EE1003
Introduction to Computer I

Chapter 4
Control Statement: Part II

Andy, Yu-Guang Chen
Assistant Professor, Department of EE
National Central University
andyygchen@ee.ncu.edu.tw

Learning Objectives

In this chapter you'll learn:

- The essentials of counter-controlled repetition.
- To use `for` and `do`...`while` to execute statements in a program repeatedly.
- To implement multiple selection using the `switch` selection statement.
- How `break` and `continue` alter the flow of control.
- To use the logical operators to form conditional expressions in control statements.
- To avoid confusing the equality and assignment operators.

Outline

4.1 Introduction

- `for`, `do`…`while` and `switch` statements.
- counter-controlled repetition.
- Introduce the `break` and `continue` program control statements.
- Logical operators for more powerful conditional expressions.
- Examine the common error of confusing the equality (==) and assignment (=) operators, and how to avoid it.
- Summarize C++'s control statements.

4.2 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires
  - the name of a control variable (or loop counter)
  - the initial value of the control variable
  - the loop-continuation condition that tests for the final value of the control variable (i.e., whether looping should continue)
  - the increment (or decrement) by which the control variable is modified each time through the loop.
- In C++, it's more precise to call a declaration that also reserves memory a definition.

4.2 Essentials of Counter-Controlled Repetition

```
// Fig. 4.1: fig04_01.cpp
// Counter-controlled repetition.
#include <iostream>
using namespace std;

int main()
{
   int counter = 1; // declare and initialize control variable

   while ( counter <= 10 ) // loop-continuation condition
   {
      cout << counter << " ";
      counter++; // increment control variable by 1
   } // end while

   cout << endl; // output a newline
} // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 4.1 | Counter-controlled repetition.

4.3 for Repetition Statement

- The for repetition statement specifies the counter-controlled repetition details in a single line of code.
- The initialization occurs once when the loop is encountered.
- The condition is tested next and each time the body completes.
- The body executes if the condition is true.
- The increment occurs after the body executes.
- Then, the condition is tested again.
- If there is more than one statement in the body of the `for`, braces are required to enclose the body of the loop.

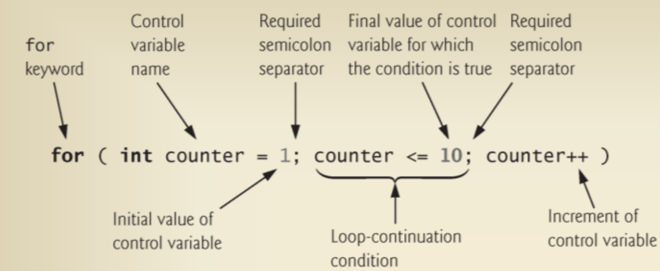
4.3 for Repetition Statement

```cpp
// Fig. 4.2: fig04_02.cpp
// Counter-controlled repetition with the for statement.
#include <iostream>
using namespace std;

int main()
{
   // for statement header includes initialization,
   // loop-continuation condition and increment.
   for ( int counter = 1; counter <= 10; counter++ )
      cout << counter << " ";

   cout << endl; // output a newline
} // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 4.2 | Counter-controlled repetition with the `for` statement.

4.3 for Repetition Statement



Good Programming Practice 4.4

Using the final value in the condition of a `while` *or* `for` *statement and using the* `<=` *relational operator will help avoid off-by-one errors. For a loop used to print the values 1 to 10, for example, the loop-continuation condition should be* `counter <= 10` *rather than* `counter < 10` *(which is an off-by-one error) or* `counter < 11` *(which is nevertheless correct). Many programmers prefer so-called **zero-based counting**, in which, to count 10 times through the loop,* `counter` *would be initialized to zero and the loop-continuation test would be* `counter < 10`.

4.3 for Repetition Statement

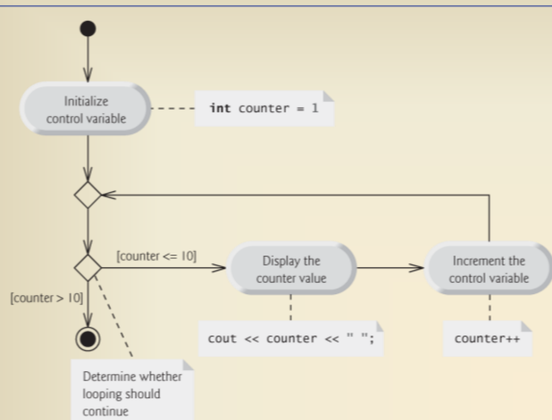


Fig. 4.4 | UML activity diagram for the `for` statement in Fig. 4.2.

4.3 for Repetition Statement

- The `for` repetition statement's UML activity diagram is similar to that of the `while` statement (Fig. 4.6).
- Figure 4.4 shows the activity diagram of the `for` statement in Fig. 4.2.
- The diagram makes it clear that initialization occurs once before the loop-continuation test is evaluated the first time, and that incrementing occurs each time through the loop *after* the body statement executes.

