Assignment 1: The Preprocessor

# Preprocessor Directives:

#include- The #include directive tells the preprocessor to treat the contents of a specified file as if those contents had appeared in the source program at the point where the directive appears. You can organize constant and macro definitions into include files and then use #include directives to add these definitions to any source file. Include files are also useful for incorporating declarations of external variables and complex data types. You need to define and name the types only once in an include file created for that purpose.

#include "path-spec"

#include <path-spec>

The path-spec is a filename optionally preceded by a directory specification. The filename must name an existing file. The syntax of the path-spec depends on the operating system on which the program is compiled.

#define, #undef - Creates and releases compile-time constants.

You can use the #DEFINE and #UNDEF preprocessor directives to create compile-time constants in programs. By creating constants with #DEFINE instead of using variables, you can reduce memory consumption, increase performance, and simplify programs.

To create a constant with #DEFINE, specify the constant's name with ConstantName and its value with eExpression. When the program is compiled, text substitution is performed and the constant value expression is substituted for the constant name wherever it appears in the program. You can stop the substitution for the constant by issuing #UNDEF.

#DEFINE ConstantName eExpression...

#UNDEF ConstantName

#if, #ifdef, #ifndef, #elif, #endif – The #if directive, with the #elif, #else, and #endif directives, controls compilation of portions of a source file. If the expression you write (after the #if) has a nonzero value, the line group immediately following the #if directive is retained in the translation unit.

The #ifdef and #ifndef directives perform the same task as the #if directive when it is used with defined( identifier ).

#if acts just like an if statement but is used for directives, which allows us to easily read and interpret the other 4 directives in simpler English. #ifdef means “if it is defined”, #ifndef means “if it is not defined”, #elif means acts just like else if in standard programming but for directives, therefore meaning “else if”, and #endif ends an if statement just like a } (closed curly bracket) in standard programming.

Each #if directive in a source file must be matched by a closing #endif directive. Any number of #elif directives can appear between the #if and #endif directives, but at most one #else directive is allowed. The #else directive, if present, must be the last directive before #endif.

The #if, #elif, #else, and #endif directives can nest in the text portions of other #if directives. Each nested #else, #elif, or #endif directive belongs to the closest preceding #if directive.

All conditional-compilation directives, such as #if and #ifdef, must be matched with closing #endif directives prior to the end of file; otherwise, an error message is generated. When conditional-compilation directives are contained in include files, they must satisfy the same conditions: There must be no unmatched conditional-compilation directives at the end of the include file.

Macro replacement is performed within the part of the command line that follows an #elif command, so a macro call can be used in the constant-expression.

The preprocessor selects one of the given occurrences of text for further processing. A block specified in text can be any sequence of text. It can occupy more than one line. Usually text is program text that has meaning to the compiler or the preprocessor.

The preprocessor processes the selected text and passes it to the compiler. If text contains preprocessor directives, the preprocessor carries out those directives. Only text blocks selected by the preprocessor are compiled.

The preprocessor selects a single text item by evaluating the constant expression following each #if or #elif directive until it finds a true (nonzero) constant expression. It selects all text (including other preprocessor directives beginning with #) up to its associated #elif, #else, or #endif.

If all occurrences of constant-expression are false, or if no #elif directives appear, the preprocessor selects the text block after the #else clause. If the #else clause is omitted and all instances of constant-expression in the #if block are false, no text block is selected.

#error - Error directives produce compiler-time error messages.

#error token-string

Example:

#if !defined(\_\_cplusplus)

#error C++ compiler required.

#endif

When #error directives are encountered, compilation terminates.

#pragma - Each implementation of C and C++ supports some features unique to its host machine or operating system. Some programs, for instance, need to exercise precise control over the memory areas where data is placed or to control the way certain functions receive parameters. The #pragma directives offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages. Pragmas are machine- or operating system-specific by definition, and are usually different for every compiler.

Pragmas can be used in conditional statements, to provide new preprocessor functionality, or to provide implementation-defined information to the compiler.

#pragma token-string

If the compiler finds a pragma it does not recognize, it issues a warning, but compilation continues.

Some pragmas provide the same functionality as compiler options. When a pragma is encountered in source code, it overrides the behavior specified by the compiler option.

