7.3. Directives

Directives are always introduced by a line that starts with a `#` character, optionally preceded by white space characters (although it isn't common practice to indent the `#`). Table 7.1 below is a list of the directives defined in the Standard.

Directive	Meaning
`# include`	include a source file
`# define`	define a macro
`# undef`	undefine a macro
`# if`	conditional compilation
`# ifdef`	conditional compilation
`# ifndef`	conditional compilation
`# elif`	conditional compilation
`# else`	conditional compilation
`# endif`	conditional compilation
`# line`	control error reporting
`# error`	force an error message
`# pragma`	used for implementation-dependent control
`#`	null directive; no effect

Table 7.1. Preprocessor directives

The meanings and use of these features are described in the following sections. Make a note that the `#` and the following keyword, if any, are individual items. They may be separated by white space.

7.3.1. The null directive

This is simple: a plain `#` on a line by itself does nothing!

7.3.2. # define

There are two ways of defining *macros*, one of which looks like a function and one which does not. Here is an example of each:

```
#define FMAC(a,b) a here, then b

#define NONFMAC some text here
```

Both definitions define a macro and some *replacement text*, which will be used to replace later occurrences of the macro name in the rest of the program. After those definitions, they can be used as follows, with the effect of the macro replacement shown in comments:

```
NONFMAC
/* some text here */

FMAC(first text, some more)
/* first text here, then some more */
```

For the non-function macro, its name is simply replaced by its replacement text. The function macro is also replaced by its replacement text; wherever the replacement text contains an identifier which is the name of one of the macro's ‘formal parameters’, the actual text given as the argument is used in place of the identifier in the replacement text. The scope of the names of the formal parameters is limited to the body of the `#define` directive.

For both forms of macro, leading or trailing white space around the replacement text is discarded.

A curious ambiguity arises with macros: how do you define a non-function macro whose replacement text happens to start with the opening parenthesis character (? The answer is simple. If the definition of the macro has a space in front of the (, then it isn't the definition of a function macro, but a simple replacement macro instead. When you **use** function-like macros, there's no equivalent restriction.

The Standard allows either type of macro to be redefined at any time, using another `# define`, provided that there isn't any attempt to change the type of the macro and that the tokens making up both the original definition and the redefinition are identical in number, ordering, spelling and use of white space. In this context all white space is considered equal, so this would be correct:

```
# define XXX abc/*comment*/def hij
# define XXX abc def hij
```

because comment is a form of white space. The token sequence for both cases (*w-s* stands for a white-space token) is:

```
# w-s define w-s XXX w-s abc w-s def w-s hij w-s
```

7.3.2.1. Macro substitution

Where will occurrences of the macro name cause the replacement text to be substituted in its place? Practically anywhere in a program that the identifier is recognized as a separate token, except as the identifier following the # of a preprocessor directive. You can't do this:

```
#define define XXX

#define YYY ZZZ
```

and expect the second `#define` to be replaced by `#XXX`, causing an error.

When the identifier associated with a non-function macro is seen, it is replaced by the macro replacement tokens, then *rescanned* (see later) for further replacements to make.

Function macros can be used like real functions; white space around the macro name, the argument list and so on, may include the newline character:

```
#define FMAC(a, b) printf("%s %s\n", a, b)

FMAC ("hello",
      "sailor"
      );
/* results in */
printf("%s %s\n", "hello", "sailor")
```

The ‘arguments’ of a function macro can be almost any arbitrary token sequence. Commas are used to separate the arguments from each other but can be hidden by enclosing them within parentheses, `(` and `)`. Matched pairs of `(` and `)` inside the argument list balance each other out, so a `)` only ends the invocation of the macro if the corresponding (is the one that started the macro invocation.

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

CALL(printf, ("%d %d %s\n",1, 24, "urgh"));
/* results in */
printf ("%d %d %s\n",1, 24, "urgh");
```

Note very carefully that the parentheses around the second argument to CALL were preserved in the replacement: they were not stripped from the text.

If you want to use macros like `printt`, taking a variable number of arguments, the Standard is no help to you. They are not supported.

If any argument contains no preprocessor tokens then the behaviour is undefined. The same is true if the sequence of preprocessor tokens that forms the argument would otherwise have been another preprocessor directive:

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

/* undefined behaviour in each case.... */
CALL(,hello)
CALL(xyz,
#define abc def)
```

In our opinion, the second of the erroneous uses of `CALL` **should** result in defined behaviour—anyone capable of writing that would clearly benefit from the attentions of a champion weightlifter wielding a heavy leather bullwhip.