4.3 for Repetition Statement

```
for (int counter=1; counter<=10; counter++)
```

- If the *initialization* expression declares the control variable, the control variable can be used only in the body of the `for` statement
  - The control variable will be unknown outside the `for` statement.
  - This restricted use of the control variable name is known as the variable's scope.

Common Programming Error 4.3

When the control variable is declared in the initialization section of the `for` statement, using the control variable after the body is a compilation error.

4.3 for Repetition Statement (cont.)

```
for (i=1, j=1; i<=10; i++, j++)
```

- The *initialization* and *increment* expressions can be comma-separated lists of expressions.
  - Those expressions evaluate from left to right.

Good Programming Practice 4.5

Place only expressions involving the control variables in the initialization and increment sections of a `for` *statement. Manipulations of other variables should appear either before the loop (if they should execute only once, like initialization statements) or in the loop body (if they should execute once per repetition, like incrementing or decrementing statements).*

4.3 for Repetition Statement

- The general form of the `for` statement is

```
for ( initialization; loopContinuationCondition; increment )
    statement
```

- where the *initialization* expression initializes the loop's control variable, *loopContinuationCondition* determines whether the loop should continue executing and *increment* increments the control variable.
- In most cases, the `for` statement can be represented by an equivalent `while` statement, as follows:

```
initialization;

while ( loopContinuationCondition )
{
    statement
    increment;
}
```

4.3 for Repetition Statement (cont.)

- The three expressions in the `for` statement header are optional (but the two semicolon separators are required).
- If the *loopContinuationCondition* is omitted, C++ assumes that the condition is true, thus creating an infinite loop.
- One might omit the *initialization* expression if the control variable is initialized earlier in the program.
- One might omit the *increment* expression if the increment is calculated by statements in the body of the `for` or if no increment is needed.

4.3 for Repetition Statement (cont.)

- The initialization, loop-continuation condition and increment expressions of a `for` statement can contain arithmetic expressions.
- The expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

  are all equivalent in the incrementing portion of the `for` statement's header (when no other code appears there).
- The "increment" of a `for` statement can be negative, in which case the loop actually counts downward.
- If the loop-continuation condition is initially false, the body of the `for` statement is not performed.

4.3 for Repetition Statement (cont.)

Common Programming Error 4.5

Placing a semicolon immediately to the right of the right parenthesis of a `for` header makes the body of that `for` statement an empty statement. This is usually a logic error.

```
for ( counter=1; counter<=10; counter++ );
```

→ Do nothing for 10 times

Error-Prevention Tip 4.2

Although the value of the control variable can be changed in the body of a `for` statement, avoid doing so, because this practice can lead to subtle logic errors.

4.4 Examples Using the for Statement

- Vary the control variable from 1 to 100 in increments of 1.
  - `for ( int i = 1; i <= 100; i++ )`
- Vary the control variable from 100 down to 1 in decrements of 1.
  - `for ( int i = 100; i >= 1; i-- )`
- Vary the control variable from 7 to 77 in steps of 7.
  - `for ( int i = 7; i <= 77; i += 7 )`
- Vary the control variable from 20 down to 2 in steps of -2.
  - `for ( int i = 20; i >= 2; i -= 2 )`
- Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.
  - `for ( int i = 2; i <= 17; i += 3 )`
- Vary the control variable over the following sequence of values: 99, 88, 77, 66, 55.
  - `for ( int i = 99; i >= 55; i -= 11 )`

4.4 Examples Using the for Statement (cont.)

◆ The program of Fig. 4.5 uses a `for` statement to sum the even integers from 2 to 20.

```
// Fig. 4.5: fig04_05.cpp
// Summing integers with the for statement.
#include <iostream>
using namespace std;

int main()
{
   int total = 0; // initialize total

   // total even integers from 2 through 20
   for ( int number = 2; number <= 20; number += 2 )
      total += number;

   cout << "Sum is " << total << endl; // display results
} // end main
```

```
Sum is 110
```


4.4 Examples Using the for Statement (cont.)

◆ A person invests $1000.00 in a savings account yielding 5 percent interest.

◆ Use the following formula to calculate and print the amount of money in the account at the end of each year for 10 years. :

$$a = p\,(1 + r)^n$$

where

p is the original amount invested (i.e., the principal),
r is the annual interest rate,
n is the number of years and
a is the amount on deposit at the end of the *n*th year.

◆ This problem involves a loop that performs the indicated calculation for each of the 10 years.

