CHAPTER 13

Supporting Facilities

At this point you've covered the complete C language, as well as the important library functions. You should be reasonably confident in programming all aspects of the language. If you aren't, that's simply because you need more practice. Once you've learned the elements of the language, competence comes down to practice, practice, practice.

In this final chapter, I'll tie up a few loose ends. You'll delve deeper into the capabilities you have available through the preprocessor, and you'll look at a few more library functions that you're sure to find useful.

In this chapter you'll learn the following:

- More about the preprocessor and its operation
- How to write preprocessor macros
- What logical preprocessor directives are and how you can use them
- What conditional compilation is and how you can apply it
- More about the debugging methods that are available to you
- How you use some of the additional library functions available

Preprocessing

As you are certainly aware by now, preprocessing of your source code occurs before it's compiled to machine instructions. The preprocessing phase can execute a range of service operations specified by preprocessing directives, which are identified by the # symbol as the first character of each preprocessor directive. The preprocessing phase provides an opportunity for manipulating and modifying your C source code prior to compilation. Once the preprocessing phase is complete and all directives have been analyzed and executed, all such preprocessing directives will no longer appear in the source code that results. The compiler begins the compile phase proper, which generates the machine code equivalent of your program.

You've already used preprocessor directives in all the examples, and you're familiar with both the `#include` and `#define` directives. There are a number of other directives that add considerable flexibility to the way in which you specify your programs. Keep in mind as you proceed that all these are preprocessing operations that occur before your program is compiled. They modify the set of statements that constitute your program. They aren't involved in the execution of your program at all.

Including Header Files in Your Programs

A header file is any external file whose contents are included into your program by use of the `#include` preprocessor directive. You're completely familiar with statements such as this:

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

This fetches the standard library header file supporting input/output operations into your program. This is a particular case of the general statement for including standard libraries into your program:

```
#include <standard_library_file_name>
```

Any library header file name can appear between the angled brackets. If you include a header file that you don't use, the only effect, apart from slightly confusing anyone reading the program, is to extend the compilation time.

You can include your own source files into your program with a slightly different `#include` statement. A typical example might be this:

```
#include "myfile.h"
```

This statement will introduce the contents of the file named between double quotes into the program in place of the `#include` directive. The contents of any file can be included into your program by this means. You simply specify the name of the file between quotes as shown in the example.

You can also give the file whatever name you like within the constraints of the operating system. In theory you don't have to use the extension `.h`, although it's a convention commonly adhered to by most programmers in C, so I strongly recommend that you use it too. The difference between using this form and using angled brackets lies in the source that will be assumed for the required file. The precise operation is compiler-dependent and will be described in the compiler documentation, but usually the first form will search the default header file directory for the required file, whereas the second form will search the current source directory before the default header file directory is searched.

Header files cannot include implementation, by which I mean executable code. You use header files to contain declarations, not function definitions or initialized global data. All your function definitions and initialized globals are placed in source files with the extension `.c`. You can use the header file mechanism to divide your program source code into several files and, of course, to manage the declarations for any library functions of your own. A very common use of this facility is to create a header file containing all the function prototypes and type declarations for a program. These can then be managed as a separate unit and included at the beginning of any source file for the program. You need to avoid duplicating information if you include more than one header file into a source file. Duplicate code will often cause compilation errors. You'll see later in this chapter in the "Conditional Compilation" section how you can ensure that any given block of code will appear only once in your program, even if you inadvertently include it several times.

■ **Note** A file introduced into your program by an `#include` directive may also contain another `#include` directive. If so, preprocessing will deal with the second `#include` in the same way as the first and continue replacing such directives with the contents of the corresponding file until there are no more `#include` directives in the program.

External Variables and Functions

With a program that's made up of several source files, you'll frequently find that you want to use a global variable that's defined in another file. You can do this by declaring the variable as external to

the current file using the `extern` keyword. For example, if you have variables defined in another file as **global** (which means outside of any of the functions) using the statements

```
int number = 0;
double in_to_mm = 2.54;
```

then in a function in which you want to access these, you can specify that these variable names are external by using these statements:

```
extern int number;
extern double in_to_mm;
```

These statements don't create these variables—they just identify to the compiler that these names are defined elsewhere, and this assumption about these names should apply to the rest of this source file. The variables you specify as `extern` must be declared and defined somewhere else in the program, usually in another source file. If you want to make these external variables accessible to all functions within the current file, you should declare them as external at the very beginning of the file, prior to any of the function definitions. With programs consisting of several files, you could place all initialized global variables at the beginning of one file and all the `extern` statements in a header file. The `extern` statements can then be incorporated into any program file that needs access to these variables by using an `include` statement for the header file.

■ **Note** Only one declaration of each global variable is allowed in a file. Of course, the global variables may be declared as external in as many files as necessary.

Substitutions in Your Program Source Code

There are preprocessor directives for replacing symbols in your source code before it is compiled. The simplest kind of symbol substitution you can make is one you've already seen. For example, the preprocessor directive to substitute the specified numeric value, wherever the character string `PI` occurs, is as follows:

```
#define PI 3.14159265
```

Although the identifier `PI` *looks* like a variable, it is not a variable and has nothing to do with variables. Here `PI` is a token, rather like a voucher, that is exchanged for the sequence of digits specified in the `#define` directive during the preprocessing phase. When your program is ready to be compiled after preprocessing has been completed, the string `PI` will no longer appear, having been replaced by its definition wherever it occurs in the source file. The general form of this sort of preprocessor directive is the following:

```
#define identifier sequence_of_characters
```

Here, `identifier` conforms to the usual definition of an identifier in C: any sequence of letters and digits, the first of which is a letter, and underline characters count as letters. Note that `sequence_of_characters`, which is the replacement for `identifier`, is any sequence of characters and need not be just digits.

A very common use of the `#define` directive is to define array dimensions by way of a substitution, to allow a number of array dimensions to be determined by a single token. Only one directive in the program then needs to be modified to alter the dimensions of a number of arrays in the program. This helps considerably in minimizing errors when such changes are necessary, as shown in the following example:

```
#define MAXLEN 256
char *buffer[MAXLEN];
char *str[MAXLEN];
```

Here, the dimensions of both arrays can be changed by modifying the single `#define` directive, and of course the array declarations that are affected can be anywhere in the program file. The advantages of this approach in a large program involving dozens or even hundreds of functions should be obvious. Not only is it easy to make a change, but also using this approach ensures that the same value is being used through a program. This is especially important with large projects involving several programmers working together to produce the final product.

Of course, you can also define a value such as `MAXLEN` as a `const` variable:

```
const size_t MAXLEN = 256;
```

The difference between this approach and using the `#define` directive is that `MAXLEN` here is no longer a token but is a variable of a specific type with the name `MAXLEN`. The `MAXLEN` in the `#define` directive does not exist once the source file has been preprocessed because all occurrences of `MAXLEN` in the code will be replaced by `256`.

