**13.1 Introduction**

13.1 Preprocessing occurs

(a) before a program is compiled.

(b) during compilation.

(c) after compilation but before execution.

(d) immediately before execution.

ANS: (a)

13.2 Preprocessor directives

(a) begin with include

(b) are C statements.

(c) are ignored if whitespace characters before them on the same line.

(d) do not end in a semicolon.

ANS: (d)

13.3. Which is *not* a capability of the preprocessor?

a) inclusion of other files in the file being compiled

b) definition of functions

c) definition of symbolic constants

d) definition of macros

ANS: (b)

13.4. Which statement about the preprocessor is *false*?

a) it can perform conditional compilation of program code

b) it can perform conditional execution of preprocessor directives

c) all preprocessor directives begin with #

d) all preprocessor directives must end in a semicolon.

ANS: (d)

**13.2 #include Preprocessor Directive**

13.5. The #includepreprocessor directive causes a(n) \_\_\_\_\_\_\_\_\_\_\_\_ to be included in place of the directive.

(a) copy of a file

(b) # character

(c) pointer to a file

(d) bitfield

ANS: (a)

13.6 Which include statement is usually appropriate for user-defined files?

(a) include <filename>

(b) #include "filename"

(c) Both (a) and (b) are appropriate.

(d) Neither (a) nor (b) is appropriate.

ANS: (b)

13.7. The #include directive

a) must begin in column 1

b) includes a file in the program at execution time

c) includes a file in the program before compilation

d) appends a file to the end of the program

ANS: (c)

13.8. Which is a correct form of the #include directive

a) #include ‘filename’

b) #include /filename/

c) #include #filename

d) #include <filename>

ANS: (d)

13.9. Which statement about the #include directive is *false*?

a) The difference between the various forms of the #include directive is the location the preprocessor searches for the file to be included.

b) If the file name is enclosed in quotes, the preprocessor searches in the same directory as the file being compiled for the file to be included.

c) If the file name is enclosed in angle brackets, the preprocessor searches in the same directory as the file being compiled for the file to be included.

d) The preprocessor searches for standard library header files in an implementation-dependent manner.

ANS: (c)

13.10. Which statement about the #include preprocessor directive is *false*?

a) It is used with programs consisting of several source files that are to be compiled together.

b) A header file containing definitions common to separate program files is often created and included in a program with #include.

c) #include header files often contain structure and union definitions, enumerations and function prototypes.

d) It includes standard library header files such as standardio.h.

ANS: (d)

**13.3 #define Preprocessor Directive: Symbolic Constants**

13.11. const variables are preferred to symbolic constants (from #define) because

(a) constvariables take up less memory.

(b) symbolic constants can be changed.

(c) const variable names are visible to the compiler

(d) const variables do not have to be of a specific data type.

ANS: (c)

13.12 Which of the following statements is correct?

(a) #define X = 3

(b) #define X 3, Y 4

(c) #define X 3

(d) #define X:3

ANS: (c)

13.13. Which statement is *false*?

a) Symbolic constants are constants represented as symbols.

b) Macros are operations defined as symbols.

c) All text replacement with symbolic constants and macros occurs before the program is compiled.

d) Symbolic constants may be redefined with new values.

ANS: (d)

13.14. Which statement about symbolic constants is *true*?

a) They make programs run faster.

b) They always make programs clearer.

c) They make it more difficult to modify programs.

d) Using meaningful names for symbolic constants helps make programs more self-documenting.

ANS: (d)

13.15. What is a problem with the preprocessor statement:

#define PI 3.14159;

a) It will make a program run slower.

b**)** #define should be #def

c) PI should be spelled with lowercase letters.

d) The semicolon is part of the substitution text, so 3.14159; will be substituted whereever PI is used and this could lead to syntax errors.

ANS: (d)

**13.4 #define Preprocessor Directive: Macros**

13.16. If the macro

#define RECTANGLE\_AREA(x, y) ((x) \* (y))

has been defined. Then the line

rectArea = RECTANGLE\_AREA(a + 4, b + 7);

will be expanded to

(a) rectArea = 11;

(b) rectArea = (a + 4 \* b + 7);

(c) rectArea = ((a + 4) \* (b + 7));

(d) RECTANGLE\_AREA(a + 4 , b + 7);

ANS: (c)

13.17. The #undef preprocessor directive

(a) can only be used once per macro name.

(b) can only be used on symbolic constants.

(c) allows macros to be redefined with #define later in the program.

(d) must be called for all symbolic constants before the end of the file.

ANS: (c)

13.18. Which of the following is *false*? Macros

(a) must be defined with arguments.

(b) are replaced by their replacement text during execution time.

(c) cannot be redefined once defined.

(d) have no data type checking.

ANS: (a)

13.19. Macros have the disadvantage, when compared to functions, of

(a) having the overhead of a function call.

(b) increasing program size (if the macros are called from many places in the program).

(c) having to fit the macro definition on a single line.

(d) taking only one argument.

