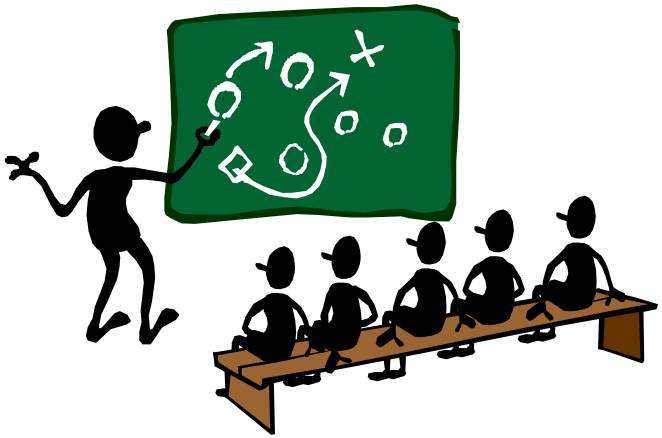
C++ Programming Language
Chapter 2 Flow of Control



Juinn-Dar Huang

Associate Professor

jdhuang@mail.nctu.edu.tw

February 2011

Learning Objectives

- Boolean expressions and operators
- Operator evaluation and precedence rules
- Branches
  - if-else
  - nested if-else structures
  - switch
- Loops
  - while, do-while, for
  - nested loop structures

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Evaluating Boolean Expressions

- Boolean expression
  - any expression can be evaluated true or false

- Data type bool
  - its value is either true or false
  - true, false are predefined constants

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Logical & Comparison Operators

- Logical Operators
  - logical AND (&&)
  - logical OR (||)
  - logical NOT (!)

```
if ( (a >= 10) && (b == 15) )
if ( (a != 12) || (b < 21) )
if (! bvalue)
```

Display 2.1 Comparison Operators

MATH SYMBOL	ENGLISH	C++ NOTATION	C++ SAMPLE	MATH EQUIVALENT
=	Equal to	==	x + 7 == 2*y	x + 7 = 2y
≠	Not equal to	!=	ans != 'n'	*ans* ≠ *'n'*
<	Less than	<	count < m + 3	*count* < m + 3
≤	Less than or equal to	<=	time <= limit	*time* ≤ *limit*
>	Greater than	>	time > limit	*time* > *limit*
≥	Greater than or equal to	>=	age >= 21	*age* ≥ 21

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Truth Tables of Logical Operators

Display 2.2 Truth Tables

AND

Exp_1	*Exp_2*	*Exp_1* && *Exp_2*
true	true	true
true	false	false
false	true	false
false	false	false

OR

Exp_1	*Exp_2*	*Exp_1* || *Exp_2*
true	true	true
true	false	true
false	true	true
false	false	false

NOT

Exp	!(*Exp*)
true	false
false	true

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Precedence of Operators (1/4)

Display 2.3 Precedence of Operators

::	Scope resolution operator	LR
.	Dot operator	LR
->	Member selection	
[]	Array indexing	
( )	Function call	
++	Postfix increment operator (placed after the variable)	
--	Postfix decrement operator (placed after the variable)	
++	Prefix increment operator (placed before the variable)	RL
--	Prefix decrement operator (placed before the variable)	
!	Not	
-	Unary minus	
+	Unary plus	
*	Dereference	
&	Address of	
new	Create (allocate memory)	
delete	Destroy (deallocate)	
delete[]	Destroy array (deallocate)	
sizeof	Size of object	
( )	Type cast	

Highest precedence (done first)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Precedence of Operators (2/4)

*	Multiply	LR
/	Divide	
%	Remainder (modulo)	
+	Addition	LR
–	Subtraction	
<<	Insertion operator (console output)	LR
>>	Extraction operator (console input)	

Lower precedence
(done later)

Precedence of Operators (3/4)

Display 2.3 Precedence of Operators

All operators in part 2 are of lower precedence than those in part 1.