I use numerical substitution in the last two examples, but as I said, you're in no way limited to this. You could, for example, write the following:

```
#define Black White
```

This will cause any occurrence of `Black` in your program to be replaced with `White`. The sequence of characters that is to replace the token identifier can be anything at all.

Macro Substitutions

A **macro** is based on the ideas implicit in the `#define` directive examples you've seen so far, but it provides a greater range of possible results by allowing what might be called **multiple parameterized substitutions**. This not only involves substitution of a fixed sequence of characters for a token identifier, but also allows parameters to be specified, which may themselves be replaced by argument values, wherever the parameter appears in the substitution sequence.

Let's look at an example:

```
#define Print(My_var) printf("%d", Myvar)
```

This directive provides for two levels of substitution. There is the substitution for `Print(My_var)` by the string immediately following it in the `#define` statement, and there is the possible substitution of alternatives for `My_var`. You could, for example, write the following:

```
Print(ival);
```

This will be converted during preprocessing to this statement:

```
printf("%d", ival);
```

You could use this directive to specify a `printf()` statement for an integer variable at various points in your program. A common use for this kind of macro is to allow a simple representation of a complicated function call in order to enhance the readability of a program.

Macros That Look Like Functions

The general form of the kind of substitution directive just discussed is the following:

```
#define macro_name( list_of_identifiers ) substitution_string
```

This shows that in the general case, multiple parameters are permitted, so you're able to define more complex substitutions.

To illustrate how you use this, you can define a macro for producing a maximum of two values with the following directive:

```
#define max(x, y) x>y ? x : y
```

You can then put the statement in the program:

```
result = max(myval, 99);
```

This will be expanded during preprocessing to produce the following code:

```
result = myval>99 ? myval : 99;
```

It's important to be conscious of the substitution that is taking place and not to assume that this is a function. You can get some strange results otherwise, particularly if your substitution identifiers include an explicit or implicit assignment. For example, the following modest extension of the last example can produce an erroneous result:

```
result = max(myval++, 99);
```

The substitution process will generate this statement:

```
result = myval++>99 ? myval++ : 99;
```

The consequence of this is that if the value of `myval` is larger than 99, `myval` will be incremented twice. Note that it does *not* help to use parentheses in this situation. If you write the statement as

```
result = max((myval++), 99);
```

preprocessing will convert this to

```
result = (myval++>99) ? (myval++) : 99;
```

You need to be very cautious if you're writing macros that generate expressions of any kind. In addition to the multiple substitution trap you've just seen, precedence rules can also catch you out. A simple example will illustrate this. Suppose you write a macro for the product of two parameters:

```
#define product(m, n)  m*n
```

You then try to use this macro with the following statement:

```
result = product(x, y + 1);
```

Of course, everything works fine so far as the macro substitution is concerned, but you don't get the result you want, as the macro expands to this:

```
result = x*y + 1;
```

It could take a long time to discover that you aren't getting the product of the two parameters at all in this case, as there's no external indication of what's going on. There's just a more or less erroneous value propagating through the program. The solution is very simple. If you use macros to generate expressions, put parentheses around everything. So you should rewrite the example as follows:

```
#define product(m, n)  ((m)*(n))
```

Now everything will work as it should. The inclusion of the outer parentheses may seem excessive, but because you don't know the context in which the macro expansion will be placed, it's better to include them. If you write a macro to sum its parameters, you will easily see that without the outer

parentheses, there are many contexts in which you will get a result that's different from what you expect. Even with parentheses, expanded expressions that repeat a parameter, such as the one you saw earlier that uses the conditional operator, will still not work properly when the argument involves the increment or decrement operators.

Preprocessor Directives on Multiple Lines

A preprocessor directive must be a single logical line, but this doesn't prevent you from using the statement continuation character, \.

You could write the following:

```
#define min(x, y) \
              ((x)<(y) ? (x) : (y))
```

Here, the directive definition continues on the second line with the first nonblank character found, so you can position the text on the second line wherever you feel looks like the nicest arrangement. Note that the \ must be the last character on the line, immediately before you press Enter.

Strings As Macro Arguments

String constants are a potential source of confusion when used with macros. The simplest string substitution is a single-level definition such as the following:

```
#define MYSTR "This string"
```

Suppose you now write the statement

```
printf("%s", MYSTR);
```

This will be converted during preprocessing into the statement

```
printf("%s", "This string");
```

This should be what you are expecting. You couldn't use the `#define` directive without the quotes in the substitution sequence and expect to be able to put the quotes in your program text instead. For example, suppose you write the following:

```
#define  MYSTR  This string
  ...
printf("%s", "MYSTR" );
```

There will be no substitution for `MYSTR` in the `printf()` function in this case. Anything in quotes in your program is assumed to be a literal string, so it won't be analyzed during preprocessing.

There's a special way of specifying that the substitution for a macro argument is to be implemented as a string. For example, you could specify a macro to display a string using the function `printf()` as follows:

```
#define PrintStr(arg) printf("%s", #arg)
```

The # character preceding the appearance of the `arg` parameter in the macro expansion indicates that the argument is to be surrounded by double quotes when the substitution is generated. Therefore, if you write the following statement in your program

```
PrintStr(Output);
```

it will be converted during preprocessing to

```
printf("%s", "Output");
```

You may be wondering why this apparent complication has been introduced into preprocessing. Well, without this facility you wouldn't be able to include a variable string in a macro definition at all. If you were to put the double quotes around the macro parameter, it wouldn't be interpreted as a variable; it would be merely a string with quotes around it. On the other hand, if you put the quotes in the macro expansion, the string between the quotes wouldn't be interpreted as an identifier for a parameter; it would be just a string constant. So what might appear to be an unnecessary complication at first sight is actually an essential tool for creating macros that allow strings between quotes to be created.

A common use of this mechanism is for converting a variable name to a string, such as in this directive:

```
#define show(var) printf("\n%s = %d", #var, var);
```

If you now write

```
show(number);
```

this will generate the statement

```
printf("\n%s = %d", "number", number);
```

You can also generate a substitution that would allow you to display a string with double quotes included. Assuming you've defined the macro `PrintStr` as shown previously and you write the statement

```
PrintStr("Output");
```

it will be preprocessed into the statement

```
printf("%s", "\"Output\"");
```

This is possible because the preprocessing phase is clever enough to recognize the need to put `\"` at each end to get a string that includes double quotes to be displayed correctly.

