

Chapter
2

**PREPROCESSOR
TECHNIQUES**

*Get on the
Fast Track!*



TM

**SYS-ED/
COMPUTER
EDUCATION
TECHNIQUES, INC.**

Objectives

You will learn:

- C Concepts.
- C Macro expansion.
- C C preprocessor commands.
- C Macros versus functions.
- C Manifest constants.
- C Macros with parameters.
- C Conditional compilation.
- C Header and other include commands.
- C Message pragma.
- C Preprocessor directives.



1 Concepts

The C preprocessor provides a means for modifying a source file before the actual compilation process takes place. The preprocessor makes substitutions in the source file (ie. preprocesses the file); thereby extending the C language and indirectly making programs more portable.

Conceptually, a preprocessor is considered to be distinct from the compiler. However, it does not have to be separate program.

Preprocessor substitutions may be made before or during the compilation process. From the standpoint of a programmer, it is important to recognize that the compiler treats source as though it had directly typed. It is the function of the preprocessor to make substitutions for the appropriate identifiers.

Preprocessor directives comprise a quasi language within C. While program statements are instructions to the microprocessor; preprocessor directives are instructions to the compiler.

One of the preprocessor's major functions is handling simple substitutions, where the replacement is a single value such as a number or a string constant. Additional preprocessor functions include file inclusion and conditional compilation.

2 Macro Expansion

The preprocessor can also handle macros. A macro is a name that is to be replaced by other material, such as a statement or an expression. The replacement text is known as the macro body. The macro preprocessor does not expand macro names placed inside quotation marks.

When the preprocessor encounters a macro identifier, it replaces the identifier with the macro body that has been specified. For manifest constants, a direct substitution is made. For macros with arguments, the information included as the argument must also be placed into the text. The substitution text then replaces the macro. The process of exchanging the macro cell for the information it represents is known as macro expansion.

The preprocessor expands macros until all required substitutions have been made. There are some limitations on possible substitutions, especially when it comes to macros with arguments.

The replacement of manifest constants and expanding macros are substitution processes, not assignments. The general rule is not to include assignment operators in preprocessor definitions. If assignments are included as assignment operators in preprocessor definitions, there is a high likelihood that there will be unexpected results.

If a semi-colon is inadvertently placed at the end of replacement text, in most cases it will result in a compiler error.

3 Preprocessor Convention and Syntax

1. All preprocessor commands, or directives, begin with #.

Most C compilers, allow a preprocessor command to be placed anywhere on the line, provided there is only a blank space before the #.

After the directive name, the command includes the constant or macro name and the replacement text for the macro or constant. Each portion of the command is separated by one or more blanks.

2. If the entire command won't fit on one line, it will be necessary to inform the preprocessor that the command continues on the next line. This entails ending the current line with a backslash (\).

When the preprocessor sees the backslash, it continues reading the line and the subsequent continuation line including the spaces. This is the reason why preprocessor commands do not have semi-colons at the end.

3. Statements may appear anywhere in the source code.

The exception is Conditional Compilation, where statements are usually located at the top.

4. The scope of the statement is the remainder of the source file.

3.1 C Preprocessor Commands

#define	Defines an identifier for substitution.
#elif	Combination of else and if.
#else	Result of false test for constant-expression.
#endif	Terminates action for conditional compilation.
#error	Allows you to include error messages.
#if	Tests a constant-expression for true.
#ifdef	Tests if an identifier is defined.
#ifndef	Tests if an identifier is not defined.
#include	Places another file in a line of code of current file.
#line	Resets line number to another value.
#pragma	Tells compiler to do something during compilation of the program
#undef	Undefines a previously defined identifier.

3.2 The #define Command

The most fundamental use of the #define directive is for assigning names to constants. It is strongly advised that the preprocessor's #define feature be used for all constants in programs.

A constant's value may have to be changed at a future date and it is convenient to be able to make all the changes at once by rewriting a single #define. With large programs (1000's of line of code) it is imperative that this technique be utilized.

Consider the following:

```
#define PI 3.14159
```

The preprocessor searches for all program lines beginning with the pound sign (#). After finding the #define directive, it goes through the entire program; substituting the phrase 3.14159 at every instance where it finds PI.

Why Use #define?

