Computer Programming

Basic Control Flow - Decisions

Adapted from C++ for Everyone and Big C++ by Cay Horstmann, John Wiley & Sons □

Objectives

- To be able to implement decisions using if statements
- To learn how to compare integers, floating-point numbers, and strings
- To understand the Boolean data type
- To develop strategies for validating user input

The `if` Statement

Decision making

(a necessary thing in non-trivial programs)

The **`if`** *statement*

allows a program to carry out different actions depending on the nature of the data being processed

The `if` Statement

The **`if`** statement is used to implement a decision.

- When a condition is fulfilled,
  one set of statements is executed.
- Otherwise,
  another set of statements is executed.

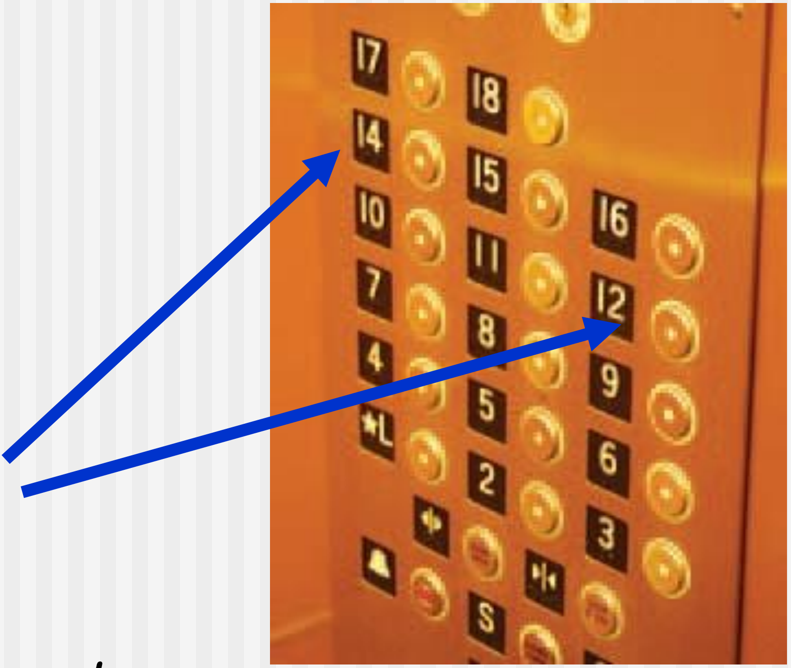
The `if` Statement

```
if it’s quicker to the candy mountain,
    we’ll go that way
else
    we go that way
```

The `if` Statement

The thirteenth floor!

It's missing!



Of course floor 13 is not usually left empty, it is simply called floor 14.

The `if` Statement

We must write the code to control the elevator.

How can we skip the 13th floor?

We will model a person choosing a floor by getting input from the user:

```
int floor;
cout << "Floor: ";
cin >> floor;
```

