that calls the function. The simplest way to accomplish this is to put the inline function definition in a header file and then include the header file in those files that use the function.

```
// eatline.h
#ifndef EATLINE_H_
#define EATLINE_H_
inline static void eatline()
{
    while (getchar() != '\n')
         continue;
}
#endif
```

An inline function is an exception to the rule of not placing executable code in a header file. Because the inline function has internal linkage, defining one in several files doesn't cause problems.

C, unlike C++, also allows a mixture of inline definitions with external definitions (function definitions with external linkage). For example, a program has the following three files:

```
//file1.c
...
inline static double square(double);
double square(double x) { return x * x; }

int main()
{
    double q = square(1.3);
...

//file2.c
...
double square(double x) { return (int) (x*x); }
void spam(double v)
{
    double kv = square(v);
```

```
...

//file3.c
...
inline double square(double x) { return (int) (x * x + 0.5); }
void masp(double w)
{
    double kw = square(w);
...
```

One has an `inline static` definition, as before. One has an ordinary function definition, hence having external linkage. And one has an `inline` definition that omits the `static` qualifier.

What happens? The `spam()` function in `file2.c` uses the `square()` definition in that file. That definition, having external linkage, is visible to the other files, but `main()` in `file1.c` uses the local `static` definition of `square()`. Because this definition also is `inline`, the compiler may (or may not) optimize the coding, perhaps inlining it. Finally, for `file3.c`, the compiler is free to use either (or both!) the inline definition of `file3.c` or the external linkage definition from `file2.c`. If you omit `static` from an `inline` definition, as in `file3`.c, the `inline` definition is considered as an alternative that could be used instead of the external definition.

Note that GCC implemented inline functions prior to C99 using somewhat different rules, so the GCC interpretation of `inline` can depend on which compiler flags you use.

`_Noreturn` Functions (C11)

When C99 added the `inline` keyword, that keyword became the sole example of a function specifier. (The keywords `extern` and `static` are termed storage-class specifiers and can be applied to data objects as well as to functions.) C11 adds a second function specifier, `_Noreturn`, to indicate a function that, upon completion, does not return to the calling function. The `exit()` function is an example of a `_Noreturn` function, for once `exit()` is called, the calling function never resumes. Note that this is different from the `void` return type. A typical `void` function does return to the calling function; it just doesn't provide an assignable value.

The purpose of `_Noreturn` is to inform the user and the compiler that a particular function won't return control to the calling program. Informing the user helps to prevent misuse of the function, and informing the compiler may enable it to make some code optimizations.

The C Library

Originally, there was no official C library. Later, a de facto standard emerged based on the Unix implementation of C. The ANSI C committee, in turn, developed an official standard library, largely based on the de facto standard. Recognizing the expanded C universe, the committee then sought to redefine the library so that it could be implemented on a wide variety of systems.

We've already discussed some I/O functions, character functions, and string functions from the library. In this chapter, we'll browse through several more. First, however, let's talk about how to use a library.

Gaining Access to the C Library

How you gain access to the C library depends on your implementation, so you need to see how the more general statements apply to your system. First, there are often several different places to find library functions. For example, `getchar()` is usually defined as a macro in the file `stdio.h`, but `strlen()` is usually kept in a library file. Second, different systems have different ways to reach these functions. The following sections outline three possibilities.

Automatic Access

On many systems, you just compile the program and the more common library functions are made available automatically.

Keep in mind that you should declare the function type for functions you use. Usually you can do that by including the appropriate header file. User manuals describing library functions tell you which files to include. On some older systems, however, you might have to enter the function declarations yourself. Again, the user manual indicates the function type. Also, Appendix B, "Reference Section," summarizes the ANSI C library, grouping functions by header file.

In the past, header filenames have not been consistent among different implementations. The ANSI C standard groups the library functions into families, with each family having a specific header file for its function prototypes.

File Inclusion

If a function is defined as a macro, you can include the file containing its definition by using the `#include` directive. Often, similar macros are collected in an appropriately named header file. For example, since the introduction of ANSI C, C compilers come with a `ctype.h` file containing several macros that determine the nature of a character: uppercase, digit, and so forth.

Library Inclusion

At some stage in compiling or linking a program, you might have to specify a library option. Even a system that automatically checks its standard library can have other libraries of functions less frequently used. These libraries have to be requested explicitly by using a compile-time option. Note that this process is distinct from including a header file. A header file provides a function declaration or prototype. The library option tells the system where to find the function code. Clearly, we can't go through all the specifics for all systems, but these discussions should alert you to what you should look for.

Using the Library Descriptions

We haven't the space to discuss the complete library, but we will look at some representative examples. First, though, let's take a look at documentation.

You can find function documentation in several places. Your system might have an online manual, and integrated environments often have online help. C vendors may supply printed user's guides describing library functions, or they might place equivalent material on a reference CD-ROM or online. Several publishers have issued reference manuals for C library functions. Some are generic in nature, and some are targeted toward specific implementations. And, as mentioned earlier, Appendix B in this book provides a summary.

The key skill you need in reading the documentation is interpreting function headings. The idiom has changed with time. Here, for instance, is how `fread()` is listed in older Unix documentation:

```
#include <stdio.h>

fread(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;
```

First, the proper `include` file is given. No type is given for `fread()`, `ptr`, `sizeof(*ptr)`, or `nitems`. By default, in the old days, they were taken to be type `int`, but the context makes it clear that `ptr` is a pointer. (In C's early days, pointers were handled as integers.) The `stream` argument is declared as a pointer to `FILE`. The declaration makes it look as though you are supposed to use the `sizeof` operator as the second argument. Actually, it's saying that the value of this argument should be the size of the object pointed to by `ptr`. Often, you would use `sizeof` as illustrated, but any type `int` value satisfies the syntax.

Later, the form changed to this:

```
#include <stdio.h>

int fread(ptr, size, nitems, stream;)
char *ptr;
int size, nitems;
FILE *stream;
```

Now all types are given explicitly, and `ptr` is treated as a pointer-to-`char`.

The ANSI C90 standard provides the following description:

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

First, it uses the new prototype format. Second, it changes some types. The `size_t` type is defined as the unsigned integer type that the `sizeof` operator returns. Usually, it is either `unsigned int` or `unsigned long`. The `stddef.h` file contains a `typedef` or a `#define` for `size_t`, as do several other files, including `stdio.h`, typically by including `stddef.h`. Many functions, including `fread()`, often incorporate the `sizeof` operator as part of an actual argument. The `size_t` type makes that formal argument match this common usage.

