CHAPTER 3

SELECTION CONTROL STRUCTURE

Chapter’s Outcome

At the end of this chapter, student should be able to:

a) Understand how decisions are made in programs

b) Understand how true and false is represented in C++

c) Use relational operators.

d) Use logical operators.

e) Use the if structure.

f) Use the if/else structure.

g) Use nested if structures.

h) Use the switch structure.

Chapter’s Outline

1) Understand how decisions are made in programs.
2) Boolean values (how true and false is represented in C++)
3) Relational operators
4) Logical operators
5) If structure
6) If and else structure
7) Nested if structures
8) Switch structure

Understand how decisions are made in programs

- Everything that happens inside of a computer is a result of the operation of its circuits.
- A **circuit** quite simply allows one out of two choices (decision/selection) to be made depending on its inputs.
- Within all non-trivial computer programs, there are many decision that in essence are like the hardware circuits inside of the computer.
- Practically all computer programs, when modeled with flowcharts, demonstrate that branching (decision) occurs within their algorithms.
- This gives the user a more interactive experience and it allows the programs to actually solve problems.

Understand how true and false is represented in C++

- When decisions are made in a computer program, they are simply the result of a computation in which the final result is either true or false (boolean values).

- Computer programs do not have intuition or "gut feelings"; they simply make thousands of yes-or-no-like decisions (boolean decisions) at amazing speeds.

- The computer programmer must set up these decisions within the algorithm of a computer program to achieve the desired results.

Relational Operators

- **Relational operators** provide the tools with which programs make decisions with true and false evaluations.
- **Relational operators:**

  == equal to NOTE: this is two equals symbols next to each other, not to be confused with the assignment operator, =

