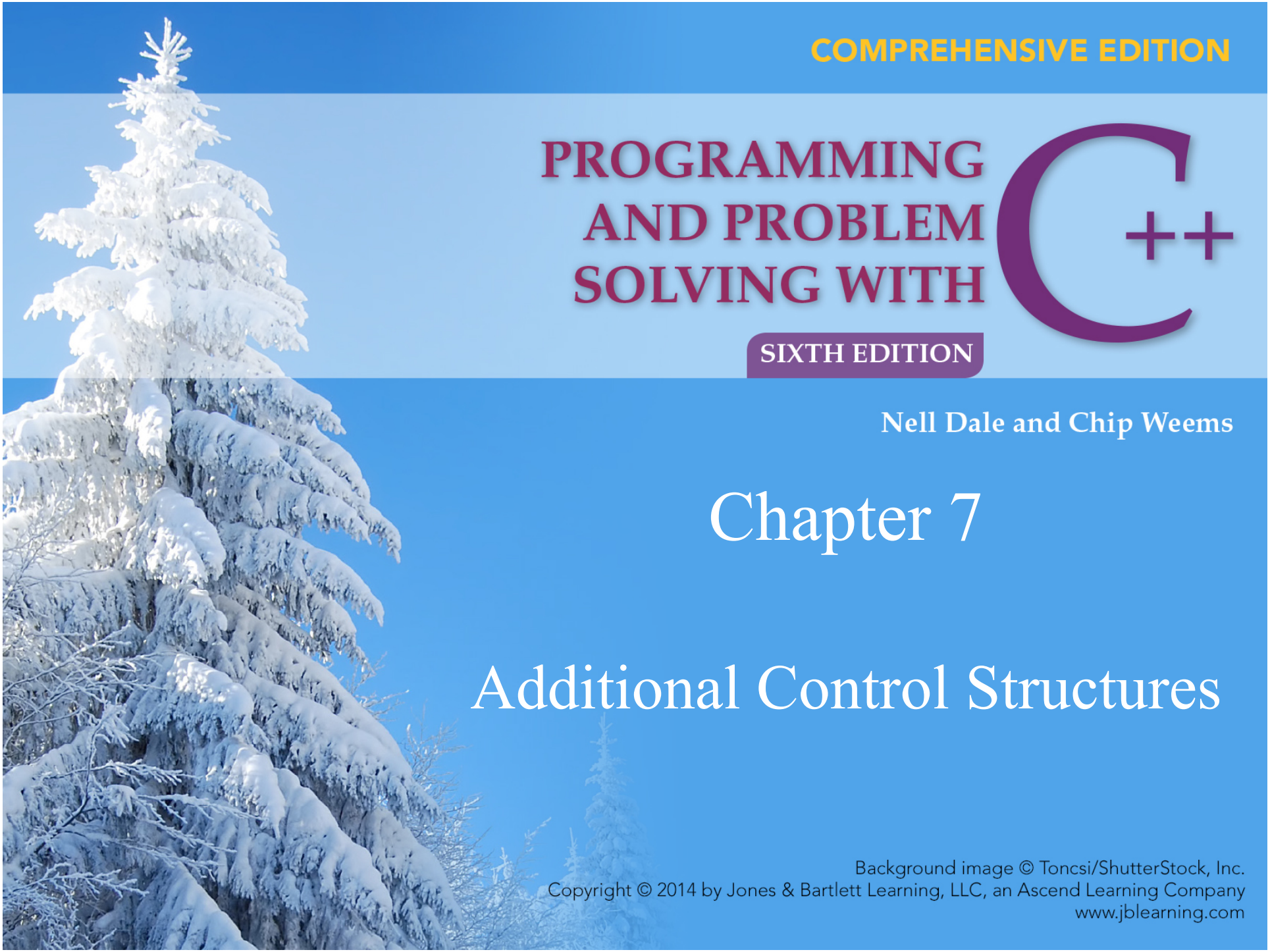


COMPREHENSIVE EDITION

PROGRAMMING AND PROBLEM SOLVING WITH C++

SIXTH EDITION

Nell Dale and Chip Weems

Chapter 7

Additional Control Structures

Chapter 7 Topics

- **Switch Statement for Multi-Way Branching**
- **Do-While Statement for Looping**
- **For Statement for Looping**
- **Using `break` and `continue` Statements**

Chapter 7 Topics

- **Additional C++ Operators**
- **Operator Precedence**
- **Type Coercion in Arithmetic and Relational Precedence**