# Macros

A macro is a series of commands and instructions that you group together as a single command to accomplish a task automatically. Macros allow you to automate repetitive actions.

# - The number-sign or "stringizing" operator (#) converts macro parameters (after expansion) to string constants. It is used only with macros that take arguments. If it precedes a formal parameter in the macro definition, the actual argument passed by the macro invocation is enclosed in quotation marks and treated as a string literal. The string literal then replaces each occurrence of a combination of the stringizing operator and formal parameter within the macro definition.

White space preceding the first token of the actual argument and following the last token of the actual argument is ignored. Any white space between the tokens in the actual argument is reduced to a single white space in the resulting string literal. Thus, if a comment occurs between two tokens in the actual argument, it is reduced to a single white space. The resulting string literal is automatically concatenated with any adjacent string literals from which it is separated only by white space.

Further, if a character contained in the argument usually requires an escape sequence when used in a string literal (for example, the quotation mark (") or backslash (\) character), the necessary escape backslash is automatically inserted before the character. The following example shows a macro definition that includes the stringizing operator and a main function that invokes the macro:

#define stringer( x ) printf( #x "\n" )

int main()

{

stringer( In quotes in the printf function call\n );

stringer( "In quotes when printed to the screen"\n );

stringer( "This: \" prints an escaped double quote" );

}

Such invocations would be expanded during preprocessing, producing the following code:

int main()

{

printf( "In quotes in the printf function call\n" "\n" );

printf( "\"In quotes when printed to the screen\"\n" "\n" );

printf( "\"This: \\\" prints an escaped double quote\"" "\n" );

}

When the program is run, screen output for each line is as follows:

In quotes in the printf function call

"In quotes when printed to the screen"

"This: \" prints an escaped double quotation mark"

## - The double-number-sign or "token-pasting" operator (##), which is sometimes called the "merging" operator, is used in both object-like and function-like macros. It permits separate tokens to be joined into a single token and therefore cannot be the first or last token in the macro definition.

If a formal parameter in a macro definition is preceded or followed by the token-pasting operator, the formal parameter is immediately replaced by the unexpanded actual argument. Macro expansion is not performed on the argument prior to replacement.

Then, each occurrence of the token-pasting operator in token-string is removed, and the tokens preceding and following it are concatenated. The resulting token must be a valid token. If it is, the token is scanned for possible replacement if it represents a macro name. The identifier represents the name by which the concatenated tokens will be known in the program before replacement. Each token represents a token defined elsewhere, either within the program or on the compiler command line. White space preceding or following the operator is optional.

This example illustrates use of both the stringizing and token-pasting operators in specifying program output:

#define paster( n ) printf( "token" #n " = %d", token##n )

int token9 = 9;

If a macro is called with a numeric argument like

paster( 9 );

the macro yields

printf( "token" "9" " = %d", token9 );

which becomes

printf( "token9 = %d", token9 );

## ANSI-Compliant Predefined Macros

The compiler recognizes 10 predefined ANSI C macros, and the Microsoft C++ implementation provides several more. These macros take no arguments and cannot be redefined. Their value, except for **\_\_LINE\_\_** and **\_\_FILE\_\_**, must be constant throughout compilation. Some of the predefined macros listed below are defined with multiple values.

\_\_LINE\_\_ - The line number in the current source file. The line number is a decimal integer constant. It can be altered with a #line directive.

\_\_FILE\_\_ - The name of the current source file. \_\_FILE\_\_ expands to a string surrounded by double quotation marks.

You can create your own wide string version of \_\_FILE\_\_ as follows:

#include <stdio.h>

#define WIDEN2(x) L ## x

#define WIDEN(x) WIDEN2(x)

#define \_\_WFILE\_\_ WIDEN(\_\_FILE\_\_)

wchar\_t \*pwsz = \_\_WFILE\_\_;

int main()

{

}

\_\_DATE\_\_ - The compilation date of the current source file. The date is a string literal of the form Mmm dd yyyy. The month name Mmm is the same as for dates generated by the library function asctime declared in TIME.H.

\_\_TIME\_\_ - The most recent compilation time of the current source file. The time is a string literal of the form hh:mm:ss.