  > greater than

  < less than

  >= greater than or equal to

  <= less than or equal to

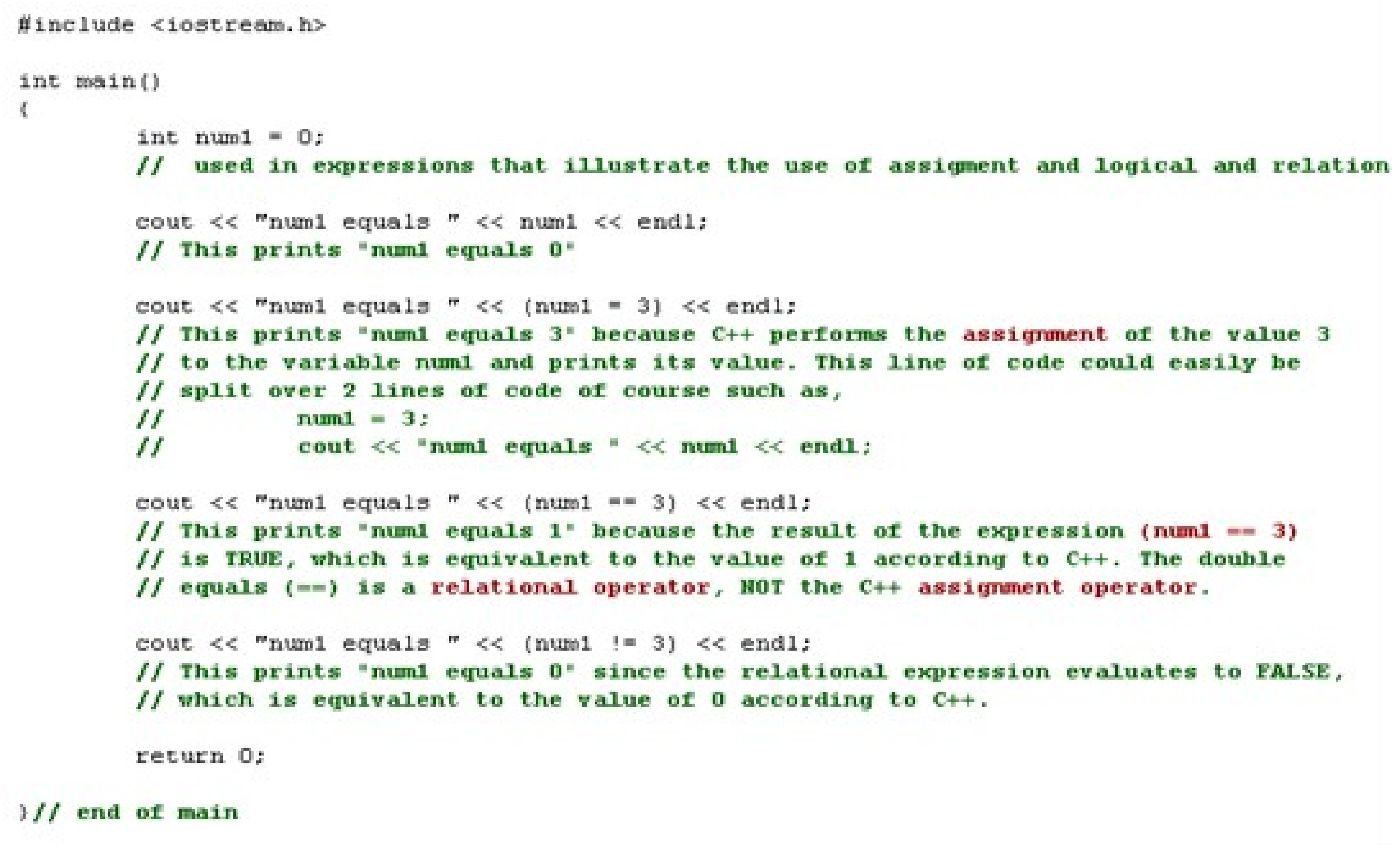
  != not equal to

Operator Precedence Chart

This table lists all the C++ operators in order of non-increasing precedence. An expression involving operators of equal precedence is evaluated according to the associatively of the operators.

OPERATOR(S)	DESCRIPTION(S)	ASSOCIATIVELY
::	Class scope resolution (binary)	left to right
::	Global scope (unary)	right to left
()	Function call	left to right
()	Value construction	left to right
[]	Array element reference	left to right
->	Pointer to class member reference	left to right
.	Class member reference	left to right
-, +	Unary minus and plus	right to left
>++, --	Increment and decrement	right to left
!, ~	Logical negation and one's complement	right to left
*, &	Pointer dereference (indirection) and address	right to left
sizeof	Size of an object	right to left
(type)	Type cast (coercion)	right to left
new, delete	Create free store object and destroy free store object	right to left
>*	Pointer to member selector	left to right
*	Pointer to member selector	left to right
*, /, %	Multiplication, division and modulus (remainder)	left to right
+, -	Addition and subtraction	left to right
<<, >>	Shift left and shift right	left to right
<, <=, >, >=	Less than, less than or equal, greater than, greater than or equal	left to right
==, !=	Equality and inequality	left to right
&	Bitwise AND	left to right
^	Bitwise XOR	left to right
|	Bitwise OR	left to right
&&	Logical AND	left to right
||	Logical OR	left to right
? :	Conditional expression	right to left
=, *=, /=, %=, +=, =, &=, ^=, |=, >>=, <<=	Assignment	right to left
,	Comma	left to right

Example of Program on Relational Operators



```
#include <iostream.h>

int main()
{
        int num1 = 0;
        //  used in expressions that illustrate the use of assigment and logical and relation

        cout << "num1 equals " << num1 << endl;
        // This prints "num1 equals 0"

        cout << "num1 equals " << (num1 = 3) << endl;
        // This prints "num1 equals 3" because C++ performs the assignment of the value 3
        // to the variable num1 and prints its value. This line of code could easily be
        // split over 2 lines of code of course such as,
        //          num1 = 3;
        //          cout << "num1 equals " << num1 << endl;

        cout << "num1 equals " << (num1 == 3) << endl;
        // This prints "num1 equals 1" because the result of the expression (num1 == 3)
        // is TRUE, which is equivalent to the value of 1 according to C++. The double
        // equals (==) is a relational operator, NOT the C++ assignment operator.

        cout << "num1 equals " << (num1 != 3) << endl;
        // This prints "num1 equals 0" since the relational expression evaluates to FALSE,
        // which is equivalent to the value of 0 according to C++.

        return 0;

}// end of main
```

Logical Operators

- When complex decisions must be coded into an algorithm, it may be necessary to **"chain together" a few relational expressions** (that use relational operators) called **compound relational expressions**.
- This is done with **logical operators** (also **called Boolean operators**).
- **Logical operators:**

  && is the logical **AND** operator

  || is the logical **OR** operator

  ! is the logical **NOT** operator

Logical Operators (cont…)

- Use these truth tables to determine the results of the logical operators.

AND			OR			NOT	
A	B	A && B	A	B	A || B	A	! A
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

- Logical operators may be mixed within evaluation statements but you must follow by their **order of operations**:
- **NOT** operator (!),
- **AND** operator (&&),
- **OR** operator (||)

1) You must remember that the **&& operation is always performed before the | |** operation because **&& is similar to multiplication** in normal arithmetic while **| | is similar to addition**.

Short-circuit Evaluation

Short-circuit evaluation in C++ allows the compiler to quickly evaluate a logical expression in some cases without having to completely evaluate each part of the expression.

Example:

```
inRange = ( i > 0 && i < 11);
```

If i is not greater than zero, the program doesn't even worry about checking whether i < 11 or not (since the whole right-hand expression can never be TRUE if i <= 0).

