**PREPROCESSOR DIRECTIVES**

INTRODUCTION AND MOTIVATION

1. The first step in compiling any C program is the preprocessor, a sort of automated editor that modifies ( a copy of ) the source code before passing it on to the compiler to translate into machine language code.
2. One of the tasks of the preprocessor is to strip off all comments, which the compiler ignores.
3. The preprocessor also responds to directives in the code, which give the preprocessor explicit instructions on how to edit the source code before passing it on to the compiler.

I. MACRO SUBSTITUTION

The #define Preprocessor Directive

* The #define directive is used to "define" preprocessor "variables", which can then be used in one of three ways, as shown in the following three sections.
* It is also possible to define preprocessor variables when invoking the compiler, either as command-line arguments or through the IDE.
* Sometimes the preprocessor only needs to know whether a particular preprocessor variable is defined or undefined.
* In this case, use #define and #undef to turn such variables "on" or "off"

Then you can turn all such blocks on or off at once using

#define DEBUG

or

#undef DEBUG

* Using #define For Text Substitution

The most common use for #define is for automatic substitution of a text string.

* Using #define To Define Macros

In addition to simple variable substitutions, it is also possible to create pre-processor macros that take arguments.

Macros operate much like functions, but because they are expanded in place they are generally faster, since they do not invoke the overhead of transferring control over to a function and then transferring control back again.

However there are some potential dangers when using macros instead of functions, as described below.

Example:

1. Object like macro----🡪 #define PI 3.14159
2. Function like macro-🡪 #define SQUARE(x) (x \* x)

Explanation:

area = PI \* SQUARE( radius );

which will be seen by the compiler as:

area = 3.14159 \* (radius \* radius);

Note:  It is very important to include parentheses around all variables used in a preprocessor macro, as well as around the entire definition.

Exercise:  What will be the effects of the following?  Which is correct?:

#define SQUARE(x) x \* x

...

abSquared = SQUARE( a + b );

#define SQUARE(x) (x) \* (x)

...

a\_over\_b\_squared = a / SQUARE( b );

#define SQUARE(x)       ( (x) \* (x) )

...

answer = SQUARE( a + b ) / SQUARE( b );

Landmine!  Do not use auto-increment or decrement operators, assignment statements, or function calls in combination with preprocessor macros.  Consider:

for( i = 0; i < 10; )

printf( "%d squared = %d\n, i, SQUARE( i++ ) );

* Preprocessor macros can take multiple arguments:

#define A\_MOD\_B(a,b) ( (a) % (b) )

* Preprocessor macros can also make use of previously defined macros

#define MAX(a,b) ( (a) > (b) ? (a) : (b) )

#define MAX3(a,b,c) ( ((a) > (b)) ? MAX((a),(c)) : MAX((b),(c)) )

or #define MAX3(a,b,c) ( MAX((a), MAX((b),(c)) )

* Preprocessor macros can span multiple lines only if the end of line is explicitly escaped:

#define MIN(a,b) ( (a) < (b) ? \

(a) : \

(b) )

Note that macros do not check the types of their arguments, which can have both good and bad consequences.

II. PREDEFINED MACROS

There are a number of macros automatically pre-defined by the compiler, inlcuding the following list. Note that each one begins and ends with TWO underscore characters:

| Predefined Macro | Value |
| --- | --- |
| \_\_DATE\_\_ | String containing the current date |
| \_\_FILE\_\_ | String containing the file name |
| \_\_LINE\_\_ | Integer representing the current line number |
| \_\_STDC\_\_ | If follows ANSI standard C, then value is a nonzero integer |
| \_\_TIME\_\_ | String containing the current date. |

III.PRAGMA PREPROCESSOR DIRECTIVE

The #pragma Preprocessor Directive-🡪 It is used to give the preprocessor ( and compiler ) specific details on exactly how to compile the program. For example, specific compiler warnings can be ignored, or warning levels changed up or down for different sections of code.

