Chp23.

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required preprocessing before the actual compilation. We'll refer to the C Preprocessor as CPP. All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives: **Directive Description** #define Substitutes a preprocessor macro. #include Inserts a particular header from another file. #undef Undefines a preprocessor macro. #ifdef Returns true if this macro is defined. #ifndef Returns true if this macro is not defined. #if Tests if a compile time condition is true. #else The alternative for #if. #elif #else and #if in one statement. #endif Ends preprocessor conditional. #error Prints error message on stderr. #pragma Issues special commands to the compiler, using

Analyze the following examples to understand various directives. `#define MAX_ARRAY_LENGTH 20` This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability. `#include <stdio.h> #include "myheader.h"` These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file. `#undef FILE_SIZE #define FILE_SIZE 42` It tells the CPP to undefine existing FILE_SIZE and define it as 42. `#ifndef MESSAGE #define MESSAGE "You wish!" #endif` It tells the CPP to define MESSAGE only if MESSAGE isn't already defined. `#ifdef DEBUG /* Your debugging statements here */ #endif` It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the *-DDEBUG* flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on-the-fly during compilation. **Predefined Macros** ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Macro Description __DATE__ The current date as a character literal in "MMM DD YYYY" format. __TIME__ The current time as a character literal in "HH:MM:SS" format. __FILE__ This contains the current filename as a string literal. __LINE__ This contains the current line number as a decimal constant. __STDC__ Defined as 1 when the compiler complies with the ANSI standard. Let's try the following example: `#include <stdio.h> main() { printf("File :%s\n", __FILE__ ); printf("Date :%s\n", __DATE__ ); printf("Time :%s\n", __TIME__ ); printf("Line :%d\n", __LINE__ ); printf("ANSI :%d\n", __STDC__ ); }` When the above code in a file **test.c** is compiled and executed, it produces the following result: `File :test.c Date :Jun 2 2012 Time :03:36:24 Line :8 ANSI :1`

Preprocessor Operators The C preprocessor offers the following operators to help create macros: **The Macro Continuation (\) Operator** A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line. For example: `#define message_for(a, b) \ printf(#a " and " #b ": We love you!\n")` **Stringize (#)** The stringize or number-sign operator (#), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list. For example: `#include <stdio.h> #define message_for(a, b) \ printf(#a " and " #b ": We love you!\n") int main(void) { message_for(Carole, Debra); return 0; }` When the above code is compiled and executed, it produces the following result: `Carole and Debra: We love you!` **Token Pasting (##)** The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example: `#include <stdio.h>`

`#define tokenpaster(n) printf ("token" #n " = %d", token##n) int main(void) { int token34 = 40; tokenpaster(34); return 0; }` When the above code is compiled and executed, it produces the following result: `token34 = 40` It happened so because this example results in the following actual output from the preprocessor: `printf ("token34 = %d", token34);` This example shows the concatenation of token##n into token34 and here we have used both **stringize** and **token-pasting**. **The Defined() Operator** The preprocessor **defined** operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows: `#include <stdio.h> #if !defined (MESSAGE) #define MESSAGE "You wish!" #endif int main(void) { printf("Here is the message: %s\n", MESSAGE); return 0; }`

When the above code is compiled and executed, it produces the following result: `Here is the message: You wish!` **Parameterized Macros** One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows: `int square(int x) { return x * x; }` We can rewrite the above code using a macro as follows: `#define square(x) ((x) * (x))` Macros with arguments must be defined using the **#define** directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis. For example: `#include <stdio.h> #define MAX(x,y) ((x) > (y) ? (x) : (y)) int main(void)`

`{ printf("Max between 20 and 10 is %d\n", MAX(10, 20)); return 0; }` When the above code is compiled and executed, it produces the following result: `Max between 20 and 10 is 20`

Chp24.

A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler. You request to use a header file in your program by including it with the C preprocessing directive **#include**, like you have seen inclusion of **stdio.h** header file, which comes along with your compiler. Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program. A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in the header files and include that header file wherever it is required. **Include Syntax** Both the user and the system header files are included using the preprocessing directive **#include**. It has the following two forms: `#include <file>` This form is used for system header files. It searches for a file named 'file' in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code. `#include "file"` This form is used for header files of your own program. It searches for a file named 'file' in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code. **Operation** The **#include** directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the **#include** directive. For example, if you have a header file header.h as

follows:

`char *test (void);` and a main program called *program.c* that uses the header file, like this: `int x; #include "header.h" int main (void) { puts (test ()); }` the compiler will see the same token stream as it would if program.c read. `int x; char *test (void); int main (void) { puts (test ()); }` **Once-Only Headers** If a header file happens to be included twice, the compiler will process its contents twice and it will result in an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this: `#ifndef HEADER_FILE #define HEADER_FILE the entire header file file #endif` This construct is commonly known as a wrapper **#ifndef**. When the header is included again, the conditional will be false, because HEADER_FILE is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Computed Includes Sometimes it is necessary to select one of the several different header files to be included into your program. For instance, they might specify configuration parameters to be used on different sorts of operating systems. You could do this with a series of conditionals as follows: `#if SYSTEM_1 # include "system_1.h" #elif SYSTEM_2 # include "system_2.h" #elif SYSTEM_3 ... #endif` But as it grows, it becomes tedious, instead the preprocessor offers the ability to use a macro for the header name. This is called a **computed include**. Instead of writing a header name as the direct argument of **#include**, you simply put a macro name there: `#define SYSTEM_H "system_1.h" ... #include SYSTEM_H` SYSTEM_H will be expanded, and the preprocessor will look for system_1.h as if the **#include** had been written that way originally. SYSTEM_H could be defined by your Makefile with a -D option.