Control Expressions

- The following **control expressions** (as we will use within if statements below) have the resulting **value** of **TRUE** assuming that the **integer variables a, b, & c** have the values **a = 1, b = 2, & c = 3.**

```
(a < 2)
(a + 1 == b)
(a <= b && b < c)
(b < a || b < c)
(0 > b || b > 0 && b > 1)
(a > 0)
(a)
(-a)
(a = 3)
```

- Note that in the control expression (a = 3) where the assignment operator is sometimes accidentally used instead of the "double equals" relational operator, C++ evaluates the control expression as TRUE even if the variable a is previously assigned some other value.

Control Expressions (cont…)

- The control expression (a) where the variable a was previously assigned the value 1 is TRUE since **C++ considers any control expression that evaluates to a nonzero value to be TRUE**. That is **even if a was assigned the value -3, the control expression (a) would evaluate to TRUE.**

  The following **control expressions** evaluate to **TRUE** where **a = 1, b = 2, and c = 3:**

```
(b)
(c + a)
(2 * b)
(-30 + c)
(0 - b)
```

Control Expressions (cont…)

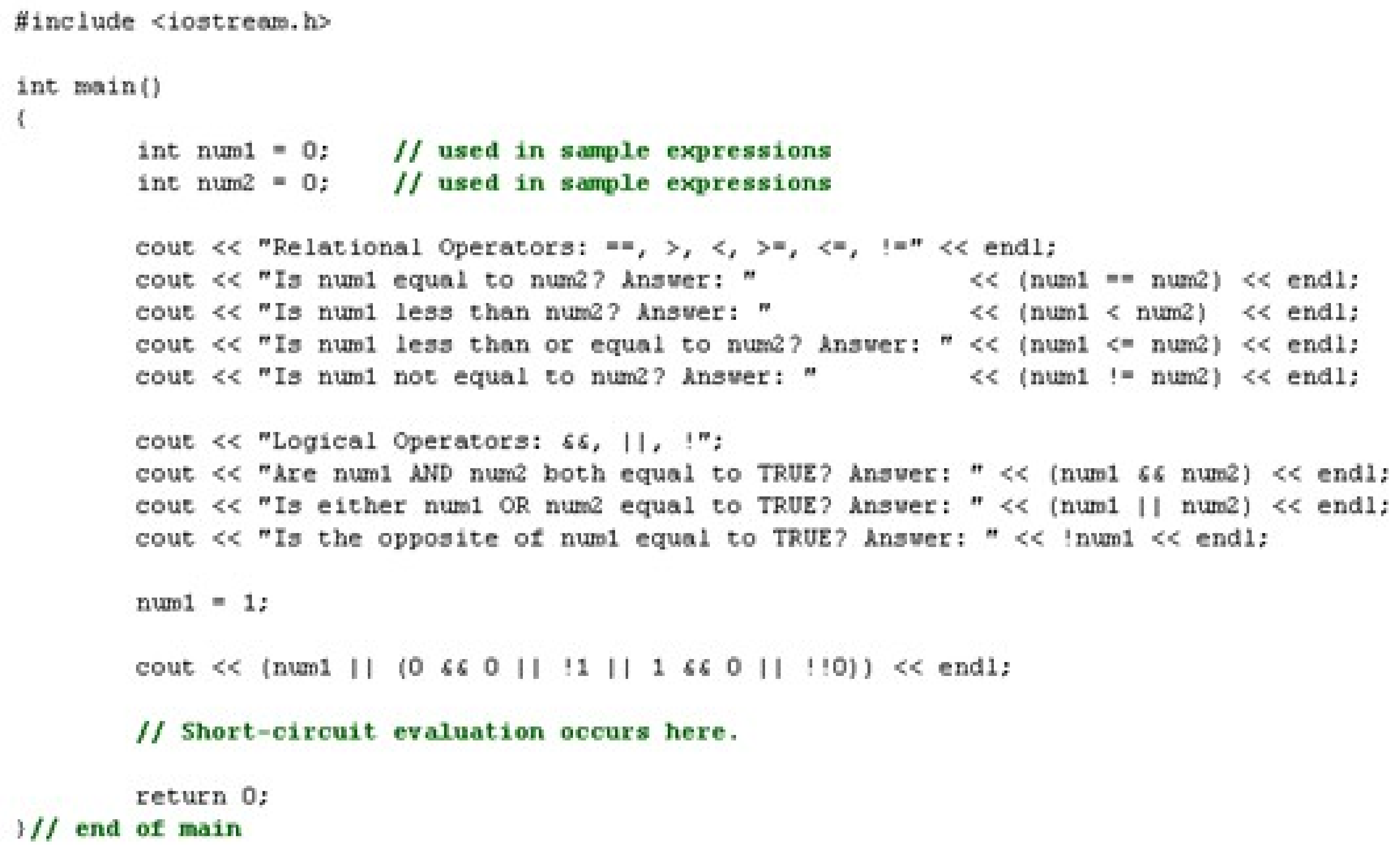
- While the following **control expressions** evaluate to **FALSE** where **a = 1, b = 2, and c = 3**:

```
(a - 1)
(!a)
(0 * c)
(c - a + b)
```

