Chapter 13
C Preprocessor

C How to Program, 8/e, GE

Objectives

In this chapter, you'll:

- Use `#include` to develop large programs.
- Use `#define` to create macros with and without arguments.
- Use conditional compilation to specify portions of a program that should not always be compiled (such as code that assists you in debugging).
- Display error messages during conditional compilation.
- Use assertions to test whether the values of expressions are correct.

13.1 Introduction

- The C preprocessor executes *before* a program is compiled.
- Some actions it performs are the inclusion of other files in the file being compiled, definition of symbolic constants and macros, conditional compilation of program code and conditional execution of preprocessor directives.
- Preprocessor directives begin with # and only whitespace characters and comments may appear before a preprocessor directive on a line.

13.2 `#include` Preprocessor Directive

- The `#include` preprocessor directive has been used throughout this text.
- The `#include` directive causes a *copy* of a specified file to be included in place of the directive.
- The two forms of the `#include` directive are:
  - `#include <filename>`
    `#include "filename"`
- The difference between these is the location the preprocessor begins searches for the file to be included.
- If the file name is enclosed in quotes, the preprocessor starts searches in the same directory as the file being compiled for the file to be included (and may search other locations, too).

13.2 `#include` Preprocessor Directive (Cont.)

- This method is normally used to include programmer-defined headers.
- If the file name is enclosed in angle brackets (< and >)—used for standard library headers—the search is performed in an implementation-dependent manner, normally through predesignated compiler and system directories.

13.2 `#include` Preprocessor Directive (Cont.)

- The `#include` directive is used to include standard library headers such as `stdio.h` and `stdlib.h` (see Fig. 5.10) and with programs consisting of *multiple source files* that are to be compiled together.
- A header containing declarations common to the separate program files is often created and included in the file.
- Examples of such declarations are *structure and union declarations, enumerations* and *function prototypes*.

13.3 `#define` Preprocessor Directive: Symbolic Constants

- The `#define` directive creates *symbolic constants*—constants represented as symbols—and macros—operations defined as symbols.
- The `#define` directive format is
  - **`#define`** ***identifier*** ***replacement-text***
- When this line appears in a file, all subsequent occurrences of *identifier* that do *not* appear in string literals will be replaced by replacement-text automatically *before* the program is compiled.

13.3 `#define` Preprocessor Directive: Symbolic Constants (Cont.)

- For example,
  - `#define PI 3.14159`

  replaces all subsequent occurrences of the symbolic constant `PI` with the numeric constant `3.14159`.
- *Symbolic constants* enable you 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 occurrences of the constant in the program will be modified accordingly.

13.3 #define Preprocessor Directive: Symbolic Constants (Cont.)

- Everything to the right of the symbolic constant name replaces the symbolic constant.] For example, `#define PI = 3.14159` causes the preprocessor to replace every occurrence of the identifier `PI` with `= 3.14159`.
- This is the cause of many subtle logic and syntax errors.
- For this reason, you may prefer to use `const` variable declarations, such as
  - `const double PI = 3.14159;`

  in preference to the preceding `#define`. Redefining a symbolic constant with a new value is also an error.

Error-Prevention Tip 13.1

Everything to the right of the symbolic constant name replaces the symbolic constant. For example, `#define PI = 3.14159` *causes the preprocessor to replace every occurrence of the identifier* `PI` *with* `= 3.14159`*. This is the cause of many subtle logic and syntax errors. For this reason, you may prefer to use* `const` *variable declarations, such as* `const double PI = 3.14159;` *in preference to the preceding* `#define`*.*

Common Programming Error 13.1

Attempting to redefine a symbolic constant with a new value is an error.

Software Engineering Observation 13.1

Using symbolic constants makes programs easier to modify. Rather than search for every occurrence of a value in your code, you modify a symbolic contant once in its `#define` *directive. When the program is recompiled, all occurrences of that constant in the program are modified accordingly.*

Good Programming Practice 13.1

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

Good Programming Practice 13.2

By convention, symbolic constants are defined using only uppercase letters and underscores.

13.4 `#define` Preprocessor Directive: Macros

- A macro is an identifier 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.
- Macros 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—i.e., the replacement-text replaces the identifier and argument list in the program.

13.4 `#define` Preprocessor Directive: Macros (Cont.)