Switch Statement

The Switch statement is a selection control structure for multi-way branching

```
switch  (IntegralExpression)
{
        case  Constant1 :
                Statement(s);           // optional
        case  Constant2 :
                Statement(s);           // optional
           .
           .
           .
        default :                       // optional
                Statement(s);           // optional
}
```

Example of Switch Statement

```
float   weightInPounds  = 165.8;
char    weightUnit;
   .  .    . // User enters letter for desired weightUnit
switch  (weightUnit)
{
    case 'P' :
    case 'p' :
            cout << weightInPounds
                 << " pounds " << endl;
            break;
    case 'O' :
    case 'o' :
             cout << 16.0 * weightInPounds
                  << " ounces " << endl;
```

Example of Switch Statement, continued

```
    break;
  case 'G' :
  case 'g' :
    cout << 454.0 * weightInPounds
         << " grams " << endl;
    break;
  default :
    cout << "That unit is not handled! "
         << endl;
    break;
}
```

Switch Statement

- The value of IntegralExpression (of char, short, int, long or enum type) determines which branch is executed
- Case labels are constant (possibly named) integral expressions
- Several case labels can precede a statement

Control in Switch Statement

- Control branches to the statement following the case label that matches the value of IntegralExpression
- Control proceeds through all remaining statements, including the default, unless redirected with break

Control in Switch Statement

- If no case label matches the value of IntegralExpression, control branches to the default label, if present
- Otherwise control passes to the statement following the entire switch statement
- Forgetting to use break can cause logical errors because after a branch is taken, control proceeds sequentially until either break or the end of the switch statement occurs

Do-While Statement

Do-While is a looping control structure in which the loop condition is tested *after* each iteration of the loop

SYNTAX

```
do
{
        Statement

} while (Expression);
```

Loop body statement can be a single statement or a block

Example of Do-While

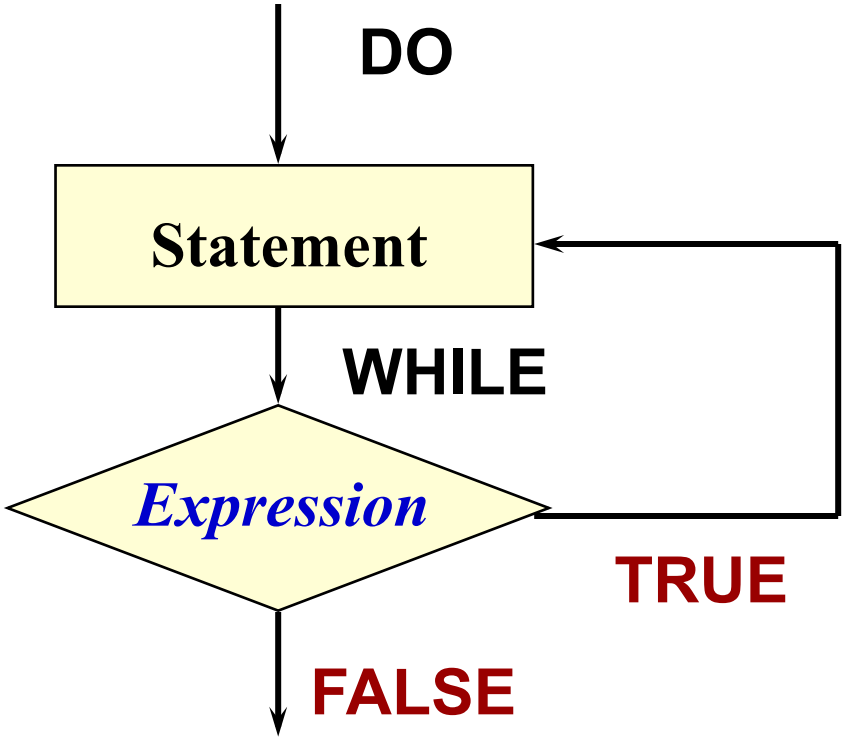
```
void GetYesOrNo (/* out */  char&  response)
//  Inputs a character from the user
//  Postcondition: response has been input
//                 && response == 'y' or 'n'
{
  do
  {
    cin >> response; // Skips leading whitespace

    if ((response != 'y') && (response != 'n'))
    cout << "Please type  y or n : ";
  } while ((response != 'y') && (response != 'n'));
}
```

Do-While Loop vs. While Loop

- POST-TEST loop (exit-condition)
- The looping condition is tested after executing the loop body
- Loop body is always executed at least once

- PRE-TEST loop (entry-condition)
- The looping condition is tested before executing the loop body
- Loop body may not be executed at all