Joining Two Results of a Macro Expansion

There are times when you may wish to generate two results in a macro and join them together with no spaces between them. Suppose you try to define a macro to do this as follows:

```
#define join(a, b) ab
```

This can't work in the way you need it to. The definition of the expansion will be interpreted as `ab`, not as the parameter `a` followed by the parameter `b`. If you separate them with a blank, the result will be separated with a blank, which isn't what you want either. The preprocessing phase provides you with another operator to solve this problem. The solution is to specify the macro as follows:

```
#define join(a, b) a##b
```

The presence of the operator consisting of the two characters ## serves to separate the parameters and to indicate that the result of the two substitutions are to be joined. For example, writing the statement

```
strlen(join(var, 123));
```

will result in the statement

```
strlen(var123);
```

This might be applied to synthesizing a variable name for some reason, or generating a format control string from two or more macro parameters.

Logical Preprocessor Directives

The last example you looked at appears to be of limited value, because it's hard to envision when you would want to simply join `var` to `123`. After all, you could always use one parameter and write `var123` as the argument. One aspect of preprocessing that adds considerably more potential to the previous example is the possibility for multiple macro substitutions where the arguments for one macro are derived from substitutions defined in another. In the last example, both arguments to the `join()` macro could be generated by other `#define` substitutions or macros. Preprocessing also supports directives that provide a logical `if` capability, which vastly expands the scope of what you can do during the preprocessing phase.

Conditional Compilation

The first logical directive I'll discuss allows you to test whether an identifier exists as a result of having been created in a previous `#define` directive. It takes the following form:

```
#if defined identifier
```

If the specified `identifier` is defined, statements that follow the `#if` are included in the program code until the directive `#endif` is reached. If the identifier isn't defined, the statements between the `#if` and the `#endif` will be skipped. This is the same logical process you use in C programming, except that here you're applying it to the inclusion or exclusion of program statements in the source file.

You can also test for the absence of an identifier. In fact, this tends to be used more frequently than the form you've just seen. The general form of this directive is:

```
#if !defined identifier
```

Here, the statements following the `#if` down to the `#endif` will be included if the identifier hasn't previously been defined. This provides you with a method of avoiding duplicating functions, or other blocks of code and directives, in a program consisting of several files, or to ensure bits of code that may occur repeatedly in different libraries aren't repeated when the `#include` statements in your program are processed.

The mechanism is simply to top and tail the block of code you want to avoid duplicating as follows:

```
#if !defined block1
  #define block1
  /* Block of code you do not */
  /* want to be repeated.     */
#endif
```

If the identifier `block1` hasn't been defined, the block following the `#if` will be included and processed, and `block1` will be defined. The following block of code down to the `#endif` will also be included in your program. Any subsequent occurrence of the same group of statements won't be included because the identifier `block1` now exists.

The `#define` directive doesn't need to specify a substitution value in this case. For the conditional directives to operate, it's sufficient for `block1` to appear in a `#define` directive. You can now include this block of code anywhere you think you might need it, with the assurance that it will never be duplicated within a program. The preprocessing directives ensure this can't happen.

■ **Note** It's a good idea to get into the habit of always protecting code in your own libraries in this fashion. You'll be surprised how easy it is to end up duplicating blocks of code accidentally, once you've collected a few libraries of your own functions.

You aren't limited to testing just one value with the `#if` preprocessor directive. You can use logical operators to test if multiple identifiers have been defined. For example, the statement

```
#if defined block1 && defined block2
```

will evaluate to `true` if both `block1` and `block2` have previously been defined, and so the code that follows such a directive won't be included unless this is the case.

A further extension of the flexibility in applying the conditional preprocessor directives is the ability to undefine an identifier you've previously defined. This is achieved using a directive such as `#undef block1`.

Now, if `block1` was previously defined, it is no longer defined after this directive. The ways in which these directives can all be combined to useful effect is only limited by your own ingenuity.

There are alternative ways of writing these directives that are slightly more concise. You can use whichever of the following forms you prefer. The directive `#ifdef block` is the same as `#if defined block`. And the directive `#ifndef block` is the same as `#if !defined block`.

Directives Testing for Specific Values

You can also use a form of the `#if` directive to test the value of a constant expression. If the value of the constant expression is nonzero, the following statements down to the next `#endif` are included in the program code. If the constant expression evaluates to zero, the following statements down to the next `#endif` are skipped. The general form of the `#if` directive is the following:

```
#if constant_expression
```

This is most frequently applied to test for a specific value being assigned to an identifier by a previous preprocessing directive. You might have the following sequence of statements, for example:

```
#if CPU == Pentium4
  printf("\nPerformance should be good." );
#endif
```

The `printf()` statement will be included in the program here only if the identifier `CPU` has been defined as `Pentium4` in a previous `#define` directive.

Multiple-Choice Selections

To complement the `#if` directives, you have the `#else` directive. This works in exactly the same way as the `else` statement does, in that it identifies a group of directives to be executed or statements to be included if the `#if` condition fails, for example:

```
#if CPU == Pentium4
  printf("\nPerformance should be good." );
#else
  printf("\nPerformance may not be so good." );
#endif
```

In this case, one or the other of the `printf()` statements will be included, depending on whether `CPU` has been defined as `Pentium4`.

The preprocessing phase also supports a special form of the #if for multiple-choice selections, in which only one of several choices of statements for inclusion in the program is required. This is the #elif directive, which has the general form

```
#elif constant_expression
```

An example of using this would be as follows:

```
#define US 0
#define UK 1
#define Australia 2
#define Country US
#if Country == US
  #define Greeting "Howdy, stranger."
#elif Country == UK
  #define Greeting "Wotcher, mate."
#elif Country == Australia
  #define Greeting "G'day, sport."
#endif
printf("\n%s", Greeting );
```

With this sequence of directives the output of the printf() statement will depend on the value assigned to the identifier Country, in this case US.

Standard Preprocessing Macros

There are usually a considerable number of standard preprocessing macros defined, which you'll find described in your compiler documentation. I'll just mention two that are of general interest and that are available to you.

The __DATE__ macro provides a string representation of the date in the form Mmm dd yyyy when it's invoked in your program. Here Mmm is the month in characters, such as Jan, Feb, and so on. The pair of characters dd is the day in the form of a pair of digits 1 to 31, where single-digit days are preceded by a blank. Finally, yyyy is the year as four digits—2006, for example.

A similar macro, __TIME__, provides a string containing the value of the time when it's invoked, in the form hh:mm:ss, which is evidently a string containing pairs of digits for hours, minutes, and seconds, separated by colons. Note that the time is when the compiler is executed, not when the program is run.