4.4 Examples Using the for Statement (cont.)

```cpp
// Fig. 4.6: fig04_06.cpp
// Compound interest calculations with for.
#include <iostream>
#include <iomanip>
#include <cmath> // standard C++ math library
using namespace std;

int main()
{
   double amount; // amount on deposit at end of each year
   double principal = 1000.0; // initial amount before interest
   double rate = .05; // interest rate

   // display headers
   cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;

   // set floating-point number format
   cout << fixed << setprecision( 2 );

```

Fig. 4.6 | Compound interest calculations with `for`. (Part 1 of 2.)

4.4 Examples Using the for Statement (cont.)

```cpp
   // calculate amount on deposit for each of ten years
   for ( int year = 1; year <= 10; year++ )
   {
      // calculate new amount for specified year
      amount = principal * pow( 1.0 + rate, year );

      // display the year and the amount
      cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
   } // end for
} // end main
```

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Fig. 4.6 | Compound interest calculations with `for`. (Part 2 of 2.)

4.4 Examples Using the for Statement (cont.)

- C++ does not include an exponentiation operator, so we use the standard library function `pow`.
  - `pow(x, y)` calculates the value of `x` raised to the `y`th power.
  - Takes two arguments of type `double` and returns a `double` value.
- This program will not compile without including header file `<cmath>`.
  - Includes information that tells the compiler to convert the value of `year` to a temporary `double` representation before calling the function.
  - Contained in `pow`'s function prototype.

4.4 Examples Using the for Statement (cont.)

- The stream manipulator `setw(4)` specifies that the next value output should appear in a field width of 4.
  - If less than 4 character positions wide, the value is right justified in the field by default.
  - If more than 4 character positions wide, the field width is extended to accommodate the entire value.
- To indicate that values should be output left justified, simply output nonparameterized stream manipulator `left` .
- Right justification can be restored by outputting nonparameterized stream manipulator `right`.

4.4 Examples Using the for Statement (cont.)

- Stream manipulator `fixed` indicates that floating-point values should be output as fixed-point values with decimal points.
- Stream manipulator `setprecision` specifies the number of digits to the right of the decimal point.
- Stream manipulators `fixed` and `setprecision` remain in effect until they're changed—such settings are called sticky settings.
- The field width specified with `setw` applies only to the next value output.

4.5 do...while Repetition Statement

- Similar to the while statement.
- The do...while statement tests the loop-continuation condition after the loop body executes
- The loop body always executes at least once.
- It's not necessary to use braces in the do...while statement if there is only one statement in the body.
- Most programmers include the braces to avoid confusion between the while and do...while statements.
- Must end a do...while statement with a semicolon.

4.5 do…while Repetition Statement

```cpp
// Fig. 4.7: fig04_07.cpp
// do...while repetition statement.
#include <iostream>
using namespace std;

int main()
{
   int counter = 1; // initialize counter

   do
   {
      cout << counter << " "; // display counter
      counter++; // increment counter
   } while ( counter <= 10 ); // end do...while

   cout << endl; // output a newline
} // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 4.7 | do…while repetition statement.

4.5 do…while Repetition Statement

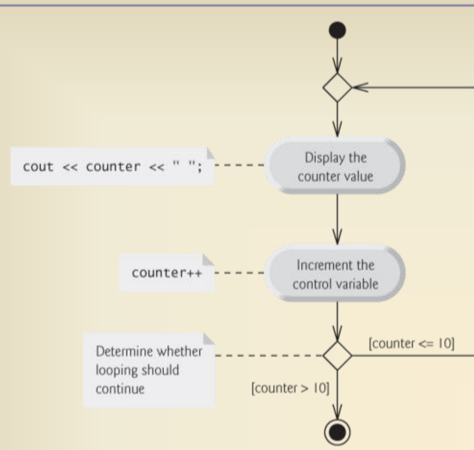


Fig. 4.8 | UML activity diagram for the do…while repetition statement of Fig. 4.7.

4.5 do...while Repetition Statement

◆ while vs. do...while

```
total = 0; grade = 0;
counter = 0;
cout << "Enter grade, ";
cout << "-1 to end: ";
cin >> grade;
while (grade != -1) {
  total = total + grade;
  counter = counter + 1;
  cout << "Enter grade, ";
  cout << "-1 to end: ";
  cin >> grade;
}
```