ANS: (b)

13.20. Macros are defined

a) in an #include preprocessor directive

b) in a #define preprocessor directive

c) in a #macro preprocessor directive

d) using keyword #MACRO in a C function.

ANS: (b)

13.21. Which statement about macros is *true*?

a) Macros always make programs run faster.

b) Macros cannot have arguments.

c) Macros cause text replacement at execution time.

d) Macro arguments should be enclosed in parentheses in the replacement text.

ANS: (d)

13.22. The parentheses around the two rightmost occurrences of x in the following preprocessor directive:

#define CIRCLE\_AREA(x) ((PI) \* (x) \* (x))

a) are always required.

b) are included to improve program readabiliy.

c) are included to eliminate bugs when an expression is substituted for x.

d) should be curly braces { }.

ANS: (c)

13.23. Macros

a) can help avoid the overhead of a function call.

b) execute before a program is compiled.

c) never add subtle bugs to programs.

d) cannot be used to replace a function call with inline code.

ANS: (a)

13.24. Symbolic constants and macros can be discarded by using the \_\_\_\_\_\_\_\_\_\_ preprocessor directive.

a) #undefine

b) #discard

c) #dscrd

d) #undef

ANS: (d)

13.25. Which statement about symbolic constants and/or macros is *true*?

a) The effects of symbolic constants and macros, once defined, cannot be modified.

b) Preprocessor directive #removedef undefines a symbolic constant or macro.

c) The scope of a symbolic constant or macro is normally from its point of definition to the end of the file, but this can be modified with a preprocessor directive.

d) A name can always be redefined with #define.

ANS: (c)

13.26. A macro commonly defined in the stdio.h header file is

a) #define getchar() getchar(stdin)

b) #define getchar() getc(stdin)

c) #define getc() getchar (stdin)

d) #define getc() getc(stdin)

ANS: (b)

13.27. Which statement is correct?

a) Expressions with side effects (i.e., variable values are modified) should not be passed to a macro because macro arguments may be evaluated more than once.

b) Expressions with side effects (i.e., variable values are modified) should not be passed to a macro because macro arguments are evaluated only once.

c) Expressions with side effects (i.e., variable values are modified) should be passed to a macro because macro arguments may be evaluated more than once.

d) Expressions with side effects (i.e., variable values are modified) should be passed to a macro because macro arguments are evaluated only once.

ANS: (a)

**13.5 Conditional Compilation**

13.28 Large portions of code can be prevented from compiling by

(a) #if 0

code prevented from compiling

#endif

(b) #nodefine

code prevented from compiling

#endif

(c) #if 1

code prevented from compiling

#endif

(d) #ifndef 0

code prevented from compiling

#endif

ANS: (a)

13.29 Which of the following is *not* a valid directive?

(a) #endif

(b) #ifdef

(c) #for

(d) #elif

ANS: (c)

13.30 Conditional compilation *cannot* be used to

(a) perform loops.

(b) ignore large blocks of code.

(c) debug programs.

(d) selectively define symbolic constants.

ANS: (a)

13.31. Which statement is *false*?

a) Cast expressions cannot be evaluated in preprocessor directives.

b) Arithmetic expressions cannot be evaluated in preprocessor directives.

c) sizeof expressions cannot be evaluated in preprocessor directives.

d) Enumeration constants cannot be evaluated in preprocessor directives.

ANS: (b)

13.32. What does the following preprocessor code do?

#if !defined(NULL)

#define NULL 0

#endif

a) The code is incorrect, so it generates an error at preprocessor time.

b) The code ensures that NULL is always 0.

c) The code defines NULL to be 0 only if NULL is undefined.

d) The code ensures that NULL is defined throughout the entire program file.

ANS: (c)

13.33. Every #if construct

a) must end with #end.

b) always affects code throughout the entire body of the program.

c) is used for debugging.

d) must end with #endif.

ANS: (d)

13.34. A shorthand for #if !defined is

a) #ifndef

b) #ifnotdef

c) #ifnotdefined

d) #ifndf

ANS: (a)

13.35. Which statement about “commenting out” code is *false*?

a) During program development, programmers often find it helpful to “comment out” portions of code to prevent it from being compiled.

b) Code may not always be “commented out” correctly simply by enclosing that code in the C comment delimiters /\* and \*/.

c) The following preprocessor code effectively comments out code

#if 0

*code prevented from compiling*

#endif

d) The following preprocessor code effectively comments out code

#if 1

*code prevented from compiling*

#endif

ANS: (d)

13.36. Which statement about conditional compilation is *false*?

a) Conditional compilation is commonly used as a debugging aid.

b) For debugging purposes, printf statements can be enclosed in conditional preprocessor directives so that the statements are compiled only while the debugging process is not completed.

c) A correct example of using a conditionally compiled printf statement only while a program is being debugged is

#ifdef DEBUG

printf(“Variable x = %d\n”, x) ;

#endif

d) A single, conditionally compiled printf statement may be inserted in a program anywhere the C compiler expects a single statement.

ANS: (d)

**13.6 #error and #pragma Preprocessor Directives**

13.37 The \_\_\_\_\_\_\_\_\_\_ directive prints out a message.

(a) #error

(b) #pragma

(c) all of the above

(d) none of the above.

