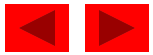
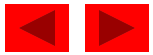
Chapter 13 - The Preprocessor

Outline



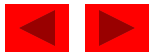
Objectives

- In this chapter, you will learn:
  - To be able to use `#include` for developing large programs.
  - To be able to use #define to create macros and macros with arguments.
  - To understand conditional compilation.
  - To be able to display error messages during conditional compilation.
  - To be able to use assertions to test if the values of expressions are correct.



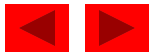
13.1 Introduction

- 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:
    - 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



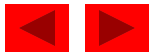
13.3 The #define Preprocessor Directive: Symbolic Constants

- `#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`"
  - Cannot redefine symbolic constants once they have been created



13.4 The #define Preprocessor Directive: Macros

- Macro
  - Operation defined in `#define`
  - A macro without arguments is treated like a symbolic constant
  - A macro with arguments has its arguments substituted for replacement text, when the macro is expanded
  - Performs a text substitution – no data type checking
  - The macro

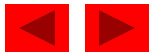
```
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
```

would cause

```
area = CIRCLE_AREA( 4 );
```

to become

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

13.4 The #define Preprocessor Directive: Macros

- Use parenthesis
  - Without them the macro

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

would cause

```
area = CIRCLE_AREA( c + 2 );
```

to become

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

- Multiple arguments

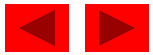
```
#define RECTANGLE_AREA( x, y )  ( ( x ) * ( y ) )
```

would cause

```
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

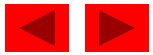
to become

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

13.4 The #define Preprocessor Directive: Macros

- #undef
  - Undefines a symbolic constant or macro
  - If a symbolic constant or macro has been undefined it can later be redefined

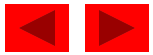


13.5 Conditional Compilation

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

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

    - Determines if symbolic constant `NULL` has been defined
      - If `NULL` is defined, `defined( NULL )` evaluates to `1`
      - If `NULL` is not defined, this function defines `NULL` to be `0`
  - Every `#if` must end with `#endif`
  - `#ifdef` short for `#if defined( name )`
  - `#ifndef` short for `#if !defined( name )`

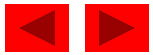


13.5 Conditional Compilation

- Other statements
  - `#elif` – equivalent of `else if` in an `if` statement
  - `#else` – equivalent of `else` in an `if` statement
- "Comment out" code
  - Cannot use `/* ... */`
  - Use

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

  - To enable code, change `0` to `1`

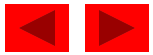


13.5 Conditional Compilation

- Debugging

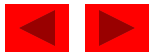
```
#define DEBUG  1
#ifdef DEBUG
     cerr << "Variable x = " << x << endl;
  #endif
```

  - Defining `DEBUG` to `1` enables code
  - After code corrected, remove `#define` statement
  - Debugging statements are now ignored



13.6 The #error and #pragma Preprocessor Directives

- #error tokens
  - Tokens are sequences of characters separated by spaces
    - "I like C++" has 3 tokens
  - Displays a message including the specified tokens as an error message
  - Stops preprocessing and prevents program compilation
- #pragma tokens
  - Implementation defined action (consult compiler documentation)
  - Pragmas not recognized by compiler are ignored



13.7 The # and ## Operators

- #
  - Causes a replacement text token to be converted to a string surrounded by quotes
  - The statement

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

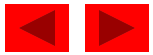
would cause

```
HELLO( John )
```

to become

```
printf( “Hello, ” “John” “\n” );
```

  - Strings separated by whitespace are concatenated when using `printf`



13.7 The # and ## Operators

- ##
  - Concatenates two tokens
  - The statement

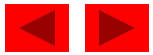
```
#define TOKENCONCAT( x, y )  x ## y
```

would cause

```
TOKENCONCAT( O, K )
```

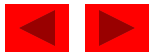
to become

```
OK
```

13.8 Line Numbers

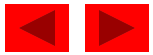
- `#line`
  - Renumbers subsequent code lines, starting with integer value
  - File name can be included
  - `#line 100 "myFile.c"`
    - Lines are numbered from 100 beginning with next source code file
    - Compiler messages will think that the error occurred in `"myfile.c"`
    - Makes errors more meaningful
    - Line numbers do not appear in source file



13.9 Predefined Symbolic Constants

- Four 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"**).



13.10 Assertions

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

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

  - If `NDEBUG` is defined
    - All subsequent assert statements ignored

```
#define NDEBUG
```