The Preprocessor - Chapter Summary

1 Introducing the Preprocessor

The preprocessor is part of a C compiler, so-called because it is used to process a program file before actual compilation takes place.

The preprocessor removes all comments from a program file.

Preprocessor control lines may be included anywhere in a C program, and under their direction, the preprocessor includes any other files needed by the program, replaces symbolic constants with their equivalent values, effects macro substitutions and decides which files will be compiled according to the conditional instructions it has received.

All control lines begin with a hash character `#`. Unlike C program statements, they do not end with a semicolon.

The preprocessor is a powerful tool, and one not found in most other high-level languages. The facilities it offers for using symbolic constants and macro substitutions enable the programmer to create code that is easier to read, while at the same time allowing easier maintenance of programs, since a change to a constant value or a parameter such as an array size need only be effected once to the appropriate `#define` control line, rather than many times throughout the program.

This last facility also improves the portability of C programs, since although important constants like the end-of-file character may have different values in different implementations of the language, the symbolic constant `EOF` will simply be replaced by the appropriate value defined in a particular implementation's version of the `stdio.h` file.

The preprocessor offers a further aid to portability by allowing the conditional compilation of system-dependent data, so that a program may contain all the information needed to run on a range of different systems, leaving it to the preprocessor to ensure that only the information appropriate to the current system is actually compiled.

2 The `#define` Control Line

The `#define` command can be used to give a symbolic name to a piece of text, in order to improve the readability and portability of program code. The general form of the control line is:

```
#define name text
```

The symbolic name in a `#define` control line is also known as a symbolic literal, and the text that the preprocessor substitutes for the macro is sometimes called the replacement string. The name of the symbolic literal must conform to the rules governing identifiers in C and should therefore begin with a letter or an underscore character, and be composed only of letters, numbers and the underscore character. By convention, the name of the symbolic literal is written in upper-case characters.

The replacement string may be a constant, an expression or even a complete C statement, but note that, being a string, it is itself a constant.

2.1 Nesting #define

Replacement strings may contain symbolic literals that other `#define` control lines will replace with further replacement strings. Note, however, that if the name occurs inside quotation marks within the replacement string, it will not be replaced:

```
#include <stdio.h>
#define LIMIT   (50)
#define LMT_STR ("\nLIMIT is %d.")
#define HALFWAY (LIMIT/2)

int main(void)
{
     int maximum = LIMIT;

     printf("\nHalfway is %d.", HALFWAY);
     printf(LMT_STR, maximum);

     return 0;
}
```

In the above example, `LIMIT` will be replaced by `50` in the first line of `main` and will be assigned to the variable `maximum`. In the second statement, `HALFWAY` will be replaced by `LIMIT/2`, which the preprocessor will then change to `50/2`, and which the `printf` statement will evaluate to 25 and include in its formatted output. However, in the final statement, although `LMT_STR` will be replaced by the string `"\nLIMIT is %d."`, the word `LIMIT` in the `printf` control string will not be further changed by the preprocessor because it occurs inside the quotes.

2.2 When to Use Symbolic Literals

It is good practice to use symbolic representations for all numerical constants, including physical constants such as PI or Planck's Constant, conversion factors and exchange rates, as this makes the code more readable. It is also a good idea to use symbolic constants for any parameters that may need to be modified as a program is developed. The following are examples of possible uses of the `#define` control line. The last is an example of a statement that might be used many times in a program and can be represented by a symbolic literal for both readability and convenience, since the complete statement need only be written once:

```
#define PLANCK      (6.626E-34 )
#define MAX_TURNS   (50 )
#define ERR_MSG printf("\nError! I cannot open %s.",source)
```

2.3 Using #define with Arguments - Macros

Symbolic literals in `#define` statements can have variables as arguments. The general form of the control line is:

```
     #define name(argument list) text
```

This instruction replaces the symbolic name with the text in the replacement string, substituting the arguments found in the argument list for the equivalent variables in the replacement string.

Thus, the control line

```
     #define DIVIDE(a,b) (a/b)
```

changes `DIVIDE(x,y)` into `x/y` and replaces `DIVIDE(n+4,m*2)` by `n+4/m*2`.

Note that there must be no spaces in the macro name itself, although they may be included in the replacement string.

2.4 Warning! Potential Problems with Macros!

The second example of the `DIVIDE` macro above demonstrates one of the potential problems with macro substitution.

Assuming that `n` and `m` are `int` variables and supposing that they have been assigned the values `12` and `4` respectively, it would seem reasonable to expect `DIVIDE(n+4, m*2)` to evaluate to `16` divided by `8`, which equals `2`. However, as shown above, the macro substitution replaces `DIVIDE(n+4, m*2)` by `n+4/m*2`, which, for the given values of `n` and `m`, and according to the rules of operator precedence, evaluates to `14`.

