Unit 11 The Preprocessor

Structure:

11.1 Introduction

In the previous unit, you studied about the structures and unions in C. You studied how those structures and unions are used to bind similar types of data structures. In this unit, you will study about a Preprocessor. Preprocessor is a unique feature of C language. The C preprocessor provides several tools that are unavailable in other high-level languages. Conceptually, the “preprocessor'' is a translation phase that is applied to your source code before the compiler proper gets its hands on it. (Once upon a time, the preprocessor was a separate program, much as the compiler and linker may still be separate programs today.) Generally, the preprocessor performs textual substitutions on your source code, in three sorts of ways:

- **File inclusion:** inserting the contents of another file into your source file, as if you had typed it all in there.
- **Macro substitution:** replacing instances of one piece of text with another.
- **Conditional compilation:** Arranging that, depending on various circumstances, certain parts of your source code are seen or not seen by the compiler at all.

The next three sections will introduce these three preprocessing functions.

The syntax of the preprocessor is different from the syntax of the rest of C in several respects. First of all, the preprocessor is “line based." Each of the preprocessor directives we're going to learn about (all of which begin with the # character) must begin at the beginning of a line, and each ends at the end of the line. (The rest of C treats line ends as just another whitespace character, and doesn't care how your program text is arranged into lines.) Secondly, the preprocessor does not know about the structure of C -- about functions, statements, or expressions. It is possible to play strange tricks with the preprocessor to turn something which does not look like C into C (or vice versa). It's also possible to run into problems when a preprocessor substitution does not do what you expected it to, because the preprocessor does not respect the structure of C statements and expressions (but you expected it to). For the simple uses of the preprocessor we'll be discussing, you shouldn't have any of these problems, but you'll want to be careful before doing anything tricky or outrageous with the preprocessor. (As it happens, playing tricky and outrageous games with the preprocessor is considered sporting in some circles, but it rapidly gets out of hand, and can lead to bewilderingly impenetrable programs.)

Objectives:

After studying this unit, you should be able to:

- explain about a preprocessor
- include Files in the source program
- explain and implement Macro definition and substitution
- use parentheses in macro definitions

11.2 File Inclusion

A line of the form

```
#include <filename.h>
```

or

```
#include "filename.h"
```

causes the contents of the file filename.h to be read, parsed, and compiled at that point. (After filename.h is processed, compilation continues on the line following the #include line.) For example, suppose you got tired of retyping external function prototypes such as

```
extern int getline(char [], int);
```

at the top of each source file. You could instead place the prototype in a header file, perhaps getline.h, and then simply place

```
#include "getline.h"
```

at the top of each source file where you called getline. (You might not find it worthwhile to create an entire header file for a single function, but if you had a package of several related function, it might be very useful to place all of their declarations in one header file.) As we may have mentioned, that's exactly what the Standard header files such as stdio.h are collections of declarations (including external function prototype declarations) having to do with various sets of Standard library functions. When you use #include to read in a header file, you automatically get the prototypes and other declarations it contains, and you *should* use header files, precisely so that you will get the prototypes and other declarations they contain.

The difference between the <> and "" forms is where the preprocessor searches for filename.h. As a general rule, it searches for files enclosed in <> in central, standard directories, and it searches for files enclosed in "" in the “current directory," or the directory containing the source file that's doing the including. Therefore, "" is usually used for header files you've written, and <> is usually used for headers which are provided for you (which someone else has written).

The extension “.h", by the way, simply stands for “header," and reflects the fact that #include directives usually sit at the top (head) of your source files, and contain global declarations and definitions which you would otherwise put there. (That extension is not mandatory, you can theoretically name your own header files anything you wish, but .h is traditional, and recommended.)

As we've already begun to see, the reason for putting something in a header file, and then using #include to pull that header file into several different source files, is when the something (whatever it is) must be declared or defined consistently in all of the source files. If, instead of using a header file, you typed the something in to each of the source files directly, and the something ever changed, you'd have to edit all those source files, and if you missed one, your program could fail in subtle (or serious) ways due to the mismatched declarations (i.e. due to the incompatibility between the new declaration in one source file and the old one in a source file you forgot to change). Placing common declarations and definitions into header files means that if they ever change, they only have to be changed in one place, which is a much more workable system.

What should you put in header files?

