SWE20004
Technical Software Development

Lecture 3
Assignment, Formatting and Selection Structures

SWINBURNE UNIVERSITY OF TECHNOLOGY

Outline

- Assignment operations
- Formatting numbers for program output
- Using mathematical library functions
- Selection criteria
- The **if-else** statement
- Nested **if** statements
- The **switch** statement
- Program testing
- Common programming errors

Assignment

- The assignment operator returns a value.

  e.g. in this statement:

  ```
  c = 25.4*12;
  ```

  - 304.8 is returned to whatever function called the assignment operation.
  - e.g.

  ```
  cout << c = 25.4*12;
  //displays 304.8 on the screen
  ```

Assignment (continued)

- Therefore:

```
 a = b = c = 25.4*12;
```

  304.8 is calculated, then

  c is set to 304.8

  b is set to 304.8 (or whatever c was set to)

  a is set to 304.8 (or whatever b was set to)

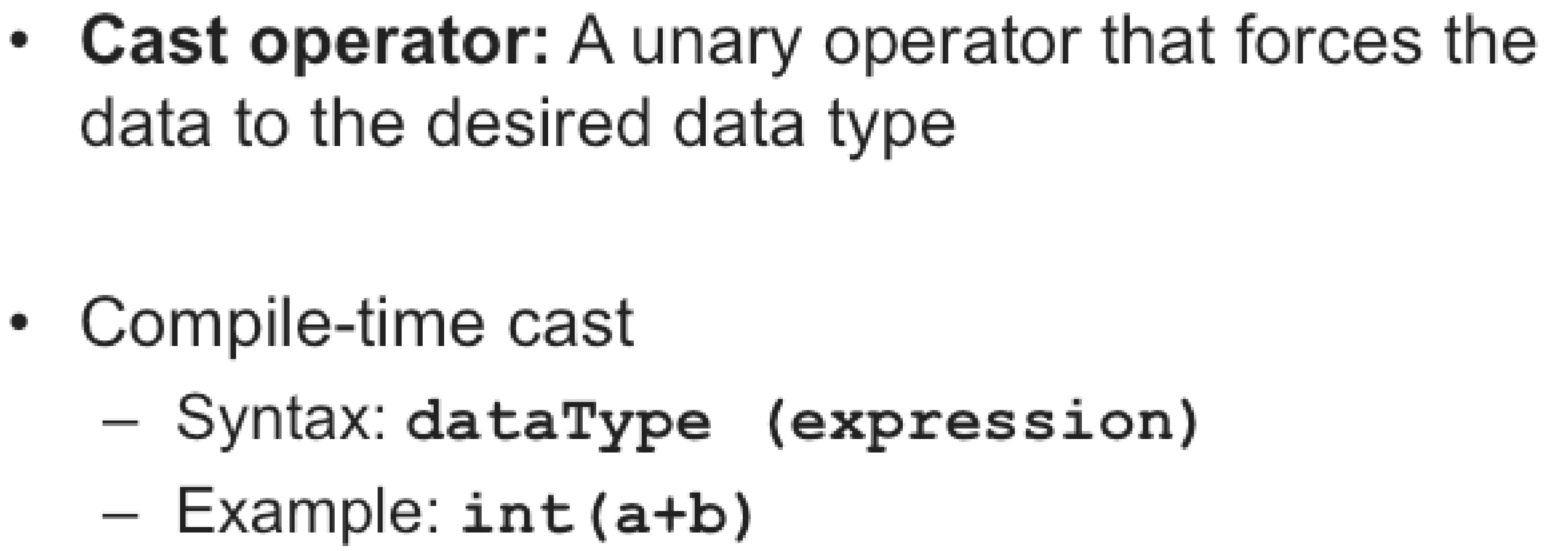
- Useful for initialising a bunch of variables:

```
x = y = x = 0.0;
```

Type mismatch

- Ideally, the Lvalue (left side) and the Rvalue (right side) are the same type.
  - both integers, both doubles, both chars
- If they are different types but the RValue can be correctly represented in the LValue, there are 3 options:
  - Coercion - C++ casts the RValue to the LValue type
  - Casting - the programmer adds code which converts the RValue to the LValue type
  - The compiler complains and the program breaks.

Assignment Operations (continued)



- **Cast operator:** A unary operator that forces the data to the desired data type

- Compile-time cast
  - Syntax: `dataType (expression)`
  - Example: `int(a+b)`

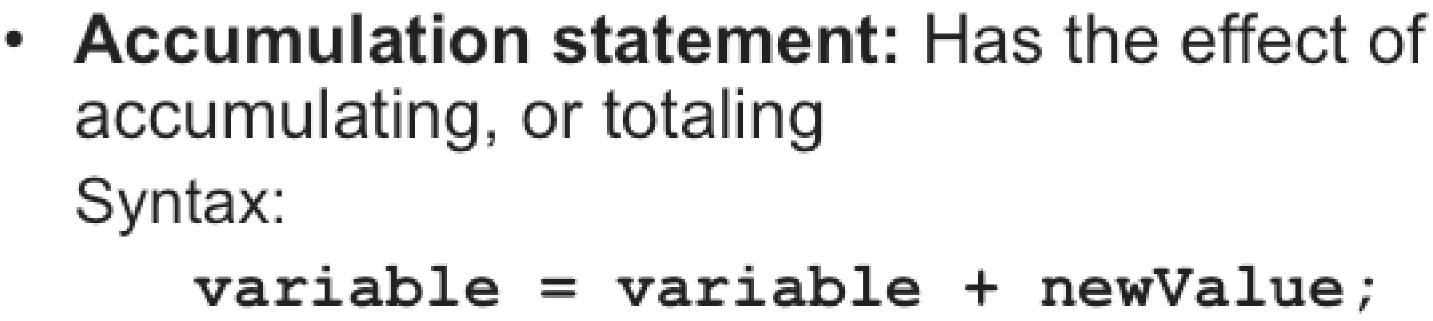
Assignment Operations (continued)

- **Run-time cast:** The requested conversion is checked at run time and applied if valid
  - Syntax:

    ```
    static_cast<data-type> (expression)
    ```
  - Example:

    ```
    static_cast<int>(a*b)
    ```

Assignment Operations (continued)

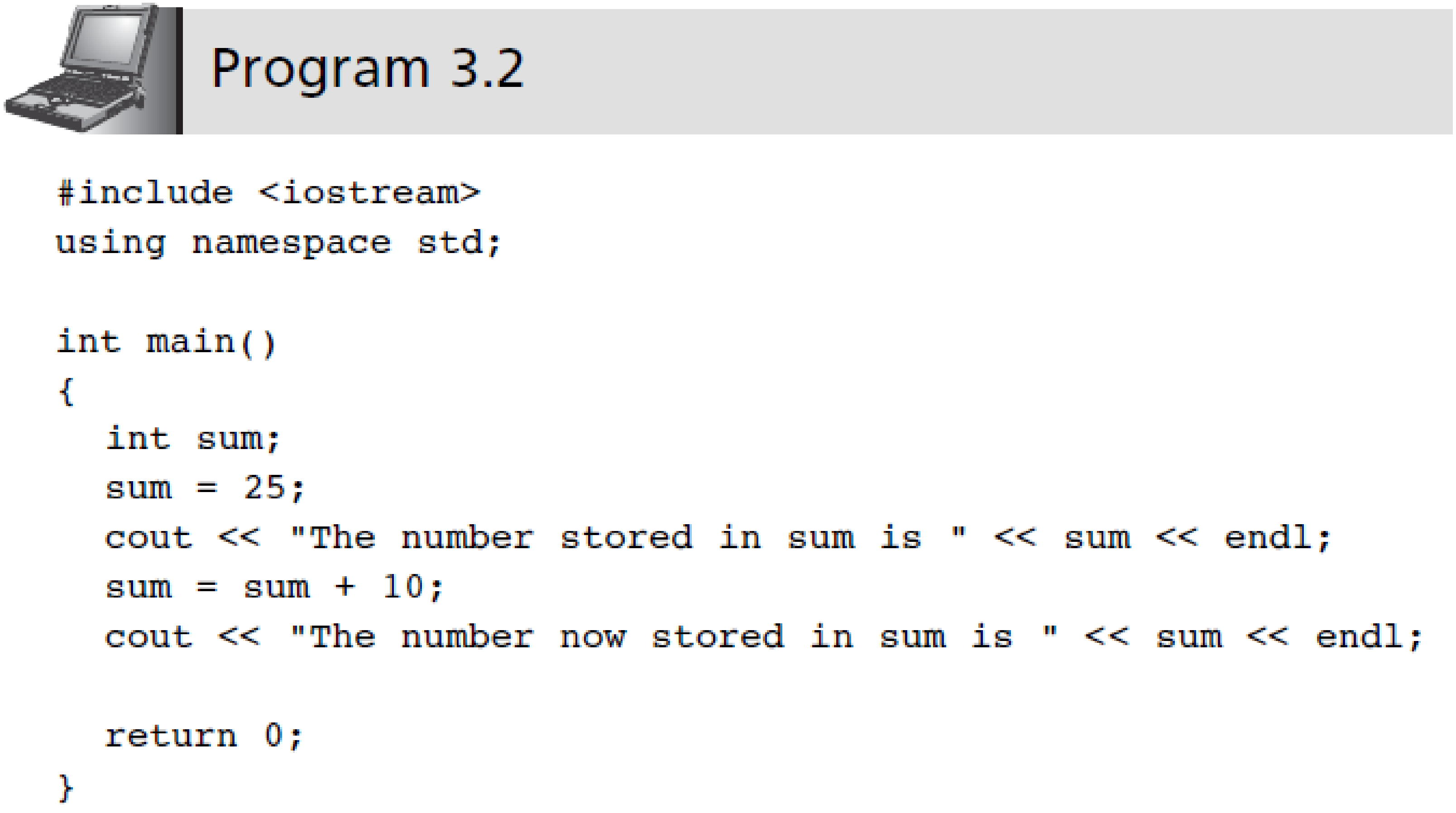


- **Accumulation statement:** Has the effect of accumulating, or totaling

  Syntax:

  ```
      variable = variable + newValue;
  ```

Assignment Operations (continued)



Program 3.2

```
#include <iostream>
using namespace std;

int main()
{
   int sum;
   sum = 25;
   cout << "The number stored in sum is " << sum << endl;
   sum = sum + 10;
   cout << "The number now stored in sum is " << sum << endl;

   return 0;
}
```

Assignment Operations (continued)

- Additional assignment operators provide short cuts: `+=`, `-=`, `*=`, `/=`, `%=`

  Example:

  ```
  sum = sum + 10;
  ```

  is equivalent to: `sum += 10;`

  ```
  price *= rate +1;
  ```

  is equivalent to:

  ```
  price = price * (rate + 1);
  ```

Assignment Operations (continued)

- **Counting statement:** Adds a fixed value to the variable's current value

  Syntax:

  ```
  variable = variable + fixedNumber;
  ```

  Example:

  ```
  i = i + 1;
  count = count + 1;
  ```

Assignment Operations (continued)

- **Increment operator ++:** Unary operator for the special case when a variable is increased by 1
- **Prefix increment operator** appears before the variable
  - Example: `++i`
- **Postfix increment operator** appears after the variable
  - Example: `i++`

Assignment Operations (continued)

- Example: `k = ++n;   //prefix increment`

  is equivalent to:

  ```
  n = n + 1;   //increment n first
   k = n;          //assign n’s value to k
  ```