The `if` Statement

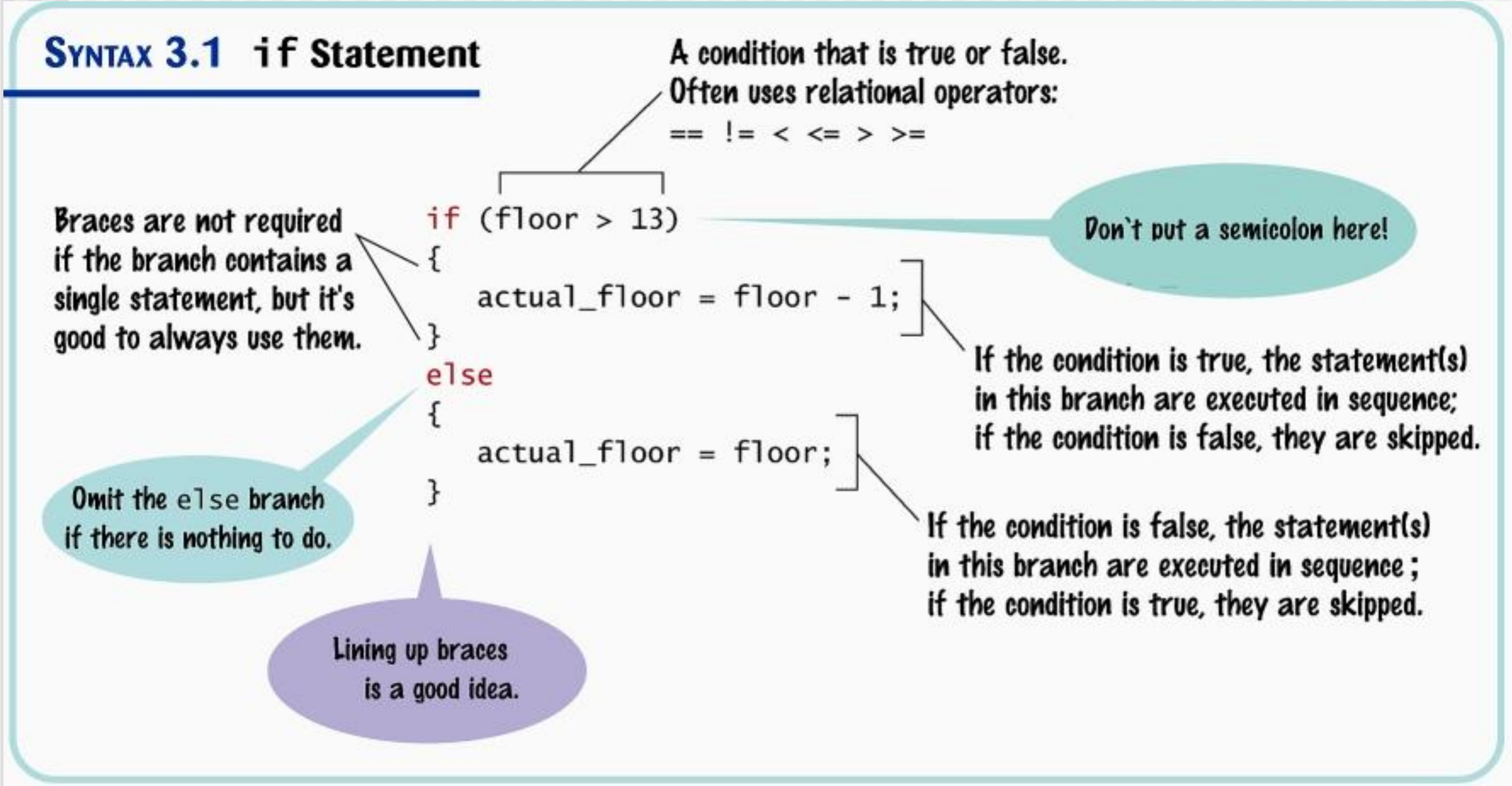
If the user inputs 20,
the program must set the actual floor to 19.

Otherwise,
we simply use the supplied floor number.

We need to decrement the input only under a certain condition:

```
int actual_floor;
if (floor > 13)
{
    actual_floor = floor - 1;
}
else
{
    actual_floor = floor;
}
```

The `if` Statement



SYNTAX 3.1 if Statement

```
if (floor > 13)
{
   actual_floor = floor - 1;
}
else
{
   actual_floor = floor;
}
```

A condition that is true or false.
Often uses relational operators:
== != < <= > >=
Don't put a semicolon here!
Braces are not required if the branch contains a single statement, but it's good to always use them.
If the condition is true, the statement(s) in this branch are executed in sequence; if the condition is false, they are skipped.
If the condition is false, the statement(s) in this branch are executed in sequence ; if the condition is true, they are skipped.
Omit the else branch if there is nothing to do.
Lining up braces is a good idea.

The `if` Statement

Sometimes, it happens that there is nothing to do in the **`else`** branch of the statement.

So don’t write it.