- [*Note:* A symbolic constant is a type of macro.]
- Consider the following *macro definition* with one *argument* for the area of a circle:

```
#define CIRCLE_AREA(x) ((PI) * (x) * (x))
```

- Wherever `CIRCLE_AREA(y)` appears in the file, the value of `y` is substituted for `x` in the replacement-text, the symbolic constant `PI` is replaced by its value (defined previously) and the macro is expanded in the program.

13.4 #define Preprocessor Directive: Macros (Cont.)

- For example, the statement
  - `area = CIRCLE_AREA(4);`

  is expanded to
  - `area = ((3.14159) * (4) * (4));`

  then, at compile time, the value of the expression is evaluated and assigned to variable `area`.
- The *parentheses* around each `x` in the replacement text *force the proper order of evaluation when the macro argument is an expression*.

13.4 #define Preprocessor Directive: Macros (Cont.)

- For example, the statement
  - `area = CIRCLE_AREA(c + 2);`

  is expanded to
  - `area = ((3.14159) * (c + 2) * (c + 2));`

  which evaluates *correctly* because the parentheses force the proper order of evaluation.

13.4 `#define` Preprocessor Directive: Macros (Cont.)

- If the parentheses in the macro definition are omitted, the macro expansion is
  - `area = 3.14159 * c + 2 * c + 2;`

  which evaluates *incorrectly* as
  - `area = (3.14159 * c) + (2 * c) + 2;`

  because of the rules of operator precedence.

Error-Prevention Tip 13.2

Enclose macro arguments in parentheses in the replacement-text to prevent logic errors.

13.4 #define Preprocessor Directive: Macros (Cont.)

- Macro `CIRCLE_AREA` could be defined more safely as a function.
- Function `circleArea`
  - ```
    double circleArea(double x)
    {
       return 3.14159 * x * x;
    }
    ```

  performs the same calculation as macro `CIRCLE_AREA`, but the function's argument is evaluated only once when the function is called.
    ```

Performance Tip 13.1

In the past, macros were often used to replace function calls with inline code *to eliminate the function-call overhead. Today's optimizing compilers often inline function calls for you, so many programmers no longer use macros for this purpose. You can also use the C standard's* `inline` *keyword (see Appendix E).*

13.4 #define Preprocessor Directive: Macros (Cont.)

- The following is a macro definition with two arguments for the area of a rectangle:
  - `#define RECTANGLE_AREA(x, y)  ((x) * (y))`
- Wherever `RECTANGLE_AREA(x, y)` appears in the program, the values of `x` and `y` are substituted in the macro replacement text and the macro is expanded in place of the macro name.
- For example, the statement
  - `rectArea = RECTANGLE_AREA(a + 4, b + 7);`
- is expanded to
  - `rectArea = ((a + 4) * (b + 7));`

13.4 `#define` Preprocessor Directive: Macros (Cont.)

- The value of the expression is evaluated at runtime and assigned to variable `rectArea`.
- The replacement text for a macro or symbolic constant is normally any text on the line after the identifier in the `#define` directive.
- If the replacement text for a macro or symbolic constant is longer than the remainder of the line, a backslash (\) must be placed at the end of the line, indicating that the replacement text continues on the next line.

13.4 `#define` Preprocessor Directive: Macros (Cont.)

- Symbolic constants and macros can be *discarded* by using the `#undef` preprocessor directive.
- Directive `#undef` "*undefines*" a symbolic constant or macro name.
- The scope of a symbolic constant or macro is from its definition until it's undefined with `#undef`, or until the end of the file.
- Once undefined, a name can be redefined with `#define`.
- Functions in the standard library sometimes are defined as macros based on other library functions.

13.4 `#define` Preprocessor Directive: Macros (Cont.)

- A macro commonly defined in the `<stdio.h>` header is
  - `#define getchar() getc(stdin)`
- The macro definition of `getchar` uses function `getc` to get one character from the standard input stream.
- Function `putchar` of the `<stdio.h>` header and the character handling functions of the `<ctype.h>` header often are implemented as macros as well.
- 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.

13.5 Conditional Compilation

- Conditional compilation enables you to control the execution of preprocessor directives and the compilation of program code.
- Each conditional preprocessor directive evaluates a constant integer expression.
- Cast expressions, `sizeof` expressions and enumeration constants cannot be evaluated in preprocessor directives.
- The conditional preprocessor construct is much like the `if` selection statement.

13.5 Conditional Compilation (Cont.)

- Consider the following preprocessor code:

```
#if !defined(MY_CONSTANT)
   #define MY_CONSTANT 0
#endif
```

  Which determines whether `MY_CONSTANT` is *defined*—that is, whether `MY_CONSTANT` has already appeared in an earlier `#define` directive.
- The expression `defined(MY_CONSTANT)` evaluates to `1` if `MY_CONSTANT` is defined and `0` otherwise.
- If the result is `0`, `!defined(MY_CONSTANT)` evaluates to `1` and `MY_CONSTANT` is defined.
- Otherwise, the `#define` directive is skipped.

13.5 Conditional Compilation (Cont.)

- Every `#if` construct ends with `#endif`.
- Directives `#ifdef` and `#ifndef` are shorthand for `#if defined(`*name*`)` and `#if !defined(`*name*`)`.
- A multiple-part conditional preprocessor construct may be tested by using the `#elif` (the equivalent of `else if` in an `if` statement) and the `#else` (the equivalent of `else` in an `if` statement) directives.
- These directives are frequently used to *prevent header files from being included multiple times in the same source file*.
- We use this technique extensively in the C++ part of this book.

13.5 Conditional Compilation (Cont.)

- During program development, it's often helpful to "comment out" portions of code to prevent them from being compiled.
- If the code contains multiline comments, /* and */ cannot be used to accomplish this task, because such comments cannot be nested.
- Instead, you can use the following preprocessor construct:
  - ```
    #if 0
        code prevented from compiling
    #endif
    ```
- To enable the code to be compiled, replace the `0` in the preceding construct with `1`.
    ```

13.5 Conditional Compilation (Cont.)

- Conditional compilation is commonly used as a *debugging* aid.
- Many C implementations include debuggers, which provide much more powerful features than conditional compilation.
- If a debugger is not available, `printf` statements are often used to print variable values and to confirm the flow of control.
- These `printf` statements can be enclosed in conditional preprocessor directives so the statements are compiled only while the debugging process is not completed.