- C Use to make the program easier to read.
- C If a variable is used throughout the program and then a decision is made to change it, the #define commands eliminate the need to manually change each statement.
- C If 3.14159 has been defined to be PI in a #define directive, and the decision has been made to make it more exact by defining PI as 3.141592, it will only be necessary to make the change once in the #define directive.

Why Not Use Variable Names Instead of the #define?

It is inefficient because the compiler can generate faster and more compact code for constants than it can for variables.

Secondly, using a variable in lieu of a constant encourages sloppy thinking and makes the program more difficult to understand.

Finally, there is the inherent risk that a variable will be altered inadvertently during execution of the program. The net outcome being that the variable no longer will have the intended constant value.

#define Directive with Argument

A #define directive can take arguments, similar to the way that a function does.

When using a #define directive do not place any spaces in the identifier:

```
#define PR (n) printf("%.2f\n",n);
```

A good practice is to put parentheses around the entire text of any define directive that uses arguments. Parenthesis should also be placed around each of the arguments.

4 Macros vs. Functions

Macros and functions have the ability to perform many of the same tasks. In order to be certain that the intended order of precedence is followed use parenthesis.

In situations where the task to be carried out by the macro is not complex, macros typically are more convenient to use than functions. Assuming the task is not complex, when should a macro be used?

Each time a macro is used, the code it generates is inserted into the executable file for the program. This leads to multiple copies of the code. The code for a function, however only appears once. Therefore; using a function is more efficient in terms of memory size.

However, when calling a macro there is no wasting of time. When calling a function, the program has to arrange for the arguments to be transferred to the function and then be returned to the function's code. Therefore, a function takes less memory, but is slower to execute.

Excessive use of macros can also make a program difficult to read, since it requires constant referencing between the directives at the beginning of the program and the identifiers in the program body.

5 Macro Substitution

A macro is used to more concisely code a statement or statements which will be reused. It enhances the readability of the program and sometimes replaces a function call.

A macro can be defined with arguments, the replacement text will be dependent on the arguments passed.

```
#define identifier(arg1, arg2,...) replacement string
```

Advantages:

- C Avoids the overhead associated with a function call.
- C Arguments may be of any data type.
- C Makes the code easier to maintain.

Disadvantages:

- C Expressions used as arguments may be evaluated more than once.
- C To assure correct evaluation, care has to be taken using parentheses.
- C Tends to enlarge the source code.

6 Manifest Constants

The identifiers replaced by the values in a preprocessor directive are known as manifest constants.

A manifest constant is an identifier without arguments.

The preprocessor substitutes the replacement text for the constant name at the appropriate points in the program file. The substitution text begins immediately after the name. There must be at least one blank between the name and the replacement value.

7 Macros with Parameters

The preprocessor command `#define` can be used for function-like macros to make more complex substitutions. These macros have "slots", or formal parameters, into which specific values can be placed when calling the macro in a program. The formal parameters are essentially placeholders.

When calling the macro, one argument is passed for each formal parameter in the macro definition. The arguments being passed are known as actual parameters.

```
/* a macro for adding two values */
```

```
#define sum(a, b)      (a) + (b)
```

The `sum(a, b)` macro has two formal parameters, `a` and `b`. Formal parameters are separated from each other by commas. Spaces between formal parameters can be left in a definition, because the preprocessor will interpret everything to the right parenthesis as being part of the formal parameter list.

Developing macros with parameters can be difficult to write. It is strongly suggested that liberal use of parentheses be utilized when writing macros.

```
/* safest form of square, add macros */  
#define square(x) ((x) * (x)) /* note outer parentheses */  
#define add(a, b) ((a) + (b)) /* note outer parentheses */
```

8 The #undef Command

This command cancels the assignment of an identifier which previously had been assigned in the source file with a preprocessor statement.

```
#undef identifier
```

#undef word

#undef four

After the program defines a constant, it removes the definition by undefining the constant. The #undef command removes the definition of the constant or macro name following the command.

9 Conditional Compilation

Conditional compilation tests a constant expression or identifier. If the conditional compilation is true, then execution continues with the next line and carries on until reaching the end or `#endif`.

```
#if constant-expression
    #else
#endif
```

```
#ifdef identifier
    #else
#endif
```

```
#ifndef identifier
    #else
#endif
```

10 Conditional Commands

10.1 The #if and #endif Directives

If the constant expression is true, then the #if directive instructs the compiler to compile the statements that follow it.