Do-While Loop



When the expression is tested and found to be false, the loop is exited and control passes to the statement that follows the Do-while statement

For Loop

SYNTAX

```
for (initialization;  test expression; update)
{
        Zero or more statements to repeat

}
```

For Loop

For loop contains

- An initialization
- An expression to test for continuing
- An update to execute after each iteration of the body

Example of For Loop

```
int   num;

for (num = 1; num <= 3; num++)
{
    cout  <<  num  <<  "Potato"
          <<  endl;
}
```

Example of Repetition

num ?

```
int   num;
```

```
for (num = 1; num <= 3; num++)
    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

Example of Repetition

num 1

```
int   num;

for (num = 1; num <= 3; num++)
    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

num 1

Example of Repetition

```
int   num;

for(num = 1; num <= 3; num++)

     cout  <<  num  <<  "Potato"
           <<  endl;
```

true

OUTPUT

Example of Repetition

num 1

```
int   num;

for (num = 1; num <= 3; num++)
    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

```
1Potato
```

num 2

Example of Repetition

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

```
1Potato
```

Example of Repetition

num 2

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"
          <<  endl;
```

true

OUTPUT

```
1Potato
```

num 2

Example of Repetition

```
int   num;

for (num = 1; num <= 3; num++)
    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

```
1Potato
2Potato
```

num 3

Example of Repetition

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

```
1Potato
2Potato
```

num 3

Example of Repetition

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"
          <<  endl;
```

true

OUTPUT

```
1Potato
2Potato
```

num 3

Example of Repetition

```
int   num;

for(num = 1; num <= 3; num++)
    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

```
1Potato
2Potato
3Potato
```

Example of Repetition

num 4

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"
          <<  endl;
```

OUTPUT

```
1Potato
2Potato
3Potato
```

Example of Repetition

num **4**

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"
          <<  endl;
```

false

OUTPUT

```
1Potato
2Potato
3Potato
```

Example of Repetition

num 4

```
int   num;

for (num = 1; num <= 3; num++)

    cout  <<  num  <<  "Potato"
          <<  endl;
```

false

When the loop control condition is evaluated and has value false, the loop is said to be "satisfied" and control passes to the statement following the For statement

Output

The output was
1Potato
2Potato
3Potato

Count-controlled Loop

```
int  count;

for  (count = 4; count > 0; count--)
{
    cout  <<  count <<  endl;
}
cout   <<  "Done"  <<  endl;
```

OUTPUT: 4

3

2

1

Done

What is output?

```
int  count;

for (count = 0; count < 10; count++)
{
    cout  <<  "*";
}
```

Answer

****

The 10 asterisks are all on one line. Why?

What output from this loop?

```
int  count;

for (count = 0;  count < 10; count++);
{
    cout  <<  "*";
}
```

Answer

- No output from the for loop! *Why?*
- The semicolon after the () means that the body statement is a null statement

Answer

- In general, the body of the For loop is whatever statement *immediately* follows the ()
- That statement can be a single statement, a block, or a null statement
- Actually, the code outputs one * after the loop completes counting to 10

Several Statements in Body Block

```
const   int   MONTHS = 12;
int       count;
float     bill;
float     sum = 0.0;
for (count = 1;  count <= MONTHS;  count++)
{
     cout << "Enter bill: ";
     cin    >>  bill;
     sum = sum + bill;
}
cout << "Your total bill is : "  << sum <<  endl;
```

Break Statement

- The Break statement can be used with Switch or any of the 3 looping structures
- It causes an immediate exit from the Switch, While, Do-While, or For statement in which it appears
- If the Break statement is inside nested structures, control exits only the innermost structure containing it

Guidelines for Choosing Looping Statement

- For a simple count-controlled loop, use the For statement
- For an event-controlled loop whose body always executes once, use of Do-While statement
- For an event-controlled loop about which nothing is known, use a While statement
- When in doubt, use a While statement

Continue Statement

- The Continue statement is valid only within loops
- It terminates the current loop iteration, but not the entire loop
- In a For or While, Continue causes the rest of the body of the statement to be skipped; in a For statement, the update is done
- In a Do-While, the exit condition is tested, and if true, the next loop iteration is begun

Problem

Given a character, a length, and a width, draw a box

For example, given the values '&', 4, and 6, you would display