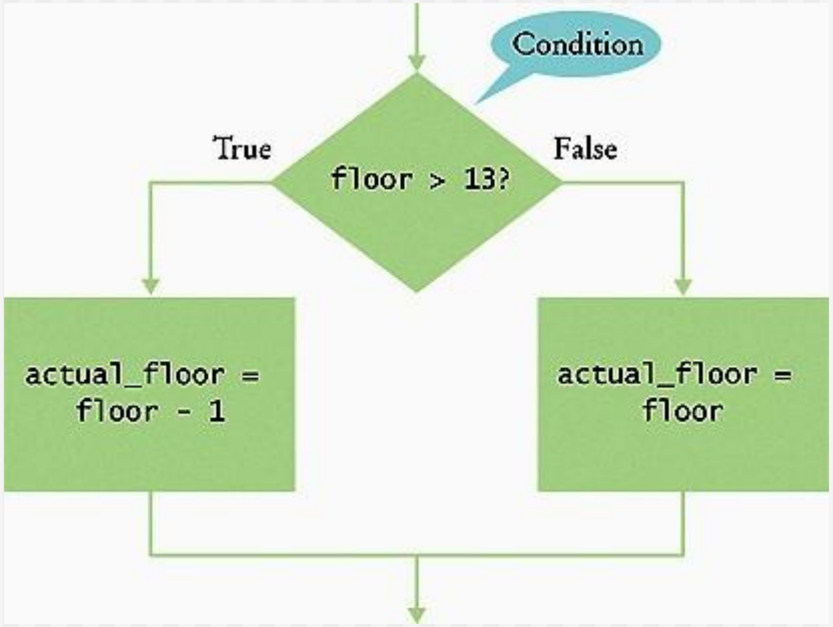
Here is another way to write this code:

We only need to decrement when the floor is greater than 13.

We can set **`actual_floor`** before testing:

```
int actual_floor = floor;
if (floor > 13)
{
    actual_floor--;
} // No else needed
```

The `if` Statement – The Flowchart



The `if` Statement – A Complete Elevator Program

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

int main()
{
   int floor;
   cout << "Floor: ";
   cin >> floor;
   int actual_floor;
   if (floor > 13)
   {
      actual_floor = floor - 1;
   }
   else
   {
      actual_floor = floor;
   }

   cout << "The elevator will travel to the actual floor "
      << actual_floor << endl;

   return 0;
}
```

The `if` Statement – Brace Layout

- Making your code easy to read is good practice.
- Lining up braces vertically helps.

```
if (floor > 13)
{
    floor--;
}
```

- As long as the ending brace clearly shows what it is closing, there is no confusion.

Some programmers prefer this style —it saves a vertical line in the code.

```
if (floor > 13) {
      floor--;
}
```

The `if` Statement – Always Use Braces

When the body of an **`if`** statement consists of a single statement, you need not use braces:

```
if (floor > 13)
   floor--;
```

However, it is a good idea to always include the braces:

- the braces makes your code easier to read, and
- you are less likely to make errors such as …

The `if` Statement – Common Error – The Do-nothing Statement

Can you see the error?

```
if (floor > 13) ;   ERROR
{
    floor--;
}
```

The `if` Statement – Indent when Nesting

Block-structured code has the property that *nested* statements are indented by one or more levels.

```
int main()
{
   int floor;
   ...
   if (floor > 13)
   {
      floor--;
   }
   ...
   return 0;
}
```

0 1 2

Indentation level

The `if` Statement – Removing Duplication

```
if (floor > 13)
{
        actual_floor = floor - 1;
        cout << "Actual floor: " << actual_floor <<
endl;
}
else
{
        actual_floor = floor;
        cout << "Actual floor: " << actual_floor <<
endl;
}
```

Do you find anything curious in this code?

The `if` Statement – Removing Duplication

