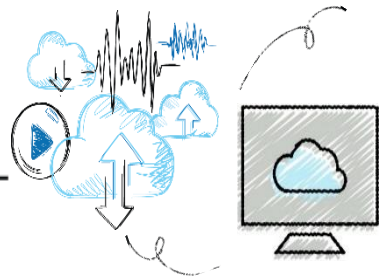
Lecture Notes on Feb/07/2023

Chapters 4 and 5

ECE 111: Introduction to C and C++ Programming

Instructor: Dr. Shayan (Sean) Taheri

Gannon University (GU)



Control Structures (2 of 2)

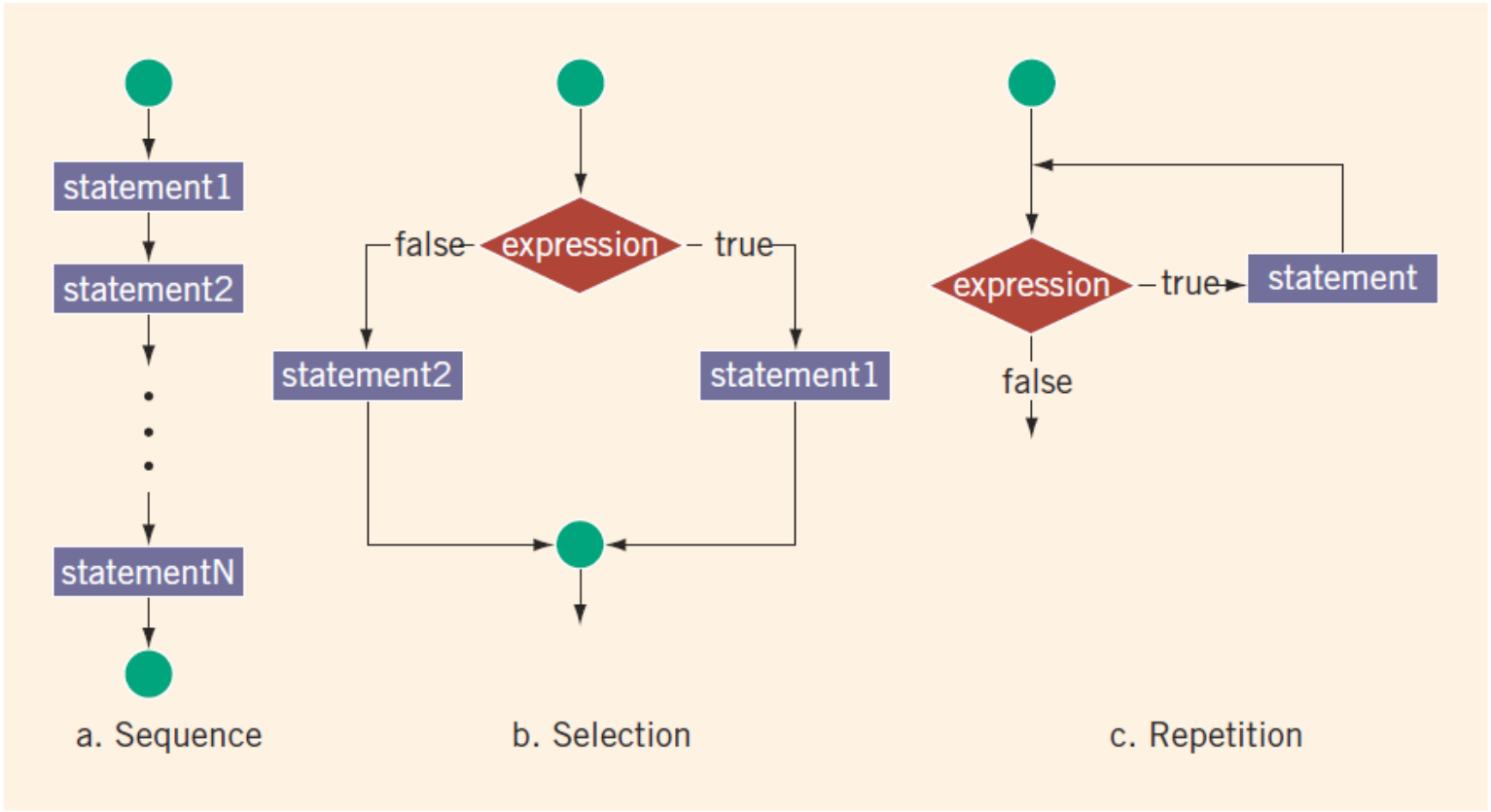


FIGURE 4-1 Flow of execution

Selection: `if` and `if...else`

- An expression that evaluates to **`true`** or **`false`** is called a logical expression
  - "**`8 is greater than 3`**" is true

CENGAGE Learning

Relational Operators

TABLE 4-1 Relational Operators in C++

Operator	Description
==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

- Each relational operator is a binary operator (requires two operands)
- Expressions using these operators always evaluate to `true` or `false`

Relational Operators and Simple Data Types

- You can use the relational operators with all three simple data types

EXAMPLE 4-1

Expression	Meaning	Value
`8 < 15`	`8` is less than `15`	`true`
`6 != 6`	`6` is not equal to `6`	`false`
`2.5 > 5.8`	`2.5` is greater than `5.8`	`false`
`5.9 <= 7.5`	`5.9` is less than or equal to `7.5`	`true`
`7 <= 10.4`	`7` is less than or equal to `10.4`	`true`

Comparing Characters

- In an expression of **`char`** values using relational operators:
  - The result depends on the machine's collating sequence
    - ASCII character set
- Logical (Boolean) expressions:
  - Include expressions such as **`4 < 6`** and **`'R' > 'T'`**
  - Return an integer value of **1** if the logical expression evaluates to **`true`**
  - Return an integer value of **0** otherwise

CENGAGE Learning

One-Way Selection (1 of 2)

- One-way selection syntax

```
if (expression)
    statement
```

- The statement is:
  - Executed if the value of the expression is **`true`**
  - Bypassed if the value is **`false`**; program goes to the next statement
- The **`expression`** is also called a decision maker
- The statement following the **`expression`** is also called the action statement

One-Way Selection (2 of 2)

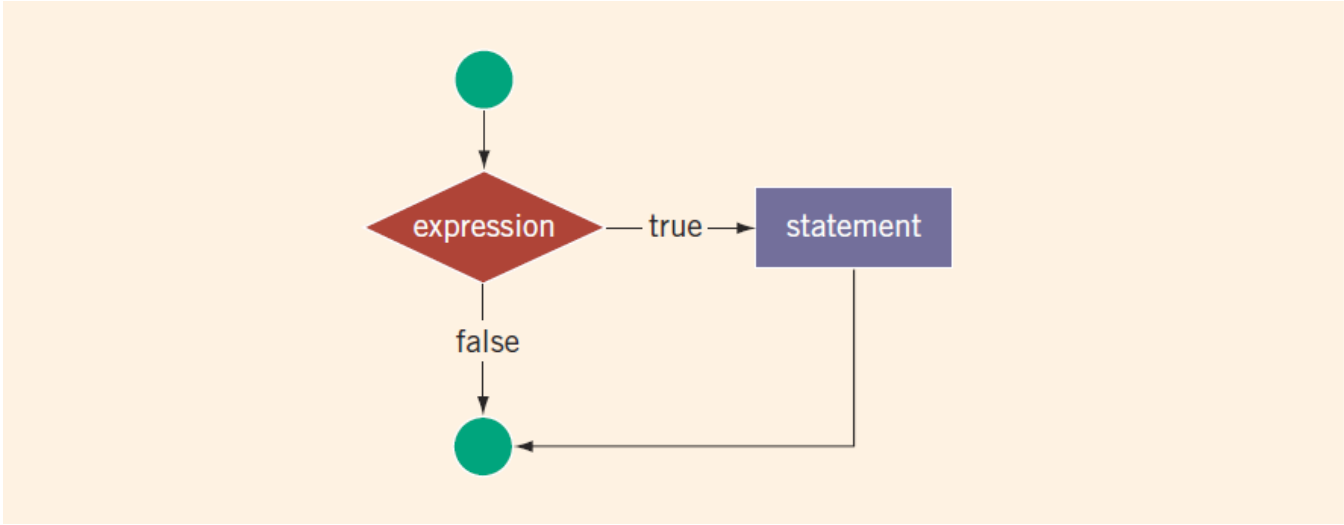


FIGURE 4-2 One-way selection

Two-Way Selection (1 of 2)

- Two-way selection syntax

```
if (expression)
     statement1
else
     statement2
```

- If **`expression`** is **`true`**, **`statement1`** is executed; otherwise, **`statement2`** is executed
  - **`statement1`** and **`statement2`** are any C++ statements

Two-Way Selection (2 of 2)

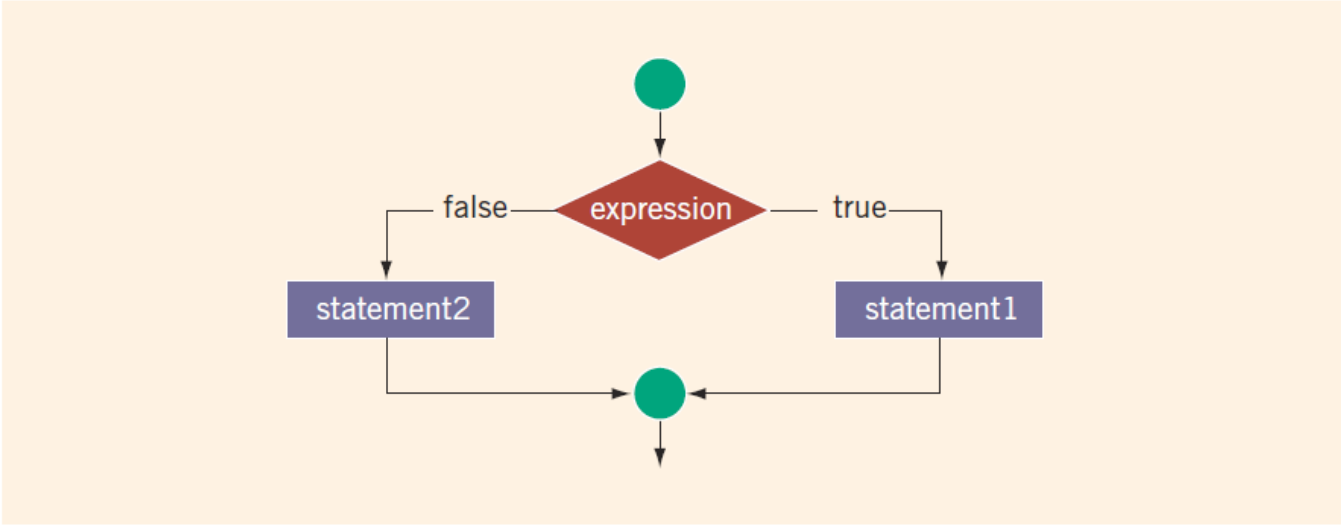


FIGURE 4-3 Two-way selection

`int` Data Type and Logical (Boolean) Expressions

- Earlier versions of C++ did not provide built-in data types that had Boolean values
- Logical expressions evaluate to either **1** or **0**
  - Logical expression value was stored in a variable of the data type **`int`**
- You can use the **`int`** data type to manipulate logical (Boolean) expressions

`bool` Data Type and Logical (Boolean) Expressions

- The data type **`bool`** has logical (Boolean) values **`true`** and **`false`**
- **`bool`**, **`true`**, and **`false`** are reserved words
- The identifier **`true`** has the value **1**
- The identifier **`false`** has the value **0**

Logical (Boolean) Operators and Logical Expressions (1 of 5)

- Logical (Boolean) operators enable you to combine logical expressions

TABLE 4-2 Logical (Boolean) Operators in C++

Operator	Description
!	not
&&	and
||	or

Logical (Boolean) Operators and Logical Expressions (2 of 5)

TABLE 4-3 The ! (Not) Operator

Expression	!(Expression)
`true` (nonzero)	`false` (0)
`false` (0)	`true` (1)

EXAMPLE 4-10

Expression	Value	Explanation
`!('A' > 'B')`	`true`	Because `'A' > 'B'` is `false`, `!('A' > 'B')` is `true`.
`!(6 <= 7)`	`false`	Because `6 <= 7` is `true`, `!(6 <= 7)` is `false`.

Logical (Boolean) Operators and Logical Expressions (3 of 5)

TABLE 4-4 The `&&` (And) Operator

Expression1	Expression2	Expression1 && Expression2
`true` (nonzero)	`true` (nonzero)	`true` (1)
`true` (nonzero)	`false` (0)	`false` (0)
`false` (0)	`true` (nonzero)	`false` (0)
`false` (0)	`false` (0)	`false` (0)

EXAMPLE 4-11

Expression	Value	Explanation
`(14 >= 5) && ('A' < 'B')`	`true`	Because `(14 >= 5)` is `true`, `('A' < 'B')` is `true`, and `true && true` is `true`, the expression evaluates to `true`.
`(24 >= 35) && ('A' < 'B')`	`false`	Because `(24 >= 35)` is `false`, `('A' < 'B')` is `true`, and `false && true` is `false`, the expression evaluates to `false`.

Logical (Boolean) Operators and Logical Expressions (4 of 5)

TABLE 4-5 The || (Or) Operator

Expression1	Expression2	Expression1 || Expression2
`true` (nonzero)	`true` (nonzero)	`true` (1)
`true` (nonzero)	`false` (0)	`true` (1)
`false` (0)	`true` (nonzero)	`true` (1)
`false` (0)	`false` (0)	`false` (0)

Logical (Boolean) Operators and Logical Expressions (5 of 5)

EXAMPLE 4-12

Expression	Value	Explanation
`(14 >= 5) || ('A' > 'B')`	`true`	Because `(14 >= 5)` is `true`, `('A' > 'B')` is `false`, and `true || false` is `true`, the expression evaluates to `true`.
`(24 >= 35) || ('A' > 'B')`	`false`	Because `(24 >= 35)` is `false`, `('A' > 'B')` is `false`, and `false || false` is `false`, the expression evaluates to `false`.
`('A' <= 'a') || (7 != 7)`	`true`	Because `('A' <= 'a')` is `true`, `(7 != 7)` is `false`, and `true || false` is `true`, the expression evaluates to `true`.

Order of Precedence (1 of 5)

- Relational and logical operators are evaluated from left to right
  - The associativity is left to right
- Parentheses can override precedence

Order of Precedence (2 of 5)

TABLE 4-6 Precedence of Operators

Operators	Precedence
!, +, – (unary operators)	first
*, /, %	second
+, –	third
<, <=, >=, >	fourth
==, !=	fifth
&&	sixth
||	seventh
= (assignment operator)	last

Order of Precedence (3 of 5)

EXAMPLE 4-13

Suppose you have the following declarations:

```
bool found = true;
int age = 20;
double hours = 45.30;
double overTime = 15.00;
int count = 20;
char ch = 'B';
```

Order of Precedence (4 of 5)

Expression	Value / Explanation
`!found`	`false` Because `found` is `true`, `!found` is `false`.
`hours > 40.00`	`true` Because `hours` is `45.30` and `45.30 > 40.00` is `true`, the expression `hours > 40.00` evaluates to `true`.
`!age`	`false` `age` is `20`, which is nonzero, so `age` evaluates to `true`. Therefore `!age` is `false`.
`!found && (age >= 18)`	`false` `!found` is `false`; `age > 18` is `20 > 18` is `true`. Therefore, `!found && (age >= 18)` is `false && true`, which evaluates to `false`.
`!(found && (age >= 18))`	`false` Now, `found && (age >= 18)` is `true && true`, which evaluates to `true`. Therefore, `!(found && (age >= 18))` is `!true`, which evaluates to `false`.

Order of Precedence (5 of 5)

`hours + overTime <= 75.00`	`true` Because `hours + overTime` is `45.30 + 15.00 = 60.30` and `60.30 <= 75.00` is `true`, it follows that `hours + overTime <= 75.00` evaluates to `true`.
`(count >= 0) &&` `(count <= 100)`	`true` Now `count` is `20`. Because `20 >= 0` is `true`, `count >= 0` is `true`. Also, `20 <= 100` is `true`, so `count <= 100` is `true`. Therefore, `(count >= 0) && (count <= 100)` is `true && true`, which evaluates to `true`.
`('A' <= ch && ch <= 'Z')`	`true` Here, `ch` is `'B'`. Because `'A' <= 'B'` is `true`, `'A' <= ch` evaluates to `true`. Also, because `'B' <= 'Z'` is `true`, `ch <= 'Z'` evaluates to `true`. Therefore, `('A' <= ch && ch <= 'Z')` is `true && true`, which evaluates to `true`.

Relational Operators and the `string` Type (1 of 5)

- Relational operators can be applied to variables of type **`string`**
  - Strings are compared character by character, starting with the first character
  - Comparison continues until either a mismatch is found or all characters are found equal
  - If two strings of different lengths are compared and the comparison is equal to the last character of the shorter string
    - The shorter string is less than the larger string

Relational Operators and the `string` Type (2 of 5)

EXAMPLE 4-13

Suppose you have the following declarations:

```
bool found = true;
int age = 20;
double hours = 45.30;
double overTime = 15.00;
int count = 20;
char ch = 'B';
```

Relational Operators and the `string` Type (3 of 5)

Expression	Value/Explanation
`str1 < str2`	`true` `str1 = "Hello"` and `str2 = "Hi".` The first character of `str1` and `str2` are the same, but the second character `'e'` of `str1` is less than the second character `'i'` of `str2`. Therefore, `str1 < str2` is `true`.
`str1 > "Hen"`	`false` `str1 = "Hello".` The first two characters of `str1` and `"Hen"` are the same, but the third character `'l'` of `str1` is less than the third `character 'n'` of `"Hen"`. Therefore, `str1 > "Hen"` is `false`.
`str3 < "An"`	`true` `str3 = "Air".` The first characters of `str3` and `"An"` are the same, but the second character `'i'` of `"Air"` is less than the second character `'n'` of `"An".` Therefore, `str3 < "An"` is `true`.

Relational Operators and the `string` Type (4 of 5)

Expression	Value/Explanation
`str1 == "hello"`	`false` `str1 = "Hello"`. The first character `'H'` of `str1` is less than the first character `'h'` of `"hello"` because the ASCII value of `'H'` is 72, and the ASCII value of `'h'` is 104. Therefore, `str1 == "hello"` is `false`.
`str3 <= str4`	`true` `str3 = "Air"` and `str4 = "Bill".` The first character `'A'` of `str3` is less than the first character `'B'` of `str4`. Therefore, `str3 <= str4` is `true`.
`str2 > str4`	`true` `str2 = "Hi"` and `str4 = "Bill".` The first character `'H'` of `str2` is greater than the first character `'B'` of `str4`. Therefore, `str2 > str4` is `true`.

Relational Operators and the `string` Type (5 of 5)

Expression	Value/Explanation
`str4 >= "Billy"`	`false` `str4 = "Bill"`. It has four characters and `"Billy"` has five characters. Therefore, `str4` is the shorter string. All four characters of `str4` are the same as the corresponding first four characters of `"Billy"`, and `"Billy"` is the larger string. Therefore, `str4 >= "Billy"` is `false`.
`str5 <= "Bigger"`	`true` `str5 = "Big"`. It has three characters and `"Bigger"` has six characters. Therefore, `str5` is the shorter string. All three characters of `str5` are the same as the corresponding first three characters of `"Bigger"`, and `"Bigger"` is the larger string. Therefore, `str5 <= "Bigger"` is `true`.

CENGAGE Learning

Compound (Block of) Statements (1 of 2)

- A compound statement (block of statements) has this form:

```
{
  statement_1
  statement_2
    .
    .
    .
  statement_n
}
```

- A compound statement functions like a single statement

CENGAGE Learning

Compound (Block of) Statements (2 of 2)

```
if (age > 18)
{
    cout << "Eligible to vote." << endl;
    cout << "No longer a minor." << endl;
}
else
{
    cout << "Not eligible to vote." << endl;
    cout << "Still a minor." << endl;
}
```

CENGAGE Learning

Multiple Selections: Nested `if` (1 of 2)

- When one control statement is located within another, it is said to be nested
- An **`else`** is associated with the most recent **`if`** that has not been paired with an **`else`**

Multiple Selections: Nested `if` (2 of 2)

EXAMPLE 4-17

Assume that `score` is a variable of type `int`. Based on the value of `score`, the following code outputs the grade:

```
if (score >= 90)
    cout << "The grade is A." << endl;
else if (score >= 80)
    cout << "The grade is B." << endl;
else if (score >= 70)
    cout << "The grade is C." << endl;
else if (score >= 60)
    cout << "The grade is D." << endl;
else
    cout << "The grade is F." << endl;
```

Comparing `if…else` Statements with a Series of `if` Statements (1 of 2)

a.
```
if (month == 1)                          //Line 1
    cout << "January" << endl;           //Line 2
else if (month == 2)                     //Line 3
    cout << "February" << endl;          //Line 4
else if (month == 3)                     //Line 5
    cout << "March" << endl;             //Line 6
else if (month == 4)                     //Line 7
    cout << "April" << endl;             //Line 8
else if (month == 5)                     //Line 9
    cout << "May" << endl;               //Line 10
else if (month == 6)                     //Line 11
    cout << "June" << endl;              //Line 12
```

Comparing `if…else` Statements with a Series of `if` Statements (2 of 2)

b.
```
if (month == 1)
    cout << "January" << endl;
if (month == 2)
    cout << "February" << endl;
if (month == 3)
    cout << "March" << endl;
if (month == 4)
    cout << "April" << endl;
if (month = 5)
    cout << "May" << endl;
if (month == 6)
    cout << "June" << endl;
```

Short-Circuit Evaluation

- Short-circuit evaluation: evaluation of a logical expression stops as soon as the value of the expression is known

EXAMPLE 4-21

Consider the following expressions:

```
(age >= 21) || ( x == 5)          //Line 1
(grade == 'A') && (x >= 7)        //Line 2
```

Comparing Floating-Point Numbers for Equality: A Precaution

- Comparison of floating-point numbers for equality may not behave as you would expect
  - Example:

    `1.0 == 3.0/7.0 + 2.0/7.0 + 2.0/7.0` evaluates to `false`

    Why? `3.0/7.0 + 2.0/7.0 + 2.0/7.0 = 0.9999999999999989`
- A solution is checking for a tolerance value
  - Example: `if fabs(x - y) < 0.000001`

Associativity of Relational Operators: A Precaution (1 of 2)

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Enter an integer: ";
    cin >> num;
    cout << endl;
    if (0 <= num <= 10)
    cout << num << " is within 0 and 10." << endl;
    else
        cout << num << " is not within 0 and 10." <<
    endl;
    return 0;
}
```

CENGAGE Learning

Associativity of Relational Operators: A Precaution (2 of 2)

- `num = 5`

`0 <= num <= 10`	`= 0 <= 5 <= 10`	
	`= (0 <= 5) <= 10`	(Because relational operators are evaluated from left to right)
	`= 1 <= 10`	(Because `0 <= 5` is `true`, `0 <= 5` evaluates to `1`)
	`= 1 (true)`	

- `num = 20`

`0 <= num <= 10`	`= 0 <= 20 <= 10`	
	`= (0 <= 20) <= 10`	(Because relational operators are evaluated from left to right)
	`= 1 <= 10`	(Because `0 <= 20` is `true`, `0 <= 20` evaluates to `1`)
	`= 1 (true)`	

CENGAGE Learning

Avoiding Bugs by Avoiding Partially Understood Concepts and Techniques

- Must use concepts and techniques correctly
  - Otherwise solution will be either incorrect or deficient
- If you do not understand a concept or technique completely
  - Do not use it
  - Save yourself an enormous amount of debugging time

Input Failure and the `if` Statement

- If an input stream enters a fail state:
  - All subsequent input statements associated with that stream are ignored
  - Program continues to execute
  - The code may produce erroneous results
- Use **`if`** statements to check status of input stream
- If the input stream enters the fail state, include instructions that stop program execution

Confusion Between the Equality (==) and Assignment (=) Operators

- C++ allows you to use any expression that can be evaluated to either `true` or `false` as an expression in the `if` statement

```
if (x = 5)
    cout << "The value is five." << endl;
```

- The appearance of = in place of == resembles a *silent killer*
  - It is not a syntax error
  - It is a logical error

Conditional Operator (`?:`)

- Conditional operator (`?:`)
  - Ternary operator: takes three arguments
- Syntax for the conditional operator

```
expression1 ? expression2 : expression3
```

- If **`expression1`** is **`true`**, the result of the conditional expression is **`expression2`**
  - Otherwise, the result is **`expression3`**
- Example: **`max = (a >= b) ? a : b;`**

Program Style and Form (Revisited): Indentation

- A properly indented program:
  - Helps you spot and fix errors quickly
  - Shows the natural grouping of statements
- Insert a blank line between statements that are naturally separate
- Two commonly used styles for placing braces
  - On a line by themselves
  - Or left brace is placed after the expression, and the right brace is on a line by itself

Using Pseudocode to Develop, Test, and Debug a Program

- Pseudocode (or just pseudo) is an informal mixture of C++ and ordinary language
  - Helps you quickly develop the correct structure of the program and avoid making common errors
- Use a wide range of values in a walk-through to evaluate the program

switch Structures (1 of 4)

- switch structure is an alternate to **if-else**
- **switch** (integral) expression is evaluated first
- Value of the expression determines which corresponding action is taken
- Expression is sometimes called the selector

```
switch (expression)
{
case value1:
    statements1
    break;
case value2:
    statements2
    break;
    .
    .
    .
case valuen:
    statementsn
    break;
default:
    statements
}
```

CENGAGE Learning

`switch` Structures (2 of 4)

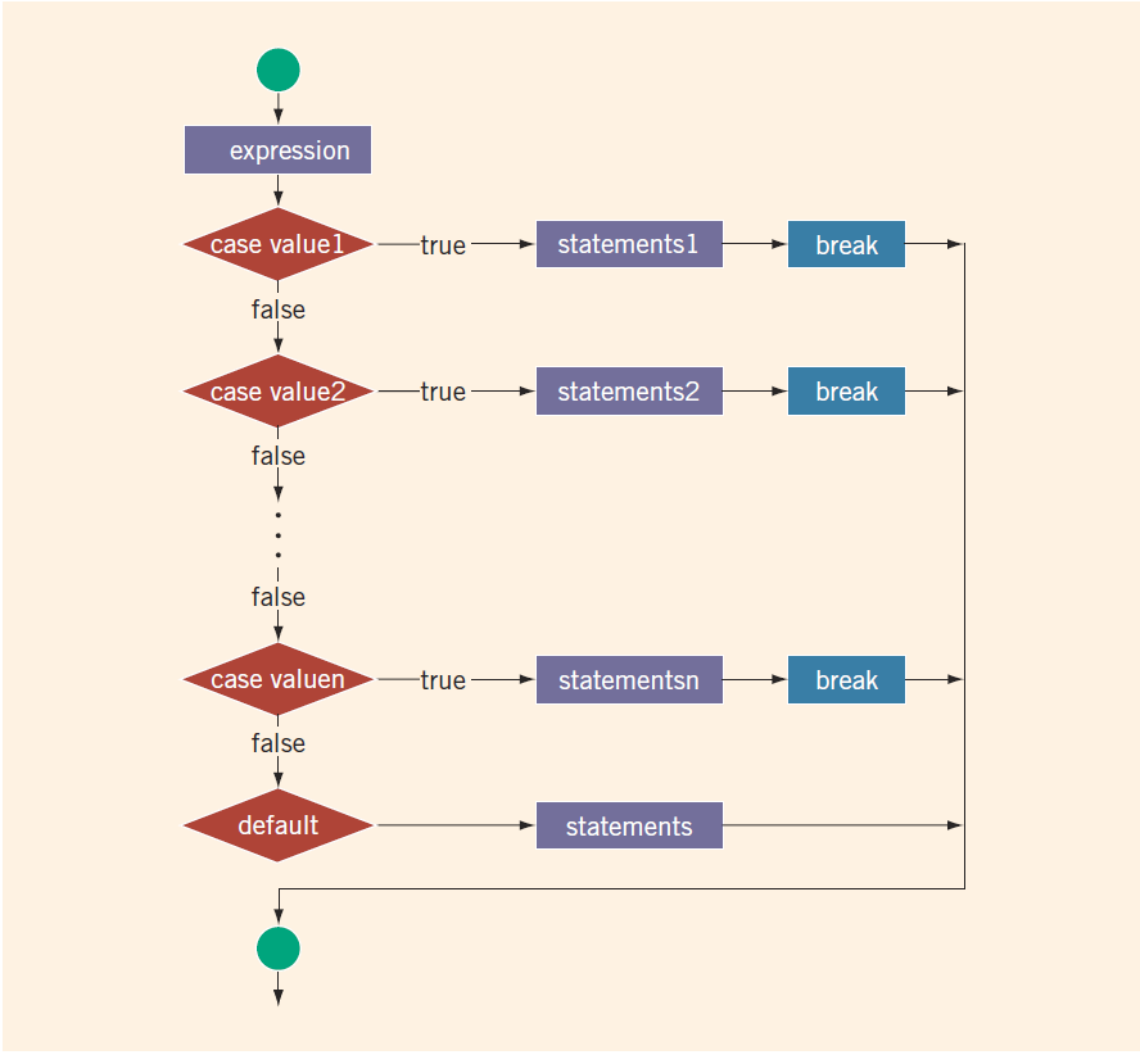


FIGURE 4-4 `switch` statement

`switch` Structures (3 of 4)

- One or more statements may follow a **`case`** label
- Braces are not needed to turn multiple statements into a single compound statement
- When a **`case`** value is matched, all statements after it execute until a break is encountered
- The **`break`** statement may or may not appear after each statement
- **`switch`**, **`case`**, **`break`**, and **`default`** are reserved words

switch Structures (4 of 4)

EXAMPLE 4-22

Consider the following statements, in which grade is a variable of type char:

```
switch (grade)
{
case 'A':
     cout << "The grade point is 4.0.";
     break;
case 'B':
     cout << "The grade point is 3.0.";
     break;
case 'C':
     cout << "The grade point is 2.0.";
     break;
case 'D':
     cout << "The grade point is 1.0.";
     break;
case 'F':
     cout << "The grade point is 0.0.";
     break;
default:
     cout << "The grade is invalid.";
}
```

Avoiding Bugs: Revisited

- To output results correctly
  - Consider whether the **`switch`** structure must include a **`break`** statement after each **`cout`** statement

Terminating a Program with the `assert` Function

- Certain types of errors are very difficult to catch
  - Example: division by zero
- The **`assert`** function is useful in stopping program execution when certain elusive errors occur

The `assert` Function (1 of 2)

- Syntax

```
assert(expression);
```

  - **`expression`** is any logical expression
- If **`expression`** evaluates to **`true`**, the next statement executes
- If **`expression`** evaluates to **`false`**, the program terminates and indicates where in the program the error occurred
- To use **`assert`**, include **`cassert`** header file

The `assert` Function (2 of 2)

- **`assert`** is useful for enforcing programming constraints during program development
- After developing and testing a program, remove or disable **`assert`** statements
- The preprocessor directive **`#define NDEBUG`** must be placed before the directive **`#include <cassert>`** to disable the **`assert`** statement

CENGAGE Learning

Reading Assignment – Very Important for "GU – ECE 111"

- Malik, D.S., 2014. **C++ programming: Program design including data structures**. Cengage Learning.
  - "**Chapter 4**: **Control Structures I (Selection)**".

CENGAGE Learning

Objectives (1 of 2)

- In this chapter, you will:
  - Learn about repetition (looping) control structures
  - Learn how to use a **`while`** loop in a program
  - Explore how to construct and use counter-controlled, sentinel-controlled, flag-controlled, and EOF-controlled repetition structures
  - Learn how to use a **`for`** loop in a program
  - Learn how to use a **`do`**…**`while`** loop in a program

Objectives (2 of 2)

- Examine **`break`** and **`continue`** statements
- Discover how to form and use nested control structures
- Learn how to avoid bugs by avoiding patches
- Learn how to debug loops

CENGAGE Learning