When a function macro is being processed, the steps are as follows:

1. All of its arguments are identified.
2. Except in the cases listed in item 3 below, if any of the tokens in an argument are themselves candidates for macro replacement, the replacement is done until no further replacement is possible. If this introduces commas into the argument list, there is no danger of the macro suddenly seeming to have a different number of arguments; the arguments are **only** determined in the step above.
3. In the macro replacement text, identifiers naming one of the macro formal arguments are replaced by the (by now expanded) token sequence supplied as the actual argument. The replacement is suppressed only if the identifier is preceded by one of `#` or `##`, or followed by `##`.

7.3.2.2. Stringizing

There is special treatment for places in the macro replacement text where one of the macro formal parameters is found preceded by `#`. The token list for the actual argument has any leading or trailing white space discarded, then the `#` and the token list are turned into a single string literal. Spaces between the tokens are treated as space characters in the string. To prevent ‘unexpected’ results, any `"` or `\` characters within the new string literal are preceded by `\`.

This example demonstrates the feature:

```
#define MESSAGE(x) printf("Message: %s\n", #x)

MESSAGE (Text with "quotes");
/*
* Result is
* printf("Message: %s\n", "Text with \"quotes\"");
*/
```

7.3.2.3. Token pasting

A `##` operator may occur anywhere in the replacement text for a macro except at the beginning or end. If a parameter name of a function macro occurs in the replacement text preceded or followed by one of these operators, the actual token sequence for the corresponding macro argument is used to replace it. Then, for both function and non-function macros, the tokens surrounding the `##` operator

are joined together. If they don't form a valid token, the behaviour is undefined. Then rescanning occurs.

As an example of token pasting, here is a multi-stage operation, involving rescanning (which is described next).

```
#define REPLACE some replacement text
#define JOIN(a, b) a ## b

JOIN(REP, LACE)
becomes, after token pasting,
REPLACE
becomes, after rescanning
some replacement text
```

7.3.2.4. Rescanning

Once the processing described above has occurred, the replacement text plus the following tokens of the source file is rescanned, looking for more macro names to replace. The one exception is that, within a macro's replacement text, the name of the macro itself is not expanded. Because macro replacement can be nested, it is possible for several macros to be in the process of being replaced at any one point: none of their names is a candidate for further replacement in the ‘inner’ levels of this process. This allows redefinition of existing functions as macros:

```
#define exit(x) exit((x)+1)
```

These macro names which were not replaced now become tokens which are immune from future replacement, even if later processing might have meant that they had become available for replacement. This prevents the danger of infinite recursion occurring in the preprocessor. The suppression of replacement is only if the macro name results directly from **replacement** text, not the other source text of the program. Here is what we mean:

```
#define m(x) m((x)+1)
/* so */
m(abc);
/* expands to */
m((abc)+1);
/*
* even though the m((abc)+1) above looks like a macro,
* the rules say it is not to be re-replaced
*/

m(m(abc));
/*
* the outer m( starts a macro invocation,
* but the inner one is replaced first (as above)
* with m((abc)+1), which becomes the argument to the outer call,
* giving us effectively
*/
m(m((abc+1));
/*
* which expands to
*/
m((m((abc+1))+1);
```

If that doesn't make your brain hurt, then go and read what the Standard says about it, which will.

7.3.2.5. Notes

There is a subtle problem when using arguments to function macros.

```
/* warning - subtle problem in this example */
#define SQR(x)  ( x * x )
/*
* Wherever the formal parameters occur in
* the replacement text, they are replaced
* by the actual parameters to the macro.
*/
printf("sqr of %d is %d\n", 2, SQR(2));
```

The formal parameter of `SQR` is `x`; the actual argument is `2`. The replacement text results in

```
printf("sqr of %d is %d\n", 2, ( 2 * 2 ));
```

The use of the parentheses should be noticed. The following example is likely to give trouble:

```
/* bad example */
#define DOUBLE(y) y+y

printf("twice %d is %d\n", 2, DOUBLE(2));
printf("six times %d is %d\n", 2, 3*DOUBLE(2));
```

The problem is that the last expression in the second printf is replaced by

```
3*2+2
```

which results in `8`, not `12`! The rule is that when using macros to build expressions, careful parenthesizing is necessary. Here's another example:

```
SQR(3+4)

/* expands to */

( 3+4 * 3+4 )
/* oh dear, still wrong! */
```

so, when formal parameters occur in the replacement text, you should look carefully at them too. Correct versions of `SQR` and `DOUBLE` are these:

```
#define SQR(x) ((x)*(x))
#define DOUBLE(x) ((x)+(x))
```

Macros have a last little trick to surprise you with, as this shows.

```
#include <stdio.h>
#include <stdlib.h>
#define DOUBLE(x) ((x)+(x))

main(){
      int a[20], *ip;

      ip = a;
      a[0] = 1;
      a[1] = 2;
      printf("%d\n", DOUBLE(*ip++));
      exit(EXIT_SUCCESS);
}
```

Example 7.1

Why is this going to cause problems? Because the replacement text of the macro refers to `*ip++` twice, so `ip`gets incremented twice. Macros should never be used with expressions that involve side effects, unless you check very carefully that they are safe.