```
if (floor > 13)
{
      actual_floor = floor - 1;
      }
else
{
      actual_floor = floor;
}
cout << "Actual floor: " << actual_floor << endl;
```

You should remove this duplication.

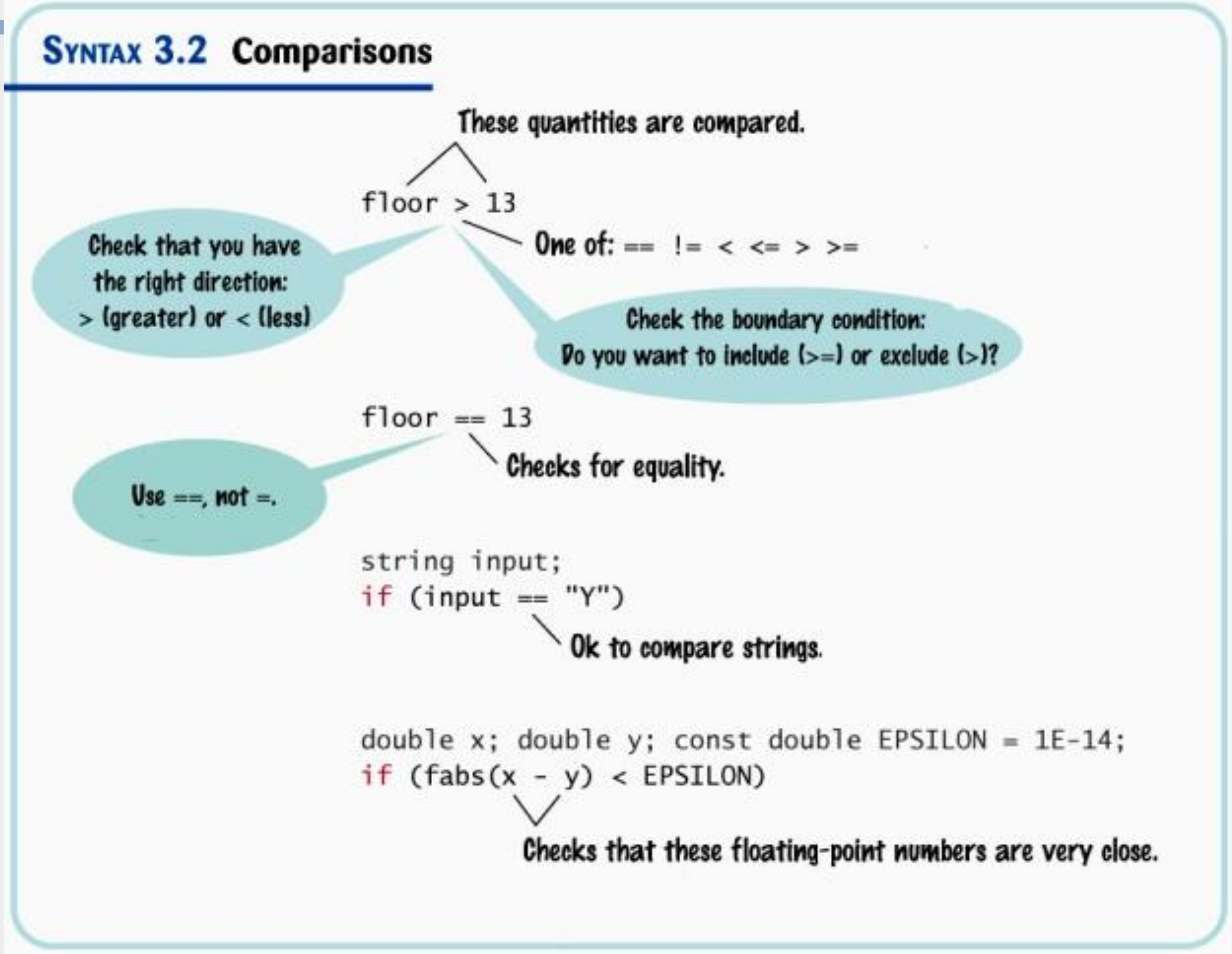
Relational Operators

Relational operators

```
 <    >=
 >    <=
==    !=
```

are used to compare numbers and strings.

Relational Operators



SYNTAX 3.2 Comparisons

These quantities are compared.

```
floor > 13
```

One of: == != < <= > >=

Check that you have the right direction: > (greater) or < (less)

Check the boundary condition: Do you want to include (>=) or exclude (>)?

```
floor == 13
```

Checks for equality.

Use ==, not =.

```
string input;
if (input == "Y")
```

Ok to compare strings.

```
double x; double y; const double EPSILON = 1E-14;
if (fabs(x - y) < EPSILON)
```

Checks that these floating-point numbers are very close.

Relational Operators

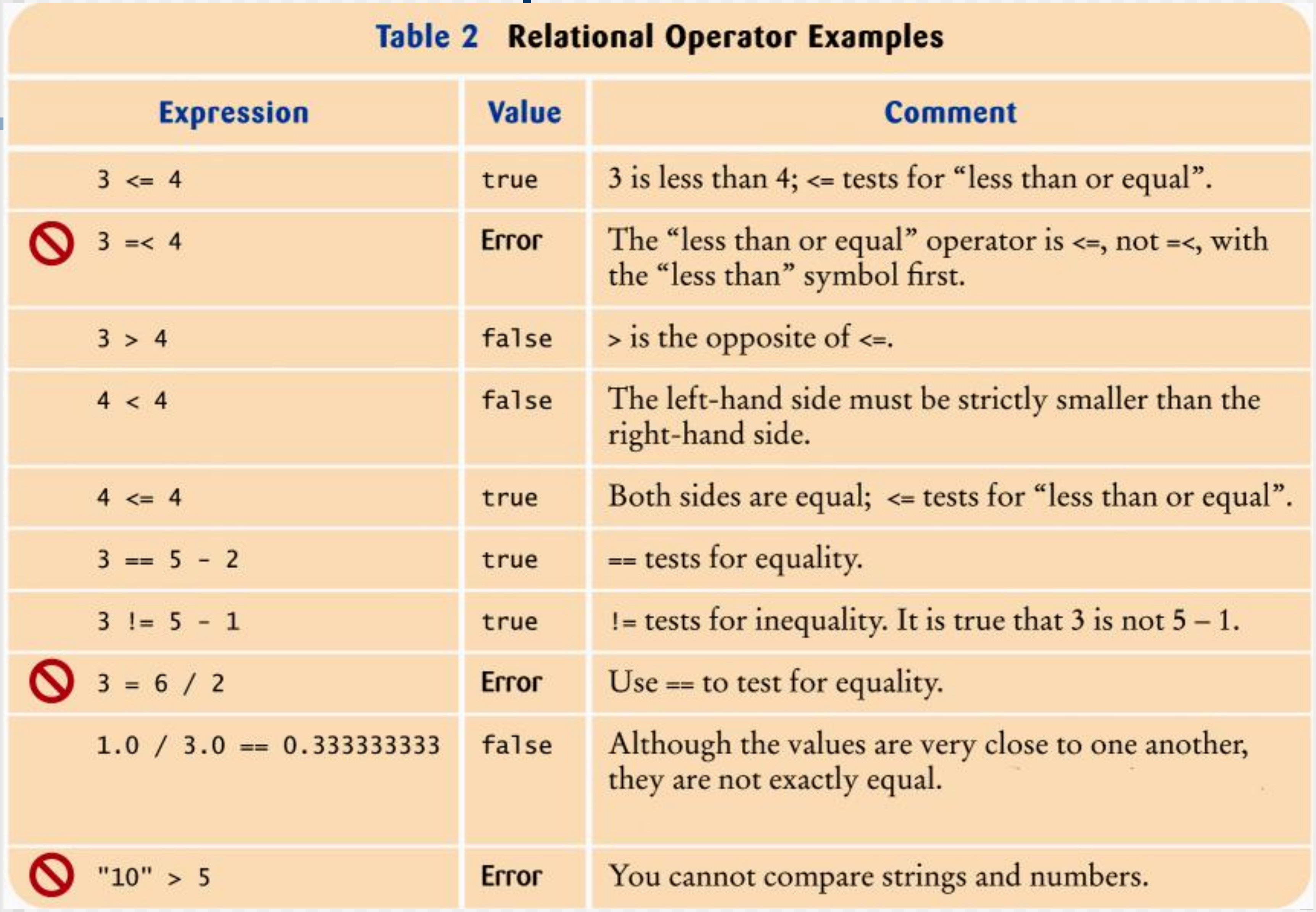


Table 2 Relational Operator Examples

Expression	Value	Comment
`3 <= 4`	`true`	3 is less than 4; <= tests for "less than or equal".
🚫 `3 =< 4`	Error	The "less than or equal" operator is <=, not =<, with the "less than" symbol first.
`3 > 4`	`false`	> is the opposite of <=.
`4 < 4`	`false`	The left-hand side must be strictly smaller than the right-hand side.
`4 <= 4`	`true`	Both sides are equal; <= tests for "less than or equal".
`3 == 5 - 2`	`true`	== tests for equality.
`3 != 5 - 1`	`true`	!= tests for inequality. It is true that 3 is not 5 – 1.
🚫 `3 = 6 / 2`	Error	Use == to test for equality.
`1.0 / 3.0 == 0.333333333`	`false`	Although the values are very close to one another, they are not exactly equal.
🚫 `"10" > 5`	Error	You cannot compare strings and numbers.

Relational Operators – Some Notes

Computer keyboards do not have keys for:

≥

≤

≠

but these operators:

```
>=
<=
!=
```

look similar (and you can type them).

Relational Operators – Some Notes

The == operator is initially confusing to beginners.

In C++, = already has a meaning, namely <u>assignment</u>

The == operator denotes equality testing:

```
floor = 13; // Assign 13 to floor
if (floor == 13)
//Test whether floor equals 13
```

You can compare strings as well:

```
if (input == "Quit") ...
```

Relational Operators – Common Error `==` vs. `=`

Furthermore, in C and C++ assignments have values. The value of the assignment expression **`floor = 13`** is 13.

These two features conspire to make a horrible pitfall:

```
if (floor = 13) …
```

is legal C++.

Relational Operators – Common Error == vs. =

You must remember:

Use **==** *in*side tests.

Use **=** *out*side tests.

Multiple Alternatives

Multiple `if` statements can be combined to evaluate complex decisions.

Multiple Alternatives

How would we write code to deal with Richter scale values?

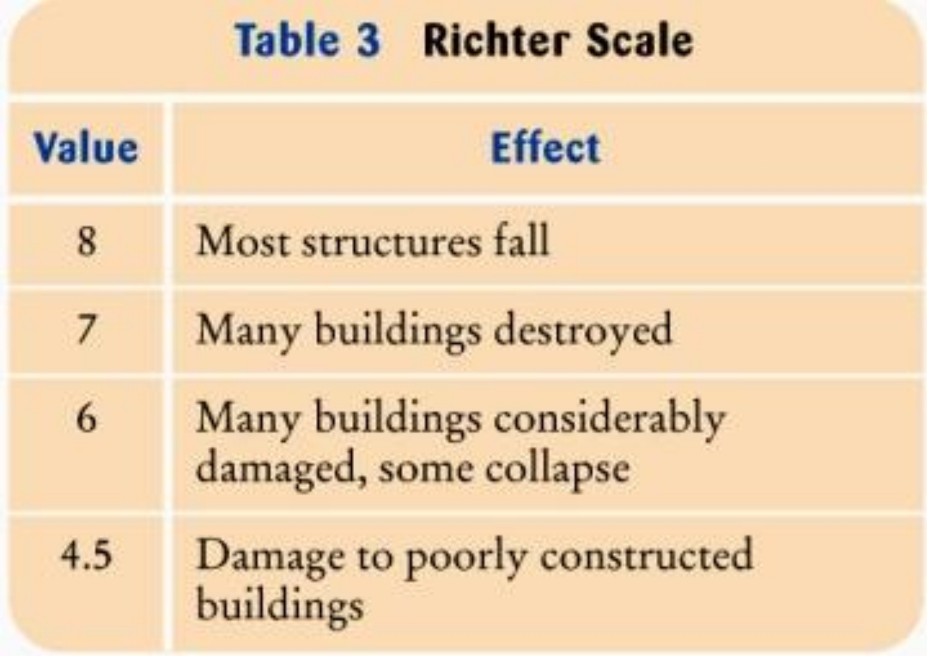


Table 3 Richter Scale

Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

Multiple Alternatives

In this case, there are five branches:

one each for the four descriptions of damage,

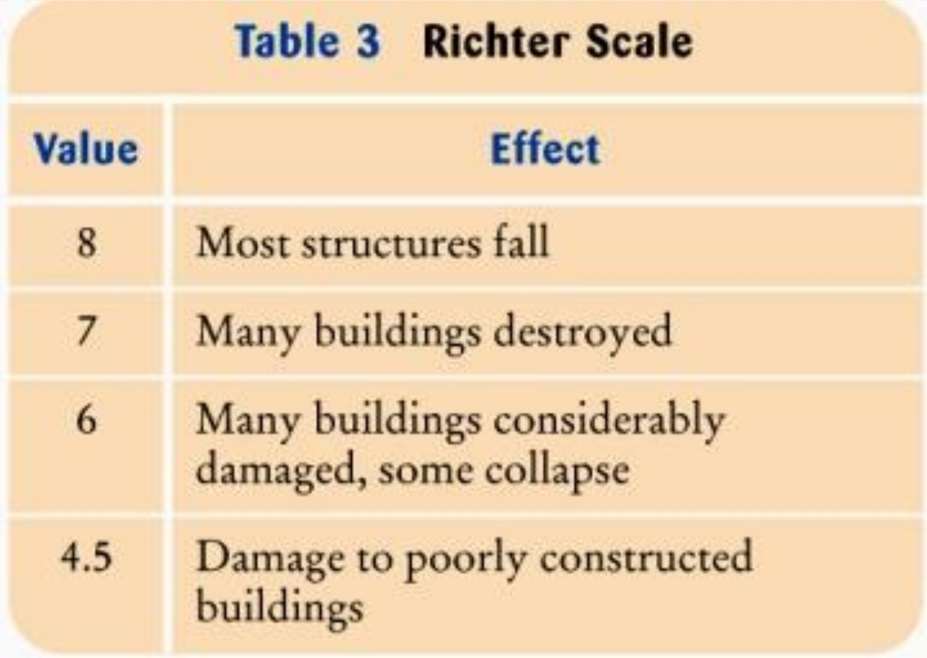
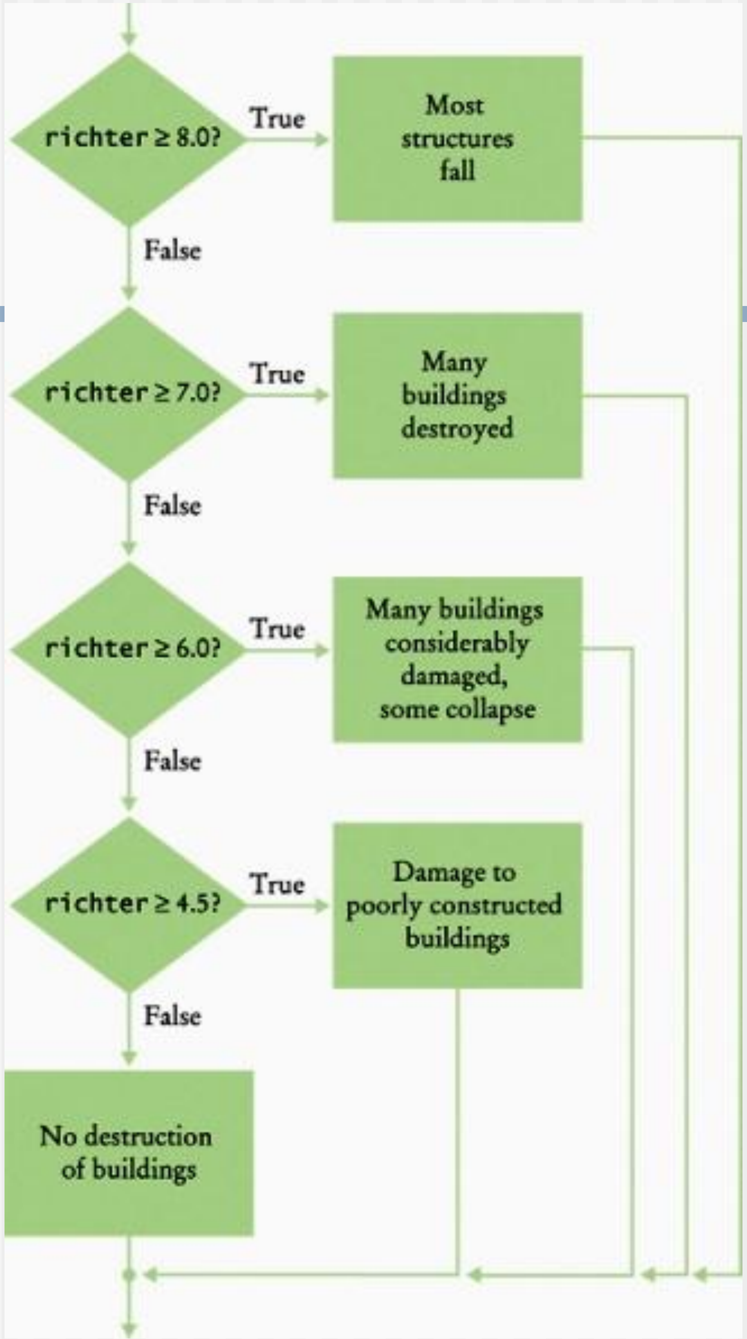


Table 3 Richter Scale

Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

and one for no destruction.

Richter flowchart



Multiple Alternatives