```
total = 0; grade = 0;
counter = -1;
do {
  total = total + grade;
  counter = counter + 1;
  cout << "Enter grade, ";
  cout << "-1 to end: ";
  cin >> grade;
} while (grade != -1);
```

◆ Duplicate input statements !!

◆ Initial values are all zero.

- No duplicate input statements !!
- Initial counter starts from -1.

4.6 switch Multiple-Selection Statement

◆ The switch multiple-selection statement performs many different actions based on the possible values of a variable or expression.

◆ Each action is associated with the value of a constant integral expression

- i.e., any combination of character and integer constants that evaluates to a constant integer value.

Common Programming Error 4.11

Specifying a nonconstant integral expression in a `switch`*'s* `case` *label is a syntax error.*

4.6 switch Multiple-Selection Statement (cont.)

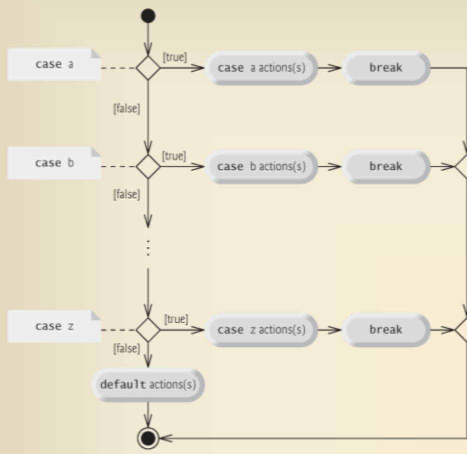


Fig. 4.10 | `switch` multiple-selection statement UML activity diagram with

4.6 switch Multiple-Selection Statement (cont.)

```
// Fig. 4.9: fig04_09.cpp
// Using a switch statement to count A, B, C, D and F grades.
#include <iostream>
using namespace std;

int main()
{
   int grade; // letter grade entered by user
   int aCount; // count of A grades
   int bCount; // count of B grades
   int cCount; // count of C grades
   int dCount; // count of D grades
   int fCount; // count of F grades

   cout << "Enter the letter grades." << endl
      << "Enter the EOF character to end input." << endl;

```

Fig. 4.9 | Using a `switch` statement to count A, B, C, D and F grades. (Part 1 of 5.)

4.6 switch Multiple-Selection Statement (cont.)

```
   // loop until user types end-of-file key sequence
   while ( ( grade = cin.get() ) != EOF )
   {
      // determine which grade was entered
      switch ( grade ) // switch statement nested in while
      {
         case 'A': // grade was uppercase A
         case 'a': // or lowercase a
            aCount++; // increment aCount
            break; // necessary to exit switch

         case 'B': // grade was uppercase B
         case 'b': // or lowercase b
            bCount++; // increment bCount
            break; // exit switch

         case 'C': // grade was uppercase C
         case 'c': // or lowercase c
            cCount++; // increment cCount
            break; // exit switch

```

Fig. 4.9 | Using a `switch` statement to count A, B, C, D and F grades. (Part 2 of 5.)

4.6 switch Multiple-Selection Statement (cont.)

```
         case 'D': // grade was uppercase D
         case 'd': // or lowercase d
            dCount++; // increment dCount
            break; // exit switch

         case 'F': // grade was uppercase F
         case 'f': // or lowercase f
            fCount++; // increment fCount
            break; // exit switch

         case '\n': // ignore newlines,
         case '\t': // tabs,
         case ' ': // and spaces in input
            break; // exit switch

         default: // catch all other characters
            cout << "Incorrect letter grade entered."
               << " Enter a new grade." << endl;
            break; // optional; will exit switch anyway
      } // end switch
   } // end while

```

Fig. 4.9 | Using a `switch` statement to count A, B, C, D and F grades. (Part 3 of 5.)

4.6 switch Multiple-Selection Statement (cont.)

```
    // output summary of results
    cout << "\n\nNumber of students who received each letter grade:"
       << "\nA: " << aCount // display number of A grades
       << "\nB: " << bCount // display number of B grades
       << "\nC: " << cCount // display number of C grades
       << "\nD: " << dCount // display number of D grades
       << "\nF: " << fCount // display number of F grades
       << endl;
} // end function main
```

Fig. 4.9 | Using a `switch` statement to count A, B, C, D and F grades. (Part 4 of 5.)

4.6 switch Multiple-Selection Statement (cont.)

```
Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z

Number of students who received each letter grade:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Fig. 4.9 | Using a `switch` statement to count A, B, C, D and F grades. (Part 5 of 5.)

4.6 switch Multiple-Selection Statement (cont.)

- The `switch` statement consists of a series of case labels and an optional default case.
- The `switch` statement compares the value of the controlling expression with each `case` label.
- If a match occurs, the program executes the statements for that `case`.
- Listing `case`s consecutively with no statements between them enables the `case`s to perform the same set of statements.
- The `break` statement causes program control to proceed with the first statement after the `switch`.

```
switch ( grade ) // switch statement nested in while
{
   case 'A': // grade was uppercase A
   case 'a': // or lowercase a
      aCount++; // increment aCount
      break; // necessary to exit switch
```


4.6 switch Multiple-Selection Statement (cont.)