- External declarations of global variables and functions. We said that a global variable must have exactly one *defining instance*, but that it can have *external declarations* in many places. We said that it was a grave error to issue an external declaration in one place saying that a variable or function has one type, when the defining instance in some other place actually defines it with another type. (If the two places are two source files, separately compiled, the compiler will probably not even catch the discrepancy.) If you put the external declarations in a header file, however, and include the header wherever it's needed, the declarations are virtually guaranteed to be consistent. It's a good idea to include the header in the source file where the defining instance appears, too, so that the compiler can check that the declaration and definition match. (That is, if you ever change the type, you do still have to change it in two places: in the source file where the defining instance occurs, and in the header file where the external declaration appears. But at least you don't have to change it in an arbitrary number of places, and, if you've set things up correctly, the compiler can catch any remaining mistakes.)
- Preprocessor macro definitions (which we'll meet in the next section).
- Structure definitions
- Typedef declarations

However, there are a few things *not* to put in header files:

- Defining instances of global variables. If you put these in a header file, and include the header file in more than one source file, the variable will end up multi defined.
- Function bodies (which are also defining instances). You don't want to put these in headers for the same reason -- it's likely that you'll end up with multiple copies of the function and hence “multiply defined" errors. People sometimes put commonly-used functions in header files and then use #include to bring them (once) into each program where they use that function, or use #include to bring together the several source files making up a program, but both of these are poor ideas. It's much better

to learn how to use your compiler or linker to combine together separately-compiled object files.

Since header files typically contain only external declarations, and should *not* contain function bodies, you have to understand just what does and doesn't happen when you #include a header file. The header file may provide the declarations for some functions, so that the compiler can generate correct code when you call them (and so that it can make sure that you're calling them correctly), but the header file does *not* give the compiler the functions themselves. The actual functions will be combined into your program at the end of compilation, by the part of the compiler called the *linker*. The linker may have to get the functions out of libraries, or you may have to tell the compiler/linker where to find them. In particular, if you are trying to use a third-party library containing some useful functions, the library will often come with a header file describing those functions. Using the library is therefore a two-step process: you must #include the header in the files where you call the library functions, *and* you must tell the linker to read in the functions from the library itself.

Self Assessment Questions:

1. # include is called ________.
2. Nesting of included files is not allowed. (True/False)
3. Defining instances of global variables is not recommended in the header files. (True/False)

11.3 Macro Definition and Substitution

A preprocessor line of the form

#define *name text*

defines a *macro* with the given name, having as its *value* the given replacement text. After that (for the rest of the current source file), wherever the preprocessor sees that name, it will replace it with the replacement text. The name follows the same rules as ordinary identifiers (it can contain only letters, digits, and underscores, and may not begin with a digit). Since macros behave quite differently from normal variables (or functions), it is customary to give them names which are all capital letters (or at least which begin with a capital letter). The replacement text can be absolutely anything--it's not restricted to numbers, or simple strings, or anything.

The most common use for macros is to propagate various constants around and to make them more self-documenting. We've been saying things like

```
char line[100];
...
getline(line, 100);
```

but this is neither readable nor reliable; it's not necessarily obvious what all those 100's scattered around the program are, and if we ever decide that 100 is too small for the size of the array to hold lines, we'll have to remember to change the number in two (or more) places. A much better solution is to use a macro:

```
#define MAXLINE 100
char line[MAXLINE];
...
getline(line, MAXLINE);
```

Now, if we ever want to change the size, we only have to do it in one place, and it's more obvious what the words MAXLINE sprinkled through the program mean than the magic numbers 100 did.

Since the replacement text of a preprocessor macro can be anything, it can also be an expression, although you have to realize that, as always, the text is substituted (and perhaps evaluated) later. No evaluation is performed when the macro is defined. For example, suppose that you write something like

```
#define A 2
#define B 3
#define C A + B
```

(this is a pretty meaningless example, but the situation does come up in practice). Then, later, suppose that you write

```
int x = C * 2;
```

If A, B, and C were ordinary variables, you'd expect x to end up with the value 10. But let's see what happens.

The preprocessor always substitutes text for macros exactly as you have written it. So it first substitutes the replacement text for the macro C, resulting in

```
int x = A + B * 2;
```

Then it substitutes the macros A and B, resulting in

```
int x = 2 + 3 * 2;
```

Only when the preprocessor is done doing all this substituting does the compiler get into the act. But when it evaluates that expression (using the normal precedence of multiplication over addition), it ends up initializing x with the value 8!

To guard against this sort of problem, it is always a good idea to include explicit parentheses in the definitions of macros which contain expressions. If we were to define the macro C as

```
#define C (A + B)
```

then the declaration of x would ultimately expand to

```
int x = (2 + 3) * 2;
```

and x would be initialized to 10, as we probably expected.