If the constant expression is false, then the #else directive is followed. Execution continues until the #endif is reached.

The #ifdef and #endif Command

The #ifdef command causes the preprocessor to check whether it has a definition for the identifier. If a definition exists, the preprocessor processes any command it encounters in the file until one of the following is reached:

· #endif

· #else

· #elif

Each #ifdef command must be followed by its own #endif, #else, or #elif function.

The #elif command is a combination of the #else followed by an #if. It is most frequently used in the middle of a sequence of if-else loops.

10.2 The #ifndef Command

The #ifndef is another conditional command. The #ifndef is carried out if the name specified by the #ifndef has not been defined. A programmer can use the command to impose selective compilation on portions of the source file.

Conversely, this implies that other portions of the source file will not be compiled. This capability is frequently used in larger programs. The directive if (along with other directives) permits sections of a program to be compiled under certain selective circumstances.

A combination of preprocessor directives, #define, #if, and #endif, are often used.

Conditional compilation is useful in situations where portions of a program are to be tested, without having to test the entire program. When testing a program, start and stop statements can be inserted within the code and then removed when testing is complete. It is much easier to test code with a conditional compile. In this situation only #define statement will need to be changed in order to test a subroutine. It therefore becomes acceptable for all of the test statements to be kept in the listing.

Conditional compilation is also useful in cases where you require multiple versions of a program for running under different operating environments. For example, two versions of a program can be created, one to run on NT and the other to run on UNIX. Instead of creating two complete source files, statements can be that varied between the two versions could be grouped together. The statements would be enclosed by the appropriate #if, #else and #endif statements.

Then, by inserting a single statement such as #define UNIX, the program could be converted.

11 Header and Other Include Commands

The preprocessor is a programming tool. It helps to make programs easier to develop and the program code cleaner and faster. Utilizing macros to represent more complex expressions or arbitrary values, will result in programs being more readable and easier to revise.

Grouping definitions, either in one place in the program listing or in a separate file, will make it easier for to use different sets of definitions and macros.

The use of header files is an important step in program modification. Different versions of a program can be created using different header files. This technique serves to make programs more transportable. It entails putting specific values in the macro definitions in the header files rather than the original source code. Changing the macro definition, will serve to change the text that will be substituted during macro expansion.

Preprocessor commands can be collected in separate files. Likewise, C functions can also be collected in separate files. This approach makes it easier to build, read, and revise programs.

The most efficient way of using functions in other programs is to compile the file and then save the compiled version in the library file(s) that your program requires.

The INCLUDE Subdirectory contains the header or the include files.

11.1 The #include Command

This command instructs the preprocessor to substitute the contents of the designated file at the point in the source file where the #include command is found. It causes one source file to be included in another.

```
#include "proginfo.h"
```

This example causes the contents of file proginfo.h to be written into the source file location where the command was found. The same results could have been achieved by typing the contents of the proginfo.h directly into the source file.

There are two forms of the #include command.

- C Within quotation marks (" ") for files in the current directory.

The file name is specified as a string constant between double quotes ("proginfo.h"). When the file name is specified this way, the system searches for the file in the current directory. This form of the command is useful for including header files.

- C Within angle brackets < > for files in a default "include file" directory.

The file name is written without quotes.

The #include command instructs the preprocessor to search for the file in a directory specified for the implementation. This directory usually will be where the particular implementation keeps its header files. This form of the #include command is used primarily for including files used by the developers of the C compiler.

The #include commands can be inside files; thereby enabling the include files to be nested. Each compiler varies in the number of nesting levels that can be used.

This preprocessor command replaces the include statement with the contents of the named file. It is used for handling collection of #defines, and structure declarations. Included files are referred to as header files, and by convention are suffixed '.h'.

```
#include "filename"  
    or  
#include <Name>
```

First "filename" is searched for in the current directory, and then in the standard list (unless the full pathname is given).

The string name is searched for in the system standard list.

12 Other Preprocessor Commands

The remaining preprocessor commands are highly specialized.

12.1 The #error Command

The #error command allows specification of an error message to be displayed by the preprocessor. The message is displayed only if it finds particular kinds of situations for which it is testing.