- Each `case` can have multiple statements.
  - The `switch` selection statement does not require braces around multiple statements in each `case`.
- Without `break` statements, the statements for that `case` and subsequent `case`s are all executed when a match occurs
  - Until a `break` statement or the end of the `switch` is encountered.
  - Referred to as “falling through” to the subsequent `case`s.
- If no match occurs between the controlling expression’s value and a `case` label, the `default` case executes.
- If a `switch` statement does not contain a `default` case, program control continues with the first statement after the `switch` when no match occurs

```
   case '\n': // ignore newlines,
   case '\t': // tabs,
   case ' ': // and spaces in input
      break; // exit switch

   default: // catch all other characters
      cout << "Incorrect letter grade entered."
         << " Enter a new grade." << endl;
      break; // optional; will exit switch anyway
} // end switch
```

4.6 switch Multiple-Selection Statement (cont.)

Common Programming Error 4.8

Forgetting a `break` *statement when one is needed in a* `switch` *statement is a logic error.*

Common Programming Error 4.9

Omitting the space between the word `case` *and the integral value being tested in a* `switch` *statement—e.g., writing* `case3:` *instead of* `case 3:`*—is a logic error. The* `switch` *statement will not perform the appropriate actions when the controlling expression has a value of 3.*

4.6 switch Multiple-Selection Statement (cont.)

Good Programming Practice 4.10

Provide a `default` *case in* `switch` *statements. Cases not explicitly tested in a* `switch` *statement without a* `default` *case are ignored. Including a* `default` *case focuses you on the need to process exceptional conditions. There are situations in which no* `default` *processing is needed. Although the* `case` *clauses and the* `default` *case clause in a* `switch` *statement can occur in any order, it's common practice to place the* `default` *clause last.*

Good Programming Practice 4.11

The last `case` *in a* `switch` *statement does not require a* `break` *statement. Some programmers include this* `break` *for clarity and for symmetry with other cases.*

4.6 switch Multiple-Selection Statement (cont.)

```
while ( ( grade = cin.get() ) != EOF )
```

- The `cin.get()` function reads one character from the keyboard.
- A character is stored as a "number" in the computer according to its ASCII code.
- Normally, characters are stored in variables of type **char**; however, characters can be stored in any integer data type.
- Can treat a character either as an integer or as a character, depending on its use. For example:

```
cout << "The character (" << 'a' << ") has the value "
    << static_cast< int > ( 'a' ) << endl;
```

prints the character `a` and its integer value as follows:

```
The character (a) has the value 97
```


ASCII code

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

4.6 switch Multiple-Selection Statement (cont.)

- `EOF` stands for "end-of-file". Commonly used as a sentinel value for characters.
  - *You cannot type the value –1 as the sentinel value*. (ASCII code is 0~255)
  - Type a system-dependent keystroke combination that means "end-of-file" to indicate that you have no more data to enter.
- `EOF` is a symbolic integer constant defined in the `<iostream>` header file.
  - The `EOF` has type int
- The keystroke combinations for entering *end-of-file* are system dependent.
  - Windows: Ctrl-Z ; UNIX: Ctrl-D

4.6 switch Multiple-Selection Statement (cont.)

- To have the program read the characters, we must send them to the computer by pressing the *Enter key*.
- This places a newline character in the input after the character we wish to process.
  - Often, this newline character must be specially processed.
- The `cin.get()` function can ignore the newline character automatically
  - Some functions can do this, but some functions cannot.

4.7 break and continue Statements

- The break statement, when executed in a `while`, `for`, `do…while` or `switch` statement, causes immediate exit from that statement.
- Program execution continues with the next statement.
- Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch` statement.

4.7 break and continue Statements (cont.)

```
// Fig. 4.11: fig04_11.cpp
// break statement exiting a for statement.
#include <iostream>
using namespace std;

int main()
{
   int count; // control variable also used after loop terminates

   for ( count = 1; count <= 10; count++ ) // loop 10 times
   {
      if ( count == 5 )
         break; // break loop only if x is 5

      cout << count << " ";
   } // end for

   cout << "\nBroke out of loop at count = " << count << endl;
} // end main
```

```
1 2 3 4
Broke out of loop at count = 5
```

Fig. 4.11 | break statement exiting a for statement.

4.7 break and continue Statements (cont.)

- The continue statement skips the remaining statements in its body and proceeds with the next iteration of the loop.
  - When executed in a `while`, `for` or `do…while` statement
- In `while` and `do…while` statements, the loop-continuation test evaluates immediately after the `continue` statement executes.
- In the `for` statement, the increment expression executes, then the loop-continuation test evaluates.