Also, ANSI C uses pointer-to-`void` as a kind of generic pointer for situations in which pointers to different types may be used. For example, the actual first argument to `fread()` may be a pointer to an array of `double` or to a structure of some sort. If the actual argument is, say, a pointer-to-array-of-20-`double` and the formal argument is pointer-to-`void`, the compiler makes the appropriate type version without complaining about type clashes.

More recently, the C99/C11 standards incorporate the new keyword `restrict` into the description:

```
#include <stdio.h>
size_t fread(void * restrict ptr, size_t size,
             size_t nmemb, FILE * restrict stream);
```

Now let's turn to some specific functions.

The Math Library

The math library contains many useful mathematical functions. The `math.h` header file provides the function declarations or prototypes for these functions. Table 16.2 lists several functions declared in `math.h`. Note that all angles are measured in radians (one radian = 180/π = 57.296 degrees). Reference Section V, "The Standard ANSI C Library with C99 Additions," supplies a complete list of the functions specified by the C99 standard.

Table 16.2 **Some ANSI C Standard Math Functions**

Prototype	**Description**
`double acos(double x)`	Returns the angle (0 to π radians) whose cosine is `x`
`double asin(double x)`	Returns the angle (—π/2 to π/2 radians) whose sine is `x`
`double atan(double x)`	Returns the angle (–π/2 to π/2 radians) whose tangent is `x`
`double atan2(double y, double x)`	Returns the angle (–π to π radians) whose tangent is `y / x`
`double cos(double x)`	Returns the cosine of `x` (`x` in radians)
`double sin(double x)`	Returns the sine of `x` (`x` in radians)
`double tan(double x)`	Returns the tangent of `x` (`x` in radians)
`double exp(double x)`	Returns the exponential function of `x` (e^x)

Prototype	Description
`double log(double x)`	Returns the natural logarithm of `x`
`double log10(double x)`	Returns the base 10 logarithm of `x`
`double pow(double x, double y)`	Returns `x` to the `y` power
`double sqrt(double x)`	Returns the square root of `x`
`double cbrt(double x)`	Returns the cube root of `x`
`double ceil(double x)`	Returns the smallest integral value not less than `x`
`double fabs(double x)`	Returns the absolute value of `x`
`double floor(double x)`	Returns the largest integral value not greater than `x`

A Little Trigonometry

Let's use the math library to solve a common problem: converting from x/y coordinates to magnitudes and angles. For example, suppose you draw, on a grid work, a line that transverses 4 units horizontally (the `x` value) and 3 units vertically (the `y` value). What is the length (magnitude) of the line and what is its direction? Trigonometry tells us the following:

magnitude = square root $(x^2 + y^2)$

and

```
angle = arctangent (y/x)
```

The math library provides a square root function and a couple arctangent functions, so you can express this solution in a C program. The square root function, called `sqrt()`, takes a `double` argument and returns the argument's square root, also as a type `double` value.

The `atan()` function takes a double argument—the tangent—and returns the angle having that value as its tangent. Unfortunately, the `atan()` function is confused by, say, a line with `x` and `y` values of `–5` and `–5`. Because (–5)/(–5) is 1, `atan()` would report 45°, the same as it does for a line with `x` and `y` values of `5` and `5`. In other words, `atan()` doesn't distinguish between a line of a given angle and one 180° in the opposite direction. (Actually, `atan()` reports in radians, not degrees; we'll discuss that conversion soon.)

Fortunately, the C library also provides the `atan2()` function. It takes two arguments: the x value and the y value. That way, the function can examine the signs of x and y and figure out the correct angle. Like `atan()`, `atan2()` returns the angle in radians. To convert to degrees, multiply the resulting angle by 180 and divide by pi. You can have the computer calculate pi by using the expression `4 * atan(1)`. Listing 16.14 illustrates these steps. It also gives you a chance to review structures and the `typedef` facility.

Listing 16.14 **The `rect_pol.c` Program**

```
/* rect_pol.c -- converts rectangular coordinates to polar */
#include <stdio.h>
#include <math.h>

#define RAD_TO_DEG (180/(4 * atan(1)))

typedef struct polar_v {
    double magnitude;
    double angle;
} Polar_V;

typedef struct rect_v {
    double x;
    double y;
} Rect_V;

Polar_V rect_to_polar(Rect_V);

int main(void)
{
    Rect_V input;
    Polar_V result;

    puts("Enter x and y coordinates; enter q to quit:");
    while (scanf("%lf %lf", &input.x, &input.y) == 2)
    {
        result = rect_to_polar(input);
        printf("magnitude = %0.2f, angle = %0.2f\n",
                result.magnitude, result.angle);
    }
    puts("Bye.");

    return 0;
}

Polar_V rect_to_polar(Rect_V rv)
{
    Polar_V pv;

    pv.magnitude = sqrt(rv.x * rv.x + rv.y * rv.y);
    if (pv.magnitude == 0)
        pv.angle = 0.0;
    else
        pv.angle = RAD_TO_DEG * atan2(rv.y, rv.x);

    return pv;
}
```

Here's a sample run:

```
Enter x and y coordinates; enter q to quit:
10 10
magnitude = 14.14, angle = 45.00
-12 -5
magnitude = 13.00, angle = -157.38
q
Bye.
```

If, when you compile, you get a message such as

```
Undefined:     _sqrt
```

or

```
'sqrt': unresolved external
```

or something similar, your compiler-linker is not finding the math library. Unix systems may require that you instruct the linker to search the math library by using the `-lm` flag:

```
cc rect_pol.c –lm
```

Note that the `–lm` flag comes at the end of the command. That's because the linker comes into play after the compiler compiles the C file. The GCC compiler on Linux may behave in the same fashion:

```
gcc rect_pol.c -lm
```

Type Variants

The basic floating-point math functions take type `double` arguments and return a type `double` value. You can pass them type `float` or type `long double` arguments, and the functions still work because the arguments are converted to type `double`. That's convenient but not necessarily optimal. If double precision isn't needed, the computations might be faster if done using single precision `float` values. And type `long double` value will lose precision when passed to a type `double` parameter; the value might not even be representable. To deal with these potential problems, the C standard provides type `float` and type `long double` versions of the standard functions, using an `f` or an `l` ("ell") suffix on the function name. So `sqrtf()` is a type `float` version of `sqrt()`, and `sqrtl()` is a type `long double` version.

