Chapter 13 - The Preprocessor

Outline

13.1 Introduction

- Preprocessing
  - Occurs before a program is compiled
  - Inclusion of other files
  - Definition of *symbolic constants* and *macros*
  - *Conditional compilation* of program code
  - *Conditional execution of preprocessor directives*
- Format of preprocessor directives
  - Lines begin with **#**
  - Only whitespace characters before directives on a line

13.2 The #include Preprocessor Directive

- **#include**
  - Copy of a specified file included in place of the directive
    `#include <filename>`
    - Searches standard library for file
    - Use for standard library files
    `#include "filename"`
    - Searches current directory, then standard library
    - Use for user-defined files
- Used for
  - Loading header files (`#include <iostream>`)
  - Programs with multiple source files to be compiled together
  - Header file - has common declarations and definitions (classes, structures, function prototypes)
    - **#include** statement in each file

13.3 The #define Preprocessor Directive: Symbolic Constants

- **#define**
  - Preprocessor directive used to create symbolic constants and macros.
- Symbolic constants
  - When program compiled, all occurrences of symbolic constant replaced with replacement text
- Format
  `#define identifier replacement-text`
  - Example: `#define PI 3.14159`
  - everything to right of identifier replaces text
    `#define PI = 3.14159`
    - replaces "**PI**" with "**= 3.14159**", probably results in an error
  - Cannot redefine symbolic constants with more **#define** statements

13.4 The #define Preprocessor Directive: Macros

- Macro
  - Operation defined in **#define**
  - Macro without arguments: treated like a symbolic constant
  - Macro with arguments: arguments substituted for replacement text, macro expanded
  - Performs a text substitution - no data type checking

Example:

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

area = CIRCLE_AREA( 4 );
```

is expanded to

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```


13.4 The #define Preprocessor Directive: Macros

- Use parenthesis
  - Without them:

```
#define CIRCLE_AREA( x ) ((PI) * ( x ) * ( x ))
#define CIRCLE_AREA( x )  PI *  x  *  x
area = CIRCLE_AREA( c + 2 );
```

becomes

```
area = 3.14159 * c + 2 * c + 2;
```

  - Evaluates incorrectly
- **Macor's advantage is that avoiding function overhead**
  - **Macro inserts code directly.**
- **Macro's disadvantage is that its argument may be evaluated more than once.**

```
double circleArea ( double x )
{    return 3.1415926 * x * x ;
}
```


13.4 The #define Preprocessor Directive: Macros

- Multiple arguments

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

becomes

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


13.4 The #define Preprocessor Directive: Macros

- **#undef**
  - Undefines a symbolic constant or macro, which can later be redefined
- #define getchar() getc (stdin)

13.5 Conditional Compilation

- Conditional compilation
  - Control preprocessor directives and compilation
  - Cast expressions, **sizeof,** enumeration constants cannot be evaluated
- Structure similar to **if**

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

  - Determines if symbolic constant **NULL** defined
    - If **NULL** is defined, **defined(NULL)** evaluates to **1**
    - If **NULL** not defined, defines **NULL** as **0**
  - Every **#if** ends with **#endif**
  - **#ifdef** short for **#if defined(***name***)**
  - **#ifndef** short for **#if !defined(***name***)**

13.5 Conditional Compilation (II)

- Other statements
  - **#elif -** equivalent of **else if** in an **if** structure
  - **#else -** equivalent of **else** in an **if** structure
- "Comment out" code
  - Cannot use **/* ... */**
  - Use /* to prevent it from being compiled */

```
#if 0
   code commented out
#endif
```

  to enable code, change **0** to **1**

13.5 Conditional Compilation (III)

- Debugging

```
#define DEBUG 1
#ifdef DEBUG
   printf (" Variable x = %d \n", x) ;
#endif
```

  - Defining **DEBUG** enables code
  - After code corrected, remove **#define** statement
  - Debugging statements are now ignored

13.7 The # and ## Operators

- **#**
  - Replacement text token converted to string with quotes

```
#define HELLO( x ) printf (" Hello, " #x "\n");

HELLO(John)
```

  becomes

```
printf (" Hello, " "John" "\n");
printf (" Hello, John\n");
```

  Notice #

    - Strings separated by whitespace are concatenated
- **##**
  - Concatenates two tokens

```
#define TOKENCONCAT( x, y )  x ## y

TOKENCONCAT( O, K )
```

  becomes

```
OK
```

13.10 Assertions

- **`assert`** macro
  - Header **`<assert.h>`**
  - Tests value of an expression
  - If **`0`** (false) prints error message and calls **`abort`**

```
assert( x <= 10 );
```

- If **`NDEBUG`** defined...
  - All subsequent **`assert`** statements ignored
  - **`#define NDEBUG`**