4.7 break and continue Statements (cont.)

```
// Fig. 4.12: fig04_12.cpp
// continue statement terminating an iteration of a for statement.
#include <iostream>
using namespace std;

int main()
{
   for ( int count = 1; count <= 10; count++ ) // loop 10 times
   {
      if ( count == 5 ) // if count is 5,
         continue;       // skip remaining code in loop

      cout << count << " ";
   } // end for

   cout << "\nUsed continue to skip printing 5" << endl;
} // end main
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Fig. 4.12 | `continue` statement terminating a single iteration of a `for` statement.

Appendix Usage of Infinite Loop

- Infinite loops are helpful when the termination condition is generated inside the loop

```
while (1) {
  ......
  ans = a * b;
   if (ans == 0) break;
   ......
}
```

1 (non-zero value) means always TRUE

- Should be used with `break` to terminate the loop
  - Make sure the condition will eventually become TRUE
- If sentinel-controlled loop can be used instead, use it !!
  - Infinite loops are not easy to debug

Appendix Nested Loops

- Nested loops (loop inside a loop) are allowed in C/C++

```
for (i=0; i<n; i++) {
  ......
  for (j=0; j<m; j++)
  {  ......  }
}
```

outer loop: run inner loop for n times

inner loop: repeated actions

- Similar to migrating 1-dimensional problems into multi-dimensional problems
  - One loop: f(0), f(1), f(2), …
  - Two loops: f(0,0), f(0,1), …, f(0,n), f(1,0), f(1,1), …
- The most important thing:
  - Obtain the changing rules of the running index

Nested Loops: Examples (1/3)

- Execute multi-dimensional operations

```
for (i=1; i<=4; i++) {
   cout << "i=" << i << ":\n";
   for (j=1; j<=3; j++) {
      cout<<i<<"x"<<j<<"="<<i*j;
   }
   cout << endl;
}
```

```
i=1:
1x1=1 1x2=2 1x3=3
i=2:
2x1=2 2x2=4 2x3=6
i=3:
3x1=3 3x2=6 3x3=9
i=4:
4x1=4 4x2=8 4x3=12
```

- Please pay special attention to the index changing sequence
  - Column first in this case

    (1,1) -> (1,2) -> (1,3) -> (2,1) -> ...
  - So, column is changed in the inner loop

Nested Loops: Examples (2/3)

- Repeat a loop for n times

```
for (i=1; i<=5; i++) {
   for (j=1; j<=6; j++)
   {  cout << "*";  }
   cout << endl;
}
```

6 stars

```
******
******
******
******
******
```

5 times

- Inner loop control the repeated actions
  - Print 6 stars in this case
- Outer loop control the number of times
  - Print 5 rows of stars in this case

Nested Loops: Examples (3/3)

◆ Inner loop is controlled by outer loop

```
for (i=1; i<=5; i++)
{  for (j=1; j<=i; j++)
   {  printf("*");  }
   printf("\n");
}
```

stars changed with i

```
*
**
***
****
*****
```

5 times

◆ Outer loop control the number of rows
- ➢ Print 5 rows of stars in this case

◆ Inner loop control the number of stars
- ➢ Change its termination condition
- ➢ i=1 --> for (j=1;j<=1;j++) --> 1 star
- ➢ i=2 --> for (j=1;j<=2;j++) --> 2 stars
- ➢ ......

break in Nested Loops

◆ In nested loops, *break/continue* can only affect the most inner loop where the *break/continue* stands

```
for (i=1; i<=5; i++)
{  for (j=1; j<=3; j++)
   {  printf("(%d,%d) ",i,j);
      if (i==3) break;
   }
   printf("\n");
}
```

break inner loop j only

```
(1,1) (1,2) (1,3)
(2,1) (2,2) (2,3)
(3,1)
(4,1) (4,2) (4,3)
(5,1) (5,2) (5,3)
```

Jump out the inner loop and start from here

◆ If *break* is used to skip the following *switch* statements, it has no effects on the outside loop
- ➢ One-time use only

```
/* loop until user types end-of-file key sequence */
while ( ( grade = getchar() ) != EOF ) {

   /* determine which grade was input */
   switch ( grade ) { /* switch nested in while */

      case 'A': /* grade was uppercase A */
      case 'a': /* or lowercase a */
         ++aCount; /* increment aCount */
         break; /* necessary to exit switch */
```

4.8 Logical Operators

- C++ provides logical operators that are used to form more complex conditions by combining simple conditions.
  - && (logical AND)
  - || (logical OR)
  - ! (logical NOT, also called logical negation).

4.8 Logical Operators (cont.)

- The && (logical AND) operator is used to ensure that two conditions are *both* `true`.
- The simple condition to the left of the && operator evaluates first.
- The right side of a logical AND expression is evaluated only if the left side is `true`.
- Figure 4.13 shows all four possible combinations of `false` and `true` values for *expression1* and *expression2*.
  - Such tables are often called truth tables.

4.8 Logical Operators (cont.)

- The || (logical OR) operator determines if either *or both* of two conditions are `true`.
- Figure 4.14 is a truth table for the logical OR operator (||).
- The && operator has a higher precedence than the || operator.
- Both operators associate from left to right.
- An expression containing && or || operators evaluates only until the truth or false hood of the expression is known.

4.8 Logical Operators (cont.)

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 4.13 | && (logical AND) operator truth table.

expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 4.14 | || (logical OR) operator truth table.

4.8 Logical Operators (cont.)

Performance Tip 4.5

In expressions using operator `&&`*, if the separate conditions are independent of one another, make the condition most likely to be* `false` *the leftmost condition. In expressions using operator* `||`*, make the condition most likely to be* `true` *the leftmost condition. This use of short-circuit evaluation can reduce a program's execution time.*

4.8 Logical Operators (cont.)

- C++ provides the ! (logical NOT, also called logical negation) operator to "reverse" a condition's meaning.
- The unary logical negation operator has only a single condition as an operand.
- You can often avoid the ! operator by using an appropriate relational or equality operator.
- Figure 4.15 is a truth table for the logical negation operator (!).

expression	!expression
`false`	`true`
`true`	`false`

Fig. 4.15 | ! (logical negation)

4.8 Logical Operators (cont.)

- Figure 4.16 demonstrates the logical operators by producing their truth tables.
- The output shows each expression that is evaluated and its `bool` result.
- By default, `bool` values `true` and `false` are displayed by `cout` and the stream insertion operator as `1` and `0`, respectively.
- Stream manipulator boolalpha (a sticky manipulator) specifies that the value of each `bool` expression should be displayed as either the word "true" or the word "false."

4.8 Logical Operators (cont.)

```
// Fig. 4.16: fig04_16.cpp
// Logical operators.
#include <iostream>
using namespace std;

int main()
{
   // create truth table for && (logical AND) operator
   cout << boolalpha << "Logical AND (&&)"
      << "\nfalse && false: " << ( false && false )
      << "\nfalse && true: " << ( false && true )
      << "\ntrue && false: " << ( true && false )
      << "\ntrue && true: " << ( true && true ) << "\n\n";

   // create truth table for || (logical OR) operator
   cout << "Logical OR (||)"
      << "\nfalse || false: " << ( false || false )
      << "\nfalse || true: " << ( false || true )
      << "\ntrue || false: " << ( true || false )
      << "\ntrue || true: " << ( true || true ) << "\n\n";

```

Fig. 4.16 | Logical operators. (Part 1 of 2.)

4.8 Logical Operators (cont.)

```
    // create truth table for ! (logical negation) operator
    cout << "Logical NOT (!)"
       << "\n!false: " << ( !false )
       << "\n!true: " << ( !true ) << endl;
 } // end main
```

```
Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Logical NOT (!)
!false: true
!true: false
```

Fig. 4.16 | Logical operators. (Part 2 of 2.)

4.9 Confusing the Equality (==) and Assignment (=) Operators

- Accidentally swapping the operators == (equality) and = (assignment) is not easy to be found.
- The statements with these errors tend to compile correctly but generate incorrect results through runtime logic errors.
  - *Any nonzero value is interpreted as* `true`
- Variable names are said to be ***lvalues*** (for “left values”)
  - Used on the left side of an assignment operator.
- Constants are said to be ***rvalues*** (for “right values”)
  - Used on only the right side of an assignment operator.
- *Lvalues* can also be used as *rvalues*, but not vice versa.

4.9 Confusing the Equality (==) and Assignment (=) Operators

Error-Prevention Tip 4.3

Programmers normally write conditions such as `x == 7` *with the variable name on the left and the constant on the right. By placing the constant on the left, as in* `7 == x`*, you'll be protected by the compiler if you accidentally replace the* `==` *operator with* `=`*. The compiler treats this as a compilation error, because you can't change the value of a constant. This will prevent the potential devastation of a runtime logic error.*

Error-Prevention Tip 4.4

Use your text editor to search for all occurrences of `=` *in your program and check that you have the correct assignment operator or logical operator in each place.*

4.10 Structured Programming Summary

- The structured programs are easier than unstructured programs to understand, test, debug, modify, and even prove correct in a mathematical sense.
- Figure 4.20 uses activity diagrams to summarize C++'s control statements.
- The initial and final states indicate the single entry point and the single exit point of each control statement.

4.10 Structured Programming Summary

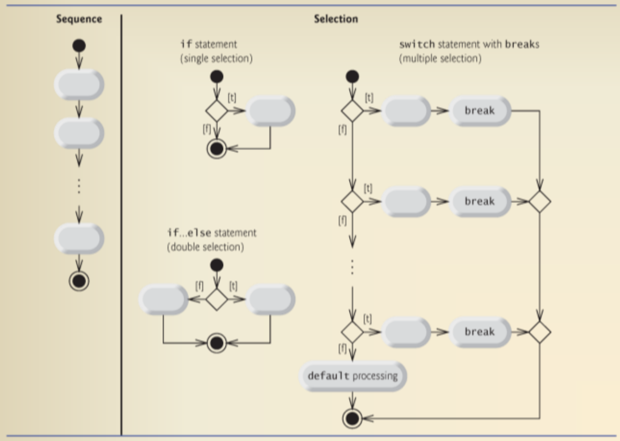


Fig. 4.18 | C++'s single-entry/single-exit sequence, selection and repetition

4.10 Structured Programming Summary

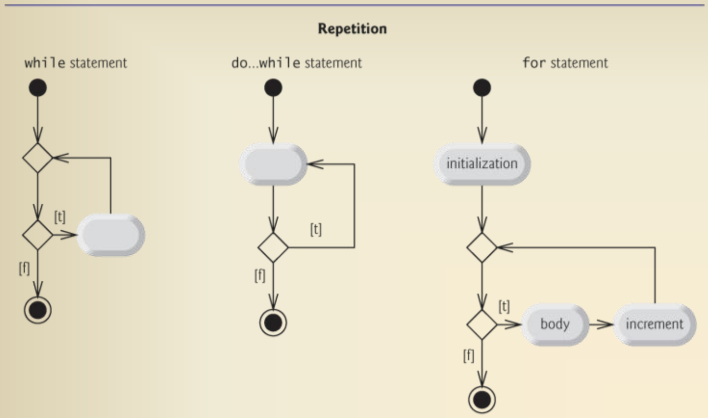


Fig. 4.18 | C++'s single-entry/single-exit sequence, selection and repetition statements. (Part 2 of 2.)

4.10 Structured Programming Summary (cont.)

- Figure 4.21 shows the rules for forming structured programs.
- The rules assume that action states may be used to indicate any action.
  - Rule 2 is the stacking rule; Rule 3 is the nesting rule.

Rules for forming structured programs	
1)	Begin with the "simplest activity diagram" (Fig. 4.20).
2)	Any action state can be replaced by two action states in sequence.
3)	Any action state can be replaced by any control statement (sequence, `if`, `if...else`, `switch`, `while`, `do...while` or `for`).
4)	Rules 2 and 3 can be applied as often as you like and in any order.

Fig. 4.19 | Rules for forming structured programs.

4.10 Structured Programming Summary (cont.)

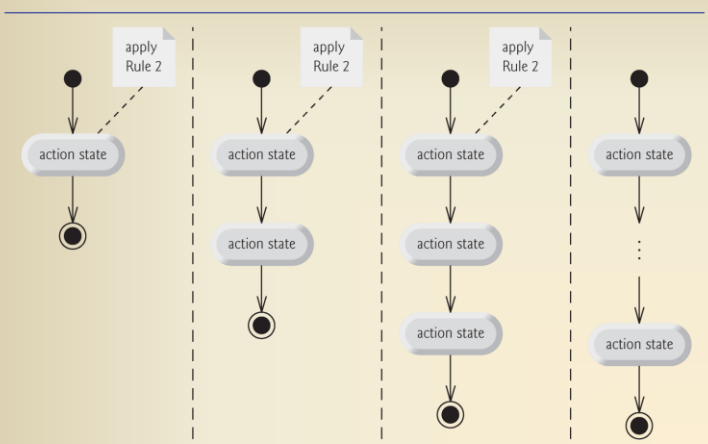


Fig. 4.21 | Repeatedly applying Rule 2 of Fig. 4.19 to the simplest activity diagram.

4.10 Structured Programming Summary (cont.)

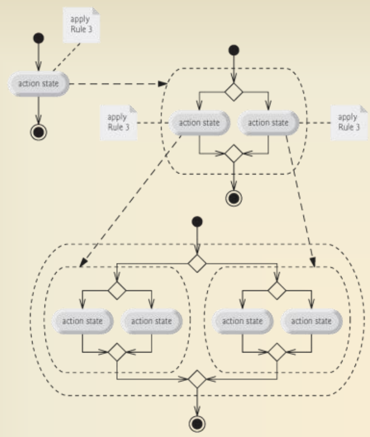


Fig. 4.22 | Applying Rule 3 of Fig. 4.19 to the simplest activity diagram several times.

4.10 Structured Programming Summary (cont.)

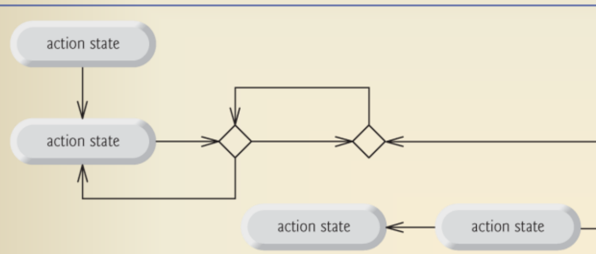


Fig. 4.23 | Activity diagram with illegal syntax.

Summary

- for loop
- do…while loop
- switch
- break and continue
- Nested Loops
- logical operators to form more complex conditions

Andy, Yu-Guang Chen
Assistant Professor, Department of EE, NCU
Email: andyygchen@ee.ncu.edu.tw