The C11 addition of the generic selection expression lets us define a generic macro that chooses the most appropriate version of a math function based on the argument type. Listing 16.15 shows two approaches.

Listing 16.15 **The `generic.c` Program**

```
//  generic.c  -- defining generic macros

#include <stdio.h>
```

```
#include <math.h>
#define RAD_TO_DEG (180/(4 * atanl(1)))

// generic square root function
#define SQRT(X) _Generic((X),\
    long double: sqrtl, \
    default: sqrt, \
    float: sqrtf)(X)

// generic sine function, angle in degrees
#define SIN(X) _Generic((X),\
    long double: sinl((X)/RAD_TO_DEG),\
    default:     sin((X)/RAD_TO_DEG),\
    float:       sinf((X)/RAD_TO_DEG)\
)

int main(void)
{
    float x = 45.0f;
    double xx = 45.0;
    long double xxx =45.0L;

    long double y = SQRT(x);
    long double yy= SQRT(xx);
    long double yyy = SQRT(xxx);
    printf("%.17Lf\n", y);   // matches float
    printf("%.17Lf\n", yy);  // matches default
    printf("%.17Lf\n", yyy); // matches long double
    int i = 45;
    yy = SQRT(i);            // matches default
    printf("%.17Lf\n", yy);
    yyy= SIN(xxx);           // matches long double
    printf("%.17Lf\n", yyy);

    return 0;
}
```

Here is the output:

```
6.70820379257202148
6.70820393249936942
6.70820393249936909
6.70820393249936942
0.70710678118654752
```

As you can see, `SQRT(i)` has the same return value as `SQRT(xx)`, as both argument types (`int` and `double`) correspond to the `default` label.

A point of interest is how to get a macro using `_Generic` to act like a function. The definition for `SIN()` takes perhaps the more obvious approach: Each labeled value is a function call, so the value of the `_Generic` expression is a particular function call, such as `sinf((X)/RAD_TO_DEG)`, with the argument to `SIN()` replacing the `X`.

The `SQRT()` definition is perhaps more elegant. In this case the value of the `_Generic` expression is the name of a function, such as `sinf`. The name of a function is replaced by the address of the function, so the value of the `_Generic` expression is a pointer to a function. However, the entire `_Generic` expression is followed by `(X)`, and the combination of *function-pointer(argument)* calls the pointed-to function with the indicated argument.

In short, for `SIN()`, the function call is inside the generic selection expression, while for `SQRT()` the generic selection expression evaluates to a pointer, which is then used to invoke a function.

The `tgmath.h` Library (C99)

The C99 standard provides a `tgmath.h` header file that defines type-generic macros similar in effect to those in Listing 16.15. If a `math.h` function is defined for each of the three types `float`, `double`, and `long double`, the `tgmath.h` file creates a type-generic macro with the same name as the double version. For instance, it defines a `sqrt()` macro that expands to the `sqrtf()`, `sqrt()`, or `sqrtl()` function, depending on the type of argument provided. In other words, the `sqrt()` macro behaves like the `SQRT()` macro in Listing 16.15.

If the compiler supports complex arithmetic, it supports the `complex.h` header file, which declares complex analogs to math functions. For example, it declares `csqrtf()`, `csqrt()`, and `csqrtl()`, which return the complex square roots of type `float complex`, `double complex`, and `long double complex`, respectively. When such support is provided, the `tgmath.h` `sqrt()` macro also can expand to the corresponding complex square root function.

If you want to, say, invoke the `sqrt()` function instead of the `sqrt()` macro even though `tgmath.h` is included, you can enclose the function name in parentheses:

```
#include <tgmath.h>
...
    float x = 44.0;
    double y;
    y = sqrt(x);   // invoke macro, hence sqrtf(x)
    y = (sqrt)(x); // invoke function sqrt()
```

This works because a function-like macro name has to be followed by an opening parenthesis, which using enclosing parentheses circumvents. Otherwise, aside from order of operations, parentheses don't affect enclosed expressions, so enclosing a function name in parentheses still results in a function call. Indeed, because of C's strangely contradictory rules about function pointers, you also can also use `(*sqrt)()` to invoke the `sqrt()` function.

What C11 adds with `_Generic` expressions is a simple way to implement the macros of `tgmath.h` without resorting to mechanisms outside the C standard.

The General Utilities Library

The general utilities library contains a grab bag of functions, including a random-number generator, searching and sorting functions, conversion functions, and memory-management functions. You've already seen `rand()`, `srand()`, `malloc()`, and `free()` in Chapter 12, "Storage Classes, Linkage, and Memory Management." Under ANSI C, prototypes for these functions exist in the `stdlib.h` header file. Appendix B, Reference Section V lists all the functions in this family; we'll take a closer look at a few of them now.

The `exit()` and `atexit()` Functions

We've already used `exit()` explicitly in several examples. In addition, the `exit()` function is invoked automatically upon return from `main()`. The ANSI standard has added a couple nice features that we haven't used yet. The most important addition is that you can specify particular functions to be called when `exit()` executes. The `atexit()` function provides this feature by registering the functions to be called on exit; the `atexit()` function takes a function pointer as its argument. Listing 16.16 shows how this works.

Listing 16.16 **The `byebye.c` Program**

```
/* byebye.c -- atexit() example */
#include <stdio.h>
#include <stdlib.h>
void sign_off(void);
void too_bad(void);

int main(void)
{
    int n;

    atexit(sign_off);    /* register the sign_off() function */
    puts("Enter an integer:");
    if (scanf("%d",&n) != 1)
    {
        puts("That's no integer!");
        atexit(too_bad); /* register the too_bad()  function */
        exit(EXIT_FAILURE);
    }
    printf("%d is %s.\n", n,  (n % 2 == 0)? "even" : "odd");

    return 0;
}

void sign_off(void)
{
    puts("Thus terminates another magnificent program from");
```

```
    puts("SeeSaw Software!");
}

void too_bad(void)
{
    puts("SeeSaw Software extends its heartfelt condolences");
    puts("to you upon the failure of your program.");
}
```

Here's one sample run:

```
Enter an integer:
212
212 is even.
Thus terminates another magnificent program from
SeeSaw Software!
```

You might not see the final two lines if you are running in an IDE.