ANS: (a)

13.38 Which line has the most tokens?

(a) I like C.

(b) I like C++ better.

(c) I like C + +.

(d) I prefer Java.

ANS: (c)

13.39. The #error directive

#error *tokens*

a) prints only the tokens specified in the message.

b) prints the word error followed by the tokens specified in the message.

c) prints only an implementation-defined message.

d) prints an implementation-defined message including the tokens specified in the directive.

ANS: (d)

13.40. The preprocessor directive

#error 7 – Out of range error –

contains \_\_\_\_\_\_\_\_\_\_ tokens.

a) 8

b) 7

c) 9

d) 6

ANS: (b)

13.41. The preprocessor directive

#pragma *tokens*

a) prints only the tokens specified in the message.

b) prints the word pragma followed by the tokens specified in the message.

c) causes an implementation-defined action.

d) prints an implementation-defined message including the tokens specified in the directive.

ANS: (c)

13.42. A pragma *not* recognized by the implementation

a) causes the compilation to terminate.

b) prints an error message and causes the compilation to terminate.

c) prints an error message and allows the complation to continue.

d) is ignored.

ANS: (d)

13.43. To customize preprocessing to the unique needs of a particular installation, you would most likely use the \_\_\_\_\_\_\_\_\_\_ directive.

a) #unique

b) #pragmatic

c) #pragma

d) #customize

ANS: (c)

**13.7 # and ## Operators**

13.44 The # operator causes \_\_\_\_\_\_\_\_\_\_ to be converted to a \_\_\_\_\_\_\_\_\_\_.

(a) any preprocessor argument, user-defined type.

(b) a text token, string surrounded by quotes.

(c) a string, concatenated string.

(d) any float or int, string.

ANS: (b)

13.45. The # preprocessor operator causes a replacement text token to be converted to

a) an integer

b) a string

c) a string surrounded by quotes

d) an integer surrounded by quotes

ANS: (c)

13.46. Given the preprocessor directive

#define HAPPY(x) printf (“Happy, ” #x “\n”);

How would you invoke this macro to generate a statement that would print

Happy BIRTHDAY (followed by a newline)

at execution time?

a) Happy(Birthday)

b) Happy(BIRTHDAY)

c) HAPPY(Birthday)

d) HAPPY(BIRTHDAY)

ANS: (d)

13.47. Operator ##

a) concatenates two tokens in a macro definition.

b) is a relational operator.

c) is the conditional-compilation operator.

d) is the symbolic-constant operator.

ANS: (a)

**13.8 Line Numbers**

13.48 The statement #line 250

(a) causes the compiler to ignore everything after line 250.

(b) causes the compiler to renumber the lines from 250 beginning with the next source code line.

(c) causes the compiler to renumber the lines from 250 beginning with this statement line.

(d) causes line number 250 to be replaced by the text following the statement.

ANS: (b)

13.49. The preprocessor directive

#line 200

a) numbers the first 200 lines of a source-code program from 1 to 200.

b) numbers the first 200 lines of a source-code program from 0 to 199.

c) starts line numbering from 200 beginning with the next source-code line

d) numbers the next line 200 and has no effect on other lines.

ANS: (c)

**13.9 Predefined Symbolic Constants**

13.50 Which of the following is *not* a predefined symbolic constant?

(a) \_\_ERROR\_\_

(b) \_\_FILE\_\_

(c) \_\_TIME\_\_

(d) \_\_LINE\_\_

ANS: (a)

13.51. Which statement about predefined symbolic constants is false?

a) The predefined symbolic constants and the defined identifier cannot be used in #define or #undef directives.

b) \_\_DATE\_\_ is replaced by the date the source file is compiled.

c) \_\_LINE\_\_is used to specify the range of line numbers in the source file.

d) The identifiers for each of the predefined symbolic constants begin and end with two underscores.

ANS: (c)

**13.10 Assertions**

13.52 Which symbolic constant must be defined to ignore all assert statements?

(a) DEBUG

(b) NDEBUG

(c) NOASSERT

(d) UNASSERT

ANS: (b)

13.53. Which statement about the following assertion is *true*?

assert(count <= 7);

a) It evaluates its argument at preprocessor time.

b) It evaluates its argument at compile time.

c) It evaluates its argument at execution time.

d) It always calls function abort.

ANS: (c)

13.54. Which statement is *false*?

a) assert is a useful debugging tool for testing if a variable has a correct value

b) If NDEBUG is defined, only the next assertion is ignored.

c) The assert macro is defined in the assert.h header file.

d) One problem with using assert is that if the assertion is false, program execution is terminated; in some situations it is more appropriate to allow the program to continue executing after the program has dealt with the problem.

ANS: (b)

**13.11 Secure C Programming**

13.55 A(n) unsafe macro is one that \_\_\_\_\_\_\_\_.

a) evaluates its argument(s) exactly once

b) evaluates its argument(s) more than once

c) does not evaluate its argument(s)

d) None of the above.

ANS: (b)