Despite these warnings, they provide a very useful feature, and one which will be used a lot from now on.

7.3.3. # undef

The name of any `#defined` identifier can be forcibly forgotten by saying

```
#undef  NAME
```

It isn't an error to `#undef` a name which isn't currently defined.

This occasionally comes in handy. Chapter 9 points out that some library functions may actually be macros, not functions, but by undefing their names you are guaranteed access to a real function.

7.3.4. # include

This comes in two flavours:

```
#include <filename>
#include "filename"
```

both of which cause a new file to be read at the point where they occur. It's as if the single line containing the directive is replaced by the contents of the specified file. If that file contains erroneous statements, you can reasonably expect that the errors will be reported with a correct file name and line number. It's the compiler writer's job to get that right. The Standard specifies that at least eight nested levels of `# include` must be supported.

The effect of using brackets `<>` or quotes `" "` around the filename is to change the places searched to find the specified file. The brackets cause a search of a number of implementation defined places, the quotes cause a search of somewhere associated with the original source file. Your implementation notes must tell you the specific details of what is meant by ‘place’. If the form using quotes can't find the file, it tries again as if you had used brackets.

In general, brackets are used when you specify standard library header files, quotes are used for private header files—often specific to one program only.

Although the Standard doesn't define what constitutes a valid file name, it does specify that there must be an implementation-defined unique way of translating file names of the form `xxx.x` (where `x` represents a ‘letter’), into source file names. Distinctions of upper and lower case may be ignored and the implementation may choose only to use six significant characters before the ‘`.`’ character.

You can also write this:

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

to get the same effect as

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

but it's a rather roundabout way of doing it, and unfortunately it's subject to implementation defined rules about how the text between `<` and `>` is treated.

It's simpler if the replacement text for `NAME` comes out to be a string, for example

```
#define NAME "stdio.h"

#include NAME
```

There is no problem with implementation defined behaviour here, but the paths searched are different, as explained above.

For the first case, what happens is that the token sequence which replaces NAME is (by the rules already given)

```
<
stdio
.
h
>
```

and for the second case

```
"stdio.h"
```

The second case is easy, since it's just a *string-literal* which is a legal token for a `# include` directive. It is implementation defined how the first case is treated, and whether or not the sequence of tokens forms a legal *header-name*.

Finally, the last character of a file which is being `included` must be a plain newline. Failure to include a file successfully is treated as an error.

7.3.5. Predefined names

The following names are predefined within the preprocessor:

`__LINE__`
: The current source file line number, a decimal integer constant.

`__FILE__`
: The ‘name’ of the current source code file, a string literal.

`__DATE__`
: The current date, a string literal. The form is

```
Apr 21 1990
```

where the month name is as defined in the library function `asctime` and the first digit of the date is a space if the date is less than 10.

`__TIME__`
: The time of the translation; again a string literal in the form produced by asctime, which has the form `"hh:mm:ss"`.

`__STDC__`
: The integer constant 1. This is used to test if the compiler is Standard-conforming, the intention being that it will have different values for different releases of the Standard.

A common way of using these predefined names is the following:

```
#define TEST(x) if(!(x))\
      printf("test failed, line %d file %s\n",\
              __LINE__, __FILE__)

/**/
```

```
TEST(a != 23);

/**/
```

Example 7.2

If the argument to `TEST` gives a false result, the message is printed, including the filename and line number in the message.

There's only one minor caveat: the use of the `if` statement can cause confusion in a case like this:

```
if(expression)
      TEST(expr2);
else
      statement_n;
```

The else will get associated with the hidden if generated by expanding the `TEST` macro. This is most unlikely to happen in practice, but will be a thorough pain to track down if it ever does sneak up on you. It's good style to make the bodies of every control of flow statement compound anyway; then the problem goes away.

None of the names `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, `__STDC__` or defined may be used in `#define` or `#undef` directives.

The Standard specifies that any other reserved names will either start with an underscore followed by an upper case letter or another underscore, so you know that you are free to use any other names for your own purposes (but watch out for additional names reserved in Library header files that you may have included).

7.3.6. #line

This is used to set the value of the built in names `__LINE__` and `__FILE__`. Why do this? Because a lot of tools nowadays actually generate C as their output. This directive allows them to control the current line number. It is of very limited interest to the ‘ordinary’ C programmer.

Its form is

```
# line number optional-string-literal newline
```

The number sets the value of `__LINE__`, the string literal, if present, sets the value of `__FILE__`.