Examples:

#pragma page( ) // Forces a form feed in the listing

#pragma warning( disable: 2044 ) // Disables warning number 2044

#pragma line 100 // Sets the current line number to 100 for listing and reporting purposes.

IV. SOURCE FILE INCLUSION DIRECTIVE

#include preprocessor directive-🡪This directive includes a file into code.

It has two possible forms:

#include <file>

or

#include ``file''

* <file> tells the compiler to look where system include files are held.   
  Usually UNIX systems store files in user include directory.
* ``file'' looks for a file in the current directory (where program was run from)

Included files usually contain C prototypes and declarations from header files and not (algorithmic) C code (SEE next Chapter for reasons)

V. CONDITIONAL COMPILATION DIRECTIVE

#if -- Conditional inclusion🡪 It evaluates a constant integer expression. You always need a #endif to delimit end of statement.

We can have else etc. as well by using #else and #elif -- else if.

Another common use of #if is with:

#ifdef

-- if defined

#ifndef

-- if not defined

These are useful for checking if macros are set -- perhaps from different program modules and header files.

#if 1

/\* This block will be included \*/

#endif

#if 0

/\* This block will not be included \*/

#endif

* The conditional expression could contain any C operator except for the assignment operators, the increment and decrement operators, the address-of operator, and the sizeof operator.
* One unique operator used in preprocessing and nowhere else is the defined operator. It returns 1 if the macro name, optionally enclosed in parentheses, is currently defined; 0 if not.
* The #endif command ends a block started by #if, #ifdef, or #ifndef.
* The #elif command is similar to #if, except that it is used to extract one from a series of blocks of code. E.g.:

#if /\* some expression \*/

:

#elif /\* another expression \*/

:

/\* imagine many more #elifs here ... \*/

:

#else

/\* The optional #else block is selected if none of the previous #if or

#elif blocks are selected \*/

:

#endif /\* The end of the #if block \*/

#ifdef and #ifndef

* The #ifdef command is similar to #if, except that the code block following it is selected if a macro name is defined. In this respect,

#ifdef NAME

is equivalent to

#if defined NAME

* The #ifndef command is similar to #ifdef, except that the test is reversed:

#ifndef NAME

is equivalent to

#if !defined NAME

Example-1:

For example, to set integer size for a portable C program between TurboC (on MSDOS) and Unix (or other) Operating systems. Recall that TurboC uses 16 bits/integer and UNIX 32 bits/integer.

Assume that if TurboC is running a macro TURBOC will be defined. So we just need to check for this:

  #ifdef TURBOC

  #define INT\_SIZE 16

#else

#define INT\_SIZE 32

#endif

Example-2:

As another example if running program on MSDOS machine we want to include file msdos.h otherwise a default.h file. A macro SYSTEM is set (by OS) to type of system so check for this:

#if SYSTEM == MSDOS

#include <msdos.h>

#else

#include ``default.h''

#endif

The -E command line is worth mentioning just for academic reasons. It is not that practical a command. The -E command will force the compiler to stop after the preprocessing stage and output the current state of your program. Apart from being debugging aid for preprocessor commands and also as a useful initial learning tool (try this option out with some of the examples above) it is not that commonly used.

  #ifdef OS\_MSDOS

  #include <msdos.h>

#elifdef OS\_UNIX

#include ``default.h''

#else

#error Wrong OS!!

#endif

VI. ERROR DIRECTIVE

#error

The #error directive halts compilation. When one is encountered the standard specifies that the compiler should emit a diagnostic containing the remaining tokens in the directive. This is mostly used for debugging purposes.

They are represented in 2 ways

* #error
* #error token-sequence

VII. LINE DIRECTIVE

# line

number "string" -- informs the preprocessor that the number is the next number of line of input. "string" is optional and names the next line of input. This is most often used with programs that translate other languages to C. For example, error messages produced by the C compiler can reference the file name and line numbers of the original source files instead of the intermediate C (translated) source files.

They are represented in 2 ways

* #line constant
* #line constant “filename”

VIII. NULL DIRECTIVE

#null--🡪 It has o effect on the code.