Programming for Engineers

C Preprocessor

ICEN 200 – Spring 2018

Prof. Dola Saha

C Preprocessor

- 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.

#include Preprocessor Directive

- A copy of a specified file will be included in place of the directive
- Two forms:
  - `#include <filename>`
    - Standard library headers
    - Searches pre-designated compiler and system directories
  - `#include “filename”`
    - Programmer defined headers
    - Searches in the same directory as the file

Symbolic Constants

- The `#define` directive format is
  - `#define` *identifier* *replacement-text*
- All occurrences of identifier will be **replaced** by replacement-text automatically before the program is compiled.
- For example, `#define PI 3.14159`

  replaces all subsequent occurrences of the symbolic constant `PI` with the numeric constant `3.14159`.
- If the constant value needs to be modified throughout the program, it can be **modified once** in the preprocessor directive.
- BEST Practice: DO NOT use numbers inside the code, always use preprocessor directive to create constants.

Example code

```
#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - FREEZING_PT) *
   SCALE_FACTOR;
  printf("Celsius equivalent is: %.1f\n",
   celsius);

  return 0;
}
```

After Preprocessing

Blank line
Blank line
Lines brought in from stdio.h
Blank line
Blank line
Blank line
Blank line

```
int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - 32.0f) * (5.0f / 9.0f);

  printf("Celsius equivalent is: %.1f\n", celsius);

  return 0;
}
```

Macro

- The macro-identifier is replaced in the program with the *replacement-text* before the program is compiled.
- Macros may be defined
  - without arguments – processed like a symbolic constant
  - 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.
    - `#define CIRCLE_AREA(x) ((PI) * (x) * (x))`

Example Macro

- `#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.
- Example Statement: `area = CIRCLE_AREA(4);`
- Expanded: `area = ((3.14159) * (4) * (4));`
- Example Statement: `area = CIRCLE_AREA(c + 2);`
- Expanded: `area = ((3.14159) * (c + 2) * (c + 2));`

Corresponding 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.
    ```

Example MACRO with two arguments

- `#define RECTANGLE_AREA(x, y)  ((x) * (y))`
- Statement
  - `rectArea = RECTANGLE_AREA(a + 4, b + 7);`
- Expanded to
  - `rectArea = ((a + 4) * (b + 7));`

Macro Error

```
#define PRODUCT(x, y) (x * x)
 int result = PRODUCT(4,5)
// returns 20
int result = PRODUCT(2+2,3+2)
// returns 10
```

MACRO Error

- 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.

MACRO Error

- An example that illustrates the need to put parentheses around a macro's replacement list:

```
#define TWO_PI 2*3.14159
/* needs parentheses around
replacement list */
```

- During preprocessing, the statement

```
conversion_factor = 360/TWO_PI;
```

becomes

```
conversion_factor = 360/2*3.14159;
```

The division will be performed before the multiplication.

MACRO Error

- Each occurrence of a parameter in a macro's replacement list needs parentheses as well:

```
#define SCALE(x) (x*10)
  /* needs parentheses around x */
```

- During preprocessing, the statement

```
j = SCALE(i+1);
```

becomes

```
j = (i+1*10);
```

This statement is equivalent to

```
j = i+10;
```

Multiline MACRO

- 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.

```
#define NUMBERS   1, \
                  2, \
                  3
int x[] = { NUMBERS };
```

Expanded: `int x[] = { 1, 2, 3 };`

#undef

- If a macro ceases to be useful, it may be *undefined* with the '#undef' directive.
- '#undef' takes a single argument, the name of the macro to undefine.
- Once a macro has been undefined, that identifier may be *redefined* as a macro by a subsequent '#define' directive.

```
#define SIZE 10
#undef SIZE
int x = SUM; // Error
#define SIZE 100
```

Preprocessor

- Cast expressions, `sizeof` expressions and enumeration constants cannot be evaluated in preprocessor directives.
- Whitespace may appear, following are same MACRO
  - `#define FOUR (2 + 2)`
  - `#define FOUR (2 + 2)`
  - `#define FOUR (2 /* two */ + 2)`

Conditional Compilation

- 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
```

determines whether `MY_CONSTANT` is defined—that is, whether `MY_CONSTANT` has already appeared in an earlier `#define` directive.

Conditional Compilation

- 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.

Comment Code

- 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.
    ```

Debug

```
#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.

Predefined Symbolic Constants

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++.

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.