In fact, the sequence of tokens following `#line` will be macro expanded. After expansion, they are expected to provide a valid directive of the right form.

7.3.7. Conditional compilation

A number of the directives control conditional compilation, which allows certain portions of a program to be selectively compiled or ignored depending upon specified conditions. The directives concerned are: `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`together with the preprocessor unary operator defined.

The way that they are used is like this:

```
#ifdef  NAME
/* compile these lines if NAME is defined */
#endif
#ifndef NAME
/* compile these lines if NAME is not defined */
```

```
#else
/* compile these lines if NAME is defined */
#endif
```

So, `#ifdef` and `#endif` can be used to test the definition or otherwise of a given macro name. Of course the`#else` can be used with `#ifdef` (and`#if` or `#elif`) too. There is no ambiguity about what a given `#else`binds to, because the use of `#endif` to delimit the scope of these directives eliminates any possible ambiguity. The Standard specifies that at least eight levels of nesting of conditional directives must be supported, but in practice there is not likely to be any real limit.

These directives are most commonly used to select small fragments of C that are machine specific (when it is not possible to make the whole program completely machine independent), or sometimes to select different algorithms depending on the need to make trade-offs.

The `#if` and `#elif` constructs take a single integral constant expression as their arguments. Preprocessor integral constant expressions are the same as other integral constant expressions except that they must not contain cast operators. The token sequence that makes up the constant expression undergoes macro replacement, except that names prefixed by defined are not expanded. In this context, the expression `defined NAME` or `defined ( NAME )`evaluates to `1` if `NAME` is currently defined, `0` if it is not. Any other identifiers in the expression **including those that are C keywords** are replaced with the value `0`. Then the expression is evaluated. The replacement even of keywords means that `sizeof` can't be used in these expressions to get the result that you would normally expect.

As with the other conditional statements in C, a resulting value of zero is used to represent ‘false’, anything else is ‘true’.

The preprocessor always must use arithmetic with at least the ranges defined in the `<limits.h>` file and treats int expressions as long int and unsigned int as unsigned long int. Character constants do not necessarily have the same values as they do at execution time, so for highly portable programs, it's best to avoid using them in preprocessor expressions. Overall, the rules mean that it is possible to get arithmetic results from the preprocessor which are different from the results at run time; although presumably only if the translation and execution are done on different machines. Here's an example.

```
#include <limits.h>

#if ULONG_MAX+1 != 0
      printf("Preprocessor: ULONG_MAX+1 != 0\n");
#endif

      if(ULONG_MAX+1 != 0)
              printf("Runtime: ULONG_MAX+1 != 0\n");
```

Example 7.3

It is conceivable that the preprocessor might perform arithmetic with a greater range than that used in the target environment. In that case, the preprocessor expression `ULONG_MAX+1`might not ‘overflow’ to give the result of `0`, whereas in the execution environment, it **must**.

The following skeleton example illustrates the use of such constants and also the ‘conditional else’, `#elif`.

```
#define NAME    100

#if     ((NAME > 50) && (defined __STDC__))
/* do something */
#elif   NAME > 25
/* do something else*/
#elif   NAME > 10
```

```
/* do something else */
#else
/* last possibility */
#endif
```

A word of warning. These conditional compilation directives do not obey the same scope rules as the rest of C. They should be used sparingly, unless your program is rapidly to become unreadable. It is impossible to read C when it is laced with these things every few lines. The urge to maim the author of a piece of code becomes very strong when you suddenly come across

```
#else
      }
#endif
```

with no `#if` or whatever immediately visible above. They should be treated like chilli sauce; essential at times, but more than a tiny sprinkle is too much.

7.3.8. #pragma

This was the Standard Committee's way of ‘opening the back door’. It allows implementation-defined things to take place. If the implementation was not expecting what you wrote (i.e. doesn't recognize it), it is ignored. Here is a possible example:

```
#pragma byte_align
```

which could be used to tell the implementation that all structure members should be aligned on byte addresses - some processor architectures are able to cope with word-sized structure members aligned on byte addresses, but with a penalty in access speed being incurred.

It could, of course, mean anything else that the implementation chooses it to mean.

If your implementation doesn't have any special meaning for this, then it will have no effect. It will **not** count as an error.

It will be interesting to see the sort of things that this gets used for.

7.3.9. #error

This directive is followed by one or more tokens at the end of the line. A diagnostic message is produced by the compiler, which includes those tokens, but no further detail is given in the Standard. It might be used like this to abort a compilation on unsuitable hardware:

```
#include <limits.h>
#if CHAR_MIN > -128
#error character range smaller than required
#endif
```

which would be expected to produce some sort of meaningful compilation error and message.