- Note that the **! symbol** (the logical NOT operator) **changes a TRUE to a FALSE.**
- The following **control expressions** have the resulting value of **FALSE** assuming that the **integer variables a, b, & c** have the values **a = 1, b = 2, & c = 3**.

```
(a > 1)

(b == 1)

(a > 0 && a < -9)

(c % 3)

(a > 0 + 4)
```

Example of Program



```
#include <iostream.h>

int main()
{
        int num1 = 0;    // used in sample expressions
        int num2 = 0;    // used in sample expressions

        cout << "Relational Operators: ==, >, <, >=, <=, !=" << endl;
        cout << "Is num1 equal to num2? Answer: "                 << (num1 == num2) << endl;
        cout << "Is num1 less than num2? Answer: "                << (num1 < num2)  << endl;
        cout << "Is num1 less than or equal to num2? Answer: "    << (num1 <= num2) << endl;
        cout << "Is num1 not equal to num2? Answer: "             << (num1 != num2) << endl;

        cout << "Logical Operators: &&, ||, !";
        cout << "Are num1 AND num2 both equal to TRUE? Answer: " << (num1 && num2) << endl;
        cout << "Is either num1 OR num2 equal to TRUE? Answer: " << (num1 || num2) << endl;
        cout << "Is the opposite of num1 equal to TRUE? Answer: " << !num1 << endl;

        num1 = 1;

        cout << (num1 || (0 && 0 || !1 || 1 && 0 || !!0)) << endl;

        // Short-circuit evaluation occurs here.

        return 0;
}// end of main
```

If Control Structure

- Practically all computer languages have decision (aka selection) structures. In C++, the **if structure** is a **one-way selection structure**. Example,

```
if (i == 3)
{
cout << "The value of i is 3";
}
```

- All if structures use a **control expression** to determine if the code in the braces is to be executed. In the example above, (`number == 3`) is the control expression. Note the use of the "double equals" relational operator `==` and not the assignment operator `=`.

- Be sure to place the semicolon at the end of each statement within the braces, which can contain many executable statements.

Compound Relational Expression

- Realize that you **must use the AND operator (&&) to form a compound relational expression**. The the following syntax is **INVALID**:

```
if (0 < number < 10)
{
      cout << number << " is greater than 0 but less than 10." << endl;
}
```

- Rather the following syntax must be used:

```
if (0 < number && number < 10)
{
      cout << number << " is greater than 0 but less than 10." << endl;
}
```

Common Mistakes of If Control Structure

- According to our **Coding Standards,** you must use curly braces around the body of an if structure even if there is only one statement.
- Avoid the following common mistakes that are made with if statements:

1) **Use the "double equals" symbol (i.e. comparison operator) rather than the assignment operator in control expressions:**

```
int num = 0:

if (num == 5)
{
cout << "hello world" << endl;
}
```

which would not display "hello world" rather than:

```
int num = 0;

if (num = 5)
{
cout << "hello world" << endl;
}
```

which would assign the value 5 to the variable num and then display "hello world“.

Common Mistakes of If Control Structure (cont…)

2) Avoid using an unnecessary semicolon after a control expression:

```
if (num > 0);
{
cout << "num is positive" << endl;
}
```

which would cause the phrase "num is positive" to be displayed even if num is negative.

