Chapter 8

The Preprocessor

The C language uses the preprocessor to extend its power and notation. In this chapter, we present a detailed discussion of the preprocessor, including new features added by the ANSI C committee. We begin by explaining the use of #include. Then we thoroughly discuss the use of the #define macro facility. Macros can be used to generate inline code that takes the place of a function call. Their use can reduce program execution time.

Lines that begin with a # are called *preprocessing directives*. These lines communicate with the preprocessor. In ANSI C, the # can be preceded on the line by white space, whereas in traditional C, it must occur in column 1. The syntax for preprocessing directives is independent of the rest of the C language. The effect of a preprocessing directive starts at its place in a file and continues until the end of that file, or until its effect is negated by another directive. It is always helpful to keep in mind that the preprocessor does not "know C."

8.1 The Use of #include

We have already used preprocessing directives such as

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

Another form of the #include facility is given by

```
#include "filename"
```

This causes the preprocessor to replace the line with a copy of the contents of the named file. A search for the file is made first in the current directory and then in other system-dependent places. With a preprocessing directive of the form

#include <*filename*>

the preprocessor looks for the file only in the other places and not in the current directory. In UNIX systems, the standard header files such as *stdio.h* and *stdlib.h* are typically found in */usr/include*. In general, where the standard header files are stored is system-dependent.

There is no restriction on what a #include file can contain. In particular, it can contain other preprocessing directives that will be expanded by the preprocessor in turn.

8.2 The Use of #define

Preprocessing directives with #define occur in two forms:

#define *identifier* $token\_string_{opt}$
#define *identifier*(*identifier*, ... , *identifier*) $token\_string_{opt}$

The *token_string* is optional. A long definition of either form can be continued to the next line by placing a backslash \ at the end of the current line. If a simple #define of the first form occurs in a file, the preprocessor replaces every occurrence of *identifier* by *token_string* in the remainder of the file, except in quoted strings. Consider the example

```
#define   SECONDS_PER_DAY   (60 * 60 * 24)
```

In this example, the token string is (60 * 60 * 24), and the preprocessor will replace every occurrence of the symbolic constant SECONDS_PER_DAY by that string in the remainder of the file.

The use of simple #defines can improve program clarity and portability. For example, if special constants such as π or the speed of light c are used in a program, they should be defined.

```
#define   PI   3.14159
#define   C    299792.458        /* speed of light in km/sec */
```

Other special constants are also best coded as symbolic constants.

```
#define   EOF     (–1)          /* typical end–of–file value */
#define   MAXINT   2147483647    /* largest 4–byte integer */
```

Program limits that are programmer decisions can also be specified symbolically.

```
#define   ITERS   50        /* number of iterations */
#define   SIZE    250       /* array size */
#define   EPS     1.0e–9    /* a numerical limit */
```

In general, symbolic constants aid documentation by replacing what might otherwise be a mysterious constant with a mnemonic identifier. They aid portability by allowing constants that may be system-dependent to be altered once. They aid reliability by restricting to one place the check on the actual representation of the constant.

Syntactic Sugar

It is possible to alter the syntax of C toward some user preference. A frequent programming error is to use the token = in place of the token == in logical expressions. A programmer could use

```
#define   EQ   ==
```

to defend against such a mistake. This superficial alteration of the programming syntax is called *syntactic sugar*. Another example of this is to change the form of the while statement by introducing "do," which is an ALGOL style construction.

```
#define   do   /* blank */
```

With these two #define lines at the top of the file, the code

```
while (i EQ 1) do {
  .....
```

will become, after the preprocessor pass,

```
while (i == 1) {
  .....
```

Keep in mind that because do will disappear from anywhere in the file, the do-while statement cannot be used. Using macros to provide a personal syntax is controversial. The advantage of avoiding == mistakes is offset by the use of an idiosyncratic style.