```
&&&&&&
&&&&&&
&&&&&&
&&&&&&
```

Additional C++ Operators

- Previously discussed C++ Operators include:
  - the assignment operator (=)
  - the arithmetic operators (+, -, *, /, %)
  - relational operators (==, !=, ,<=, >, >=)
  - logical operators (!, &&, ||)

- C++ has many specialized other operators seldom found in other programming languages

Additional C++ Operators

- See Table in 7.6 in Text for Additional C++ Operators for a full list of:
  - Combined Assignment Operators
  - Increment and Decrement Operators
  - Bitwise Operators
  - More Combined Assignment Operators
  - Other Operators

Assignment Operators and Assignment Expressions

- (=) is the basic assignment operator
- Every assignment expression has a value and a side effect, the value that is stored into the object denoted by the left-hand side
- For example, `delta = 2 * 12` has the value 24 and side effect of storing 24 into `delta`

Assignment Expressions

- In C ++, any expression become an expression statement when terminated by a semicolon
- The following are all valid C++ statements, first 2 have no effect at run time:

```
23;
2 * (alpha + beta);
delta = 2 * 12;
```

Increment and Decrement Operators

- **The increment and decrement operators (++ and --) operate only on variables, not on constants or arbitrary expressions**

1) Example of pre-incrementation

```
int1 = 14;
int2 = ++int1; // int1 == 15 && int2 == 15
```

2) Example of post-incrementation

```
int1 = 14;
int2 = int1++;  // int1 == 15 && int2 == 14
```

Bitwise Operators

- Bitwise operators (e.g., <<, >>, and |) are used for manipulating individual bits within a memory cell
- << and >> are left and right shift operators, respectively that take bits within a memory cell and shift them to the left or the right
- Do not confuse relational && and || operators used in logical expressions with & and | bitwise operators

The Cast Operation

- **Explicit type cast used to show that the type conversion is intentional**

- **In C++, the cast operation comes in three forms:**
  - `intVar = int(floatVar);` **// Functional notation**
  - `intVar = (int)floatVar;` **// Prefix notation**
  - `intVar = static_cast<int>(floatVar);` **// Keyword notation**

The Cast Operation

- Restriction on use of functional notation: Data type name must be a single identifier
- If type name consists of more than one identifier, prefix notation or keyword notation must be used
- Most software engineers now recommend use of keyword cast because it is easier to find these keywords in a large program

The sizeof Operator

- The `sizeof` operator --a unary operator that yields the size, in bytes, of its operand
- The operand can be a variable name , as in `sizeof someInt`
- Alternatively, the operand can be the name of a data type enclosed in parentheses: `(sizeof float)`

The ? Operator

- ? : operator, also called the conditional operator is a three-operand operator
- Example of its use: set a variable `max` equal to the larger of two variables `a` and `b`.
- With the `? :` operator , you can use the following assignment statement:

```
max = (a>b) ? a : b;
```

Operator Precedence

- Following Table on slide 53 and slide 54 groups discussed operators by precedence levels for C++
- Horizontal line separates each precedence level from the next-lower level
- Column level Associativity describes grouping order.
- Within a precedence level, operators group Left to right or, Right to left

Operator	Associativity	Remarks
`()` `++` `--`	Left to right Right to left	Function call and function-style cast `++`and `-` as postfix operators

`++` `--`
`!` Unary
+Unary
(cast)
`sizeof`
	Right to left
	Right to left
		`++`and `-` as prefix operators

`*` `/` `%`	Left to right	
`+` `-`	Left to right	

Operator	Associativity	Remarks
< <= > >=	Left to right	
== !=	Left to right	
&&	Left to right	
||	Left to right	
? :	Right to left	
= += -= *= /=	Right to left	

Type Coercion in Arithmetic and Relational Expressions

- If two operands are of different types, one of them is temporarily **promoted** (or **widened**) to match the data type of the other
- Rule of type coercion in an arithmetic coercion:

Step 1: Each `char`, `short`, `bool`, or enumeration value is promoted (widened) to int. If both operands are now `int`, the result is an `int` expression.

Type Coercion in Arithmetic and Relational Expressions

Step 2: If Step 1 still leaves a mixed type expression, the following precedence of types is used:

```
int, unsigned int, long, float, double,
  long double
```

Type Coercion in Arithmetic and Relational Expressions

Example: expression `someFloat+2`

- Step 1 leaves a mixed type expression
- In Step 2, `int` is a lower type than the `float` value---for example, 2.0
- Then the addition takes place, and the type of the entire expression is `float`

Type Coercion in Arithmetic and Relational Expressions

- This description also holds for relational expressions such as `someInt <= someFloat`
- Value of `someInt` temporarily coerced to floating-point representation before the comparison occurs
- Only difference between arithmetic and relational expressions:
- Resulting type of relational expression is always `bool`---the value `true` or `false`