< > <= >=	Less than Greater than Less than or equal to Greater than or equal to	LR
== !=	Equal Not equal	LR
&&	And	LR
||	Or	LR

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Precedence of Operators (4/4)

=	Assignment	RL
+=	Add and assign	
−=	Subtract and assign	
*=	Multiply and assign	
/=	Divide and assign	
%=	Modulo and assign	
? :	Conditional operator	RL
throw	Throw an exception	RL
,	Comma operator	LR

swap

Lowest precedence (done last)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

More about Operator Precedence

- The table in textbook is incomplete and even erroneous!
  - check the following link for a comprehensive survey http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B
- Be aware of the associativity too!
  - e.g., a * b / c *d ➔ (((a * b) / c) * d)
  - e.g., a += b = c; ➔ ((a += (b = c));

LR: Left-to-Right

RL: Right-to-Left

- Whenever you are not sure, use parentheses!

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Precedence Examples

- Examples
  - x + y + z ➔ ((x + y) + z)
  - x + y * z ➔ (x + (y * z))
  - (x + y) * z ➔ ((x + y) * z)
  - x + 1 > 2 || x + 1 < -3 ➔ (((x + 1) > 2) || ((x + 1) < (-3)))

- Cautions!
  - if (0 < score < 10) vs. ((0 < score) && (score < 10))
  - if (a = 10) vs. if (a == 10)
  - unfortunately, all above cases result in **no** compilation errors!

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Short-Circuit Evaluation

- Short-circuit evaluation for && and ||
  - if ((x >= 0) && (y > 1))
    - if the value of x is negative, then the value of y will **NOT** be examined
    - if the value of x is nonnegative, then the value of y will be examined
  - if ((x >= 0) || (y > 1))
    - if the value of x is nonnegative, then the value of y will **NOT** be examined
    - if the value of x is negative, then the value of y will be examined
  - **Cautions**!
    - if ((--x) && (--y)) // **Q:** assume y = 9 initially, y = ? at the end?
    - **Answer:** It depends. If x = 1 then y is still equal to 9; otherwise y = 8.
  - **Bright side**

```
if ( (kids != 0) && ( (pies / kids) >= 2) )
    cout << "Each kid can have two pieces of pies at least!" << endl;
```

Shift Operators

Right shift >> , and left shift <<

- Divided by power of 2 using right shift

```
int i = 53;          // 53_10 = 110101_2
int q = i >> 4;      // actually, q = i / 16; 2^4 = 16
```

- Multiplied by power of 2 using left shift

```
int i = 53;
int m = i << 2;      // actually, m = i * 4; 2^2 = 4
```

- Not power of 2?
  - How to get i * 20? ➔ m = (i << 4) + (i << 2);
  - How to get i * 15? ➔ m = (i << 4) – i;

Bitwise Operators

- Bitwise AND, &

  int i = 53; // $53_{10} = 110101_2$

  int r = i & 0xf; // actually, r = i % 16;

  mask: 0s to reset bits; 1s to keep bits unaltered

- Bitwise OR, |

  int i = 53; // $53_{10} = 110101_2$

  int m = i | 0x6; // $m = 110111_2 = 55_{10}$

  mask: 1s to set bits; 0s to keep bits unaltered

- Bitwise exclusive OR, ^
  - int i = 53; // $53_{10} = 110101_2$
  - int n = 53 ^ 0xf; // $n = 111010_2 = 58_{10}$
  - mask: 1s to **toggle** (**flip**) bits; 0s to keep bits unaltered

Branch Mechanisms

- if-else statements
  - choice of two mutually exclusive statements based on condition expression
  - example:

```
if (hrs > 40)
  grossPay = rate*40 + 1.5*rate*(hrs-40);  // true part
else
  grossPay = rate*hrs;                     // false part
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

if-else Statement Syntax

- Syntax:

```
if (<Boolean_expression>)
        <true_statement>
else
        <false_statement>
```

- Note each alternative contains only ONE statement!
- To have multiple statements executed in either branch
  ➔ use compound statement

Compound/Block Statement

- Only “get” one statement per branch
- Must use **compound statement** **{ }** for multiples
  - also called a block statement
- Suggestion:
  - use a block statement even for just one statement
  - better readability

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Indentation Styles

- Two common ways to do indentation:

```
if (myScore > yourScore)
{
        cout << “I win!\n”;
        wager = wager + 100;
}
else
{
        cout << “I wish these were golf scores.\n”;
        wager = 0;
}
```

```
if (myScore > yourScore) {
        cout << “I win!\n”;
        wager = wager + 100;
} else {
        cout << “I wish these were golf scores.\n”;
        wager = 0;
}
```

Common Pitfalls: = vs. ==

- Operator "=" vs. operator "=="
  - lest one means "assignment" (=)
  - right one means "equality" (==)
  - **VERY** different in C++!

  - example:

```
if (x = 12)
    Do_Something
else
    Do_Something_Else
// in this case, Do_Something ALWAYS gets executed!
```

Optional else

- else clause is optional
  - If, in the false branch (else), you want “nothing” to happen, then just leave it out
  - example:

```
if (sales >= minimum)
    salary += bonus;
cout << “Salary = %” << salary;
```

  - nothing to do for false condition, so there is no else clause!
  - execution continues with cout statement

Nested if-else Statements

- if-else statements
  - can contain compound or simple statements (we have seen)
  - can also contain another if-else statement!
  - example:

```
if (speed > 110)
    if (speed > 150)
        cout << “Too fast!\n”;
    else
        cout << “Fast.\n”;
```

  - need proper indenting!

Multiway if-else

- Not new, just different indenting
- Avoid "excessive" indenting
  - Syntax:

Multiway if–else Statement

SYNTAX

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
        .
        .
        .
else if (Boolean_Expression_n)
    Statement_n
else
    Statement_For_All_Other_Possibilities
```

Example of Multiway if-else

EXAMPLE

```
if ((temperature < -10) && (day == SUNDAY))
    cout << "Stay home.";
else if (temperature < -10) //and day != SUNDAY
    cout << "Stay home, but call work.";
else if (temperature <= 0) //and temperature >= -10
    cout << "Dress warm.";
else //temperature > 0
    cout << "Work hard and play hard.";
```

The Boolean expressions are checked in order until the first `true` Boolean expression is encountered, and then the corresponding statement is executed. If none of the Boolean expressions is *true*, then the *Statement_For_All_Other_Possibilities* is executed.

Order Does Matter…

誰來晚餐? PART 1

if (林志玲 says yes)
　我會喜極而泣的赴宴
else if (林熙蕾 says yes)
　我會欣喜若狂的赴宴
else if (林若亞 says yes)
　我會歡天喜地的赴宴
else if (林逸欣 says yes)
　我會快快樂樂的赴宴
else
　吃飯只是為了要活著罷了。。。

誰來晚餐? PART 2

if (林逸欣 says yes)
　我會快快樂樂的赴宴
else if (林若亞 says yes)
　我會歡天喜地的赴宴
else if (林熙蕾 says yes)
　我會欣喜若狂的赴宴
else if (林志玲 says yes)
　我會喜極而泣的赴宴
else
　吃飯只是為了要活著罷了。。。

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

switch Statement

- Another statement for controlling multiple branches
- Check syntax on the next slide
- Controlling expression MUST return an integral value
  - OK: char, int, bool, enum (see page 92)
  - not OK: float, double, …
- Case labels must also be integral values
- break and default are optional

switch Statement Syntax

switch Statement

SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
                .
                .
                .
    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
}
```

You need not place a **break** *statement in each case. If you omit a* **break**, *that case continues until a* **break** *(or the end of the* **switch** *statement) is reached.*

Example of switch Statement

EXAMPLE

```
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
  }
```

If you forget this **break**, *then passenger cars will pay $1.50.*

switch: Multiple case Labels

- Execution "falls thru" until break
  - a case label provides a "point of entry"
  - example:

```
case 'A':
case 'a':
    cout << "Excellent: you got an A!\n";
    break;

case 'B':
case 'b':
    cout << "Good: you got a B!\n";
    break;
```

  - note multiple labels provide an identical "entry"
    - e.g., 'a' and 'A'

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Pitfall and Tip for switch

Pitfall

- Forgetting breaks
  - No compilation error
  - Execution simply “falls thru” other cases until break

Tip

- One common use: MENUs
  - provides clearer “big-picture” view
  - shows menu structure effectively
  - each branch is one menu choice

Menu Example

- Switch statement for menus:

```
switch (response)
{
        case 1:
                // Execute menu option 1
                break;
        case 2:
                // Execute menu option 2
                break;
        case 3:
                // Execute menu option 3
                break;
        default:
                cerr << “Please enter valid response.” << endl;
}
```

Conditional Operator

$expr_1$? $expr_2$: $expr_3$

- Also called "ternary operator"
  - allows embedding a conditional into an expression
  - essentially a "shorthand if-else" operator
  - example:

```
if (n1 > n2)
    max = n1;
else
    max = n2;
```

  - can be also written as:

```
max = (n1 > n2) ? n1 : n2;
```

Word Bank

Unary
Binary
Ternary

Suggestion: Minimize the use of condition operator

Loops

- 3 Types of loops in C++
  - while
  - do-while
    - always enters the loop body at least once
  - for
    - appropriate for “counting” loops

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

while Loop Syntax

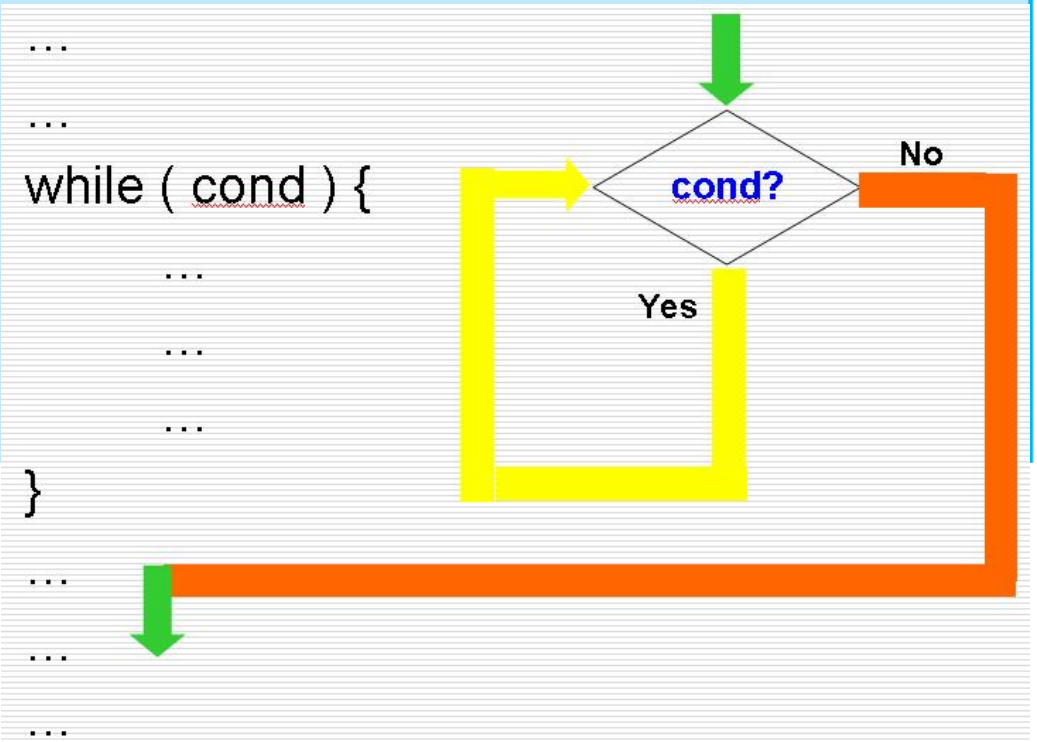
Syntax for while and do-while Statements

A while STATEMENT WITH A SINGLE STATEMENT BODY

```
while (Boolean_Expression)
    Statement
```

A while STATEMENT WITH A MULTISTATEMENT BODY

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
      .
      .
      .
    Statement_Last
}
```



while Loop Example

```
count = 0;                  // initialization
while (count < 3)           // loop condition
{
      cout << “Hi ”;        // loop body
      ++count;              // update expression
}
```

- How many times does the loop body get executed?

do-while Loop Syntax

A do-while STATEMENT WITH A SINGLE-STATEMENT BODY

```
do
    Statement
while (Boolean_Expression);
```

A do-while STATEMENT WITH A MULTISTATEMENT BODY

```
do
{
    Statement_1
    Statement_2
      .
      .
      .
    Statement_Last
} while (Boolean_Expression);
```

Do not forget the final semicolon.

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

do-while Loop Example

```
count = 0;                    // initialization
do
{
      cout << “Hi ”;          // loop body
      ++count;                // update expression
} while (count < 3);          // loop condition
```

- How many times does the loop body get executed?
- do-while loop body always gets executed **at least once**!