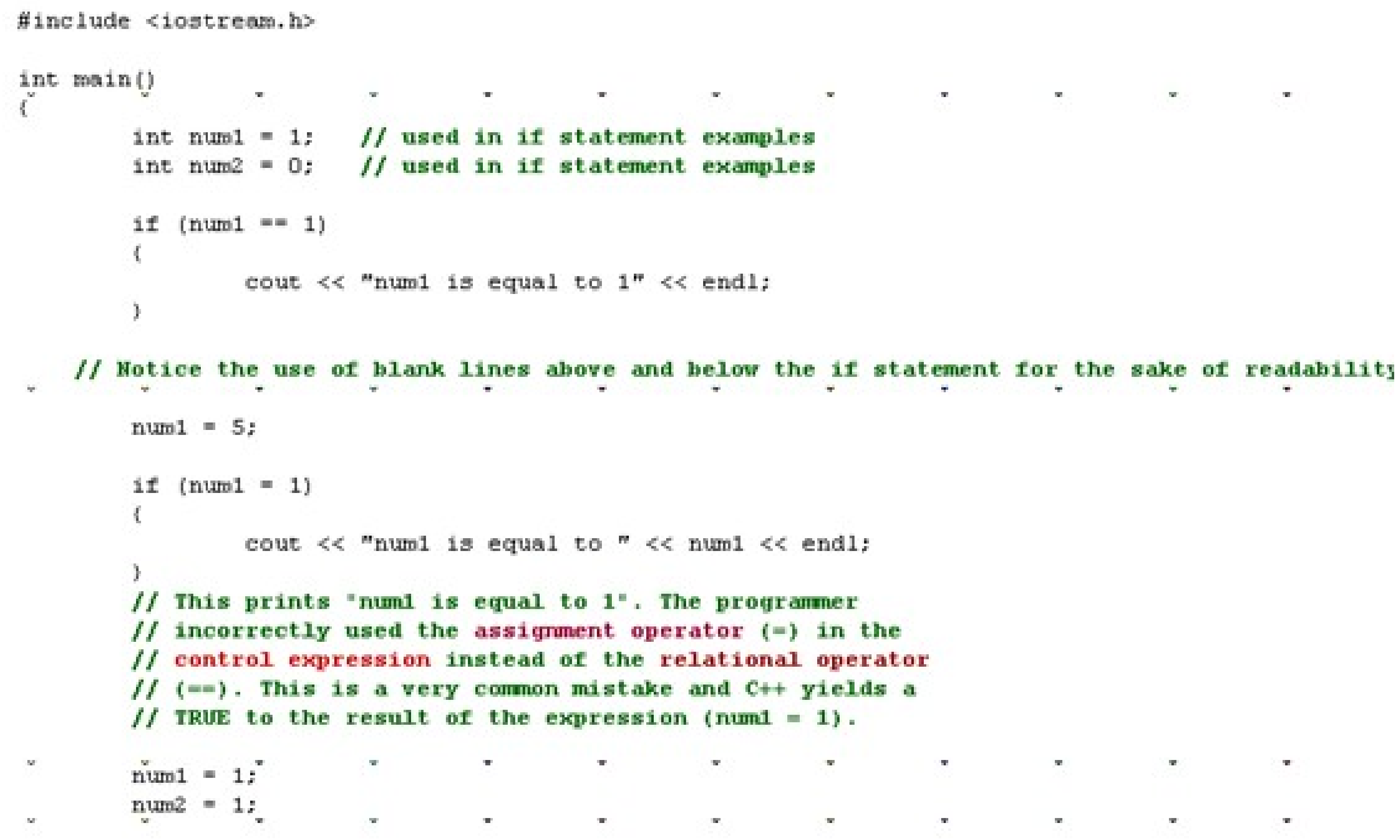
Common Mistakes of `If` Control Structure (cont...)

- To be a safe programmer that uses good style, **always use curly braces around the body of an if statement, even if the body of the if statement only contains one statement.** In the following code segment.

```
if (num > 0)
     cout << "num is positive" << endl;
     cout << "num is not zero" << endl;
```