You could use this macro to record when your program was last compiled with a statement such as this:

```
printf("\nProgram last compiled at %s on %s", __TIME__, __DATE__ );
```

Note that both __DATE__ and __TIME__ have two underscore characters at the beginning and the end. Once the program containing this statement is compiled, the values that will be output by the printf() statement are fixed until you compile it again. On subsequent executions of the program, the then current time and date will be output. Don't confuse these macros with the time function I discuss later in this chapter in the section "The Date and Time Function Library."

Debugging Methods

Most of your programs will contain errors, or **bugs**, when you first complete them. Removing such bugs from a program can represent a substantial proportion of the time required to write the program. The larger and more complex the program, the more bugs it's likely to contain, and the more time it

will take to get the program to run properly. Very large programs, such as those typified by operating systems, or complex applications, such as word processing systems or even C program development systems, can be so involved that all the bugs can never be eliminated. You may already have experience of this in practice with some of the systems on your own computer. Usually these kinds of residual bugs are relatively minor, with ways in the system to work around them.

Your approach to writing a program can significantly affect how difficult it will be to test. A well-structured program consisting of compact functions, each with a well-defined purpose, is much easier to test than one without these attributes. Finding bugs will also be easier in a program that has extensive comments documenting the operation and purpose of its component functions and has well-chosen variable and function names. Good use of indentation and statement layout can also make testing and fault finding simpler.

It's beyond the scope of this book to deal with debugging comprehensively, but in this section I introduce the basic ideas that you need to be aware of.

Integrated Debuggers

Many compilers are supplied with extensive debugging tools built into the program development environment. These can be very powerful facilities that can dramatically reduce the time required to get a program working. They typically provide a varied range of aids to testing a program that include the following:

Tracing program flow: This capability allows you to execute your program one source statement at a time. It operates by pausing execution after each statement and continuing with the next statement after you press a designated key. Other provisions of the debug environment will usually allow you to display information easily, pausing to show you what's happening to the data in your program.

Setting breakpoints: Executing a large or complex program one statement at a time can be very tedious. It may even be impossible in a reasonable period of time. All you need is a loop that executes 10,000 times to make it an unrealistic proposition. Breakpoints provide an excellent alternative. With breakpoints, you define specific selected statements in your program at which a pause should occur to allow you to check what's happening. Execution continues to the next breakpoint when you press a specified key.

Setting watches: This sort of facility allows you to identify variables that you want to track the value of as execution progresses. The values of the variables that you select are displayed at each pause point in your program. If you step through your program statement by statement, you can see the exact point at which values are changed, or perhaps not changed when you expect them to be.

Inspecting program elements: It may also be possible to examine a wide variety of program components. For example, at breakpoints the inspection can show details of a function such as its return type and its arguments. You can also see details of a pointer in terms of its address, the address it contains, and the data stored at the address contained in the pointer. Seeing the values of expressions and modifying variables may also be provided for. Modifying variables can help to bypass problem areas to allow other areas to be executed with correct data, even though an earlier part of the program may not be working properly.

The Preprocessor in Debugging

By using conditional preprocessor directives, you can arrange for blocks of code to be included in your program to assist in testing. In spite of the power of the debug facilities included with many C development systems, the addition of tracing code of your own can still be useful. You have complete control

of the formatting of data to be displayed for debugging purposes, and you can even arrange for the kind of output to vary according to conditions or relationships within the program.

TRY IT OUT: USING PREPROCESSOR DIRECTIVES

I can illustrate how you can use preprocessor directives to control execution and switch debugging output on and off through a program that calls functions at random through an array of function pointers:

```
/* Program 13.1 Debugging using preprocessing directives */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Macro to generate pseudo-random number from 0 to NumValues */
#define random(NumValues) ((int)(((double)(rand())*(NumValues))/(RAND_MAX+1.0)))

#define iterations 6
#define test                            /* Select testing output       */
#define testf                           /* Select function call trace  */
#define repeatable                      /* Select repeatable execution */

/* Function prototypes */
int sum(int, int);
int product(int, int);
int difference(int, int);

int main(void)
{
  int funsel = 0;                     /* Index for function selection       */
  int a = 10, b = 5;                  /* Starting values                    */
  int result = 0;                     /* Storage for results                */

  /* Function pointer array declaration */
  int (*pfun[])(int, int) = {sum, product, difference};

  /* Conditional code for repeatable execution */
  #ifdef repeatable
  srand(1);
  #else
  srand((unsigned int)time(NULL));    /* Seed random number generation      */
  #endif

  /* Execute random function selections */
  int element_count = sizeof(pfun)/sizeof(pfun[0]);
  for(int i = 0 ; i < iterations ; i++)
  {
    /* Generate random index to pfun array */
```

```
    funsel = random(element_count);
    if( funsel>element_count-1 )
    {
      printf("\nInvalid array index = %d", funsel);
      exit(1);
  }

   #ifdef test
    printf("\nRandom index = %d", funsel);
   #endif

    result = pfun[funsel](a , b);      /* Call random function               */
    printf("\nresult = %d", result );
  }
  return 0;
}

/* Definition of the function sum */
int sum(int x, int y)
{
#ifdef testf
  printf("\nFunction sum called args %d and %d.", x, y);
#endif

  return x + y;
}

/* Definition of the function product */
int product( int x, int y )
{
  #ifdef testf
  printf("\nFunction product called args %d and %d.", x, y);
  #endif

  return x * y;
}

/* Definition of the function difference */
int difference(int x, int y)
{
  #ifdef testf
  printf("\nFunction difference called args %d and %d.", x, y);
  #endif

  return x - y;
}
```

How It Works

You have a macro defined at the beginning of the program:

```
#define random(NumValues) ((int)(((double)(rand())*(NumValues))/(RAND_MAX+1.0)))
```

This defines the macro `random()` in terms of the function `rand()` that's declared in `stdlib.h`. The function `rand()` generates random numbers in the range 0 to `RAND_MAX`, which is a constant defined in `<stdlib.h>`. The macro maps values from this range to produce values from 0 to `NumValues-1`. You cast the value from `rand()` to `double` to ensure that computation will be carried out as type `double`, and you cast the result overall back to `int` because that's what you want in the program. It's quite possible that your version of `<stdlib.h>` may already contain a macro for `random()` that does essentially the same thing. If so, you'll get an error message as the compiler won't allow two different definitions of the same macro. In this case, just delete the definition from the program.

I defined `random()` as a macro to show how you do it, but it would be better defined as a function because this would eliminate any potential problems that might arise with argument values to the macro.