For example, suppose you want to ask whether a macro has been defined and if the macro hasn't been defined have an error message displayed. The message would only occur during preprocessing and compilation, not during program execution.

```
#ifdef my_macro
    #error "ERROR: my_macro is undefined"
#endif

main()
{
    printf ( "Hi, there.");
}
```

12.2 The #line Command

The compiler ordinarily counts lines as it processes a program; it starts counting at line 1. The #line command provides the capability for specifying an arbitrary line number at any point in your source file. The compiler treats subsequent lines as having numbers continuing from the number that has been specified previously.

```
main()      /*line 1 in source file */
{
    printf( "Hello,\n")
    /*line 2 in source file */
    /*line 3 in source file */
    /*specify a new value for current line */
    #line 100
    printf( "world\n");      /*line 100 in source file */
}                            /*line 101 in source file */
```

This capability is useful when tracing the program execution in order to locate errors.

13 The #pragma Command

Pragmas are a direct product of the ANSI Committee's efforts to standardize the C language.

Designed as preprocessor directives, pragmas provide a means by which an implementation can allow a program to include nonstandard behavior or nonportable extensions.

Partial Listing of Pragmas Used for Controlling Optimization by Microsoft C

Pragma	Description
#pragma loop_opt(on/off)	Turns loop optimization ON/OFF.
#pragma intrinsic(func1,...)	Turns ON use of intrinsics for the specified functions.
#pragma function(func1,...)	Turns OFF use of intrinsics for the specified functions.
#pragma check_stack(on/off)	Turns stack checking ON/OFF.
#pragma pack(n)	Packs structures to the nth byte boundary.
#pragma alloc_text (segment, func1,...)	Names the segment in which the functions are to be placed.
#pragma same_seg(var1,...)	Specifies variables that should be placed in the same data segment.

The preprocessor directives instruct the compiler to perform an operations before compiling the program. Another instruction to the compiler, the pragma, tells the compiler to perform an operation during compilation.

The preprocessor command #pragma directs the implementation to behave in certain ways at the specified times.

Microsoft C includes a way of specifying whether the compiler should check if there will be sufficient stack space for loading and executing a particular function.

The `check_stack` pragma takes either `on` or `off` as its argument. As with other preprocessor directives, there is no semi-colon at the end of the directive line.

In the following example, the compiler checks only for function `large_stack()`:

```
1      #define ver-message "This is in DOS mode"
2      #pragma check_stack(on)
3      void large_stack()
4      {
5          printf( "in large_stack()\n");
6      }
7      #pragma check_stack(off)
8      void small_stack()
9      {
10         printf( "in small_stack()\n");
11     }
12     main()
13     {
14         small_stack();
15         large_stack();
16         printf( "\n%s\n", ver_message);
17     }
```

13.1 Message Pragma

When the compiler encounters this pragma, the argument to message is displayed during the compilation process. The argument to the pragma consists of either zero or additional string literals in double quotes or macros that expand to string literals.

This is demonstrated in the following program:

```
#pragma message(" Hello there ")

main()
{
    printf( "Hi");
}
```

During compilation, the "Hello there " message will appear on the screen. When running the program, the "Hi" message will be displayed. The message pragma directs what the compiler will write, not what the program will write.

In Microsoft C there are over a dozen pragmas.

If an action or declaration specified with a #pragma command is not supported by the implementation, the implementation ignores the pragma.

14 Preprocessor Directives

<code>#include<stdio.h></code>	Includes file <code>stdio.h</code> in source file start search in standard directory.
<code>#include "user.h"</code>	Includes file <code>user.h</code> in source file start search in current directory.
<code>#define MAX 10</code>	Replace <code>MAX</code> by <code>10</code> in source file.
<code>#define SUM(x,y) x+y</code>	Replace <code>SUM(x,y)</code> with <code>x+y</code> for all values of <code>x</code> and <code>y</code> .
<code>#define TEST</code>	Makes <code>TEST</code> true (remove to make false).
<code>#if defined(TEST)</code>	If its a true statement, then this will be performed.
<code>#else TEST</code>	Otherwise, this will be done.
<code>#endif TEST</code>	Delimiter for <code>#ifdef</code> and <code>#else</code> .
<code>#undef TEST</code>	Undefines <code>TEST</code> .