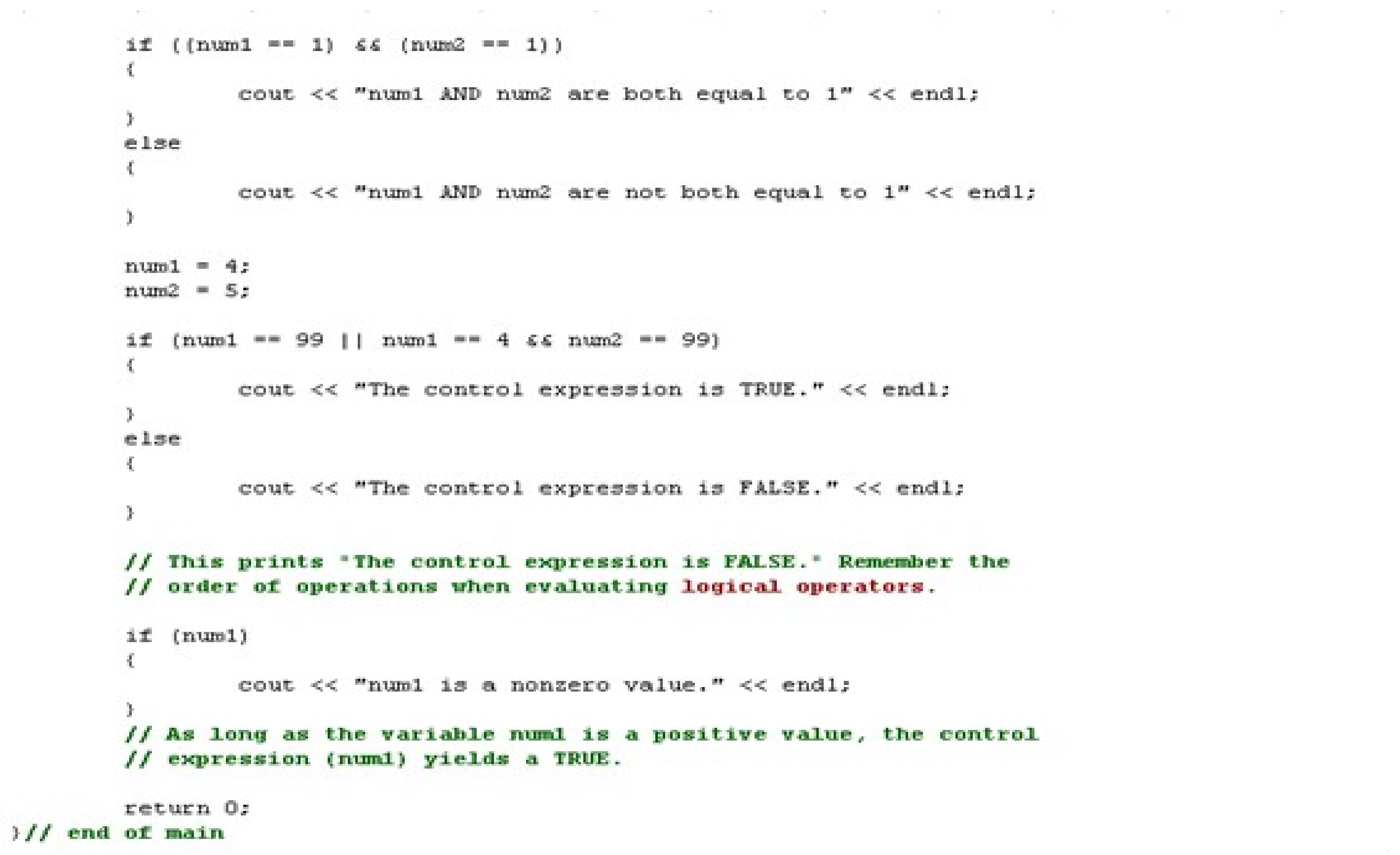
- **The body of the if statement is only considered to be the first cout statement. The second statement would execute even if num is equal to zero.** The indentation of the second statement has no effect on the way the compiler interprets this code.

Example of Program



```
#include <iostream.h>

int main()
{
        int num1 = 1;    // used in if statement examples
        int num2 = 0;    // used in if statement examples

        if (num1 == 1)
        {
                cout << "num1 is equal to 1" << endl;
        }

    // Notice the use of blank lines above and below the if statement for the sake of readabilit

        num1 = 5;

        if (num1 = 1)
        {
                cout << "num1 is equal to " << num1 << endl;
        }
        // This prints 'num1 is equal to 1'. The programmer
        // incorrectly used the assignment operator (=) in the
        // control expression instead of the relational operator
        // (==). This is a very common mistake and C++ yields a
        // TRUE to the result of the expression (num1 = 1).

        num1 = 1;
        num2 = 1;
```

Example of Program (cont…)



```
        if ((num1 == 1) && (num2 == 1))
        {
                cout << "num1 AND num2 are both equal to 1" << endl;
        }
        else
        {
                cout << "num1 AND num2 are not both equal to 1" << endl;
        }

        num1 = 4;
        num2 = 5;

        if (num1 == 99 || num1 == 4 && num2 == 99)
        {
                cout << "The control expression is TRUE." << endl;
        }
        else
        {
                cout << "The control expression is FALSE." << endl;
        }

        // This prints "The control expression is FALSE." Remember the
        // order of operations when evaluating logical operators.

        if (num1)
        {
                cout << "num1 is a nonzero value." << endl;
        }
        // As long as the variable num1 is a positive value, the control
        // expression (num1) yields a TRUE.

        return 0;
}// end of main
```

`If-else` Control Structure

- The **`if-else` structure** is considered to be a **two-way selection structure**, since either the block of code after the "if" part will be executed or the block of code after the "else" part will be executed.

- The **`"if"`** **part is executed if the control expression is TRUE while the** **`"else"`** **part is executed if the "if" part is false**, guaranteeing one part of the expression to be executed or the other. **Do not place a control expression after the else keyword.**

- **Example:**

```
if ( i < 0)
{
cout << "The number is negative.";
}
else
{
cout << "The number is zero or positive.";
}
```

- Don't forget to include **boundary cases** when you code control expressions. That is, be sure to **use <= or >= instead of < or >,** when necessary.