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

while vs. do-while

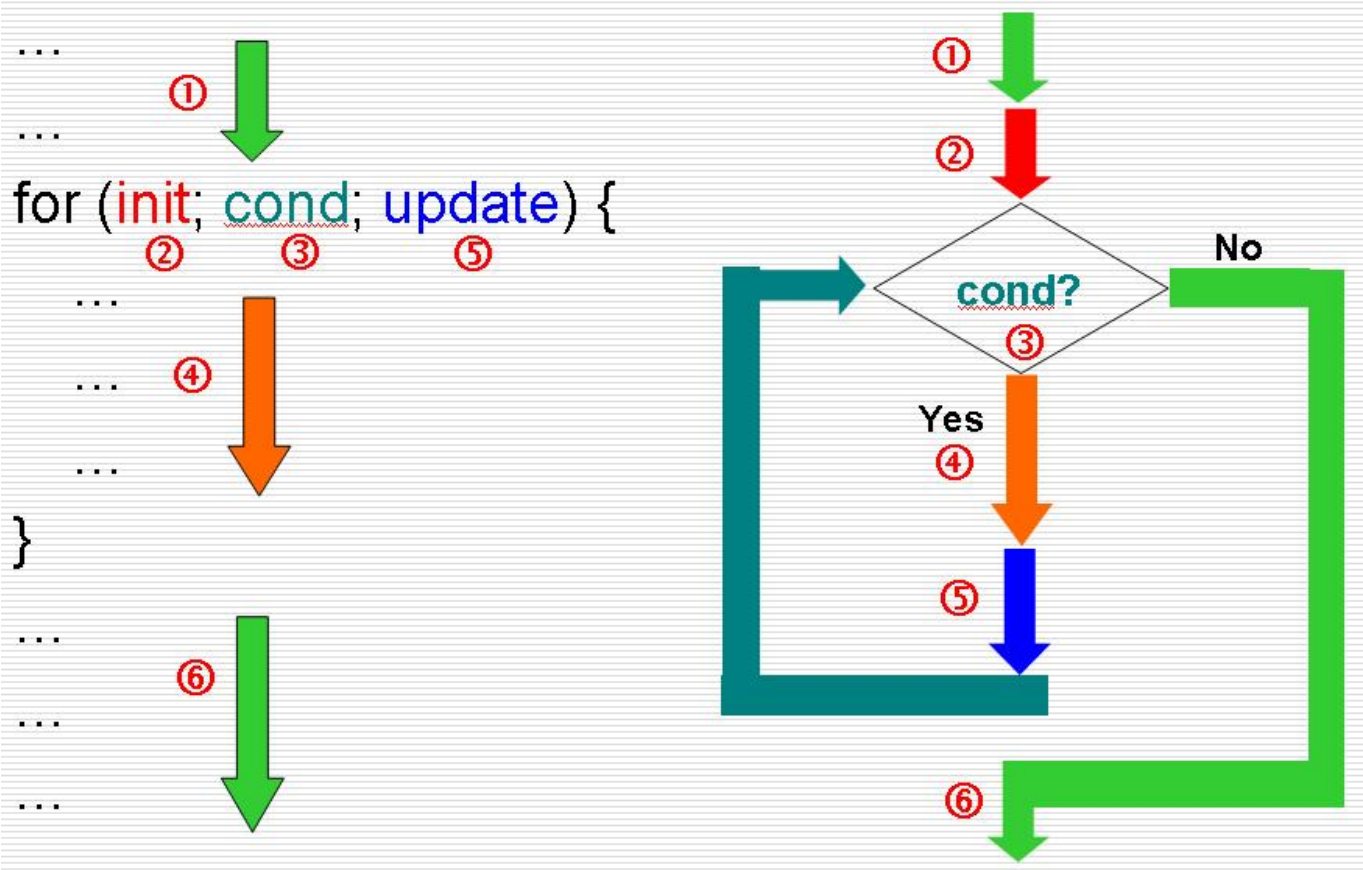
- Very similar, but …
  - one important difference
    - issue is **WHEN** Boolean expression is checked
    - while: checked **BEFORE** body is executed
    - do-while: checked **AFTER** body is executed
- Except for that, they are essentially identical!