Here's a second run:

```
Enter an integer:
what?
That's no integer!
SeeSaw Software extends its heartfelt condolences
to you upon the failure of your program.
Thus terminates another magnificent program from
SeeSaw Software!
```

You might not see the final four lines if you are running in an IDE.

Let's look at two main areas: the use of the `atexit()` and `exit()` arguments.

Using `atexit()`

Here's a function that uses function pointers! To use the `atexit()` function, simply pass it the address of the function you want called on exit. Because the name of a function acts as an address when used as a function argument, use `sign_off` or `too_bad` as the argument. Then `atexit()` registers that function in a list of functions to be executed when `exit()` is called. ANSI guarantees that you can place at least 32 functions on the list. Each function is added with a separate call to `atexit()`. When the `exit()` function is finally called, it executes these functions, with the last function added being executed first.

Notice that both `sign_off()` and `too_bad()` were called when input failed, but only `sign_off()` was called when input worked. That's because the `if` statement registers `too_bad()` only if input fails. Also note that the last function registered was the first called.

The functions registered by `atexit()` should, like `sign_off()` and `too_bad()`, be type `void` functions taking no arguments. Typically, they would perform housekeeping tasks, such as updating a program-monitoring file or resetting environmental variables.

Note that `sign_off()` is called even when `exit()` is not called explicitly; that's because `exit()` is called implicitly when `main()` terminates.

Using `exit()`

After `exit()` executes the functions specified by `atexit()`, it does some tidying of its own. It flushes all output streams, closes all open streams, and closes temporary files created by calls to the standard I/O function `tmpfile()`. Then `exit()` returns control to the host environment and, if possible, reports a termination status to the environment. Traditionally, Unix programs have used 0 to indicate successful termination and nonzero to report failure. Unix return codes don't necessarily work with all systems, so ANSI C defined a macro called `EXIT_FAILURE` that can be used portably to indicate failure. Similarly, it defined `EXIT_SUCCESS` to indicate success, but `exit()` also accepts 0 for that purpose. Under ANSI C, using the `exit()` function in a nonrecursive `main()` function is equivalent to using the keyword `return`. However, `exit()` also terminates programs when used in functions other than `main()`.

The `qsort()` Function

The "quick sort" method is one of the most effective sorting algorithms, particularly for larger arrays. Developed by C.A.R. Hoare in 1962, it partitions arrays into ever smaller sizes until the element level is reached. First, the array is divided into two parts, with every value in one partition being less than every value in the other partition. This process continues until the array is fully sorted.

The name for the C implementation of the quick sort algorithm is `qsort()`. The `qsort()` function sorts an array of data objects. It has the following ANSI prototype:

```
void qsort (void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
```

The first argument is a pointer to the beginning of the array to be sorted. ANSI C permits any data pointer type to be typecast to a pointer-to-`void`, thus permitting the first actual argument to `qsort()` to refer to any kind of array.

The second argument is the number of items to be sorted. The prototype converts this value to type `size_t`. As you may recall from several previous mentions, `size_t` is the integer type returned by the `sizeof` operator and is defined in the standard header files.

Because `qsort()` converts its first argument to a `void` pointer, `qsort()` loses track of how big each array element is. To compensate, you must tell `qsort()` explicitly the size of the data object. That's what the third argument is for. For example, if you are sorting an array of type `double`, you would use `sizeof(double)` for this argument.

Finally, `qsort()` requires a pointer to the function to be used to determine the sorting order. The comparison function should take two arguments: pointers to the two items being compared. It should return a positive integer if the first item should follow the second value, zero if the two items are the same, and a negative integer if the second item should follow the first. The `qsort()` will use this function, passing it pointer values that it calculates from the other information given to it.

The form the comparison function must take is set forth in the `qsort()` prototype for the final argument:

```
int (*compar)(const void *, const void *)
```

This states that the final argument is a pointer to a function that returns an `int` and that takes two arguments, each of which is a pointer to type `const void`. These two pointers point to the items being compared.

Listing 16.17 and the discussion following it illustrate how to define a comparison function and how to use `qsort()`. The program creates an array of random floating-point values and sorts the array.

Listing 16.17 **The `qsorter.c` Program**

```
/* qsorter.c -- using qsort to sort groups of numbers */
#include <stdio.h>
#include <stdlib.h>

#define NUM 40
void fillarray(double ar[], int n);
void showarray(const double ar[], int n);
int mycomp(const void * p1, const void * p2);

int main(void)
{
    double vals[NUM];
    fillarray(vals, NUM);
    puts("Random list:");
    showarray(vals, NUM);
    qsort(vals, NUM, sizeof(double), mycomp);
    puts("\nSorted list:");
    showarray(vals, NUM);
    return 0;
}

void fillarray(double ar[], int n)
{
    int index;

    for( index = 0; index < n; index++)
```

```
        ar[index] = (double)rand()/((double) rand() + 0.1);
}

void showarray(const double ar[], int n)
{
    int index;

    for( index = 0; index < n; index++)
    {
        printf("%9.4f ", ar[index]);
        if (index % 6 == 5)
            putchar('\n');
    }
    if (index % 6 != 0)
        putchar('\n');
}

/* sort by increasing value */
int mycomp(const void * p1, const void * p2)
{
    /* need to use pointers to double to access values   */
    const double * a1 = (const double *) p1;
    const double * a2 = (const double *) p2;

    if (*a1 < *a2)
        return -1;
    else if (*a1 == *a2)
        return 0;
    else
        return 1;
}
```

Here is a sample run:

```
Random list:
   0.0001    1.6475    2.4332    0.0693    0.7268    0.7383
  24.0357    0.1009   87.1828    5.7361    0.6079    0.6330
   1.6058    0.1406    0.5933    1.1943    5.5295    2.2426
   0.8364    2.7127    0.2514    0.9593    8.9635    0.7139
   0.6249    1.6044    0.8649    2.1577    0.5420   15.0123
   1.7931    1.6183    1.9973    2.9333   12.8512    1.3034
   0.3032    1.1406   18.7880    0.9887

Sorted list:
   0.0001    0.0693    0.1009    0.1406    0.2514    0.3032
   0.5420    0.5933    0.6079    0.6249    0.6330    0.7139
   0.7268    0.7383    0.8364    0.8649    0.9593    0.9887
```

```
  1.1406    1.1943    1.3034    1.6044    1.6058    1.6183
  1.6475    1.7931    1.9973    2.1577    2.2426    2.4332
  2.7127    2.9333    5.5295    5.7361    8.9635   12.8512
 15.0123   18.7880   24.0357   87.1828
```

Let's look at two main areas: the use of `qsort()` and the definition of `mycomp()`.

Using `qsort()`

The `qsort()` function sorts an array of data objects. The ANSI prototype, again, is this:

```
void qsort (void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
```

The first argument is a pointer to the beginning of the array to be sorted. In this program, the actual argument is `vals`, the name of an array of `double`, hence a pointer to the first element of the array. The ANSI prototype causes the `vals` argument to be typecast to type pointer-to-`void`. That's because ANSI C permits any data pointer type to be typecast to a pointer-to-`void`, thus permitting the first actual argument to `qsort()` to refer to any kind of array.

The second argument is the number of items to be sorted. In Listing 16.17, it is `N`, the number of array elements. The prototype converts this value to type `size_t`.

The third argument is the size of each element—`sizeof(double)`, in this case.

The final argument is `mycomp`, the address of the function to be used for comparing elements.

Defining `mycomp()`

As mentioned before, the `qsort()` prototype mandates the form of the comparison function:

```
int (*compar)(const void *, const void *)
```

This states that the final argument is a pointer to a function that returns an `int` and that takes two arguments, each of which is a pointer to type `const void`. We made the prototype for the `mycomp()` function agree with this prototype:

```
int mycomp(const void * p1, const void * p2);
```

Remember that the name of the function is a pointer to the function when used as argument, so `mycomp` matches the `compar` prototype.

The `qsort()` function passes the addresses of the two elements to be compared to the comparison function. In this program, then, `p1` and `p2` are assigned the addresses of two type `double` values to be compared. Note that the first argument to `qsort()` refers to the whole array, and the two arguments in the comparison function refer to two elements in the array. There is a problem. To compare the pointed-to values, you need to dereference a pointer. Because the values are type `double`, you need to dereference a pointer to type `double`. However, `qsort()` requires pointers to type `void`. The way to get around this problem is to declare pointers of the proper type inside the function and initialize them to the values passed as arguments:

```
/* sort by increasing value */
int mycomp(const void * p1, const void * p2)
{
    /* need to use pointers to double to access values   */
    const double * a1 = (const double *) p1;
    const double * a2 = (const double *) p2;

    if (*a1 < *a2)
        return -1;
    else if (*a1 == *a2)
        return 0;
    else
        return 1;
}
```

In short, `qsort()` and the comparison function use `void` pointers for generality. As a consequence, you have to tell `qsort()` explicitly how large each element of the array is, and within the definition of the comparison function, you have to convert its pointer arguments to pointers of the proper type for your application.

> **Note** *`void *`* **in C and in C++**
>
> C and C++ treat pointer-to-`void` differently. In both languages, you can assign a pointer of any type to type `void *`. The function call to `qsort()` in Listing 16.17, for example, assigns type `double *` to a type `void *` pointer. But C++ requires a type cast when assigning a `void *` pointer to a pointer of another type, whereas C doesn't have that requirement. For instance, the `mycomp()` function in Listing 16.17 has this type cast for the type `void *` pointer `p1`:
>
> ```
> const double * a1 = (const double *) p1;
> ```
>
> In C, this type cast is optional; in C++ it is mandatory. Because the type cast version works in both languages, it makes sense to use it. Then, if you convert the program to C++, you won't have to remember to change that part.

Let's look at one more example of a comparison function. Suppose you have these declarations:

```
struct names {
    char first[40];
    char last[40];
};
struct names staff[100];
```

What should a call to `qsort()` look like? Following the model in Listing 16.17, a call could look like this:

```
qsort(staff, 100, sizeof(struct names), comp);
```

Here, comp is the name of the comparison function. What should this function look like? Suppose you want to sort by last name, then by first name. You could write the function this way:

```
#include <string.h>
int comp(const void * p1, const void * p2)   /* mandatory form */
{
    /* get right type of pointer */
    const struct names *ps1 = (const struct names *) p1;
    const struct names *ps2 = (const struct names *) p2;
    int res;

    res = strcmp(ps1->last, ps2->last);  /* compare last names */
    if (res != 0)
        return res;
    else       /* last names identical, so compare first names */
        return strcmp(ps1->first, ps2->first);
}
```

This function uses the `strcmp()` function to do the comparison; its possible return values match the requirements for the comparison function. Note that you need a pointer to a structure to use the `->` operator.

The Assert Library

The assert library, supported by the `assert.h` header file, is a small one designed to help with debugging programs. It consists of a macro named `assert()`. It takes as its argument an integer expression. If the expression evaluates as false (nonzero), the `assert()` macro writes an error message to the standard error stream (`stderr`) and calls the `abort()` function, which terminates the program. (The `abort()` function is prototyped in the `stdlib.h` header file.) The idea is to identify critical locations in a program where certain conditions should be true and to use the `assert()` statement to terminate the program if one of the specified conditions is not true. Typically, the argument is a relational or logical expression. If `assert()` does abort the program, it first displays the test that failed, the name of the file containing the test, and a line number.

Using `assert`

Listing 16.18 shows a short example using `assert`. It asserts that `z` is greater than or equal to `0` before attempting to take its square root. It also mistakenly subtracts a value instead of adding it, making it possible for `z` to obtain forbidden values.

Listing 16.18 **The `assert.c` Program**

```
/* assert.c -- use assert() */
#include <stdio.h>
#include <math.h>
#include <assert.h>
int main()
{
    double x, y, z;

    puts("Enter a pair of numbers (0 0 to quit): ");
    while (scanf("%lf%lf", &x, &y) == 2
                && (x != 0 || y != 0))
    {
        z = x * x - y * y;  /* should be + */
        assert(z >= 0);
        printf("answer is %f\n", sqrt(z));
        puts("Next pair of numbers: ");
    }
    puts("Done");

    return 0;
}
```

Here is a sample run:

```
Enter a pair of numbers (0 0 to quit):
4 3
answer is 2.645751
Next pair of numbers:
5 3
answer is 4.000000
Next pair of numbers:
3 5
Assertion failed: (z >= 0), function main, file /Users/assert.c, line 14.
```

The exact wording will depend on the compiler. One potentially confusing point to note is that the message is not saying that `z >= 0`; instead, it's saying that the claim `z >= 0` failed.

You could accomplish something similar with an `if` statement:

```
if (z < 0)
{
    puts("z less than 0");
    abort();
}
```

The `assert()` approach has several advantages, however. It identifies the file automatically. It identifies the line number where the problem occurs automatically. Finally, there's a mechanism for turning the `assert()` macro on and off without changing code. If you think you've eliminated the program bugs, place the macro definition

```
#define NDEBUG
```

before the location where `assert.h` is included and then recompile the program, and the compiler will deactivate all `assert()` statements in the file. If problems pop up again, you can remove the `#define` directive (or comment it out) and then recompile, thus reactivating all the `assert()` statements.

`_Static_assert` (C11)

The `assert()` expression is a run-time check. C11 adds a feature, the `_Static_assert` declaration, that does a compile-time check. So, `assert()` can cause a running program to abort, while `_Static_assert()` can cause a program not to compile. The latter takes two arguments. The first is a constant integer expression, and the second is a string. If the first expression evaluates to `0` (or `_False`), the compiler displays the string and does not compile the program. Let's look at the short example of Listing 16.19, and then look at the differences between `assert()` and `_Static_assert()`.

Listing 16.19 **The `statasrt.c` Program**

```
//  statasrt.c
#include <stdio.h>
#include <limits.h>
_Static_assert(CHAR_BIT == 16, "16-bit char falsely assumed");
int main(void)
{
    puts("char is 16 bits.");
    return 0;
}
```

Here is a sample attempt at command-line compilation:

```
$ clang statasrt.c
statasrt.c:4:1: error: static_assert failed "16-bit char falsely assumed"
_Static_assert(CHAR_BIT == 16, "16-bit char falsely assumed");
^              ~~~~~~~~~~~~~~
1 error generated.
$
```

In terms of syntax, `_Static_assert` is treated as a declaration statement. Thus, unlike most kinds of C statements, it can appear either in a function or, as in this case, external to a function.

The requirement that the first argument to `_Static_assert` be an integer constant expression guarantees that it can be evaluated during compilation. (Recall that `sizeof` expressions count as integer constants.) So you can't substitute `_Static_assert` for `assert` in Listing 16.18, because that program used `z > 0` for a test expression, and that's a nonconstant expression that can be evaluated only while the program is running. You could use `assert(CHAR_BIT == 16)` in the body of `main()` in Listing 16.19, but that would alert you to an error only after you compiled and ran the program, which is more inefficient.

The `assert.h` header makes `static_assert` an alias for the C keyword `_Static_assert`. That's to make C more compatible with C++, which uses `static_assert` as its keyword for this feature.

memcpy() and memmove() from the string.h Library

You can't assign one array to another, so we've been using loops to copy one array to another, element by element. The one exception is that we've used the `strcpy()` and `strncpy()` functions for character arrays. The `memcpy()` and `memmove()` functions offer you almost the same convenience for other kinds of arrays. Here are the prototypes for these two functions:

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

Both of these functions copy `n` bytes from the location pointed to by `s2` to the location pointed to by `s1`, and both return the value of `s1`. The difference between the two, as indicated by the keyword `restrict`, is that `memcpy()` is free to assume that there is no overlap between the two memory ranges. The `memmove()` function doesn't make that assumption, so copying takes place as if all the bytes are first copied to a temporary buffer before being copied to the final destination. What if you use `memcpy()` when there are overlapping ranges? The behavior is undefined, meaning it might work or it might not. The compiler won't stop you from using the `memcpy()` function when you shouldn't, so it's your responsibility to make sure the ranges aren't overlapping when you use it. It's just another part of the programmer's burden.

Because these functions are designed to work with any data type, the two pointer arguments are type pointer-to-`void`. C allows you to assign any pointer type to pointers of the `void *` type. The other side of this tolerant acceptance is that these functions have no way of knowing what type of data is being copied. Therefore, they use the third argument to indicate the number of bytes to be copied. Note that for an array, the number of bytes is not, in general, the number of elements. So if you were copying an array of 10 `double` values, you would use `10*sizeof(double)`, not `10`, as the third argument.

Listing 16.20 shows some examples using these two functions. It assumes that `double` is twice the size of `int`, and it uses the C11 `_Static_assert` feature to test that assumption.

Listing 16.20 **The `mems.c` Program**

```
// mems.c -- using memcpy() and memmove()
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define SIZE 10
void show_array(const int ar[], int n);
// remove following if C11 _Static_assert not supported
_Static_assert(sizeof(double) == 2 * sizeof(int), "double not twice int size");
int main()
{
    int values[SIZE] = {1,2,3,4,5,6,7,8,9,10};
    int target[SIZE];
    double curious[SIZE / 2] = {2.0, 2.0e5, 2.0e10, 2.0e20, 5.0e30};

    puts("memcpy() used:");
    puts("values (original data): ");
    show_array(values, SIZE);
    memcpy(target, values, SIZE * sizeof(int));
    puts("target (copy of values):");
    show_array(target, SIZE);

    puts("\nUsing memmove() with overlapping ranges:");
    memmove(values + 2, values, 5 * sizeof(int));
    puts("values -- elements 0-5 copied to 2-7:");
    show_array(values, SIZE);

    puts("\nUsing memcpy() to copy double to int:");
    memcpy(target, curious, (SIZE / 2) * sizeof(double));
    puts("target -- 5 doubles into 10 int positions:");
    show_array(target, SIZE/2);
    show_array(target + 5, SIZE/2);

    return 0;
}

void show_array(const int ar[], int n)
{
    int i;

    for (i = 0; i < n; i++)
        printf("%d ", ar[i]);
    putchar('\n');
}
```

Here is the output:

```
memcpy() used:
values (original data):
1 2 3 4 5 6 7 8 9 10
target (copy of values):
1 2 3 4 5 6 7 8 9 10

Using memmove() with overlapping ranges:
values -- elements 0-5 copied to 2-7:
1 2 1 2 3 4 5 8 9 10

Using memcpy() to copy double to int:
target -- 5 doubles into 10 int positions:
0 1073741824 0 1091070464 536870912
1108516959 2025163840 1143320349 -2012696540 1179618799
```

The last call to `memcpy()` copies data from a type `double` array to a type `int` array. This shows that `memcpy()` doesn't know or care about data types; it just copies bytes from one location to another. (You could, for example, copy bytes from a structure to a character array.) Also, there is no data conversion. If you had a loop doing element-by-element assignment, the type `double` values would be converted to type `int` during assignment. In this case, the bytes are copied over "as is," and the program then interprets the bit patterns as if they were type `int`.

Variable Arguments: `stdarg.h`

Earlier, this chapter discussed variadic macros—macros that can accept a variable number of arguments. The `stdarg.h` header file provides a similar capability for functions. But the usage is a bit more involved. You have to do the following:

1. Provide a function prototype using an ellipsis.
2. Create a `va_list` type variable in the function definition.
3. Use a macro to initialize the variable to an argument list.
4. Use a macro to access the argument list.
5. Use a macro to clean up.

Let's look at these steps in more detail. The prototype for such a function should have a parameter list with at least one parameter followed by an ellipsis:

```
void f1(int n, ...);                // valid
int f2(const char * s, int k, ...); // valid
char f3(char c1, ..., char c2);     // invalid, ellipsis not last
double f3(...);                     // invalid, no parameter
```

The rightmost parameter (the one just before the ellipses) plays a special role; the standard uses the term *parmN* as a name to use in discussion. In the preceding examples, *parmN* would be `n` for the first case and `k` for the second case. The actual argument passed to this parameter will be the number of arguments represented by the ellipses section. For example, the `f1()` function prototyped earlier could be used this way:

```
f1(2, 200, 400);          // 2 additional arguments
f1(4, 13, 117, 18, 23);   // 4 additional arguments
```

Next, the `va_list` type, which is declared in the `stdargs.h` header file, represents a data object used to hold the parameters corresponding to the ellipsis part of the parameter list. The beginning of a definition of a variadic function would look something like this:

```
double sum(int lim,...)
{
    va_list ap;                  // declare object to hold arguments
```

In this example, `lim` is the *parmN* parameter, and it will indicate the number of arguments in the variable-argument list.

After this, the function will use the `va_start()` macro, also defined in `stdargs.h`, to copy the argument list to the `va_list` variable. The macro has two arguments: the `va_list` variable and the `parmN` parameter. Continuing with the previous example, the `va_list` variable is called `ap` and the `parmN` parameter is call `lim`, so the call would look like this:

```
va_start(ap, lim);           // initialize ap to argument list
```

The next step is gaining access to the contents of the argument list. This involves using `va_arg()`, another macro. It takes two arguments: a type `va_list` variable and a type name. The first time it's called, it returns the first item in the list; the next time it's called, it returns the next item, and so on. The type argument specifies the type of value returned. For example, if the first argument in the list were a `double` and the second were an `int`, you could do this:

```
double tic;
int toc;
...
tic = va_arg(ap, double);   // retrieve first argument
toc = va_arg(ap, int);      // retrieve second argument
```

Be careful. The argument type really has to match the specification. If the first argument is 10.0, the previous code for `tic` works fine. But if the argument is 10, the code may not work; the automatic conversion of `double` to `int` that works for assignment doesn't take place here.

Finally, you should clean up by using the `va_end()` macro. It may, for example, free memory dynamically allocated to hold the arguments. This macro takes a `va_list` variable as its argument:

```
va_end(ap);                  // clean up
```

After you do this, the variable `ap` may not be usable unless you use `va_start` to reinitialize it.

Because `va_arg()` doesn't provide a way to back up to previous arguments, it may be useful to preserve a copy of the `va_list` type variable. C99 has added a macro for that purpose. It's called `va_copy()`. Its two arguments are both type `va_list` variables, and it copies the second argument to the first:

```
va_list ap;
va_list apcopy;
double
double tic;
int toc;
...
va_start(ap, lim);          // initialize ap to argument list
va_copy(apcopy, ap);        // make apcopy a copy of ap
tic = va_arg(ap, double);   // retrieve first argument
toc = va_arg(ap, int);      // retrieve second argument
```

At this point, you could still retrieve the first two items from `apcopy`, even though they have been removed from `ap`.

Listing 16.21 is a short example of how the facilities can be used to create a function that sums a variable number of arguments; here, the first argument to `sum()` is the number of items to be summed.

Listing 16.21 The `varargs.c` Program

```
//varargs.c -- use variable number of arguments
#include <stdio.h>
#include <stdarg.h>
double sum(int, ...);

int main(void)
{
    double s,t;

    s = sum(3, 1.1, 2.5, 13.3);
    t = sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1);
    printf("return value for "
           "sum(3, 1.1, 2.5, 13.3):                %g\n", s);
    printf("return value for "
           "sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1): %g\n", t);

    return 0;
}

double sum(int lim,...)
{
    va_list ap;                     // declare object to hold arguments
```

```
    double tot = 0;
    int i;

    va_start(ap, lim);            // initialize ap to argument list
    for (i = 0; i < lim; i++)
       tot += va_arg(ap, double); // access each item in argument list
    va_end(ap);                   // clean up

    return tot;
}
```

Here is the output:

```
return value for sum(3, 1.1, 2.5, 13.3):                16.9
return value for sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1): 31.6
```

If you check the arithmetic, you'll find that `sum()` did add three numbers to the first function call and six numbers to the second.

All in all, using variadic functions is more involved than using variadic macros, but the functions have a greater range of application.

Key Concepts

The C standard doesn't just describe the C language; it describes a package consisting of the C language, the C preprocessor, and the standard C library. The preprocessor lets you shape the compiling process, listing substitutions to be made, indicating which lines of code should be compiled, and other aspects of compiler behavior. The C library extends the reach of the language and provides prepackaged solutions to many programming problems.

Summary

The C preprocessor and the C library are two important adjuncts to the C language. The C preprocessor, following preprocessor directives, adjusts your source code before it is compiled. The C library provides many functions designed to help with tasks such as input, output, file handling, memory management, sorting and searching, mathematical calculations, and string processing, to name a few. Appendix B, Reference Section V lists the complete ANSI C library.

Review Questions

1. Here are groups of one or more macros followed by a source code line that uses them. What code results in each case? Is it valid code? (Assume C variables have been declared.)

a.

```
#define FPM  5280     /* feet per mile */
dist = FPM * miles;
```

b.

```
#define FEET 4
#define POD FEET + FEET
plort = FEET * POD;
```

c.

```
#define SIX = 6;
nex = SIX;
```

d.

```
#define NEW(X) X + 5
y = NEW(y);
berg = NEW(berg) * lob;
est = NEW(berg) / NEW(y);
nilp = lob * NEW(-berg);
```

2. Fix the definition in part d of question 1 to make it more reliable.

3. Define a macro function that returns the minimum of two values.

4. Define the `EVEN_GT(X,Y)` macro, which returns 1 if `X` is even and also greater than `Y`.

5. Define a macro function that prints the representations and the values of two integer expressions. For example, it might print

```
3+4 is 7 and 4*12 is 48
```

   if its arguments are `3+4` and `4*12`.

6. Create `#define` statements to accomplish the following goals:

   a. Create a named constant of value `25`.

   b. Have `SPACE` represent the space character.

   c. Have `PS()` represent printing the space character.

   d. Have `BIG(X)` represent adding 3 to `X`.

   e. Have `SUMSQ(X,Y)` represent the sums of the squares of `X` and `Y`.

7. Define a macro that prints the name, value, and address of an `int` variable in the following format:

   ```
   name: fop;  value: 23;  address: ff464016
   ```

8. Suppose you have a block of code you want to skip over temporarily while testing a program. How can you do so without actually removing the code from the file?

9. Show a code fragment that prints out the date of preprocessing if the macro `PR_DATE` is defined.

10. The discussion of inline functions shows three different versions of a `square()` function. How do the three differ from one another in terms of behavior?

11. Create a macro using a generic selection expression that evaluates to the string `"boolean"` if the macro argument is type `_Bool`, and evaluates to `"not boolean"` otherwise.

12. What's wrong with this program?

    ```
    #include <stdio.h>
    int main(int argc, char argv[])
    {
        printf("The square root of %f is %f\n", argv[1],
                sqrt(argv[1]) );
    }
    ```

13. Suppose `scores` is an array of 1000 `int` values that you want to sort into descending order. And suppose you are using `qsort()` and a comparison function called `comp()`.

    a. What is a suitable call to `qsort()`?

    b. What is a suitable definition for `comp()`?

14. Suppose `data1` is an array of 100 `double` values and `data2` is an array of 300 `double` values.

    a. Write a `memcpy()` function call that copies the first 100 elements of `data2` to `data1`.

    b. Write a `memcpy()` function call that copies the last 100 elements of `data2` to `data1`.

Programming Exercises

1. Start developing a header file of preprocessor definitions that you want to use.

2. The harmonic mean of two numbers is obtained by taking the inverses of the two numbers, averaging them, and taking the inverse of the result. Use a `#define` directive to define a macro "function" that performs this operation. Write a simple program that tests the macro.

3. Polar coordinates describe a vector in terms of magnitude and the counterclockwise angle from the x-axis to the vector. Rectangular coordinates describe the same vector in terms of x and y components (see Figure 16.3). Write a program that reads the magnitude and angle (in degrees) of a vector and then displays the x and y components. The relevant equations are these:

```
x = r cos A    y = r sin A
```

To do the conversion, use a function that takes a structure containing the polar coordinates and returns a structure containing the rectangular coordinates (or use pointers to such structures, if you prefer).

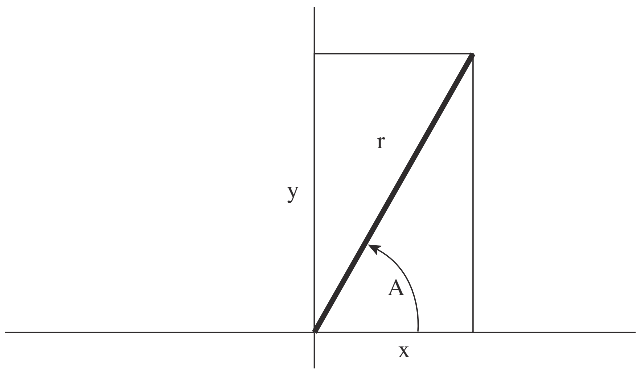

Figure 16.3 Rectangular and polar coordinates.

4. The ANSI library features a `clock()` function with this description:

```
#include <time.h>
clock_t clock (void);
```

Here, `clock_t` is a type defined in `time.h`. The function returns the processor time, which is given in some implementation-dependent units. (If the processor time is unavailable or cannot be represented, the function returns a value of `-1`.) However, `CLOCKS_PER_SEC`, also defined in `time.h`, is the number of processor time units per second. Therefore, dividing the difference between two return values of `clock()` by `CLOCKS_PER_SEC` gives you the number of seconds elapsed between the two calls. Typecasting the values to `double` before division enables you to get fractions of a second. Write a function that takes a `double` argument representing a desired time delay and then runs a loop until that amount of time has passed. Write a simple program that tests the function.

5. Write a function that takes as arguments the name of an array of type `int` elements, the size of an array, and a value representing the number of picks. The function then should select the indicated number of items at random from the array and prints them. No array element is to be picked more than once. (This simulates picking lottery numbers or jury members.) Also, if your implementation has `time()` (discussed in Chapter 12) or a similar function available, use its output with `srand()` to initialize the `rand()` random-number generator. Write a simple program that tests the function.

6. Modify Listing 16.17 so that it uses an array of `struct names` elements (as defined after the listing) instead of an array of `double`. Use fewer elements, and initialize the array explicitly to a suitable selection of names.

7. Here's a partial program using a variadic function:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
void show_array(const double ar[], int n);
double * new_d_array(int n, ...);

int main()
{
    double * p1;
    double * p2;

    p1 = new_d_array(5, 1.2, 2.3, 3.4, 4.5, 5.6);
    p2 = new_d_array(4, 100.0, 20.00, 8.08, -1890.0);
    show_array(p1, 5);
    show_array(p2, 4);
    free(p1);
    free(p2);

    return 0;
}
```

The `new_d_array()` function takes an `int` argument and a variable number of `double` arguments. The function returns a pointer to a block of memory allocated by `malloc()`. The `int` argument indicates the number of elements to be in the dynamic array, and the `double` values are used to initialize the elements, with the first value being assigned to the first element, and so on. Complete the program by providing the code for `show_array()` and `new_d_array()`.