`If-else if` Control Structure

- The `if-else if` **structure** is considered to be a **multiple selection structure**, since it **compare more than two selections**.
- You can use an `if-else if` statement as well. The else clause is optional just as it is with an if statement.
- **Example:**

```
if (number > 10)
{
     cout << number << " is greater than 10." << endl;
}
else if (number <= 9 && number >= 6)
{
     cout << number << " is greater than 5 and less than 10." << endl;
}
else
{
     cout << number << "must be less than 6." << endl;
}
```

Nested `if` Control Structure

- **`If`** structures and i**`f-else`** structures can be **nested** within one another in order to model complex decision structures. Be sure to use the braces and semicolons properly when coding such structures. Also, be sure to rigorously check the logic of your algorithm since it is quite easy to overlook possible errors.

- **Example:**

The if statement that tests for divisibility by 5 is located inside of the if statement that tests for divisibility by 3 therefore it is considered to be a **nested if statement**.

```
if (number % 3 == 0)
{
cout << number << " is divisible by 3." << endl;

if (number % 5 == 0)
{
cout << number << " is divisible by 3 and 5." << endl;
}
}
```

Multiple Selection Difficulties

Example:

```
...
...
if (total_test_score >=0 && total_test_score < 50)
cout << "You are a failure - you must study much harder.\n";
else if (total_test_score < 60)
cout << "You have just scraped through the test.\n";
else if (total_test_score < 80)
cout << "You have done quite well.\n";
else if (total_test_score <= 100)
cout << "Your score is excellent - well done.\n";
else
cout << "Incorrect score - must be between 0 and 100.\n";
...
...
```

- Because multiple selection can sometimes be difficult to follow, C++ provides an alternative method of handling this concept, called the ***`switch`*** statement. **`"Switch"`** statements can be used when several options depend on the value of a single variable or expression.

`Switch` Control Structure

- The **`switch`** **structure** is a **multiple-selection** **structure** and it allows you to model even more complicated decision statements than a two-way if-else structure allows. It chooses one of the "cases" depending on the result of the control expression.

- Only integer or single-character data types may be used as control expressions **in switch statements**.

- **Example:**

```
switch(characterEntered)
     {
     case 'A':
     cout << "You entered an A";
     break;

     case 'B':
     cout << "You entered a B";
     break;

     default:
     cout << "Illegal entry";
     break;
     }
```

Switch Control Structure (cont...)

- Example where the variable **menuChoice** is an **int variable**. Note that single quotes are **NOT** used around the integer values in each case.

```
switch (menuChoice)
{
case 1:
     cout << "You entered menu choice #1";
     break;
case 2:
     cout << "You entered menu choice #2";
     break;
case 3:
     cout << "You entered menu choice #3";
     break;
default:
     cout << "You failed to enter a valid menu choice";
     break;
}
```

Switch Control Structure (cont...)

- The **default case**, if present, will result if neither of the prior cases apply. Also, note the use of the **break statement**, **which is necessary to make sure that the program evaluates the next executable statement AFTER the switch structure**.

- For example, in the following example, the output would be "You entered a B", even if characterEntered = 'A':

```
switch(characterEntered)
{
case 'A':
case 'B':
cout << "You entered a B";
break;

default:
cout << "Illegal entry";
break;
}
```

Switch Control Structure (cont...)

However, in the following example, the would be **no output if characterEntered = 'A':**

```
switch(characterEntered)
{
case 'A':
break;
case 'B':
cout << "You entered a B";
break;

default:
cout << "Illegal entry";
break;
}
```

Switch Control Structure (cont…)

The break statement must be used within each case if you do not want following cases to evaluate once one case is found. **When the break statement is executed within a switch, C++ will execute the next statement outside of the switch statement**. However, sometimes it is desirable not to use the break statement in a particular case. **Example:**

