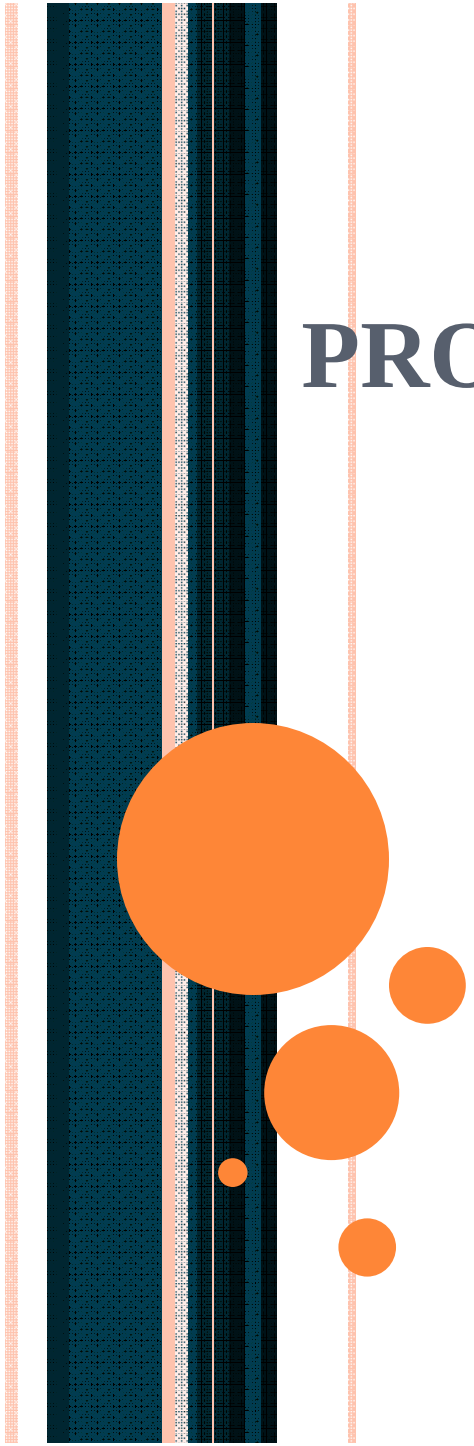


OBJECT ORIENTED PROGRAMMING USING C++

Chapter 17 - The Preprocessor

Outline

17.1 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
  - they are not C++ statements - no semicolon (;)

17.2 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

17.3 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

17.4 The #define Preprocessor Directive: Macros

- Macro - operation defined in **#define**
  - intended for C programs
  - 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 );
```

becomes

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


17.4 The #define Preprocessor Directive: Macros (II)

- use parenthesis:
  - without them,

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

    becomes

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

    which evaluates incorrectly

- multiple arguments:

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

    becomes

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

- **#undef**
  - undefines a symbolic constant or macro, which can later be redefined

17.5 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*`)`**

17.5 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

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

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

17.5 Conditional Compilation (III)

- Debugging

```
#define DEBUG 1

#ifdef DEBUG
   cerr << "Variable x = " << x << endl;
#endif
```

Defining **DEBUG** enables code. After code corrected, remove **#define** statement and debugging statements are ignored.

17.6 The `#error` and `#pragma` Preprocessor Directives

- **`#error`** *tokens*
  - tokens - sequences of characters separated by spaces
    - **`"I like C++"`** has 3 tokens
  - prints message and tokens (depends on implementation)
  - for example: when **`#error`** encountered, tokens displayed and preprocessing stops (program does not compile)

- **`#pragma`** *tokens*
  - implementation defined action (consult compiler documentation)
  - pragmas not recognized by compiler are ignored

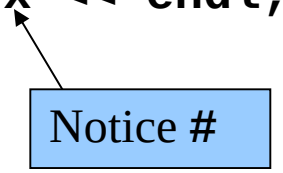
17.7 The # and ## Operators

- # - replacement text token converted to string with quotes

```
#define HELLO( x ) cout << "Hello, " #x << endl;
```

HELLO(John) becomes

```
cout << "Hello, " "John" << endl;
```



  - strings separated by whitespace are concatenated when using **cout**

- ## - concatenates two tokens

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

TOKENCONCAT(O, K) becomes

OK

17.8 Line Numbers

- **`#line`**
  - renumbers subsequent code lines, starting with integer value
  - file name can be included

- **`#line 100 "myFile.c"`**
  - lines are numbered from **`100`** beginning with next source code file
  - for purposes of errors, file name is **`"myFile.c"`**
  - makes errors more meaningful
  - line numbers do not appear in source file

17.9 Predefined Symbolic Constants

- Five predefined symbolic constants
  - cannot be used in **`#define`** or **`#undef`**

Symbolic constant	Description
`__LINE__`	The line number of the current source code line (an integer constant).
`__FILE__`	The presumed name of the source file (a string).
`__DATE__`	The date the source file is compiled (a string of the form **`"Mmm dd yyyy"`** such as **`"Jan 19 2001"`**).
`__TIME__`	The time the source file is compiled (a string literal of the form **`"hh:mm:ss"`**).

17.10 Assertions

- **`assert`** macro
  - header **`<cassert>`**
  - 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`**