8.3 Macros with Arguments

So far, we have considered only simple #define preprocessing directives. We now want to discuss how we can use the #define facility to write macro definitions with parameters. The general form is given by

#define *identifier*(*identifier*, ... , *identifier*) $token\_string_{opt}$

There can be no space between the first identifier and the left parenthesis. Zero or more identifiers can occur in the parameter list. An example of a macro definition with a parameter is

```
#define  SQ(x)  ((x) * (x))
```

The identifier x in the #define is a parameter that is substituted for in later text. The substitution is one of string replacement without consideration of syntactic correctness. For example, with the argument 7 + w the macro call

SQ(7 + w) expands to ((7 + w) * (7 + w))

In a similar fashion

SQ(SQ(*p)) expands to ((((*p) * (*p))) * (((*p) * (*p))))

This seemingly extravagant use of parentheses is to protect against the macro expanding an expression so that it led to an unanticipated order of evaluation. It is important to understand why all the parentheses are necessary. First, suppose we had defined the macro as

```
#define  SQ(x)  x * x
```

With this definition

SQ(a + b) expands to a + b * a + b

which, because of operator precedence, is not the same as

((a + b) * (a + b))

Now suppose we had defined the macro as

```
#define  SQ(x)  (x) * (x)
```

With this definition

```
4 / SQ(2)     expands to     4 / (2) * (2)
```

which, because of operator precedence, is not the same as

```
4 / ((2) * (2))
```

Finally, let us suppose that we had defined the macro as

```
#define   SQ (x)   ((x) * (x))
```

With this definition

```
SQ(7)     expands to     (x)     ((x) * (x)) (7)
```

which is not even close to what was intended. If, in the macro definition, there is a space between the macro name and the left parenthesis that follows, then the rest of the line is taken as replacement text.

A common programming error is to end a #define line with a semicolon, making it part of the replacement string when it is not wanted. As an example of this, consider

```
#define   SQ(x)   ((x) * (x));       /* error */
```

The semicolon here was typed by mistake, one that is easily made because programmers often end a line of code with a semicolon. When used in the body of a function, the line

```
x = SQ(y);     gets expanded to     x = ((y) * (y));;
```

The last semicolon creates an unwanted null statement. If we were to write

```
if (x == 2)
  x = SQ(y);
else
  ++x;
```

we would get a syntax error caused by the unwanted null statement. The extra semicolon does not allow the else to be attached to the if statement.

Macros are frequently used to replace function calls by inline code, which is more efficient. For example, instead of writing a function to find the minimum of two values, a programmer could write

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

After this definition, an expression such as

```
m = min(u, v)
```

gets expanded by the preprocessor to

```
m = (((u) < (v)) ? (u) : (v))
```

The arguments of min() can be arbitrary expressions of compatible type. Also, we can use min() to build another macro. For example, if we need to find the minimum of four values, we can write

```
#define   min4(a, b, c, d)   min(min(a, b), min(c, d))
```

A macro definition can use both functions and macros in its body. For example

```
#define   SQ(x)      ((x) * (x))
#define   CUBE(x)    (SQ(x) * (x))
#define   F_POW(x)   sqrt(sqrt(CUBE(x))) /* fractional power:3/4 */
```

A preprocessing directive of the form

#undef *identifier*

will undefine a macro. It causes the previous definition of a macro to be forgotten.

Caution: Debugging code that contains macros with arguments can be difficult. To see the output from the preprocessor, you can give the command

cc –E file.c

After the preprocessor has done its work, no further compilation takes place. (See exercise 1, on page 395.)

8.4 The Type Definitions and Macros in *stddef.h*

C provides the typedef facility so that an identifier can be associated with a specific type. A simple example is

```
typedef   char   uppercase;
```

This makes uppercase a type that is synonymous with char, and it can be used in declarations, just as other types are used. An example is

```
uppercase   c, u[100];
```

The typedef facility allows the programmer to use type names that are appropriate for a specific application. (See Section 9.1, “Structures,” on page 408, and Section 10.6, “An Example: Polish Notation and Stack Evaluation,” on page 468.)