Agenda

C Preprocessor
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.

```
1  // A first program in C.
2  #include <stdio.h>
3
4  // function main begins program execution
5  int main( void )
6  {
7     printf( “Welcome to C!\n” );
8  } // end function main
```

C Preprocessor
#include Preprocessor Directive

- 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 at which the preprocessor begins searches for the file to be included. If the filename is enclosed in angle brackets (< and >)--used for standard library headers--the search is performed in an implementation-depended manner, normally through predesignated compiler and system directories.

- 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. This method is normally used to include programmer-defined headers. If the compiler cannot find the file in the current directory, then it will search through the predesignated compiler and system directories.

- The #include directive is used to include standard library headers such as stdio.h and stdlib.h 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.

#define Preprocessor Directive: Symbolic Constans

- 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 occurences of identifiers that do not appear in string literals will be replaced by replacement text automatically before the program is compiled. For example,

#define PI 3.14159

replaces all subsequent occurences of the symbolic constant PI with the numeric constant 3.14159. Every thing to the right of the symbolic constant name replaces the symbolic constant. For example #define PI = 3.14159 causes the preprocessor to replace every occurence 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 preceeding #define. Redefining a symbolic constant with a new value is also an error.

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

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

#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. A symbolic constant is a type of macro.

- Condier 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 test, the symbol, the symbolic constant PI is replaced by its value and the macro is expanded in the program. 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. For example, the statement area = CIRCLE_AREA(c + 2); is expanded to area = ((3.14159) * (c + 2) * (c + 2)) which evaluates correctly because the parenthesis force the proper evaluation. 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.

#define Preprocessor Directive: Macros

- The following 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 placement text and the macro is expanded in the place of the macro name. For example, the statement rectArea = RECTANGLE_AREA(a + 4, b + 7) is expanded to rectArea = ((a + 4) * (b + 7)).

- 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.

- 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.

- Conditional compilation enables you to control the execution of preprocessor directives and the compilation of program code. Each conditional preprocessor directive evaluates a constatn integer expression.

- The conditional preprocessor construct is much like the if selection statement. 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.

- 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 and #else directives. These directives are frequently used to prevent header files from being included multiple times in the same source file.

During program development, it's often helpful to comment out portions of code to prevent then 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 prevent from compiling
#endif
```

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. 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 before directive #ifdef DEBUG. 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.

Conditional 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.

C Preprocessor
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) puts(“Hello, ”, #x);

- When HELLO(Alice) appears in a program file, it’s expanded to puts(“Hello, ” “Alice”). The string “Alice” replaces #x in the replacement text. Strings separated by white space are concatenated during preprocessing, so the preceding statement is equivalent to puts(“Hello, Alice”).

- The ## operator concatenates two tokens. 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.

Line Numbers

- The #line preprocessor directive causes the subsequent source code lines to be renumbered starting with the specified 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. 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.

Predefined Symbolic Constants

Standard C provides predefined symbolic constants, several of which are shown in Figure 1.1. The identifiers for each of the predefined symbolic constants begin and end with two underscore. These identifiers and the defined identifiers canned be used in #define or #undef directives

__LINE__	The line number of the current source code line (an integer contant).
__FILE__	The presumed 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 10 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.

Fig. 1.1 | Some predefined symbolic contants.

Assertions

- The assert macro--defined in the <assert.h> header--tests the value of an expression at execution time. If the value is false (0), assert prints an error message and calls function abort (of the general utilities library--<stdlib.h>) to terminate the 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. 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. 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.

- The CIRCLE_AREA macro #define CIRCLE_AREA(x) ((PI) * (x) * (x)) is considered to be 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.

- 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. 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.