You then have four directives that define symbols:

```
#define iterations 6
#define test                          /* Select testing output       */
#define testf                         /* Select function call trace  */
#define repeatable                    /* Select repeatable execution */
```

The first of these defines a symbol that specifies the number of iterations in the loop that executes one of three functions at random. The last three are all symbols that control the selection of code to be included in the program. Defining the `test` symbol causes code to be included that will output the value of the index that selects a function. Defining `testf` causes code that traces function calls to be included in the function definitions. When the `repeatable` symbol is defined, the `srand()` function is called with a fixed seed value, so the `rand()` function will always generate the same pseudo-random sequence and the same output will be produced on successive runs of the program. Having repeatable output during test runs of the program obviously makes the testing process somewhat easier. If you remove the directive that defines the `repeatable` symbol, `srand()` will be called with the current time value as the argument, so the seed will be different each time the program executes, and you will get different output on each execution of the program.

After setting up the initial variables used in `main()` you have the following statement declaring and initializing the `pfun` array:

```
int (*pfun[])(int, int) = {sum, product, difference};
```

This declares an array of pointers to functions with two parameters of type `int` and a return value of type `int`. The array is initialized using the names of the three functions so the array will contain three elements.

Next you have a directive that includes one of two alternative statements depending on whether the `repeatable` symbol is defined:

```
#ifdef repeatable
  srand(1);
#else
  srand((unsigned int)time(NULL));     /* Seed random number generation      */
#endif
```

If `repeatable` is defined, the statement that calls `srand()` with the argument value 1 will be included in the source for compilation. This will result in the same output each time you execute the program. Otherwise the statement with the result of the `time()` function as the argument will be included and you will get different output each time you run the program.

Look at the loop in `main()` that follows. The number of iterations is determined by the value of the iterations symbol; in this case it corresponds to 6. The first action in the loop is the following:

```
    funsel = random(element_count);
    if( funsel>element_count-1 )
    {
      printf("\nInvalid array index = %d", funsel);
      exit(1);
  }
```

This uses the `random()` macro with `element_count` as the argument. This is the number of elements in the `pfun` array and is calculated immediately prior to the loop. The preprocessor will substitute `element_count` in the macro expansion before the code is compiled. For safety, there is a check that we do indeed get a valid index value for the `pfun` array.

The next three lines are the following:

```
   #ifdef test
    printf("\nRandom index = %d", funsel);
   #endif
```

These include the `printf()` statement in the code when the `text` symbol is defined. If you remove the directive that defines `text`, the `printf()` will not be included in the program that is compiled.

The last two statements in the loop call a function through one of the pointers in the `pfun` array and output the result of the call:

```
    result = pfun[funsel](a , b);      /* Call random function              */
    printf("\nresult = %d", result );
```

Let's look at one of the functions that may be called, `product()` for example:

```
int product( int x, int y )
{
  #ifdef testf
  printf("\nFunction product called args %d and %d.", x, y);
  #endif

  return x * y;
}
```

The function definition includes an output statement if the `testf` symbol is defined. You can therefore control whether the statements in the `#ifdef` block are included here independently from the output block in `main()` that is controlled by `test`. With the program as written with both `test` and `testf` defined, you'll get trace output for the random index values generated and a message from each function as it's called, so you can follow the sequence of calls in the program exactly.

You can have as many different symbolic constants defined as you wish. As you've seen previously in this chapter, you can combine them into logical expressions using the `#if defined` form of the conditional directive.

Using the assert() Macro

The `assert()` macro is defined in the standard library header file `<assert.h>`. This macro enables you to insert tests of arbitrary expressions in your program that will cause the program to be terminated with a diagnostic message if a specified expression is false (that is, evaluates to 0). The argument to the `assert()` macro is an expression that results in an integer value, for example:

```
assert(a == b);
```

The expression will be true (nonzero) if a is equal to b. If a and b are unequal, the argument to the macro will be false and the program will be terminated with a message relating to the assertion. Termination is achieved by calling abort(), so it's an abnormal end to the program. When abort() is called, the program terminates immediately. Whether stream output buffers are flushed, open streams are closed, or temporary files are removed, it is implementation-dependent, so consult your compiler documentation on this.

In program 13.1 I could have used an assertion to verify that funsel is valid:

```
assert(funsel<element_count);
```

If funsel is not less than element_count, the expression will be false, so the program will assert. Typical output from the assertion looks like this:

```
Assertion failed:  funsel<element_count d:\examples\program13_01.c 44
```

The assertion facility allowing assertions to occur can be switched off by defining the symbol NDEBUG before the #include directive for assert.h, like this:

```
#define NDEBUG                          /* Switch off assertions */
#include <assert.h>
```

This will cause all assertions to be ignored.

With some systems, assertions are disabled by default, in which case you can enable them by undefining NDEBUG:

```
#undef NDEBUG                           /* Switch on assertions */
#include <assert.h>
```

By including the directive to undefine NDEBUG, you ensure that assertions are enabled for your source file. The #undef directive must appear before the #include directive for assert.h to be effective.

I can demonstrate this in operation with a simple example.

TRY IT OUT: DEMONSTRATING THE ASSERT() MACRO

Here's the code for a program that uses the assert() macro:

```
/* Program 13.2 Demonstrating assertions */
#undef NDEBUG                           /* Switch on assertions */
#include <stdio.h>
#include <assert.h>

int main(void)
{
  int y = 5;
  for(int x = 0 ; x < 20 ; x++)
  {
    printf("\nx = %d   y = %d", x, y);
    assert(x<y);
  }
  return 0;
}
```

Compiling and executing this with my compiler produces the following output:

```
x = 0   y = 5
x = 1   y = 5
x = 2   y = 5
x = 3   y = 5
x = 4   y = 5
x = 5   y = 5
Assertion failed: x<y , file Prog13_02.C, line 12
```

How It Works

At this point, apart from the `assert()` statement, the program shouldn't need much explanation, as it simply displays the values of `x` and `y` in the `for` loop.

The program is terminated by the `assert()` macro as soon as the condition `x < y` becomes false. As you can see from the output, this is when `x` reaches the value 5. The macro displays the output on `stderr`, which is always the display screen. Not only do you get the condition that failed displayed, but you also get the file name and line number in the file where the failure occurred. This is particularly useful with multifile programs in which the source of the error is pinpointed exactly.

Assertions are often used for critical conditions in a program in which, if certain conditions aren't met, disaster will surely ensue. You would want to be sure that the program wouldn't continue if such errors arise.

You could switch off the assertion mechanism in the example by replacing the `#undef` directive with `#define NDEBUG`. This must be placed before the `#include` statement for `<assert.h>` to be effective, although assertions may be disabled by default. With this `#define` at the beginning of Program 13.2, you'll see that you get output for all the values of `x` from 0 to 19, and no diagnostic message.

Additional Library Functions

The library functions are basic to the power of the C language. Although I've covered quite a range of standard library functions so far, it's beyond the scope of this book to discuss all the standard libraries. However, I can introduce a few more of the most commonly used functions you haven't dealt with in detail up to now.