- Example: `k = n++;   //postfix increment`

  is equivalent to

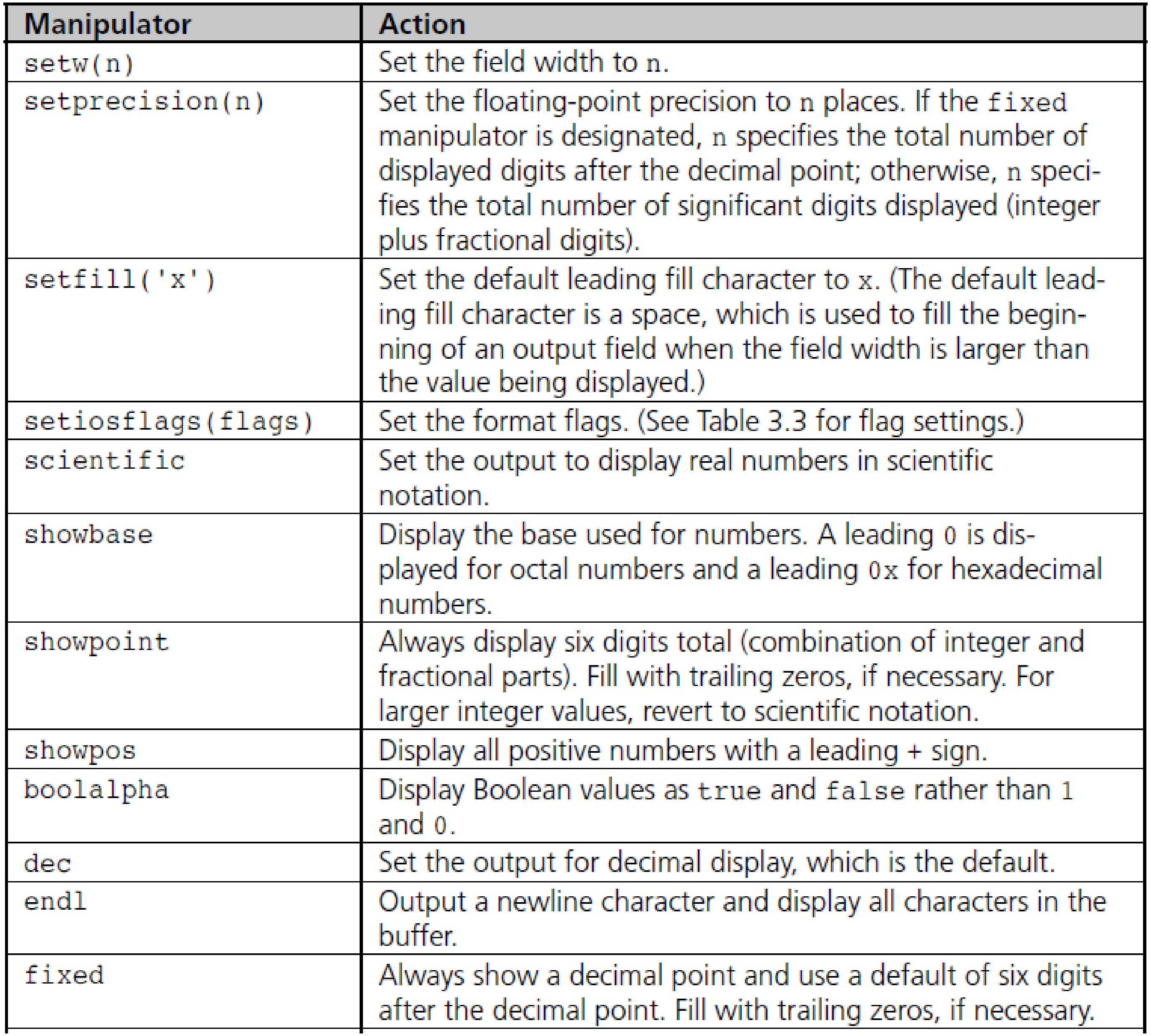
  ```
  k = n;          //assign n’s value to k
    n = n + 1;   //and then increment n
  ```

Assignment Operations (continued)

- **Decrement operator `--`:** Unary operator for the special case when a variable is decreased by 1
- **Prefix decrement operator** appears before the variable
  - Example: **`--i;`**
- **Postfix decrement operator** appears after the variable
  - Example: **`i--;`**

Formatting Numbers for Program Output

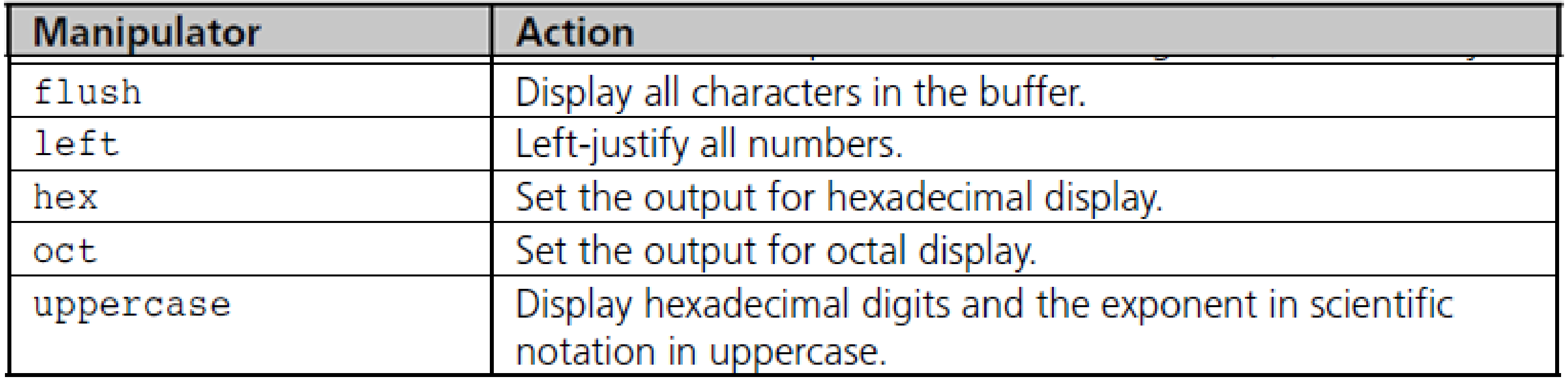
- Proper output formatting contributes to ease of use and user satisfaction
- **`cout`** with stream manipulators can control output formatting



Manipulator	Action
`setw(n)`	Set the field width to `n`.
`setprecision(n)`	Set the floating-point precision to `n` places. If the `fixed` manipulator is designated, `n` specifies the total number of displayed digits after the decimal point; otherwise, `n` specifies the total number of significant digits displayed (integer plus fractional digits).
`setfill('x')`	Set the default leading fill character to `x`. (The default leading fill character is a space, which is used to fill the beginning of an output field when the field width is larger than the value being displayed.)
`setiosflags(flags)`	Set the format flags. (See Table 3.3 for flag settings.)
`scientific`	Set the output to display real numbers in scientific notation.
`showbase`	Display the base used for numbers. A leading `0` is displayed for octal numbers and a leading `0x` for hexadecimal numbers.
`showpoint`	Always display six digits total (combination of integer and fractional parts). Fill with trailing zeros, if necessary. For larger integer values, revert to scientific notation.
`showpos`	Display all positive numbers with a leading + sign.
`boolalpha`	Display Boolean values as `true` and `false` rather than `1` and `0`.
`dec`	Set the output for decimal display, which is the default.
`endl`	Output a newline character and display all characters in the buffer.
`fixed`	Always show a decimal point and use a default of six digits after the decimal point. Fill with trailing zeros, if necessary.

Table 3.1 Commonly Used Stream Manipulators

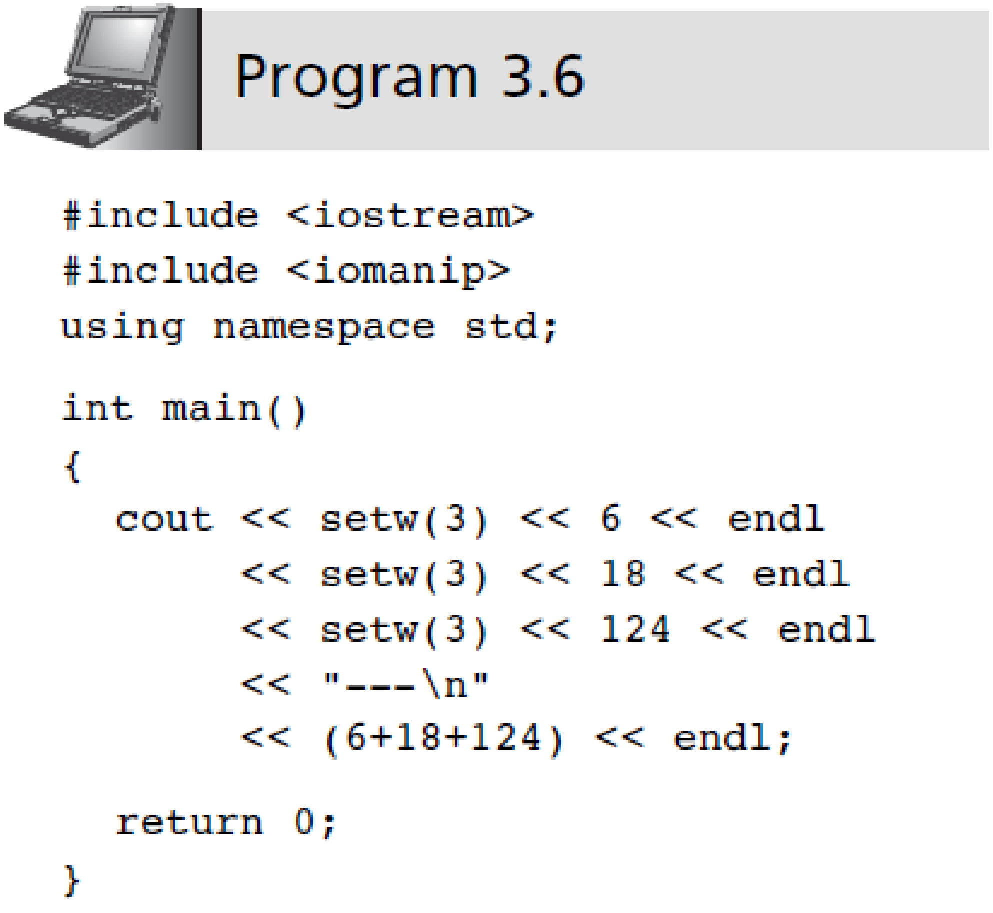
Formatting Numbers for Program Output (continued)



Manipulator	Action
`flush`	Display all characters in the buffer.
`left`	Left-justify all numbers.
`hex`	Set the output for hexadecimal display.
`oct`	Set the output for octal display.
`uppercase`	Display hexadecimal digits and the exponent in scientific notation in uppercase.

Table 3.1 Commonly Used Stream Manipulators (continued)

Formatting Numbers for Program Output (continued)



Program 3.6

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << setw(3) << 6 << endl
       << setw(3) << 18 << endl
       << setw(3) << 124 << endl
       << "---\n"
       << (6+18+124) << endl;

  return 0;
}
```

Formatting Numbers for Program Output (continued)

- The field width manipulator must be included for each value in the data stream sent to **`cout`**
- Other manipulators remain in effect until they are changed
- **`iomanip`** header file must be included to use manipulators requiring arguments

Formatting Numbers for Program Output (continued)

- Formatting floating-point numbers requires three field-width manipulators to:
  - Set the total width of the display
  - Force a decimal place
  - Set the number of significant digits after the decimal point
- Example:

```
cout << "|" << setw(10) << fixed
     << setprecision(3) << 25.67 << "|";
```

produces this output: `| 25.670|`

Formatting Numbers for Program Output (continued)

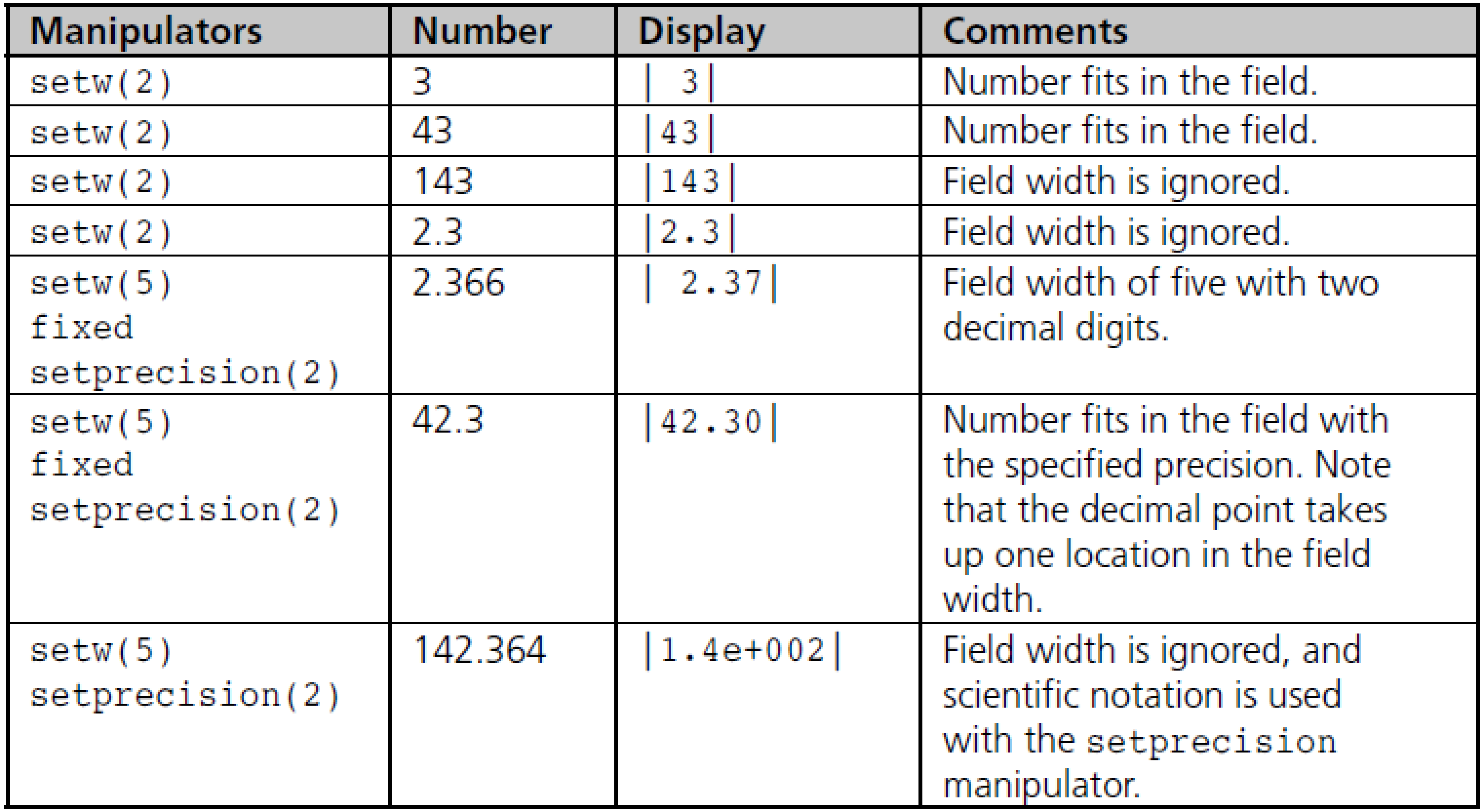
`setprecision:` Sets number of digits after decimal point if a decimal point has been explicitly forced; otherwise, it sets the total number of displayed digits

If the field width is too small, **`cout`** ignores the **`setw`** manipulator setting and allocates enough space for printing

If **`setprecision`** setting is too small, the fractional part of the value is rounded to the specified number of decimal places

If **`setprecision`** value is too large, the fractional value is displayed with its current size

Formatting Numbers for Program Output (continued)



Manipulators	Number	Display	Comments
setw(2)	3	| 3|	Number fits in the field.
setw(2)	43	|43|	Number fits in the field.
setw(2)	143	|143|	Field width is ignored.
setw(2)	2.3	|2.3|	Field width is ignored.

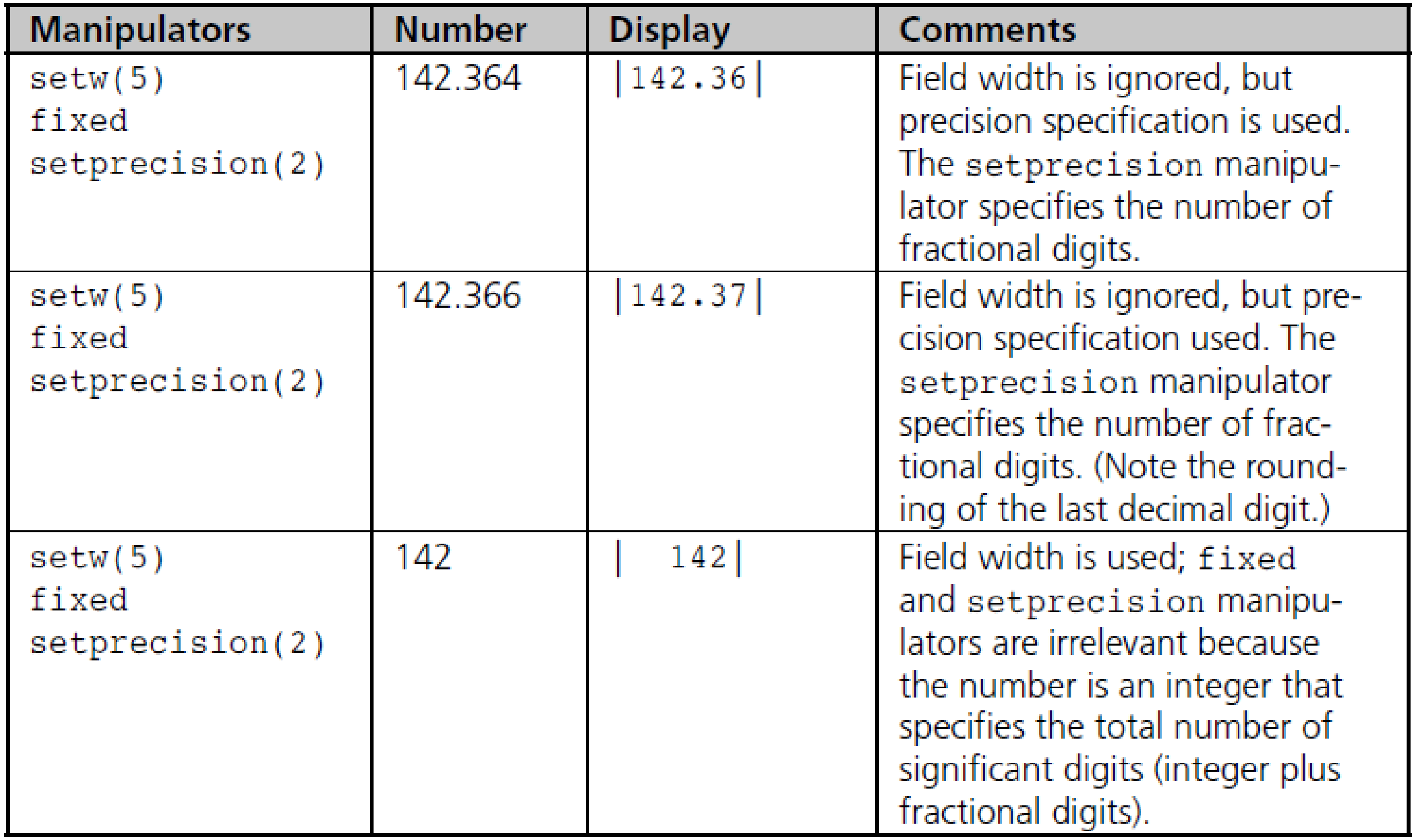
setw(5)
fixed
setprecision(2)
	2.366	| 2.37|	Field width of five with two decimal digits.

setw(5)
fixed
setprecision(2)
	42.3	|42.30|	Number fits in the field with the specified precision. Note that the decimal point takes up one location in the field width.

setw(5)
setprecision(2)
	142.364	|1.4e+002|	Field width is ignored, and scientific notation is used with the setprecision manipulator.

Table 3.2 Effect of Format Manipulators

Formatting Numbers for Program Output (continued)



Manipulators	Number	Display	Comments

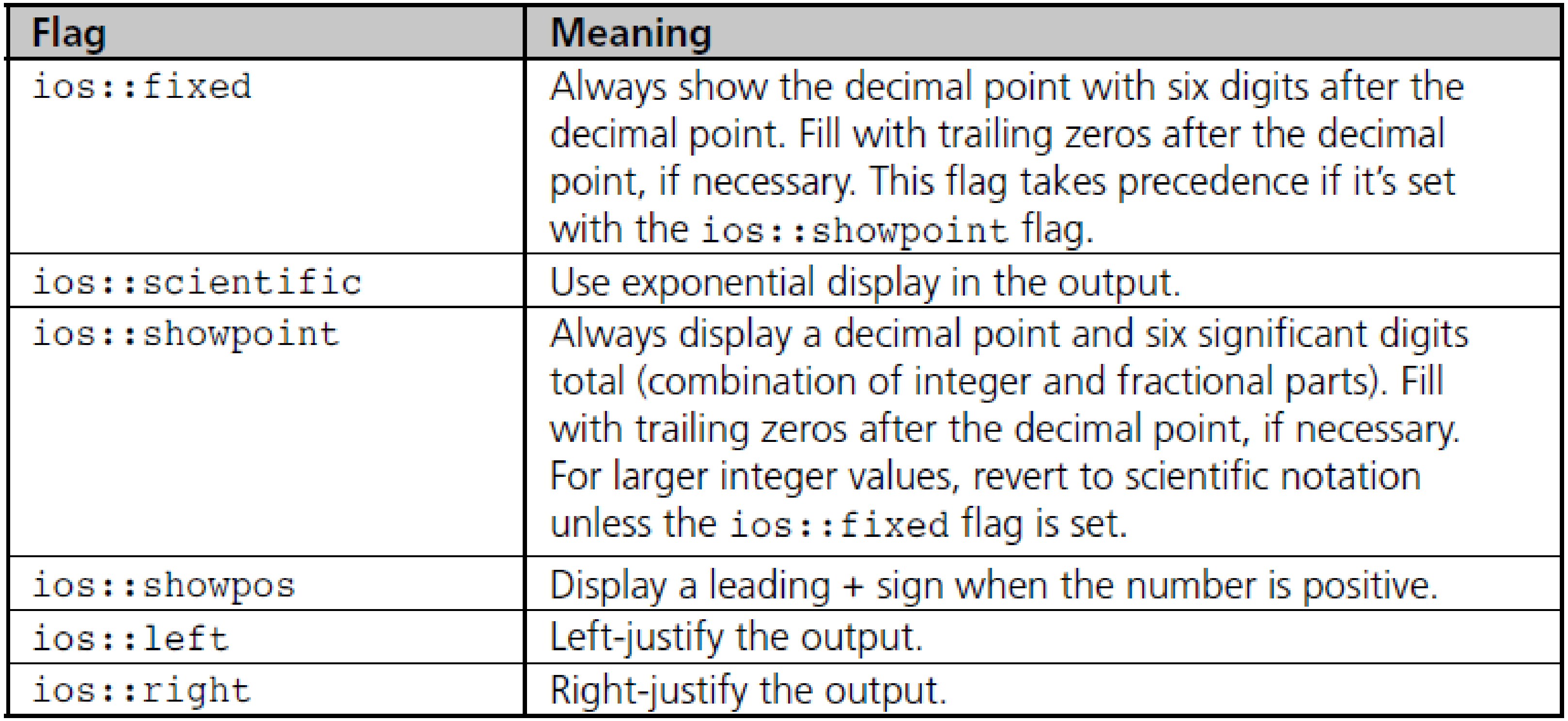
`setw(5)`
`fixed`
`setprecision(2)`
	142.364	`|142.36|`	Field width is ignored, but precision specification is used. The `setprecision` manipulator specifies the number of fractional digits.

`setw(5)`
`fixed`
`setprecision(2)`
	142.366	`|142.37|`	Field width is ignored, but precision specification used. The `setprecision` manipulator specifies the number of fractional digits. (Note the rounding of the last decimal digit.)

`setw(5)`
`fixed`
`setprecision(2)`
	142	`|  142|`	Field width is used; `fixed` and `setprecision` manipulators are irrelevant because the number is an integer that specifies the total number of significant digits (integer plus fractional digits).

Table 3.2 Effect of Format Manipulators (continued)

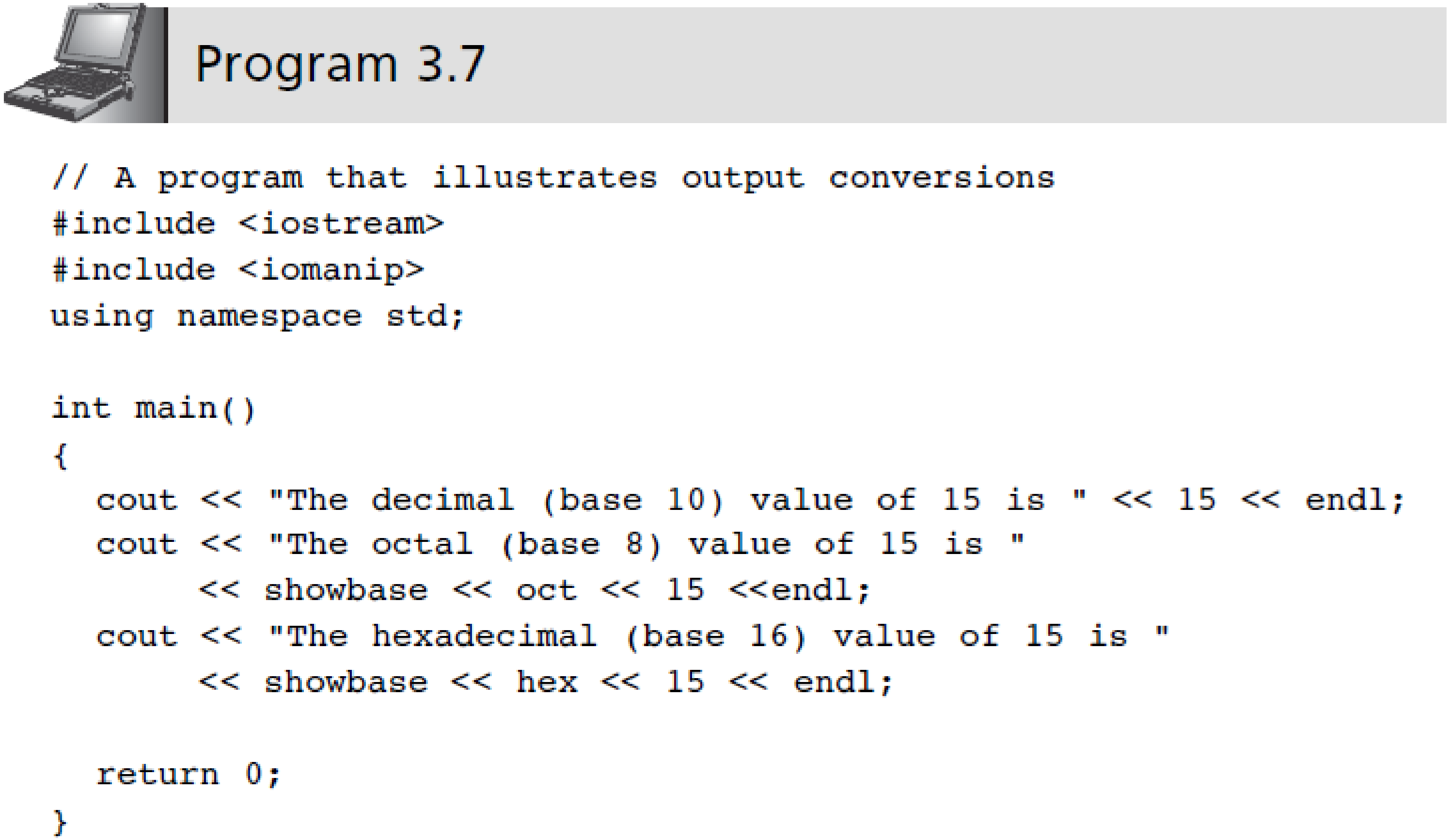
Formatting Numbers for Program Output (continued)



Flag	Meaning
`ios::fixed`	Always show the decimal point with six digits after the decimal point. Fill with trailing zeros after the decimal point, if necessary. This flag takes precedence if it's set with the `ios::showpoint` flag.
`ios::scientific`	Use exponential display in the output.
`ios::showpoint`	Always display a decimal point and six significant digits total (combination of integer and fractional parts). Fill with trailing zeros after the decimal point, if necessary. For larger integer values, revert to scientific notation unless the `ios::fixed` flag is set.
`ios::showpos`	Display a leading + sign when the number is positive.
`ios::left`	Left-justify the output.
`ios::right`	Right-justify the output.

Table 3.3 Format Flags for Use with **`setiosflags()`**

Formatting Numbers for Program Output (continued)



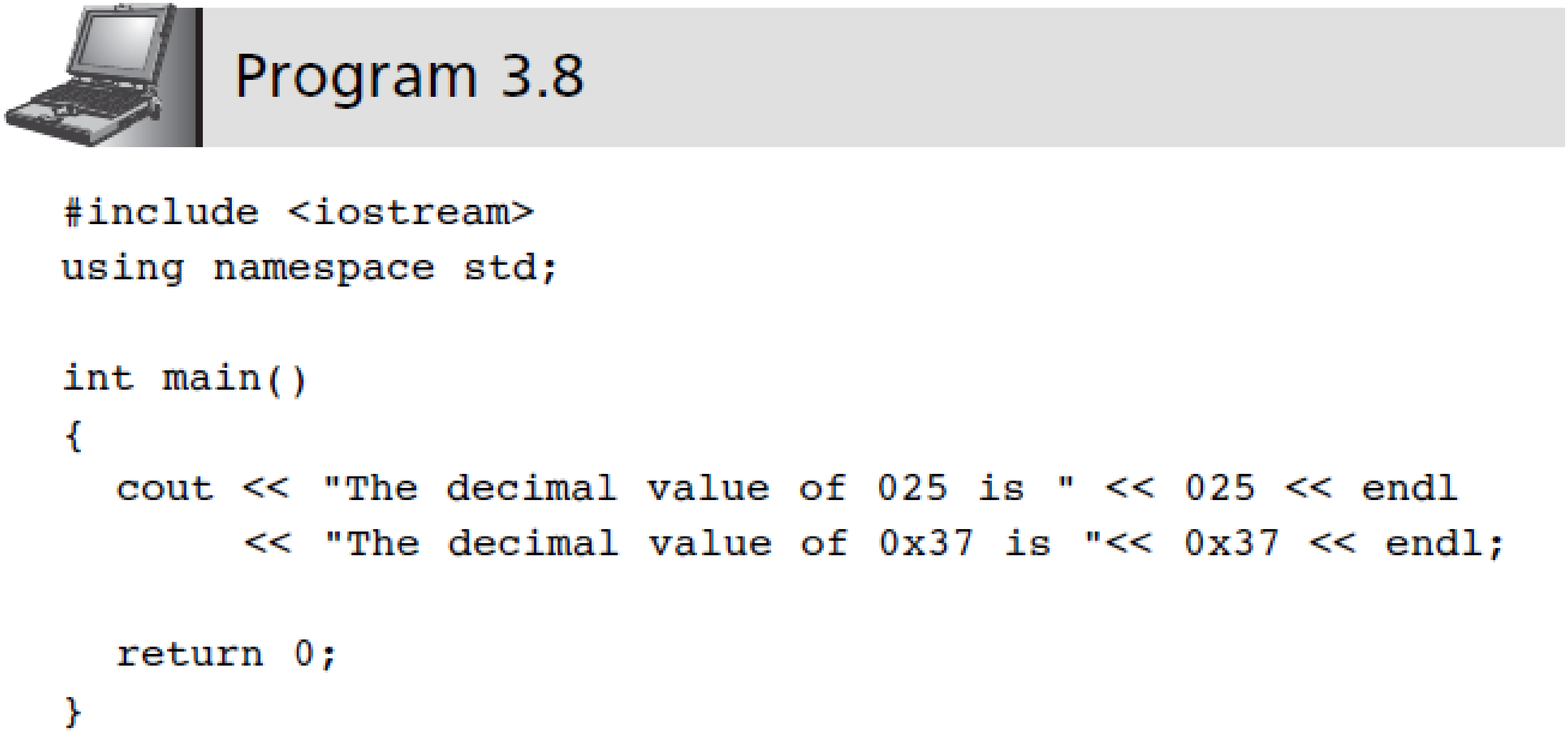
Program 3.7

```
// A program that illustrates output conversions
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  cout << "The decimal (base 10) value of 15 is " << 15 << endl;
  cout << "The octal (base 8) value of 15 is "
       << showbase << oct << 15 <<endl;
  cout << "The hexadecimal (base 16) value of 15 is "
       << showbase << hex << 15 << endl;

  return 0;
}
```

Formatting Numbers for Program Output (continued)

- To designate an octal integer constant:
  - use a leading zero
- To designate a hexadecimal integer constant:
  - use a leading 0x
- Manipulators affect only output; the value stored internally does not change

Formatting Numbers for Program Output (continued)



Program 3.8

```
#include <iostream>
using namespace std;

int main()
{
  cout << "The decimal value of 025 is " << 025 << endl
       << "The decimal value of 0x37 is "<< 0x37 << endl;

  return 0;
}
```

Formatting Numbers for Program Output (continued)

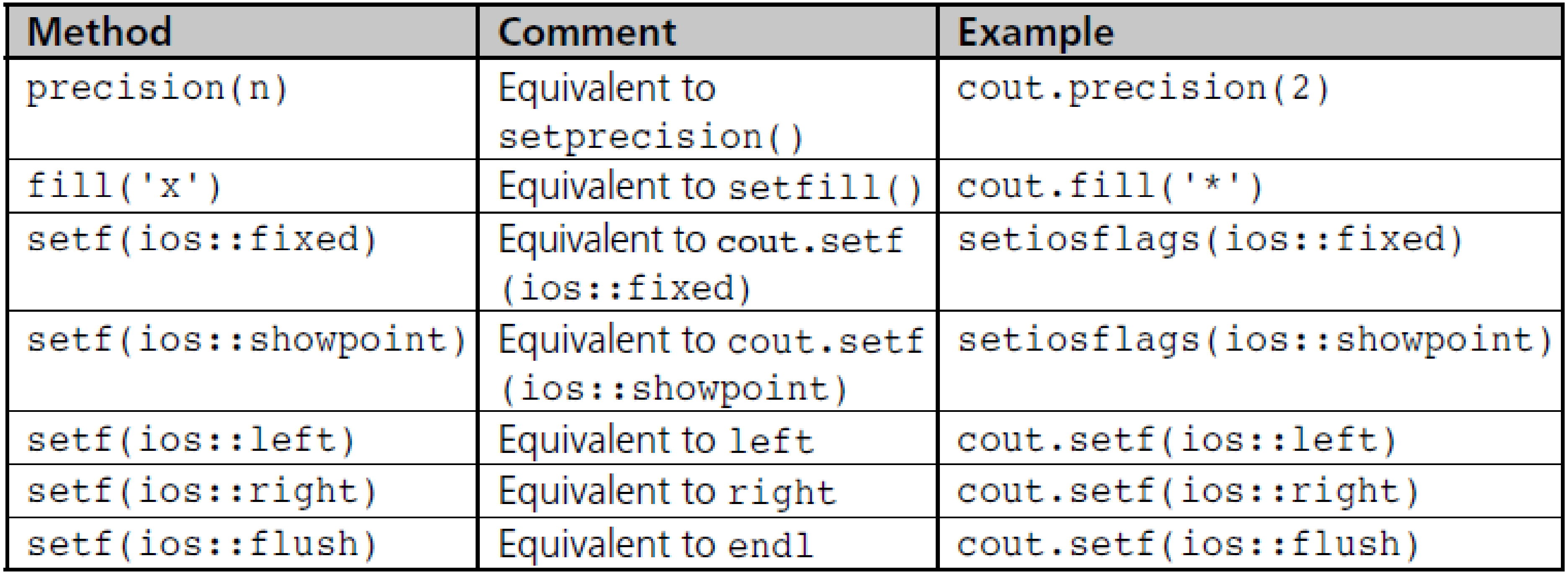
Manipulators can also be set using the **`ostream`** class methods

Separate the **`cout`** object name from the method name with a period

Example:

```
cout.precision(2)
```

Formatting Numbers for Program Output (continued)



Method	Comment	Example
`precision(n)`	Equivalent to `setprecision()`	`cout.precision(2)`
`fill('x')`	Equivalent to `setfill()`	`cout.fill('*')`
`setf(ios::fixed)`	Equivalent to `cout.setf (ios::fixed)`	`setiosflags(ios::fixed)`
`setf(ios::showpoint)`	Equivalent to `cout.setf (ios::showpoint)`	`setiosflags(ios::showpoint)`
`setf(ios::left)`	Equivalent to `left`	`cout.setf(ios::left)`
`setf(ios::right)`	Equivalent to `right`	`cout.setf(ios::right)`
`setf(ios::flush)`	Equivalent to `endl`	`cout.setf(ios::flush)`

Table 3.4 `ostream` Class Functions

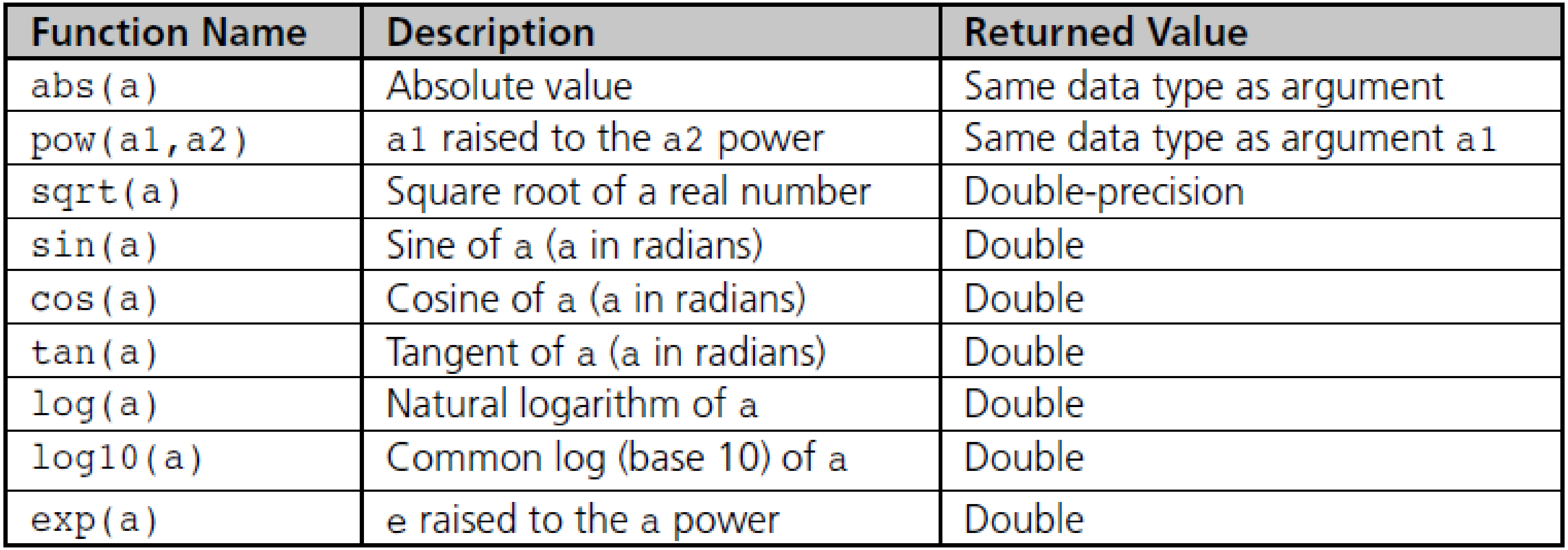
Using Mathematical Library Functions

- C++ has preprogrammed mathematical functions that can be included in a program
- You must include the **`cmath`** header file:

```
#include <cmath>
```

- Math functions require one or more arguments as input, but will return only one value
- All functions are overloaded, and can be used with integer and real arguments

Using Mathematical Library Functions (continued)

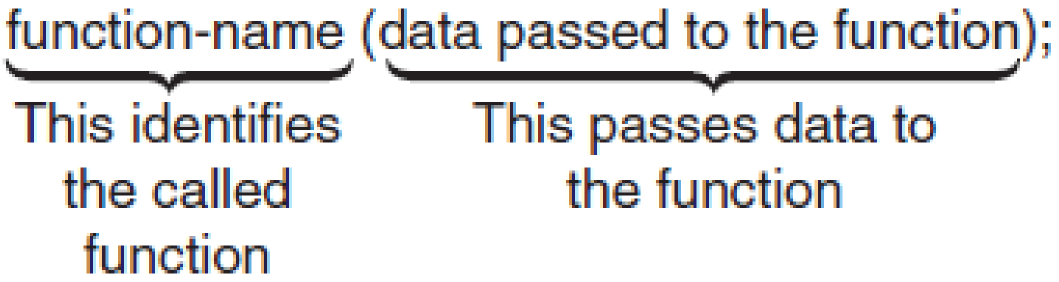


Function Name	Description	Returned Value
abs(a)	Absolute value	Same data type as argument
pow(a1,a2)	a1 raised to the a2 power	Same data type as argument a1
sqrt(a)	Square root of a real number	Double-precision
sin(a)	Sine of a (a in radians)	Double
cos(a)	Cosine of a (a in radians)	Double
tan(a)	Tangent of a (a in radians)	Double
log(a)	Natural logarithm of a	Double
log10(a)	Common log (base 10) of a	Double
exp(a)	e raised to the a power	Double

Table 3.5 Common C++ Functions

Using Mathematical Library Functions (continued)

- To use a math function, give its name and pass the input arguments within parentheses
- Expressions that can be evaluated to a value can be passed as arguments



function-name (data passed to the function);

This identifies the called function

This passes data to the function

Figure 3.10 Using and passing data to a function

- Function calls can be nested. Eg. **sqrt(sin(abs(angle)))**

Selection Structures

Selection Criteria

- **`if-else`** statement: Implements a decision structure for two alternatives

  Syntax:

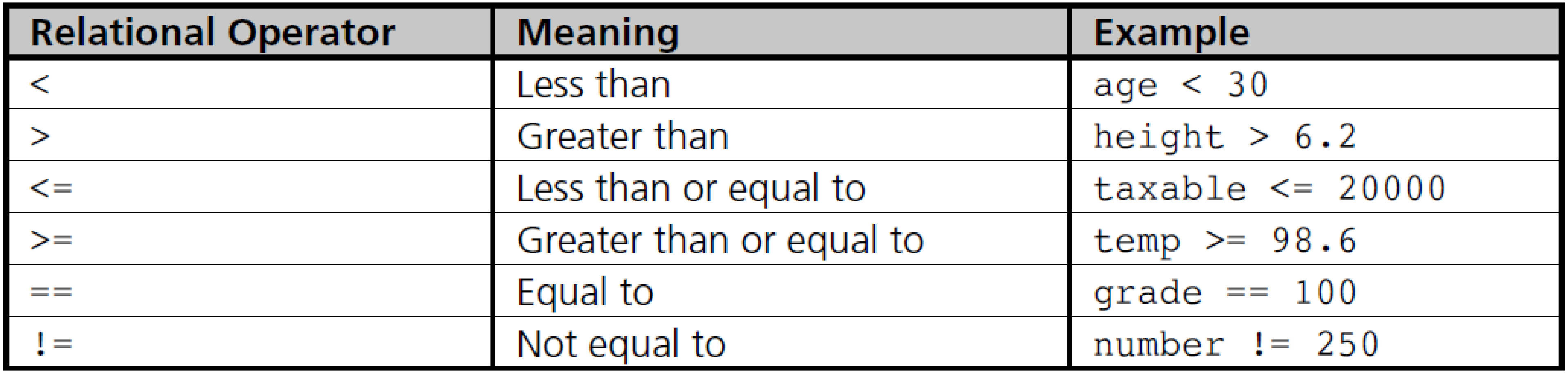
  ```
  if (condition)
      statement executed if condition is true;
  else
      statement executed if condition is false;
  ```

Selection Criteria (continued)

- The condition is evaluated to its numerical value:
  - A non-zero value is considered to be true
  - A zero value is considered to be false
- The **`else`** portion is optional
  - Executed only if the condition is false
- The condition may be any valid C++ expression

Relational Operators

- **Relational expression:** Compares two operands or expressions using **relational**



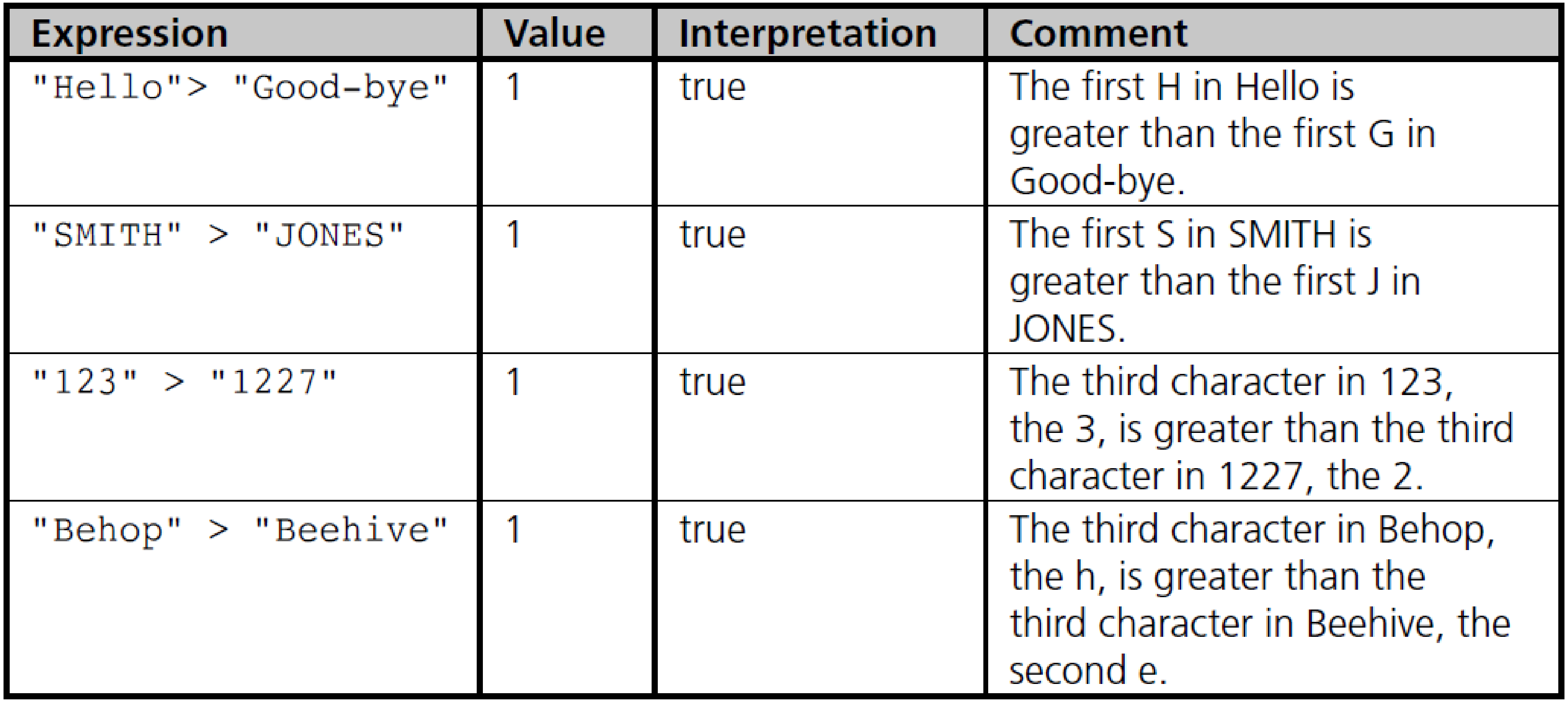
Relational Operator	Meaning	Example
`<`	Less than	`age < 30`
`>`	Greater than	`height > 6.2`
`<=`	Less than or equal to	`taxable <= 20000`
`>=`	Greater than or equal to	`temp >= 98.6`
`==`	Equal to	`grade == 100`
`!=`	Not equal to	`number != 250`

Table 4.1 C++’s Relational Operators

Relational Operators (continued)

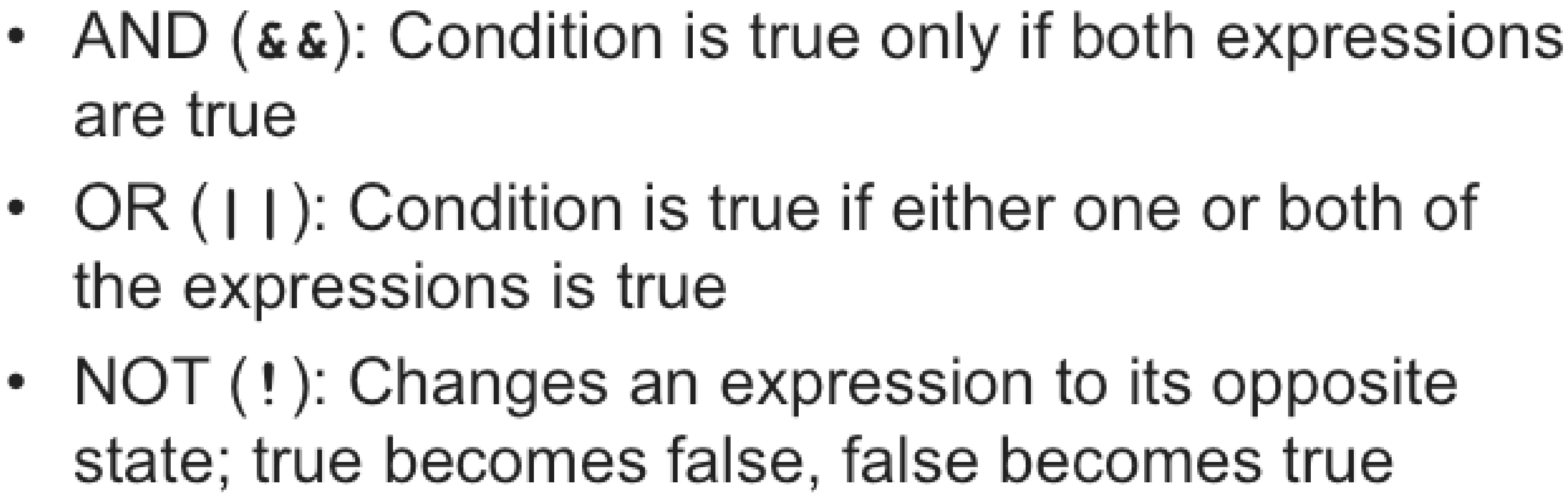
- Relational expressions are evaluated to a numerical value of 1 or 0 only:
  - If the value is 1, the expression is true
  - If the value is 0, the expression is false
- `char` values are automatically coerced to `int` values for comparison purposes
- Strings are compared on a character by character basis
  - The string with the first lower character is considered smaller

Relational Operators (continued)



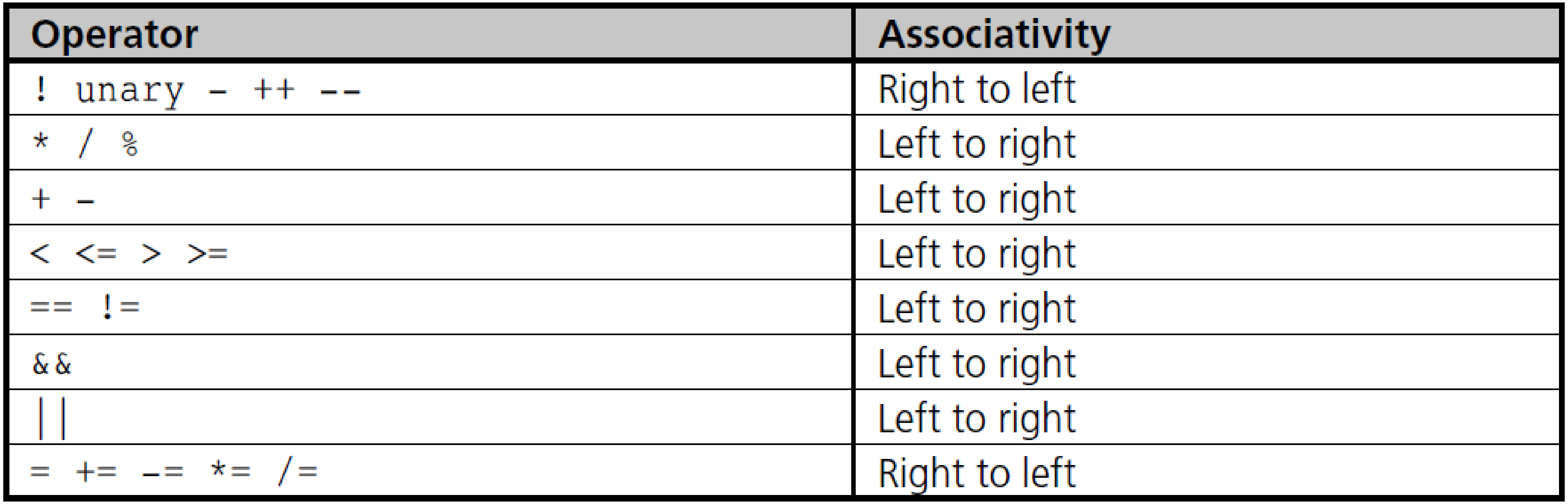
Expression	Value	Interpretation	Comment
`"Hello"> "Good-bye"`	1	true	The first H in Hello is greater than the first G in Good-bye.
`"SMITH" > "JONES"`	1	true	The first S in SMITH is greater than the first J in JONES.
`"123" > "1227"`	1	true	The third character in 123, the 3, is greater than the third character in 1227, the 2.
`"Behop" > "Beehive"`	1	true	The third character in Behop, the h, is greater than the third character in Beehive, the second e.

Logical Operators



- AND (&&): Condition is true only if both expressions are true
- OR (||): Condition is true if either one or both of the expressions is true
- NOT (!): Changes an expression to its opposite state; true becomes false, false becomes true

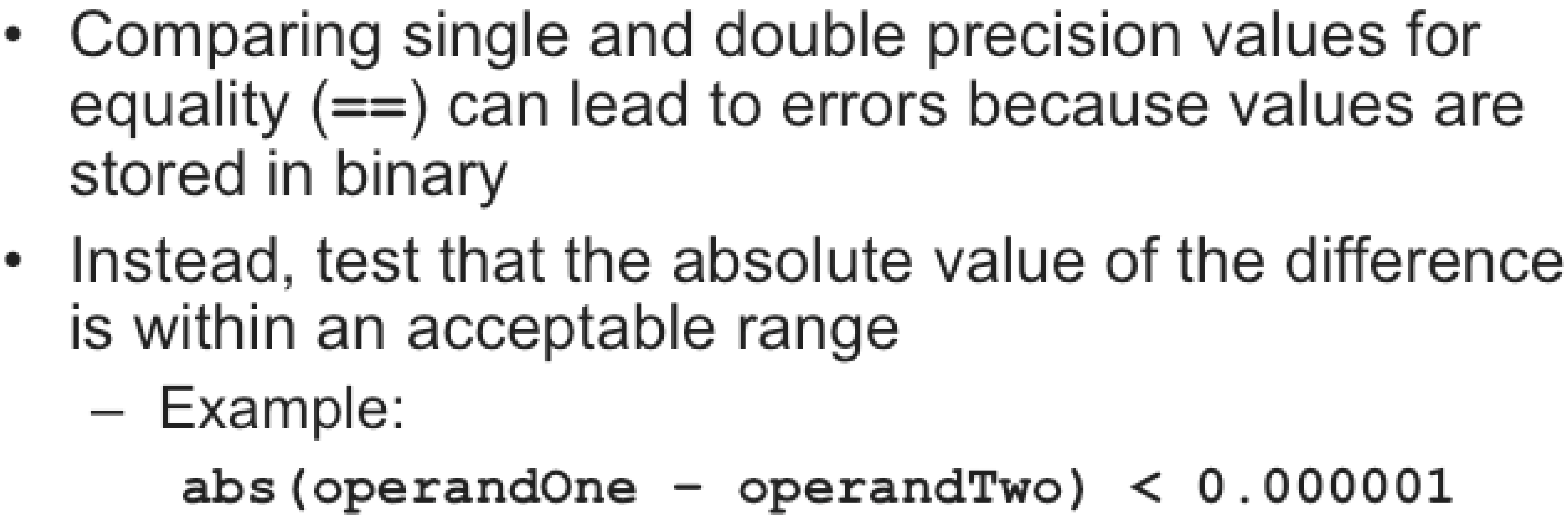
Logical Operators (continued)



Operator	Associativity
`! unary - ++ --`	Right to left
`* / %`	Left to right
`+ -`	Left to right
`< <= > >=`	Left to right
`== !=`	Left to right
`&&`	Left to right
`||`	Left to right
`= += -= *= /=`	Right to left

Table 4.2 Operator Precedence and Associativity

A Numerical Accuracy Problem

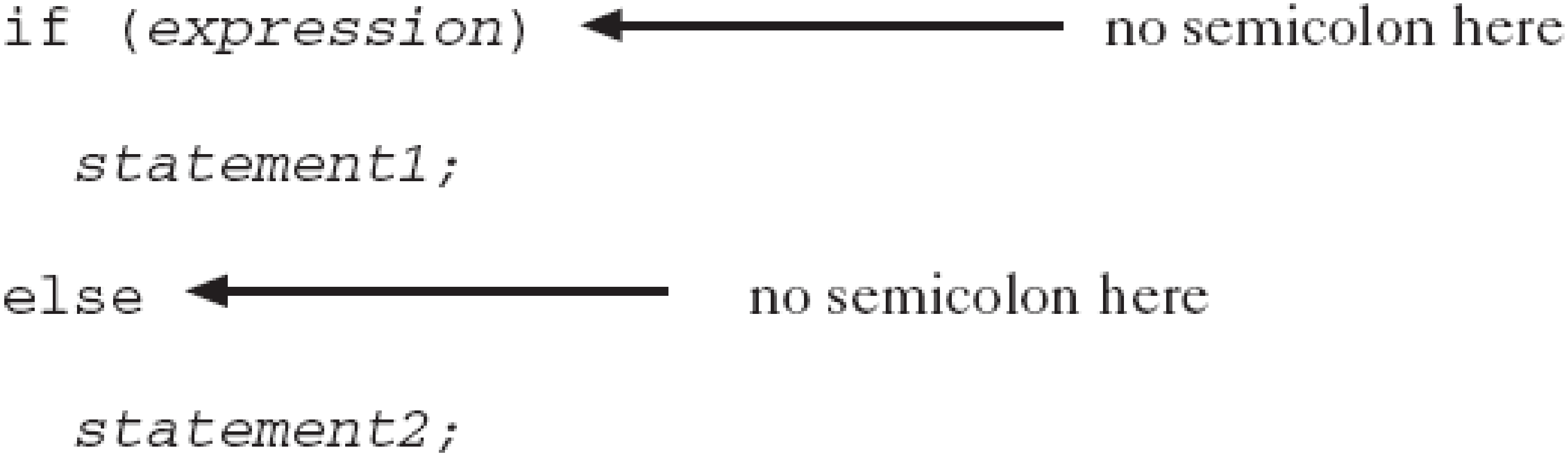


- Comparing single and double precision values for equality (==) can lead to errors because values are stored in binary
- Instead, test that the absolute value of the difference is within an acceptable range
  - Example:

```
abs(operandOne - operandTwo) < 0.000001
```

The `if-else` Statement

- **`if-else`** performs instructions based on the result of a comparison
- Place statements on separate lines for readability
- Syntax:



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

no semicolon here

no semicolon here

The `if-else` Statement (cont’ d)

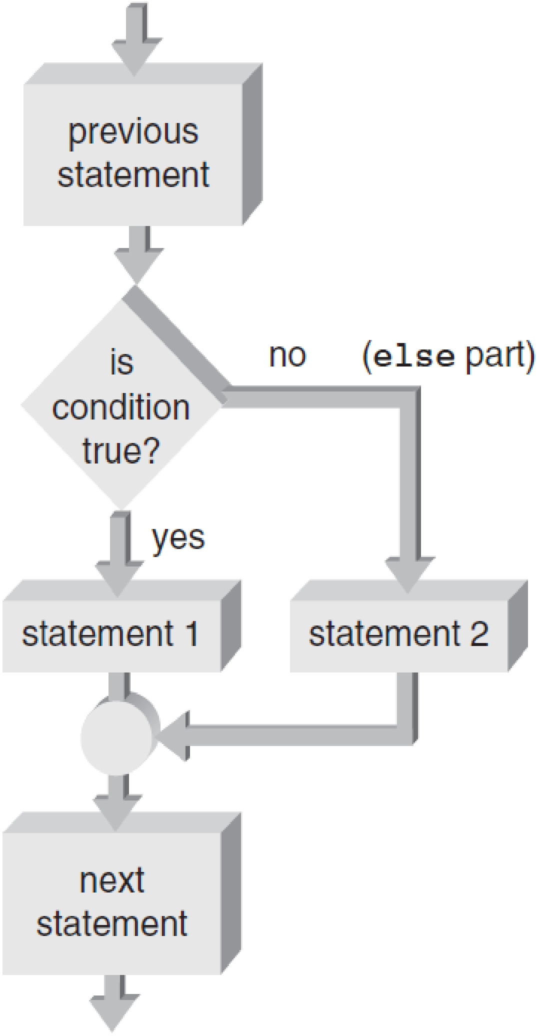
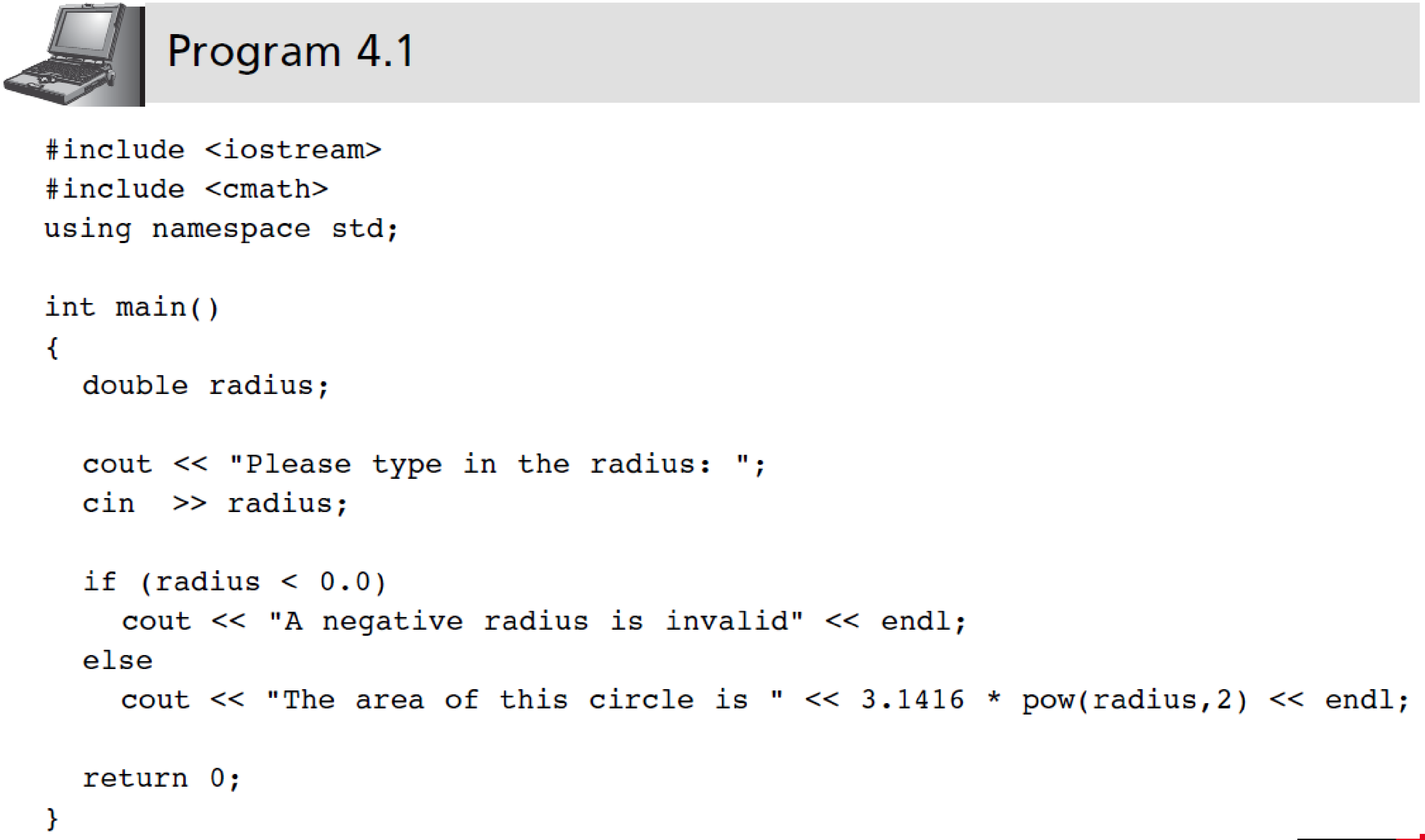


Figure 4.2
The **`if-else`** flowchart

The `if-else` Statement (continued)



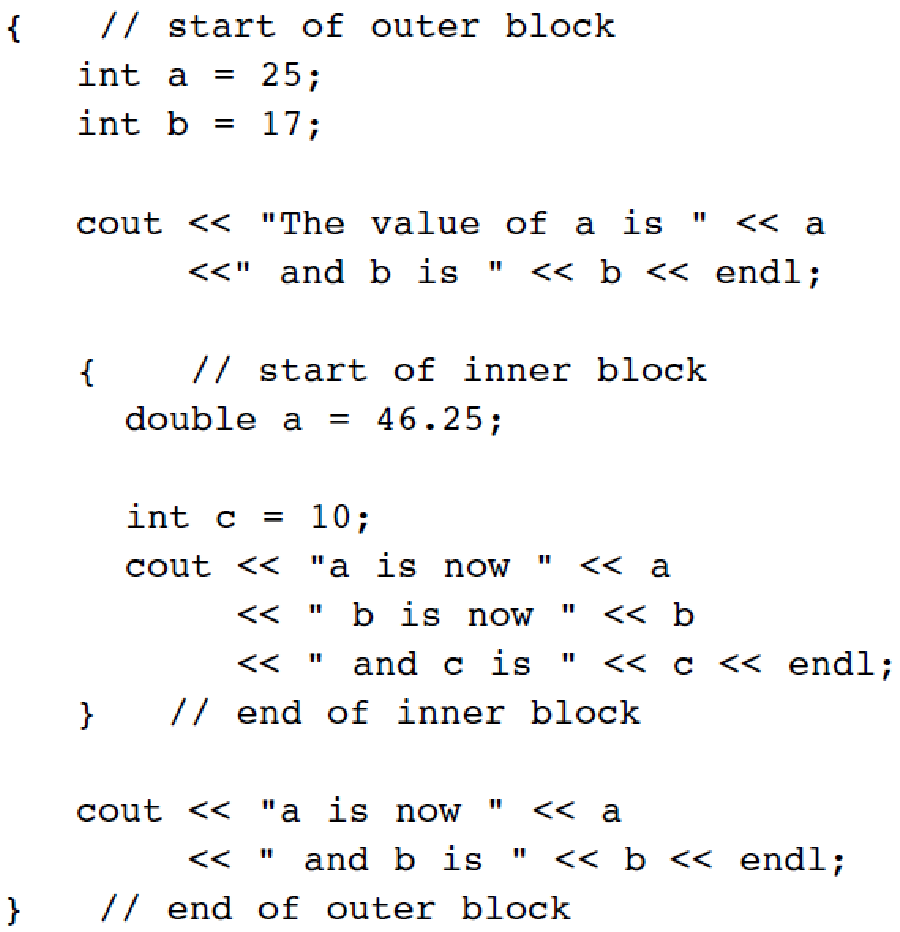
Program 4.1

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  double radius;

  cout << "Please type in the radius: ";
  cin  >> radius;

  if (radius < 0.0)
    cout << "A negative radius is invalid" << endl;
  else
    cout << "The area of this circle is " << 3.1416 * pow(radius,2) << endl;

  return 0;
}
```

Compound Statements

- **Compound statement:** A sequence of single statements contained between braces
  - Creates a block of statements
  - A block of statements can be used anywhere that a single statement is legal
  - Any variable declared within a block is usable only within that block
- **Scope:** The area within a program where a variable can be used
  - A variable's scope is based on where the variable is declared

Block Scope (continued)



```
{   // start of outer block
   int a = 25;
   int b = 17;

   cout << "The value of a is " << a
        <<" and b is " << b << endl;

   {    // start of inner block
     double a = 46.25;

     int c = 10;
     cout << "a is now " << a
          << " b is now " << b
          << " and c is " << c << endl;
   }   // end of inner block

   cout << "a is now " << a
        << " and b is " << b << endl;
}   // end of outer block
```

One-Way Selection

- **One-way selection:** An **`if`** statement without the optional **`else`** portion

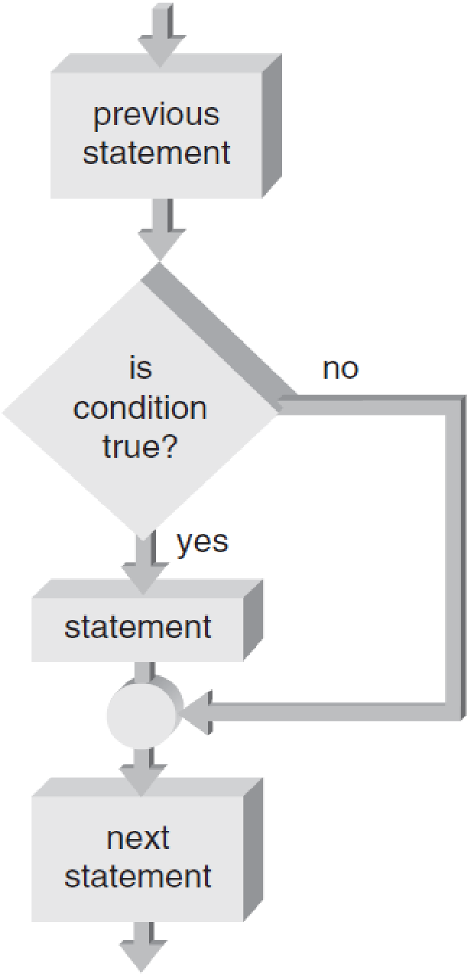


Figure 4.3 A one-way selection **`if`** statement

Problems Associated with the `if-else` Statement

- Common problems with **`if-else`** statements:
  - Misunderstanding what an expression is
  - Using the assignment operator (**=**) instead of the relational operator (**==**)

Nested `if` Statements

- **`if-else`** statement can contain any valid C++ statement, including another **`if-else`**
- Nested **`if`** statement: an **`if-else`** statement completely contained within another **`if-else`**
- Use braces to block code, especially when inner **`if`** statement does not have its own **`else`**

Nested `if` Statements (continued)

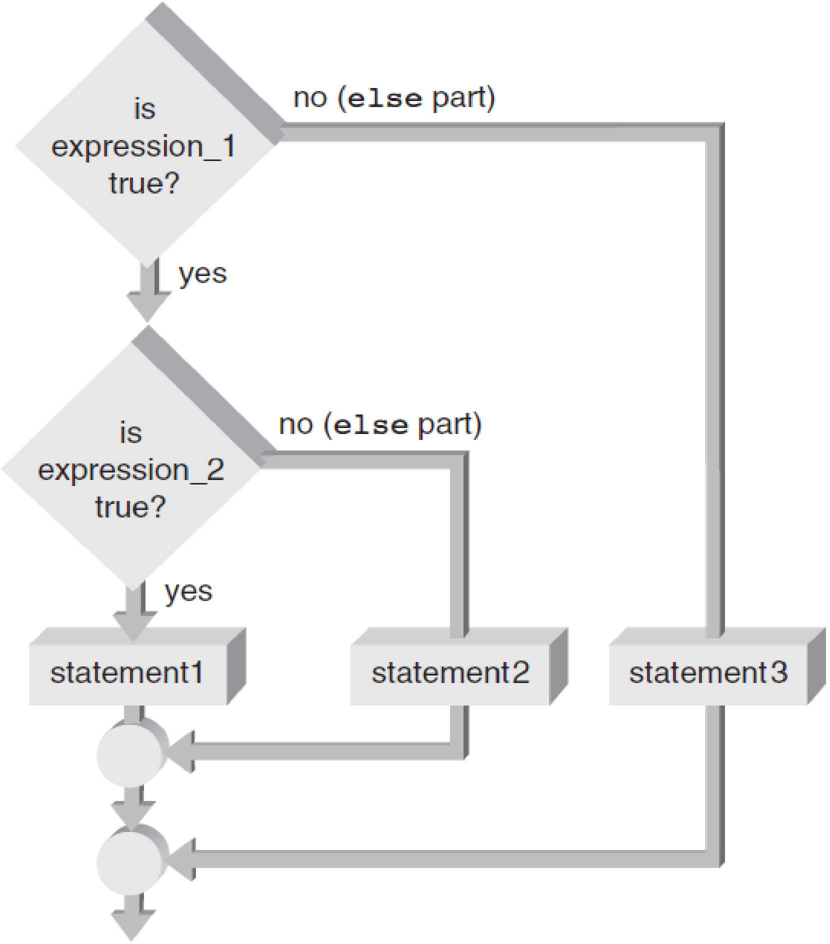


Figure 4.4a
Nested within the
`if` part

The `if-else` Chain

- **`if-else`** chain: A nested **`if`** statement occurring in the **`else`** clause of the outer **`if-else`**
- If any condition is true, the corresponding statement is executed and the chain terminates
- Final **`else`** is only executed if no conditions were true
  - Serves as a catch-all case
- **`if-else`** chain provides one selection from many possible alternatives

The `if-else` Chain (continued)

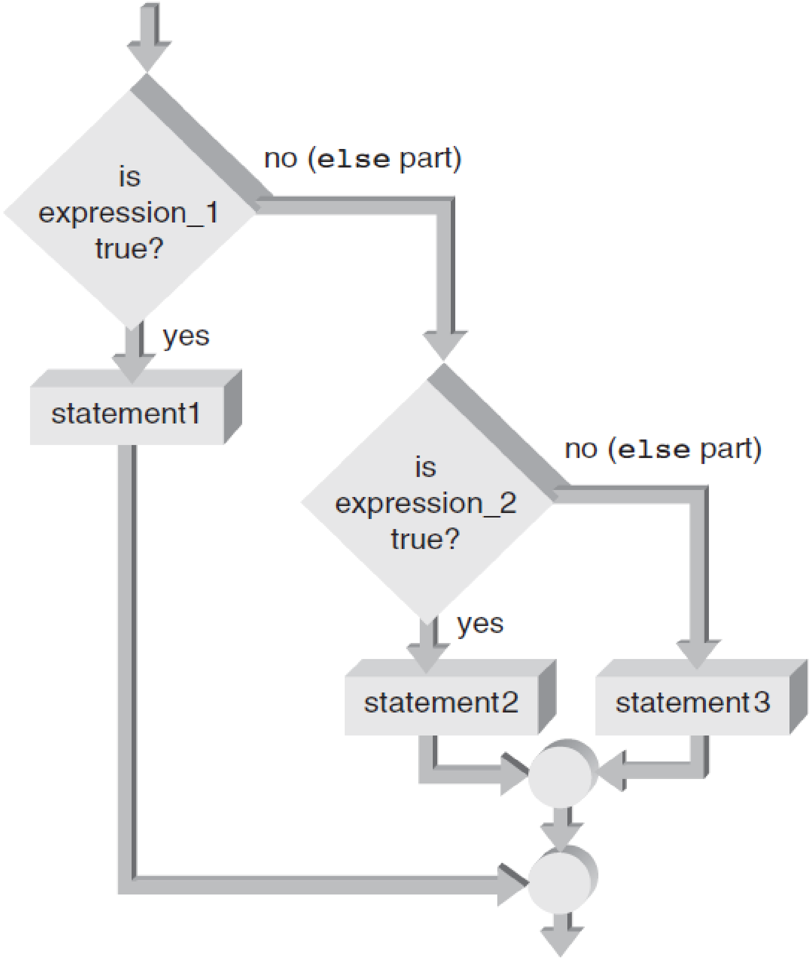
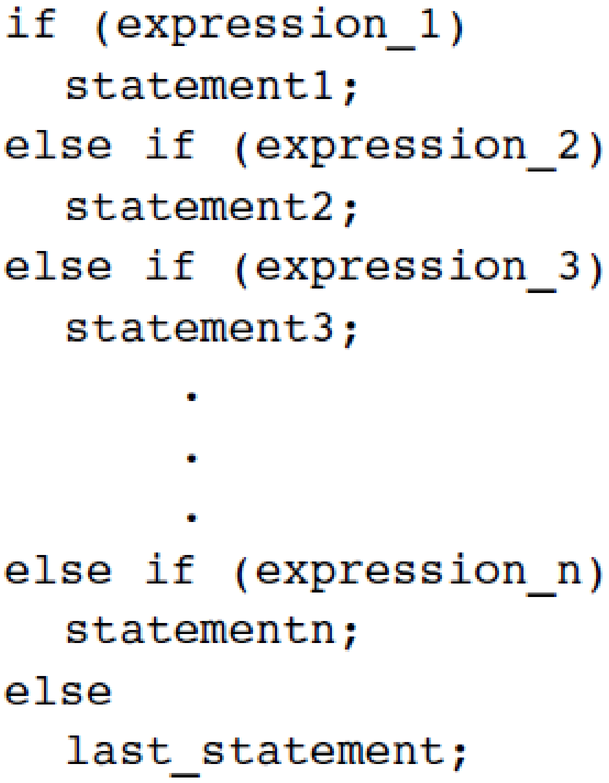


Figure 4.4b
Nested within the
`else` part

The `if-else` Chain (continued)

- General form of an **`if-else`** chain



```
if (expression_1)
   statement1;
else if (expression_2)
   statement2;
else if (expression_3)
   statement3;
         .
         .
         .
else if (expression_n)
   statementn;
else
   last_statement;
```

The `if-else` Chain (continued)

Example:

Display a one word description of an int value number as “Positive”, “Negative” or “Zero”

Answer

```
if (number >  0)
   cout << “Positive”;
else if (number < 0)
   cout << “Negative”;
else
   cout << “Zero”;
```

The `switch` Statement

- **`switch`** statement: Provides for one selection from many alternatives
- **`switch`** keyword starts the statement
  - Is followed by the expression to be evaluated
- **`case`** keyword identifies a value to be compared to the value of the switch expression
  - When a match is found, statements in this **`case`** block are executed
- All further cases after a match is found are executed unless a **`break`** statement is found
- **`default`** case is executed if no other case value matches were found
- **`default`** case is optional

`switch` (continued)

- If you find that your code has lots of **`else if`** statements and the relational expressions are all ==, you should probably change over to a **`switch`** statement.
- switch tests equivalence of an **`int`** type (incl **`char, int, long`**) against many **`const int`** values.

`switch` (continued)

```
switch (n)
{
   case 0:
         //statements
         break;
   case 1:
         //statements
         break;
   case 2:
         //statements
         break;
   default:
         //statements
}
```

must be an **`int`** type

runs case if n == 0

no braces. end code with break;

runs this code if no cases match

`switch` with char

```
switch (c)
{
   case '\n':
         //statements
         break;
   case '1':
         //statements
         break;
   case 'e':
         //statements
         break;
   case 32:
         //statements
}
```

`char c = cin.getc();`

ENTER key

e character entered

the SPACE bar (ASCII)

switch == || ==

```
switch (c)
{
   case 'a':
   case 'A':
   //statements
       break;
   case 'b':
   case 'B':
   //statements
       break;
   default:
       //statements
   }
```

remember: these must be constants

letter a OR A

You could simulate an AND by nesting a switch inside a case, but no-one is that crazy

SWINBURNE UNIVERSITY OF TECHNOLOGY

Uses for switch

- Great for
  - processing key presses,
    - message pump (`const int`s or "magic numbers" used to send messages),
    - replacing messy `else if` constructs and
    - menus.
- switch statements usually run inside a loop
- A `do while` loop is the most useful

A classification example with if..else

```
#include <iostream>
using namespace std;
int main()
{
  char code;
  cout << "Enter a specification code: ";
  cin  >> code;
  if (code == 'S')
     cout << "The item is space exploration grade.";
  else if (code == 'M')
    cout << "The item is military grade.";
  else if (code == 'C')
    cout << "The item is commercial grade.";
  else if (code == 'T')
     cout << "The item is toy grade.";
  else
     cout << "An invalid code was entered.";
  cout << endl;
  return 0;
}
```

A classification example with switch

```
#include <iostream>
using namespace std;
int main()
{
  char code;
  cout << "Enter a specification code:  ";
  cin  >> code;
  switch (code) {
     case 'S': cout << "The item is space exploration grade.";
               break;
     case 'M': cout << "The item is military grade.";
               break;
     case 'C': cout << "The item is commercial grade.";
               break;
     case 'T': cout << "The item is toy grade.";
               break;
     default: cout << "An invalid code was entered.";
  }
 cout << endl;
 return 0;
}
```

menu code

```
char c;
do
{
   cout << "Menu:\n1. read\n2. write\n3. rest\n";
   cin >> c;
   switch(c)
   {
       case '1′: cout << "reading...\n";
                 break;
       case '2′: cout << "writing...\n";
                 break;
       default: cout << ">";
   }//end of switch
}while(c !='3');
```

Using peek()

```
int main () {
  char c; int n; string str;
  cout << "Enter a number or a word: ";
  c=cin.peek();
  if ( (c >= '0') && (c <= '9') )
  {
    cin >> n;
    cout << "You have entered number " << n << endl;
  }   else   {
    cin >> str;
    cout << "You have entered word " << str << endl;
  }
  return 0;  }
```

Better menu

- `peek()` lets us look at a character in the keyboard stream
- We can use it to find out what kind of characters the user is typing
- digits? must be a number. Maybe it's a menu item
- letters? must be text. Put it in a string.