The division and multiplication operators are of equal precedence, but they associate left to right and both have higher precedence than the addition operator. The evaluation therefore proceeds through the following stages:

```
     12 + 4/4 * 2    /* first divide 4 by 4          */
     12 + 2          /* now add 12 and 2             */
     14              /* the answer!             */
```

In order to obtain the result `2`, the original `#define` control line should have been written:

```
     #define DIVIDE(a,b) ((a)/(b))
```

2.5 Automatic Conversion to String Literal

If an argument in a macro definition with a single argument appears in the replacement string preceded by a # character, the argument is made into a string literal. Thus, the directive

```
#define PRINT(str)  printf(#str , 180)
```

will replace the macro `PRINT(Score %d)` with the following:

```
printf("Score %d", 180);
```

2.6 Advantages and Disadvantages of Macros

Macros with arguments behave in many ways like functions. Note, however, that whereas when a function is called the arguments are first evaluated, and then their values are passed to the function, a macro substitution simply replaces the macro with the argument string, making no evaluation at all.

Macros provide in-line code by substituting replacement strings for each occurrence of the symbolic literal in the program. This produces longer programs that use more memory than equivalent programs using function calls, since a function need only be defined once, however many times it is invoked. On the other hand, function calls take longer to execute than in-line code, since the flow of control has to pass between the calling environment and the function and then back again every time the function is executed.

On these considerations, it is better to use function calls when memory space is at a premium, and occasionally better to use macro substitutions when a program's speed of execution is absolutely critical.

The main disadvantage of using macros instead of functions remains, however, that macros are prone to unhelpful side effects, as demonstrated in 2.4 above.

3 The `#include` Control Line

The `#include` control line has the general form:

```
#include <filename>
```
OR
```
#include "filename"
```
Either form of this control line instructs the preprocessor to replace the line by the contents of the file indicated.

Files included in this way at the top of a program file are known as header files, and by convention have names ending in the `.h` extension.

`#include` files provide a convenient way of collecting together commonly-used `#define` instructions, and prototypes and declarations of standard library functions.

Including the appropriate header file obviates the need to declare such standard data items every time they are required.

Programmers may also create their own header files of useful functions and macro definitions for sharing with different programs.

`#include` files may themselves include `#include` control lines calling on the preprocessor to include yet further files.

In larger programs, where individual files are compiled separately and then linked, it is desirable to `#include` the same header files in every file that needs to use any global data they declare.

4 Conditional Preprocessor Control Lines

The `#if` directive allows conditional compilation of sections of program code, and is typically used for achieving portability by conditionally including appropriate system-specific information.

The use of `#if` follows a similar pattern to the `if`, `if else` and `if else if` constructions, although braces are not used to tie together statements depending on the same condition. The optional `#elif` directive is the counterpart to the `else if` statement, and the optional `#else` corresponds to `else`. The end of a section of code that is dependent on a `#if` directive, or its accompanying series of options, is indicated by the use of the `#endif` directive.

The following program fragment demonstrates the use of `#if` to specify the system information required by different clients using the same program:

```
#if CLIENT == NYSONSOFT
     #include "nsoft.h"       /* client-specific header file */
     set24x80();
     puts("\nGreetings, NysonSoft!");

#elif CLIENT == DATADAY
     #include "ddata.h"
     set32x90();              /* sets screen dimensions */
     puts("\nGood day, DataDay!");

#else
     #include "anon.h"
     set40x100();
     puts("\nSalutations, valued client!"); /* default greeting */
#endif
```

Note that the lines enclosed by `#if` and `#endif` may themselves be preprocessor directives, and that `#if` control lines may be nested.

Some implementations of C specify that all preprocessor control lines must start at the left margin, but the ISO standard for C allows spaces and tabs to come before the `#`, which helpfully makes possible indentation of control lines in `#if` constructions.

5 Other Features of the Preprocessor

5.1 Concatenation of Adjacent Strings

The ISO standard for C specifies that syntactically-adjacent string literals are concatenated (i.e. joined) together into a single string.

```
     puts("\nThis works with puts()" " as well as printf().");
```
becomes:
```
     puts("\nThis works with puts() as well as printf().");
```

5.2 Location Identifiers

The preprocessor predefines the following identifiers:

`__LINE__`	gives the current line number in the source-code file
`__FILE__`	gives the name of the current file as a string literal

These identifiers are especially useful for creating traces that relate events during program execution to the source code:

```
     printf("\nThis function is at line %d in file %s.",
               __LINE__, __FILE__);
```

5.3 Undefining

Most compilers return a warning or error if an attempt is made to redefine a name that is already defined by another `#define` directive. The directive `#undef` undefines a name; that is, it cancels any previous definition and allows a new definition to be made:

```
     #define MAXIMUM (60 )
     #undef MAXIMUM
     #define MAXIMUM (100)
```

Note that it is not an error to undefine a name that has not actually been defined. This means that `#undef` can be used as a safety measure in a large program where a programmer is not sure whether the name to be defined has already been defined elsewhere in the program.

5.4 Definition as a Condition

The `#ifdef` command instructs that the subsequent lines of program should be compiled, up to the next `#else` or `#endif`, if the name specified in the `#ifdef` directive has already been defined by the preprocessor:

```
#ifdef MILE
     printf("\nA mile is %d kilometres.", MILE);
#endif
```

`#ifndef` is the negative equivalent of `#ifdef`, and directs that the ensuing instructions be followed if the name specified has not been defined by the preprocessor.

Note that even an empty definition, such as `#define MILE`, with a macro but no replacement string, is sufficient to meet the condition of the `#ifdef` directive, and to fail to meet that of `#ifndef`.