Suggestion: Use while instead of do-while whenever possible

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Loop Syntax

Syntax:
for (Init_Action; Bool_Cond; Update_Action)
Body_Statement



Loop Usage

- Like if-else, Body_Statement can be a block statement
  - much more typical
- It is intended for expressing fairly regular loop structures
  - typically controlled by a loop variable
  - how a loop variable is initialized, tested, and updated is highlighted
    - better readability and thus less errors
  - ②, ③, ⑤ are all optional
  - it's **UNWISE** to put loop variable unrelated expressions in ②, ③, ⑤
  - a for-loop can **ALWAYS** be converted to an equivalent while-loop

- Suggestion:
  - prefer while-loop to for-loop if there is no obvious loop variable

for Loop Example

```
for (count=0; count<3; ++count)
{
        cout << "Hi ";          // loop body
}
```

- How many times does the loop body get executed?
- Initialization, loop condition and update are all highlighted at the beginning of the for-loop structure!

Loop Issues

- Loop's condition expression can be ANY Boolean expression
- Examples:

```
while (count<3 && done!=0)
{
    // Do something
}

for (index=0; index<10 && entry!=99 ; ++index)
{
    // Do something
}
```


Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Loop Pitfalls: Misplaced ;

- Watch the misplaced ; (semicolon)
  - Example:

```
response = 1;
while (response != 0) ; ←
{
    cout << "Enter val: ";
    cin >> response;
}
```

- Unfortunately, there is **NO** compilation error here
- Result here: **INFINITE LOOP**!

Intentional Infinite Loops

- Commonly, loop condition must evaluate to false at some iteration through loop
  - if not ➔ infinite loop

- Sometimes, programmers create infinite loops intentionally
  - example:

```
while (true) {
    // do something
}
```

```
for( ; ;) {
    // do something
}
```

  - perfectly legal C++ loops ➔ always infinite!

- Infinite loops can be desirable
  - e.g., embedded systems

break and continue Statements (1/2)

- Flow of Control
  - loops provide clear flow of control in and out
  - In RARE instances, a programmer can alter the natural flow
- break
  - forces loop to exit immediately
    - recall what break does in switch statement
- continue
  - skips rest of loop body
- These statements violate natural flow
  - only used when absolutely necessary!

break and continue Statements (2/2)

```
int number, sum = 0, count = 0;
  cout << "Enter 4 negative numbers:\n";

  while (++count <= 4)
  {
     cin >> number;

     if (number >= 0)
     {
        cout << "ERROR: positive number"
             << " or zero was entered as the\n"
             << count << "th number! Input ends "
             << "with the " << count
             << "th number.\n";
        break;
     }

     sum = sum + number;
  }

  cout << sum << " is the sum of the first "
       << (count - 1) << " numbers.\n";
```

P. 107

```
int number, sum = 0, count = 0;
  cout << "Enter 4 negative numbers, ONE
    PER LINE:\n";

  while (count < 4)
  {
     cin >> number;

     if (number >= 0)
     {
        cout << "ERROR: positive number "
             << "(or zero)!\n"
             << "Reenter that number and "
             << "continue:\n";
        continue;
     }

     sum = sum + number;
     ++count;
  }

  cout << sum << " is the sum of the "
       << count << " numbers.\n";
```

P. 108

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Nested Loops

- ANY valid C++ statements can be inside loop body
- This includes additional loop statements!
  - called **nested loops**
- Requires careful indenting:

```
for (outer=0; outer<10; ++outer)
    for (inner=0; inner<20; ++inner)
        sum += data[outer][inner];
```

- Notice no { } in the above case since each body contains only one statement

Comma Operator (,)

- Evaluate list of expressions from the leftmost to the rightmost strictly

```
int a = 10, b = 1; t;
t = a, a = b, b = t;     // swap the values of a and b using t
```

- Return value of the last expression

```
first = (first = 2, second = first + 1);
```

  – 1: first gets assigned the value 2
  – 2: second gets assigned the value 3 (← 2 + 1)
  – 3: first gets assigned the value 3

- Most often used in a for-loop

```
for (i = 0, j = 9;  i < 10; ++i, --j)
    b_array[i] = a_array[j];
```

Summary (1/2)

- Boolean expressions
- Logical and comparison operators
- Operator precedence
- Short-circuiting evaluation for && , ||
- shift (<< , >>) and bitwise (& , | , ^) operators
- Branch statements
  - if-else
  - switch
- Loop statements
  - while
  - do-while
  - for

Summary (2/2)

- do-while loops
  - Always execute their loop body at least once
- for-loop
  - there is an obvious loop variable
- Unintentional and intentional infinite loops
- break and continue statements
- comma (,) operator

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw