Operators ++ and --
The increment operator ++ adds 1, and the decrement operator -- subtracts 1.

Postincrement operator x++
- The value of the variable x is increased by 1
- The value of the expression x++ is equal to the original value of x

Example: Suppose the value of x is 5. Then after we execute:
\[ y = x++; \quad \text{// The value of y is 5, and the value of x is 6.} \]

Preincrement operator ++x
- The value of the variable x is increased by 1
- The value of the expression ++x is equal to one more than the original value of x

Example: Suppose the value of x is 5. Then after we execute:
\[ y = ++x; \quad \text{// The value of y is 6, and the value of x is 6.} \]

Example: Suppose i = 100. Then after we execute:
\[ a = i++ * 2; \quad \text{// The value of a is 200, and the value of i is 101.} \]
\[ a = ++i * 2; \quad \text{// The value of a is 202, and the value of i is 101.} \]

Postdecrement operator x--
- The value of the variable x is decreased by 1
- The value of the expression x-- is equal to the original value of x

Example: Suppose the value of x is 5. Then after we execute:
\[ y = x--; \quad \text{// The value of y is 5, and the value of x is 4.} \]

Predecrement operator --x
- The value of the variable x is decreased by 1
- The value of the expression --x is equal to one less than the original value of x

Example: Suppose the value of x is 5. Then after we execute:
\[ y = --x; \quad \text{// The value of y is 4, and the value of x is 4.} \]

Compound Operators (-=, +=, *=, /=, and %=)
\[ x \, op\, y \quad \text{// same as } x = x \, op \, (y) \]

Example 1:
\[ x *= 2; \quad \text{// same as } x = x * 2 \]
<table>
<thead>
<tr>
<th>x’s value before</th>
<th>expression</th>
<th>x’s value after</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>x = 3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>x += 3</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>x -= 3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>x *= 3</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>x /= 3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>x %= 3</td>
<td>0</td>
</tr>
</tbody>
</table>

Example 2: What is printed?

```c
for (i = 1; i <= 5; i++) {
    cout << i << endl;
    i += 2
}
```

Answer: 1

4

More Operators and Control Flow

**break statement**
The `break` statement causes an immediate exit from the innermost `while`, `do while`, or `for` loop.

**continue statement**
The `continue` statement is similar to the `break` statement, but instead of exiting the loop we consider resuming execution of the loop. In a `while` or `do while` loop, the `continue` statement causes an immediate jump to the loop test. In a `for` loop, `continue` causes an immediate jump to the update expression `expr3` (i.e., the loop counter updater).

Example: What is printed?

```c
i = 0;
while (i < 5) {
    if (i < 3) {
        i += 2;
        cout << i << endl;
        continue;
    } else {
        cout << ++i << endl;
        break;
    }
    cout << “bottom of loop” << endl;
}
```
switch statement
The switch statement is an alternative to nested if-else statements. See Section 2.10, pages 111-114 (JK).

cast operator
The cast operator can change the data type of a variable’s value.

Example:
```cpp
int x = 5;    // x is the integer 5
(double) x    // the value of x is now converted to a floating-point number 5.0 (or 5.)
```

This can be useful when doing division.

Division of two integers truncates the result (i.e., the fractional part is dropped) in order to return another integer value. For example, \(15/4 = 3.75\); when the fractional part is dropped, the result is 3. Thus, the following code

```cpp
int x = 15, y = 4;
cout << x/y << endl; // output is 3
```

prints 3 to the terminal window.

Similarly, \(3/5 = 0.60\). Therefore, the following code would print 0 to the terminal window!

```cpp
int x = 3, y = 5;
cout << x/y << endl; // output is 0
```

To avoid this kind of truncation, you can use casting:

```cpp
int x = 3, y = 5;
cout << double(x) / double(y) << endl; // output is 0.60
```

Note that casting does NOT change the data type of the variable. In this last example, x and y are still ints throughout the casting. Only their values 3 and 5 are cast to the floating-point values 3.0 and 5.0 in order to ensure floating-point division.