The Date and Time Function Library

Because time is an important parameter to measure, C includes a standard library of functions called `<time.h>` that deal with time and dates. They provide output in various forms from the hardware timer in your PC.

The simplest function has the following prototype:

```
clock_t clock(void);
```

This function returns the processor time (not the elapsed time) used by the program since execution began, as a value of type `clock_t`. Your computer will typically be executing multiple processes at any given moment. The processor time is the total time the processor has been executing on behalf of the process that called the `clock()` function. The type `clock_t` is defined in `<time.h>` and is equivalent to type `size_t`. The value that is returned by the `clock()` function is measured in **clock ticks**, and to convert this value to seconds, you divide it by the value that is produced by the macro `CLOCKS_PER_SEC`, which is also defined in the library `<time.h>`. The value that results from executing `CLOCKS_PER_SEC` is

the number of clock ticks in one second and is of type clock_t. The clock() function returns -1 if an error occurs.

To determine the processor time used in executing a process, you need to record the time when the process starts executing and subtract this from the time returned when the process finishes. For example

```
clock_t start, end;
double cpu_time;
start = clock();

/* Execute the process for which you want the processor time */

end = clock();
cpu_time = ((double)(end-start)/CLOCKS_PER_SEC;
```

This fragment stores the total processor time used in cpu_time. The cast to type double is necessary in the last statement to get the correct result.

As you've seen, the time() function returns the calendar time as a value of type time_t. The calendar time is the current time in seconds since a fixed time and date. The fixed time and date is often 00:00:00GMT on January 1, 1970, for example, and this is typical of how time values are defined.

The prototype of the time() function is the following:

```
time_t time(time_t *timer);
```

If the argument isn't NULL, the current calendar time is also stored in the location pointed to by the argument. The type time_t is defined in the header file and is equivalent to long.

To calculate the elapsed time in seconds between two successive time_t values returned by time(), you can use the function difftime(), which has this prototype:

```
double difftime(time_t T2, time_t T1);
```

The function will return the value of T2 - T1 expressed in seconds as a value of type double. This value is the time elapsed between the two time() function calls that produce the time_t values, T1 and T2.

TRY IT OUT: USING TIME FUNCTIONS

You could define a function to log the elapsed time and processor time used between successive calls by using functions from <time.h> as follows:

```
/* Program 13.3 Test our timer function */
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <ctype.h>

int main(void)
{
  time_t calendar_start = time(NULL);       /* Initial calendar time    */
  clock_t cpu_start = clock();              /* Initial processor time   */
  int count = 0;                            /* Count of number of loops */
  const int iterations = 1000000;           /* Loop iterations          */
  char answer = 'y';
```

```
  printf("Initial clock time = %lu Initial calendar time = %lu\n",
                                    cpu_start, calendar_start);

  while(tolower(answer) == 'y')
  {
    for(int i = 0 ; i<iterations ; i++)
    {
      double x = sqrt(3.14159265);
    }
    printf("\n%ld square roots completed.", iterations*(++count));
    printf("\nDo you want to run some more(y or n)? ");

    scanf("\n%c", &answer);
  }

  clock_t cpu_end = clock();                /* Final cpu time           */
  time_t calendar_end = time(NULL);         /* Final calendar time      */

  printf("\nFinal clock time = %lu Final calendar time = %lu\n",
                                                  cpu_end, calendar_end);
  printf("\nCPU time for %ld iterations is %.2lf seconds\n",
           count*iterations, ((double)(cpu_end-cpu_start))/CLOCKS_PER_SEC );

  printf("\nElapsed calendar time to execute the program is %8.2lf\n",
                                  difftime(calendar_end, calendar_start));
  return 0;
}
```

On my machine I get the following output:

```
Initial clock time = 0 Initial calendar time = 1155841882

1000000 square roots completed.
Do you want to run some more(y or n)? y

2000000 square roots completed.
Do you want to run some more(y or n)? n

Final clock time = 7671 Final calendar time = 1155841890

CPU time for 2000000 iterations is 7.67 seconds

Elapsed calendar time to execute the program is     8.00
```

How It Works

This program illustrates the use of the functions `clock()`, `time()`, and `difftime()`. The `time()` function returns the current time in seconds, so you won't get values less than 1 second. Depending on the speed of your machine, you may want to adjust the number of iterations in the loop to reduce or increase the time required to execute this program. Note that the `clock()` function may not be a very accurate way of determining the processor time used in the program. You also need to keep in mind that measuring elapsed time using the `time()` function can be a second out.

You record and display the initial values for the processor time and the calendar time and set up the controls for the loop that follows with these statements:

```
time_t calendar_start = time(NULL);       /* Initial calendar time    */
clock_t cpu_start = clock();              /* Initial processor time   */
int count = 0;                            /* Count of number of loops */
const int iterations = 1000000;           /* Loop iterations          */
char answer = 'y';
printf("Initial clock time = %lu Initial calendar time = %lu\n",
                                  cpu_start, calendar_start);
```

With my C library, the types `clock_t` and `time_t` are defined as type `unsigned long` in the `time.h` header file. You should check how the types are defined in your library and adjust the format specifications for these values if necessary.

You then have a loop controlled by the character stored in `answer`, so the loop will execute as long as you want it to continue:

```
while(tolower(answer) == 'y')
{
  for(int i = 0 ; i<iterations ; i++)
  {
    double x = sqrt(3.14159265);
  }

   printf("\n%ld square roots completed.", iterations*(++count));
   printf("\nDo you want to run some more(y or n)? ");

  scanf("\n%c", &answer);
}
```

The inner loop calls the `sqrt()` function that is declared in the `<math.h>` header `iterations` times, so this is just to occupy some processor time. If you are leisurely in your entry of a response to the prompt for input, this should extend the elapsed time. Note the newline escape sequence in the beginning of the first argument to `scanf()`. If you leave this out, your program will loop indefinitely, because `scanf()` will not ignore whitespace characters in the input stream buffer.

Finally, you output the final values returned by `clock()` and `time()`, and calculate the processor and calendar time intervals:

```
clock_t cpu_end = clock();                /* Final cpu time           */
time_t calendar_end = time(NULL);         /* Final calendar time      */

printf("\nFinal clock time = %lu Final calendar time = %lu\n",
                                                cpu_end, calendar_end);
printf("\nCPU time for %ld iterations is %.2lf seconds\n",
         count*iterations, ((double)(cpu_end-cpu_start))/CLOCKS_PER_SEC );

printf("\nElapsed calendar time to execute the program is %8.2lf\n",
                                  difftime(calendar_end, calendar_start));
```

The library that comes with your C compiler may well have additional nonstandard functions for obtaining processor time that are more reliable than the `clock()` function.

