Chapter 13: The Preprocessor

* Preprocessing occurs before a program is compiled.
* Possible actions:
    1. inclusion of other files
    2. definition of symbolic constants and macros
    3. definition of conditional compilation
    4. conditional execution of preprocessor directives
* All preprocessor directives begin with "#"

The "#include" Preprocessor Directive

* The "#include" directive causes a copy of a specified file to be included in place of the directive.
* Two forms:
        1. #include <filename> : the preprocessor searches through predesignated directories, in an implementation-dependent manner, e.g., standard library header files.
        2. #include "filename" : the preprocessor searches in the same directory as the file being compiled for the file to be included, e.g. programmer-defined header files.

* The "#include" directive is used with programs consisting of several source files that are to be compiled together.
* A header file containing declarations common to the separate files.

The "#define" Preprocessor Directive: Symbolic Constants

* The "#define" directive creates symbolic constants - constants represented as symbols, and macros - operations defined as symbols.
* The format:
        #define identifier replacement-text
* When this line appears in a file, all subsequent occurences of "identifier" will be replaced by "replacement-text" automatically before the program is compiled.
* E.g.
        #define PI 3.14159

* Symbolic constants enable the programmer to create a name for a constant and use the name throughout the program.
* If the constant needs to be modified throughout the program, it can be modified once in the "#define" directive
* When the program is recompiled, all occurrence of the constant in the program will be modified automatically.

The "#define" Preprocessor Directive: Macros

* A macro is an operation defined in a "#define" preprocessor directive.
* As with symbolic constants, the macro-identifier is replaced in the program with the replacement-text before the program is compiled.
* Macro may be defined with or without arguments.
* A macro without arguments is processed like a symbolic constant.
* In a macro with arguments, the arguments are substituted in the replacement text, then the macro is expanded.
* E.g.

```
        #define CIRCLE_AREA(x) ( PI * (x) * (x) )
        ....
        area = CIRCLE_AREA(4);
```

is expanded to (before compile time),

```
        area = ( 3.14159 * (4) * (4) );
* The parentheses around each "x" in the replacement text force the
proper order of evaluation when the macro argument is an expression.
* E.g.
        area = CIRCLE_AREA(c+2);
is expanded to
        area = ( 3.14159 * (c+2) * (c+2) );

* Macro "CIRCLE_AREA" could be defined as a function.
* E.g.
        double circleArea(double x)
        {
            return 3.14159 * x * x;
        }
* But the overhead is associated with the function.
* A disadvantage of macro is that its argument is evaluated twice.

* If the replacement text for a macro or symbolic constant is longer
than the remainder of the line, a bachslash (\) must be placed at the
end of the line.

* Symbolic constants and macros can be discarded using the "#undef"
preprocessor directive.
* Once undefined, a name can be redefined with "#define".

Conditional Compilation
-----------------------
* Conditional compilation enables the programmer to control the
execution of preprocessor directives, and the compilation of program
code.
* Each of the conditional preprocessor directives evaluates a constant
integer expression.
* E.g.
        #if !defined(NULL)
            #define NULL 0
        #endif
* If NULL is not defined, it is defined by "#define NULL 0"
* If it is defined, the "#define" directive is skipped.
* Every "#if" ends with "#endif".

* Directives "#ifdef name" and "#ifndef name" are shorthand for "#if
define(name)" and "#if !defined(name)" respectively.
* A multiple-part conditional preprocessor may be tested using the
"#elif" and the "#else" directives.

* To comment out a large portions of code with comment,
        #if  0
            code prevented from compiling
        #endif

* To insert some debugging statement, you may
        #ifdef DEBUG
            printf("Variable x = %d\n", x);
        #endif

The "#error" and "#pragma" Preprocessor Directives
--------------------------------------------------
* The "#error" directive
        #error tokens
```

prints an implementation-dependent message including the token
specified in the directive.

* The "#pragma" directive
        #pragma tokens
cause an implementation-defined action.

Predefined Symbolic Constants
----
* There are five predefined symbolic constant.
    __LINE__   The line number of the current source code line.
    __FILE__   The presumed name of the source file.
    __DATE__   The date the source file is compiled
    __TIME__   The time the source file is compiled
    __STDC__   The indicator of ANSI compliant if it is 1.

Assertions
----
* The "assert" macro - defined in the "assert.h" header file - tests
the value of an expression.
* The the value of expression is 0, then "assert" prints an error
message and calls function "abort" (of "stdlib.h") to terminate
program execution.
* E.g.
        assert(x <= 10);
* If "x" is greater than 10, an error message containing the line
number and file name is printed, and the program terminates.
* When assertions are no longer needed, the line
        #define NDEBUG
is inserted in the program file rather than deleting each assertion
manually.