Notice that there does not have to be (and in fact there usually is *not*) a semicolon at the end of a #define line. (This is just one of the ways that the syntax of the preprocessor is different from the rest of C.) If you accidentally type

```
#define MAXLINE 100;                    /* WRONG */
```

then when you later declare

```
char line[MAXLINE];
```

the preprocessor will expand it to

```
char line[100;];                /* WRONG */
```

which is a syntax error. This is what we mean when we say that the preprocessor doesn't know much of anything about the syntax of C-- in this last example, the *value* or replacement text for the macro MAXLINE was the 4 characters 1 0 0 ; , and that's exactly what the preprocessor substituted (even though it didn't make any sense).

Simple macros like MAXLINE act sort of like little variables, whose values are constant (or constant expressions). It's also possible to have macros which look like little functions (that is, you invoke them with what looks like function call syntax, and they expand to replacement text which is a function of the actual arguments they are invoked with) but we won't be looking at these yet.

11.3.1 Macros with arguments

The preprocessor permits us to define more complex and more useful form of replacements. It takes the form:

#define *identifier*(f1, f2, …, fn) *string*

Notice that there is no space between the macro identifier and the left parenthesis. The identifiers f1, f2, …, fn are the formal macro arguments that are analogous to the formal arguments in a function definition.

There is a basic difference between the simple replacement discussed above and the replacement of macros with arguments. Subsequent occurrence of a macro with arguments is known as a *macro call*. When a macro is called, the preprocessor substitutes the *string*, replacing the formal parameters with actual parameters.

A simple example of a macro with arguments is

```
#define CUBE(x)  (x*x*x)
```

If the following statement appears later in the program

```
volume=CUBE(side);
```

then the preprocessor would expand this statement to:

```
volume=(side*side*side);
```

Consider the following statement:

```
volume=CUBE(a+b);
```

This would expand to:

```
volume=(a+b*a+b*a+b);
```

which would obviously not produce the correct results. This is because the preprocessor performs a blind text substitution of the argument a+b in place of x. This shortcoming can be corrected by using parentheses for each occurrence of a formal argument in the *string*.

Example:

```
#define CUBE(x)  ((x)*(x)*(x))
```

This would result in correct expansion of CUBE(a+b) as shown below:

```
volume=((a+b)*(a+b)*(a+b));
```

Some commonly used definitions are:

```
#define MAX(a,b)  (((a) > (b))?(a):(b))
#define MIN(a,b)   (((a) <(b))?(a):(b))
#define ABS(x)     (((x) > 0)?(x):(-(x)))
```

11.3.2 Nesting of Macros

We can also use one macro in the definition of another macro. That is, macro definitions may be nested. For instance, consider the following macro definitions:

```
#define  M       5
#define  N       M+1
#define  SQUARE(x)  ((x)*(x))
#define CUBE(x)        (SQUARE(x)*(x))
#define SIXTH(x)       (CUBE(x)*CUBE(x))
```

Self Assessment Questions

4. The role of preprocessor in macro substitution is ______________ substitution according to the macro definition.
5. Macros and functions are synonyms. (True/False)
6. Nesting of macros is not allowed. (True/False)

11.4 Conditional Compilation

The last preprocessor directive we're going to look at is #ifdef. If you have the sequence

```
#ifdef name
program text
#else
more program text
#endif
```

in your program, the code that gets compiled depends on whether a preprocessor macro by that *name* is defined or not. If it is (that is, if there has been a #define line for a macro called *name*), then “*program text* is compiled and “*more program text*’’ is ignored. If the macro is not defined, “*more program text*” is compiled and “*program text*’’ is ignored. This looks a lot like an if statement, but it behaves completely differently: an if statement controls which statements of your program are executed at run time, but #ifdef controls which parts of your program actually get compiled.

Just as for the if statement, the #else in an #ifdef is optional. There is a companion directive #ifndef, which compiles code if the macro is *not* defined (although the “#else clause" of an #ifndef directive will then be compiled if the macro *is* defined). There is also an #if directive which compiles code depending on whether a compile-time expression is true or false. (The expressions which are allowed in an #if directive are somewhat restricted, however, so we won't talk much about #if here.)

Conditional compilation is useful in two general classes of situations:

- You are trying to write a portable program, but the way you do something is different depending on what compiler, operating system, or computer you're using. You place different versions of your code, one for each situation, between suitable #ifdef directives, and when you compile the program in a particular environment, you arrange to have the macro names defined which select the variants you need in that environment. (For this reason, compilers usually have ways of letting you define macros from the invocation command line or in a configuration file, and many also predefine certain macro names related to the operating system, processor, or compiler in use. That way, you don't have to change the code to change the #define lines each time you compile it in a different environment.)