■ **Caution** Note that the processor clock can wrap around, and the resolution with which processor time is measured can vary between different hardware platforms. For example, if the processor clock is a 32-bit value that has a microsecond resolution, the clock will wrap back to zero roughly every 72 minutes.

Getting the Date

Having the time in seconds since a date over a quarter of a century ago is interesting, but it's often more convenient to get today's date as a string. You can do this with the function `ctime()`, which has this prototype:

```
char *ctime(const time_t *timer);
```

The function accepts a pointer to a `time_t` variable as an argument that contains a calendar time value returned by the `time()` function. It returns a pointer to a 26-character string containing the day, the date, the time, and the year, which is terminated by a newline and a `'\0'`.

A typical string returned might be the following:

```
"Mon Aug 25  10:45:56 2003\n\0"
```

You might use the `ctime()` function like this:

```
time_t calendar = 0;
calendar = time(NULL);                  /* Store calendar time                 */
  printf("\n%s", ctime(&calendar));     /* Output calendar time as date string */
```

You can also get at the various components of the time and date from a calendar time value by using the library function `localtime()`. This function has the following prototype:

```
struct tm *localtime(const time_t *timer);
```

This function accepts a pointer to a `time_t` value and returns a pointer to a structure type `tm`, which is defined in the `<time.h>` header file. The structure contains the members listed in Table 13-1.

Table 13-1. *Members of the `tm` Structure*

Member	Description
`tm_sec`	Seconds after the minute on 24-hour clock
`tm_min`	Minutes after the hour on 24-hour clock (0 to 59)
`tm_hour`	The hour on 24-hour clock (0 to 23)
`tm_mday`	Day of the month (1 to 31)
`tm_mon`	Month (0 to 11)
`tm_year`	Year (current year minus 1900)
`tm_wday`	Weekday (Sunday is 0; Saturday is 6)
`tm_yday`	Day of year (0 to 365)
`tm_isdst`	Daylight saving flag. Positive for daylight saving time 0 for not daylight saving time –1 for not known

All these structure members are of type `int`. The `localtime()` function returns a pointer to the same structure each time you call it and the structure members are overwritten on each call. If you want to keep any of the member values, you need to copy them elsewhere before the next call to `localtime()`, or you could create your own `tm` structure and save the whole lot if you really need to.

The time that the `localtime()` function produces is local to where you are. If you want to get the time in a `tm` structure that reflects UTC (Coordinated Universal Time) you can use the `gmtime()` function. This also expects an argument of type `time_t` and returns a pointer to a `tm` structure.

Here's a code fragment that will output the day and the date from the members of the `tm` structure:

```
time_t calendar = 0;                /* Holds calendar time        */
struct tm *time_data;               /* Holds address of tm struct */
const char *days[] = {"Sunday",   "Monday", "Tuesday", "Wednesday",
                      "Thursday", "Friday", "Saturday"              };
constchar *months[] = {"January", "February", "March",
                       "April",    "May",     "June",
                       "July",    "August",   "September",
                       "October", "November", "December"  };

calendar = time(NULL);              /* Get current calendar time */
printf("\n%s", ctime(&calendar));
time_data = localtime(&calendar);
printf("Today is %s %s %d %d\n",
                    days[time_data->tm_wday], months[time_data->tm_mon],
                    time_data->tm_mday,       time_data->tm_year+1900);
```

You've defined arrays of strings to hold the days of the week and the months. You use the appropriate member of the structure that has been set up by the call to the `localtime()` function. You use the day in the month and the year values from the structure directly. You can easily extend this to output the time.

TRY IT OUT: GETTING THE DATE

It's very easy to pick out the members you want from the structure of type `tm` returned from the function `localtime()`. You can demonstrate this with the following example:

```
/* Program 13.4        Getting date data with ease */
#include <stdio.h>
#include <time.h>
int main(void)
{
  const char *Day[7] = {
                  "Sunday"  , "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"
                      };
  const char *Month[12] = {
                    "January",   "February", "March",    "April",
                    "May",       "June",     "July",     "August",
                    "September", "October",  "November", "December"
                       };
  const char *Suffix[4] = { "st", "nd", "rd", "th" };
  enum sufindex { st, nd, rd, th } sufsel = th;       /* Suffix selector */
```

```
  struct tm *OurT = NULL;              /* Pointer for the time structure */
  time_t Tval = 0;                     /* Calendar time                  */

  Tval = time(NULL);                   /* Get calendar time              */
  OurT = localtime(&Tval);             /* Generate time structure        */

  switch(OurT->tm_mday)
  {
    case 1: case 21: case 31:
      sufsel= st;
      break;
    case 2: case 22:
      sufsel= nd;
      break;
    case 3: case 23:
      sufsel= rd;
      break;
    default:
      sufsel= th;
      break;
  }

  printf("Today is %s the %d%s %s %d", Day[OurT->tm_wday],
    OurT->tm_mday, Suffix[sufsel], Month[OurT->tm_mon], 1900 + OurT->tm_year);
  printf("\nThe time is %d : %d : %d",
    OurT->tm_hour, OurT->tm_min, OurT->tm_sec );
  return 0;
}
```

Here's an example of output from this program:

```
Today is Friday the 18th August 2006
The time is 11 : 49 : 53
```

How It Works

In this example, the first declarations in `main()` are as follows:

```
  const char *Day[7] = {
                  "Sunday"  , "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"
                      };
  const char *Month[12] = {
                    "January",   "February", "March",    "April",
                    "May",       "June",     "July",     "August",
                    "September", "October",  "November", "December"
                        };
  const char *Suffix[] = { "st", "nd", "rd", "th" };
```

Each defines an array of pointers to `char`. The first holds the days of the week, the second contains the months in the year, and the third holds the suffixes to numerical values for the day in the month when representing dates. You could leave out the array dimensions in the first two declarations and the compiler would compute them for you, but in this case you're reasonably confident about both these numbers, so this is an instance in which putting them

in helps to avoid an error. The const qualifier specifies that the strings pointed to are constants and should not be altered in the code.

The enumeration provides a mechanism for selecting an element from the Suffix array:

```
enum sufindex { st, nd, rd, th } sufsel = th;       /* Suffix selector */
```

The enumeration constants, st, nd, rd, and th will be assigned values 0 to 3 by default, so we can use the sufsel variable as an index to access elements in the Suffix array.

You also declare a structure variable in the following declaration:

```
struct tm *OurT = NULL;              /* Pointer for the time structure */
```

This provides space to store the pointer to the structure returned by the function local_time().