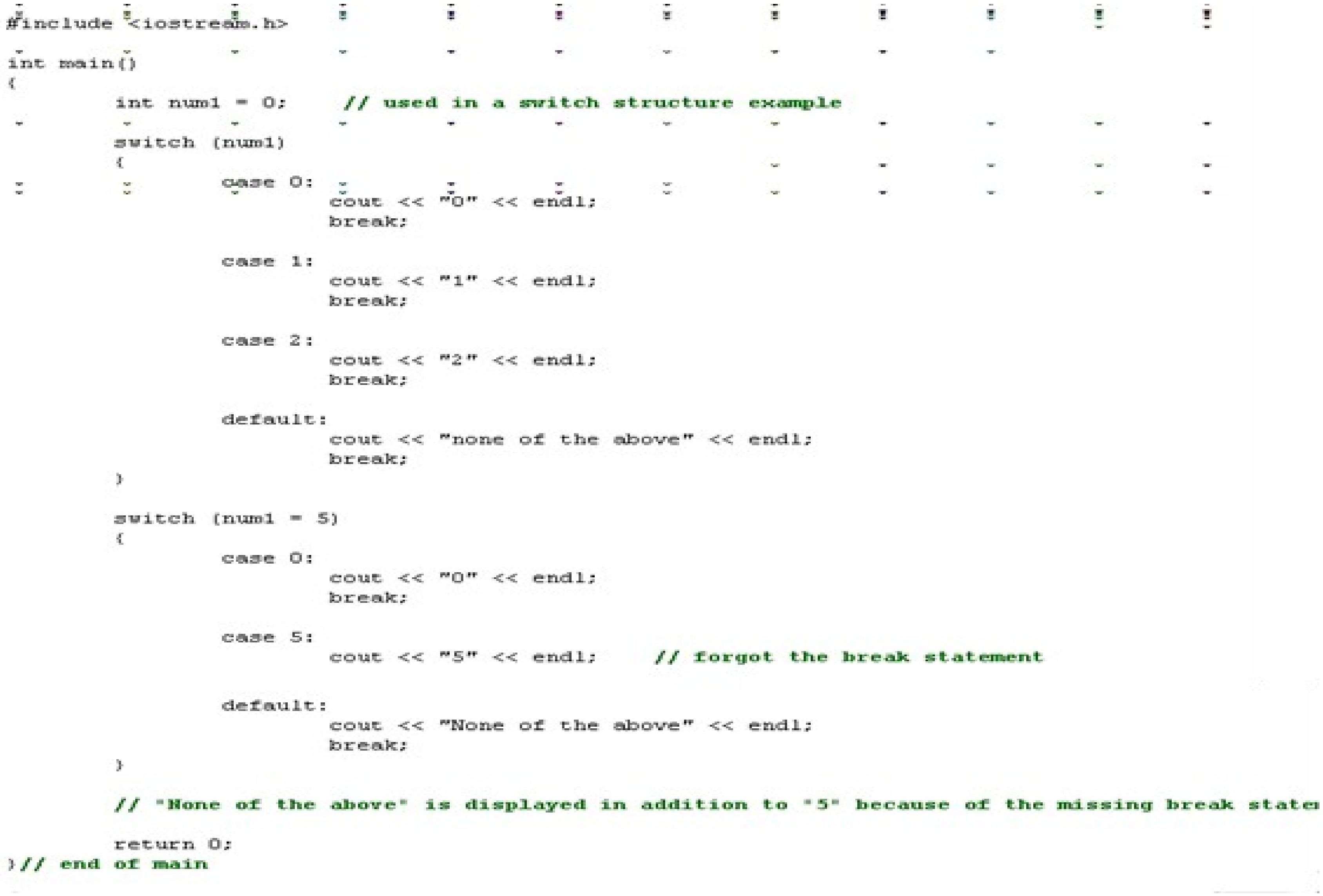
```
switch (donationLevel)
{
case 1:
   cout << "You donated over $1,000, therefore you will receive a "   << "complimentary Wyomissing pocket watch." << endl;
case 2:
   cout << "You donated over $500, therefore you can have a "
           << "complimentary Wyomissing 14K gold pen & pencil set. "<<endl;
case 3:
   cout << "You donated over $250, therefore you will receive a
            << "complimentary Wyomissing necktie." << endl;
case 4:
   cout << "You donated over $100, therefore you will receive a "
           << "complimentary Wyomissing bumper sticker. " << endl;
   break;
default:
   cout << "You are cheap and ungenerous." << endl;
   break;
   }
```

Switch Control Structure (cont...)

You can save redundancy in your code by placing the cases next to each other as in the following example.

```
switch (number)
{
case 1:
case 3:
case 5:
case 7:
case 9:
         cout << number << " is an even number." << endl;
         break;
case 2:
case 4:
case 6:
case 8:
         cout << number << " is an odd number. " << endl;
         break;
default:
         cout << number << " is not a value between or including 1 and 9."<<endl;
         break;
}
```

Example of Program



```
#include <iostream.h>

int main()
{
        int num1 = 0;     // used in a switch structure example

        switch (num1)
        {
                case 0:
                        cout << "0" << endl;
                        break;

                case 1:
                        cout << "1" << endl;
                        break;

                case 2:
                        cout << "2" << endl;
                        break;

                default:
                        cout << "none of the above" << endl;
                        break;
        }

        switch (num1 = 5)
        {
                case 0:
                        cout << "0" << endl;
                        break;

                case 5:
                        cout << "5" << endl;     // forgot the break statement

                default:
                        cout << "None of the above" << endl;
                        break;
        }

        // "None of the above" is displayed in addition to "5" because of the missing break state

        return 0;
}// end of main
```