  For example, in ANSI C, the function to delete a file is remove. On older Unix systems, however, the function was called unlink. So if filename is a variable containing the name of a file you want to delete, and if you want to be able to compile the program under these older Unix systems, you might write

  ```
  #ifdef unix
          unlink(filename);
  #else
          remove(filename);
  #endif
  ```

  Then, you could place the line
  #define unix

at the top of the file when compiling under an old Unix system. (Since all you're using the macro unix for is to control the #ifdef, you don't need to give it any replacement text at all. *Any* definition for a macro, even if the replacement text is empty, causes an #ifdef to succeed.) (In fact, in this example, you wouldn't even need to define the macro unix at all, because C compilers on old Unix systems tend to predefine it for you, precisely so you can make tests like these.)

- You want to compile several different versions of your program, with different features present in the different versions. You bracket the code for each feature with #ifdef directives, and (as for the previous case) arrange to have the right macros defined or not to build the version you want to build at any given time. This way, you can build the several different versions from the same source code. (One common example is whether you turn debugging statements on or off. You can bracket each debugging printout with #ifdef DEBUG and #endif, and then turn on debugging only when you need it.)

  For example, you might use lines like this:

  ```
  #ifdef DEBUG
  printf("x is %d\n", x);
  #endif
  ```

  to print out the value of the variable x at some point in your program to see if it's what you expect. To enable debugging printouts, you insert the line

  ```
  #define DEBUG
  ```

  at the top of the file, and to turn them off, you delete that line, but the debugging printouts quietly remain in your code, temporarily deactivated, but ready to reactivate if you find yourself needing them again later. (Also, instead of inserting and deleting the #define line, you might use a compiler flag such as -DDEBUG to define the macro DEBUG from the compiler invocating line.)

  Conditional compilation can be very handy, but it can also get out of hand. When large chunks of the program are completely different depending on, say, what operating system the program is being compiled for, it's often better to place the different versions in separate source files, and then only use one of the files (corresponding to one of the versions) to build the program on any given system. Also, if you are using an ANSI Standard compiler and you are writing ANSI-compatible code, you usually won't need so much conditional compilation, because the Standard specifies exactly how the compiler must do certain things, and exactly which library functions it much provide, so you don't have to work so hard to accommodate the old variations among compilers and libraries.

Self Assessment Questions

7. In ANSI C, the function to delete a file is __________ .
8. On older Unix systems, the function was called ______________.
9. You can use _______________ directive which compiles code depending on whether a compile-time expression is true or false.

11.5 Summary

Preprocessor is a unique feature of C language. The C preprocessor provides several tools that are unavailable in other high-level languages. The preprocessor is a program that processes the source code before it passes through the compiler. Macro substitution is a process where an identifier in a program is replaced by a predefined text. The preprocessor permits us to define more complex and more useful form of replacements with macros with arguments. Nesting of macros is also allowed to simplify the complex macro definitions. An external file containing functions or macro definitions can be included as a part of a program.

11.6 Terminal Questions

1. What is the difference between a macro call and a function call?
2. List some of the Compiler Control preprocessor directives.
3. Distinguish between the following macro definitions:

   #include “filename”   and   #include <filename>
4. What is the preprocessor directive used to undefine a macro? Show with an example.
5. Write a macro definition to find the square of a number. Explain it with the help of an example.

11.7 Answers to Self Assessment Questions

1. Preprocessor directive
2. False
3. True
4. Text
5. False
6. False
7. remove
8. unlink.
9. #if

11.8 Answers to Terminal Questions

1. When a macro is called, the macro definition with string substitution will be inserted in the place of call in the source program whereas in a function call the execution control will transfer to the function.
2. #ifdef, #ifndef, #if, #else
3. Preprocessor searches for files enclosed in <> in central, standard directories, and it searches for files enclosed in "" in the current directory.
4. #undef (Refer the entire unit for more details)
5. #define SQUARE(x) ((x)*(x)) (Refer the entire unit for more details)

11.9 Exercises

1. Write a macro definition to find the largest of three numbers.
2. Write a macro definition to compare two numbers.
3. What are the advantages of using macro definitions in a program?
4. Define a macro, PRINT_VALUE which can be used to print two values of arbitrary type.
5. Identify errors, if any, in the following macro definition.
   #define ABS(x) (x>0)?(x):(-x)