You first obtain the current time in Tval using the function time(). You then use this value to generate the values of the members of the structure returned by the function localtime(). If you want to keep the data in the structure, you need to copy it before calling the localtime() function again, as it would be overwritten. Once you have the structure from localtime(), you execute the switch:

```
switch( OurT->tm_mday )
{
    case 1: case 21: case 31:
      sufsel= st;
      break;
    case 2: case 22:
      sufsel= nd;
      break;
    case 3: case 23:
      sufsel= rd;
      break;
    default:
      sufsel= th;
      break;
}
```

The sole purpose of this is to select what to append to the date value. Based on the member tm_mday, the switch selects an index to the array Suffix[] for use when outputting the date by setting the sufsel variable to the appropriate enumeration constant value.

The day, the date, and the time are displayed, with the day and month strings obtained by indexing the appropriate array with the corresponding structure member value. The addition of 1900 to the value of the member tm_year is because this value is measured relative to the year 1900.

You can also use the mktime() function to determine the day of the week for a given date. The function has the following prototype:

```
time_t mktime(struct tm *ptime);
```

You can pass a pointer to a tm structure to a function with the tm_mon, tm_day, and tm_year values set to a date you are interested in. The values of the tm_wkday and tm_yday members of the structure will be ignored, and if the operation is successful, the values will be replaced with the values that are correct for the date you have supplied. The function returns the calendar time as a value of type time_t if the operation is successful, or –1 if the date cannot be represented as a time_t value, causing the operation to fail. Let's see it working in an example.

TRY IT OUT: GETTING THE DAY FOR A DATE

It's very easy to pick out the members you want from the structure of type `tm` returned from the function `localtime()`. You can demonstrate this with the following example:

```
/* Program 13.5        Getting the day for a given date */
#include <stdio.h>
#include <time.h>
int main(void)
{
  const char *Day[7] = {
                  "Sunday"  , "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"
                      };
  const char *Month[12] = {
                    "January",   "February", "March",    "April",
                    "May",       "June",     "July",     "August",
                    "September", "October",  "November", "December"
                        };
  const char *Suffix[4] = { "st", "nd", "rd", "th" };
  enum sufindex { st, nd, rd, th } sufsel = th;  /* Suffix selector */

  int day = 0;                          /* Stores a day...           */
  int month = 0;                        /* month...                  */
  int year = 0;                         /* and year for a date       */

  struct tm birthday;                   /* A birthday time structure */

  /* Set the structure members we don't care about */
  birthday.tm_hour = birthday.tm_min = 0;
  birthday.tm_sec = 1;
  birthday.tm_isdst = -1;

  printf("Enter your birthday as integers, day month year."
            "\ne.g. Enter 1st February 1985 as 1 2 1985. : ");
  scanf(" %d %d %d", &day, &month, &year);

  birthday.tm_mon = month-1;
  birthday.tm_mday = day;
  birthday.tm_year = year-1900;

  if(mktime(&birthday) == (time_t)-1)
  {
    printf("\nOperation failed.");
    return 0;
  }

  switch(birthday.tm_mday)
  {
      sufsel= st;
      break;
    case 2: case 22:
      sufsel= nd;
      break;
```

```
 case 3: case 23:
      sufsel= rd;
      break;
    default:
      sufsel= th;
      break;
  }

  printf("\nYour birthday, the %d%s %s %d, was a %s",
                birthday.tm_mday, Suffix[sufsel], Month[birthday.tm_mon],
                           1900 + birthday.tm_year, Day[birthday.tm_wday]);

  return 0;
}
```

Here's a sample of output from this example:

```
Enter your birthday as integers, day month year.
e.g. Enter 1st February 1985 as 1 2 1985. : 15 6 1985

Your birthday, the 15th June 1985, was a Saturday
```

How It Works

You create arrays of constant strings for the day month and date suffixes as you did in Program 13.4. You then create a `tm` structure and initialize some of its members:

```
struct tm birthday;                  /* A birthday time structure */

/* Set the structure members we don't care about */
birthday.tm_hour = birthday.tm_min = 0;
birthday.tm_sec = 1;
birthday.tm_isdst = -1;
```

You set the value of the `tm_isdst` member to –1 because you don't know whether it applies for the date that will be entered.

After outputting a prompt, you read values for the day, month, and year of a birthday date from the keyboard and set the values entered in the birthday structure:

```
printf("\nEnter your birthday as integers, day month year."
          "\ne.g. Enter 1st February 1985 as 1 2 1985. : ");
scanf(" %d %d %d", &day, &month, &year);

birthday.tm_mon = month-1;
birthday.tm_mday = day;
birthday.tm_year = year-1900;
```

With the date set, you can get the `tm_wday` and `tm_yday` members set according by calling the `mktime()` function:

```
if(mktime(&birthday) == (time_t)-1)
{
  printf("\nOperation failed.");
  return 0;
}
```

The `if` statement checks whether the function returns –1, indicating that the operation has failed. In this case you simply output a message and terminate the program.

Finally, the day corresponding to the birth date entered is displayed in the same way as in the previous example.

Summary

In this chapter I discussed the preprocessor directives that you use to manipulate and transform the code in a source file before it is compiled. Your standard library header files are an excellent source of examples of coding preprocessing directives. You can view these examples with any text editor. Virtually all of the capabilities of the preprocessor are used in the libraries, and you'll find a lot of C source code there too. It's also useful to familiarize yourself with the contents of the libraries, as you can find many things not necessarily described in the library documentation. If you aren't sure what the type `clock_t` is, for example, just look in the library header file `<time.h>`, where you'll find the definition.

The debugging capability that the preprocessor provides is useful, but you will find that the tools that are provided for debugging with many C programming systems are much more powerful. For serious program development the debugging tools are as important as the efficiency of the compiler.

If you've reached this point and are confident that you understand and can apply what you've read, you should now be comfortable with programming in C. All you need now is more practice to improve your expertise. To get better at programming, there's no alternative to practice, and the more varied the types of programs you write, the better. You can always improve your skills, but, fortunately—or unfortunately, depending on your point of view—it's unlikely that you'll ever reach perfection in programming, as it's almost impossible to sit down and write a bug-free program of any size from the outset. However, every time you get a new piece of code to work as it should, it will always generate a thrill and a feeling of satisfaction. Enjoy your programming!

Exercises

The following exercises enable you to try out what you learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Download area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 13-1. Define a macro, `COMPARE(x, y)`, that will result in the value –1 if `x < y`, 0 if `x == y`, and 1 if `x > y`. Write an example to demonstrate that your macro works as it should. Can you see any advantage that your macro has over a function that does the same thing?

Exercise 13-2. Define a function that will return a string containing the current time in 12-hour format (a.m./p.m.) if the argument is 0, and in 24-hour format if the argument is 1. Demonstrate that your function works with a suitable program.

Exercise 13-3. Define a macro, `print_value(expr)`, that will output on a new line `exp = result` where `result` is the value that results from evaluating `expr`. Demonstrate the operation of your macro with a suitable program.