13.5 Conditional Compilation (Cont.)

- For example,
  - ```
    #ifdef DEBUG
       printf("Variable x = %d\n", x);
    #endif
    ```

  causes a `printf` statement to be compiled in the program if the symbolic constant DEBUG has been defined (`#define` DEBUG) before directive `#ifdef` DEBUG.
    ```

13.5 Conditional Compilation (Cont.)

- When debugging is completed, the `#define` directive is removed from the source file (or commented out) and the `printf` statements inserted for debugging purposes are ignored during compilation.
- In larger programs, it may be desirable to define several different symbolic constants that control the conditional compilation in separate sections of the source file.
- Many compilers allow you to define and undefine symbolic constants with a compiler flag so that you do not need to change the code.

Error-Prevention Tip 13.3

When inserting conditionally compiled `printf` *statements in locations where C expects a single statement (e.g., the body of a control statement), ensure that the conditionally compiled statements are enclosed in blocks.*

13.6 #error and #pragma Preprocessor Directives

- The #error directive
  - **#error** ***tokens***

  prints an implementation-dependent message including the *tokens specified in the directive.*
- The tokens are sequences of characters separated by spaces.
- For example,
  - **#error 1 - Out of range error**

  contains 6 tokens.
- When a #error directive is processed on some systems, the tokens in the directive are displayed as an error message, preprocessing stops and the program does not compile.

13.6 `#error` and `#pragma` Preprocessor Directives (Cont.)

- The `#pragma` directive
  - **`#pragma`** ***tokens***

  causes an implementation-defined action.
- A pragma not recognized by the implementation is ignored.

13.7 # and ## Operators

- The # and ## preprocessor operators are available in Standard C.
- The # operator causes a replacement text token to be converted to a string surrounded by quotes.
- Consider the following macro definition:
  - `#define HELLO(x) printf("Hello, " #x "\n");`
- When `HELLO(John)` appears in a program file, it's expanded to
  - `printf("Hello, " "John" "\n");`
- The string `"John"` replaces `#x` in the replacement text.

13.7 # and ## Operators (Cont.)

- Strings separated by white space are concatenated during preprocessing, so the preceding statement is equivalent to
  - `printf("Hello, John\n");`
- The # operator must be used in a macro with arguments because the operand of # refers to an argument of the macro.
- The ## operator concatenates two tokens.

13.7 # and ## Operators (Cont.)

- Consider the following macro definition:
  - `#define TOKENCONCAT(x, y) x ## y`
- When `TOKENCONCAT` appears in the program, its arguments are concatenated and used to replace the macro.
- For example, `TOKENCONCAT(O, K)` is replaced by `OK` in the program.
- The ## operator must have two operands.

13.8 Line Numbers

- The `#line` preprocessor directive causes the subsequent source code lines to be renumbered starting with the specified constant integer value.
- The directive
  - `#line 100`

  starts line numbering from `100` beginning with the next source code line.
- A filename can be included in the `#line` directive.

13.8 Line Numbers (Cont.)

- The directive
  - `#line 100 "file1.c"`

  indicates that lines are numbered from `100` beginning with the next source code line and that the name of the file for the purpose of any compiler messages is `"file1.c"`.
- The directive normally is used to help make the messages produced by syntax errors and compiler warnings more meaningful.
- The line numbers do not appear in the source file.

13.9 Predefined Symbolic Constants

- Standard C provides predefined symbolic constants, several of which are shown in Fig. 13.1——the rest are in Section 6.10.8 of the C standard document.
- The identifiers for each of the predefined symbolic constants begin and end with *two* underscores.
- These identifiers and the `defined` identifier (used in Section 13.5) cannot be used in `#define` or `#undef` directives.

Symbolic constant	Explanation
`__LINE__`	The line number of the current source-code line (an integer constant).
`__FILE__`	The name of the source file (a string).
`__DATE__`	The date the source file was compiled (a string of the form `"Mmm dd yyyy"` such as `"Jan 19 2002"`).
`__TIME__`	The time the source file was compiled (a string literal of the form `"hh:mm:ss"`).
`__STDC__`	The value `1` if the compiler supports Standard C; `0` otherwise. Requires the compiler flag `/Za` in Visual C++.

Fig. 13.1 | Some predefined symbolic constants.

13.10 Assertions

- The `assert` macro—defined in the `<assert.h>` header—tests the value of an expression at execution time.
- If the value of the expression is false (`0`), `assert` prints an error message and calls function `abort` (of the general utilities library—`<stdlib.h>`) to terminate program execution.
- This is a useful *debugging* tool for testing whether a variable has a correct value.
- For example, suppose variable `x` should never be larger than `10` in a program.

13.10 Assertions (Cont.)

- An assertion may be used to test the value of `x` and print an error message if the value of `x` is incorrect.
- The statement would be
  - `assert(x <= 10);`
- If `x` is greater than `10` when the preceding statement is encountered in a program, an error message containing the line number and filename is printed and the program *terminates*.

13.10 Assertions (Cont.)

- You may then concentrate on this area of the code to find the error.
- If the symbolic constant NDEBUG is defined, subsequent assertions will be *ignored*.
- Thus, when assertions are no longer needed, the line
  - `#define NDEBUG`

  is inserted in the program file rather than each assertion being deleted manually.

13.10 Assertions (Cont.)

- [*Note*: The new C standard includes a capability called `_Static_assert`, which is essentially a compile-time version of assert that produces a compilation error if the assertion fails. We discuss `_Static_assert` in Appendix F.]

Software Engineering Observation 13.2

Assertions are not meant as a substitute for error handling during normal runtime conditions. Their use should be limited to finding logic errors during program development.

13.10 Secure C Programming

- The `CIRCLE_AREA` macro defined in Section 13.4

```
#define CIRCLE_AREA(x) ((PI) * (x) *
(x))
```

  is considered to be an unsafe macro because it evaluates its argument `x` more than once.
- This can cause subtle errors.
- If the macro argument contains side effects—such as incrementing a variable or calling a function that modifies a variable's value—those side effects would be performed multiple times.

13.10 Secure C Programming (Cont.)

- For example, if we call `CIRCLE_AREA` as follows:

```
result = CIRCLE_AREA(++radius);
```

  the call to the macro `CIRCLE_AREA` is expanded to:

```
result = ((3.14159) * (++radius) * (++radius));
```

  which increments radius twice in the statement.

- In addition, the result of the preceding statement is undefined because C allows a variable to be modified *only once* in a statement.

13.10 Secure C Programming (Cont.)

- In a function call, the argument is evaluated *only once before* it's passed to the function.
- So, functions are always preferred to unsafe macros.