```
if (richter >= 8.0)
{
   cout << "Most structures fall";
}
else if (richter >= 7.0)
{
   cout << "Many buildings destroyed";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
{
   cout << "Damage to poorly constructed buildings";
}
else
{
   cout << "No destruction of buildings";
}
. . .
```

Multiple Alternatives

```
if (richter >= 8.0)
{
   cout << "Most structures fall";
}
else if (richter >= 7.0)
{
   cout << "Many buildings destroyed";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
{
   cout << "Damage to poorly constructed buildings";
}
else
{
   cout << "No destruction of buildings";
}
. . .
```

If a test is false,

Multiple Alternatives

```
if (   false    )
{
   cout << "Most structures fall";
}
else if (richter >= 7.0)
{
   cout << "Many buildings destroyed";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
{
   cout << "Damage to poorly constructed buildings";
}
else
{
   cout << "No destruction of buildings";
}
. . .
```

If a test is false,

Multiple Alternatives

```
if (richter >= 8.0)
{
   cout << "Most structures fall";
}
else if (richter >= 7.0)
{
   cout << "Many buildings destroyed";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
{
   cout << "Damage to poorly constructed buildings";
}
else
{
   cout << "No destruction of buildings";
}
. . .
```

If a test is false, that block is skipped

Multiple Alternatives

```
if (richter >= 8.0)
{
   cout << "Most structures fall";
}
else if (richter >= 7.0)
{
   cout << "Many buildings destroyed";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
{
   cout << "Damage to poorly constructed buildings";
}
else
{
   cout << "No destruction of buildings";
}
. . .
```

**If a test is false,
that block is skipped and
the next test is made.**

Multiple Alternatives

```
if (richter >= 8.0)
{
   cout << "Most structures fall";
}
else if (richter >= 7.0)
{
   cout << "Many buildings destroyed";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
{
   cout << "Damage to poorly constructed buildings";
}
else
{
   cout << "No destruction of buildings";
}
. . .
```

As soon as one of the four tests succeeds,

Multiple Alternatives

```
if (richter >= 8.0)
{
   cout << "Most structures fall";
}
else if (   true    )
{
   cout << "Many buildings destroyed";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
{
   cout << "Damage to poorly constructed buildings";
}
else
{
   cout << "No destruction of buildings";
}
. . .
```

As soon as one of the four tests succeeds,

Multiple Alternatives

```
if (richter >= 8.0)
{
   cout << "Most structures fall";
}
else if (richter >= 7.0)
{
   cout << "Many buildings destroyed";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
{
   cout << "Damage to poorly constructed buildings";
}
else
{
   cout << "No destruction of buildings";
}
. . .
```

As soon as one of the four tests succeeds, that block is executed, displaying the result,

Multiple Alternatives

```
if (richter >= 8.0)
{
   cout << "Most structures fall";
}
else if (richter >= 7.0)
{
   cout << "Many buildings destroyed";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
else if (richter >= 4.5)
{
   cout << "Damage to poorly constructed buildings";
}
else
{
   cout << "No destruction of buildings";
}
. . .
```

As soon as one of the four tests succeeds, that block is executed, displaying the result,

and no further tests are attempted.

Multiple Alternatives – Wrong Order of Tests

Because of this execution order,
when using multiple **`if`** statements,
pay attention to the order of the conditions.

Multiple Alternatives – Wrong Order of Tests

```
if (richter >= 4.5)    // Tests in wrong order
{
   cout << "Damage to poorly constructed buildings";
}
else if (richter >= 6.0)
{
   cout << "Many buildings considerably damaged, some collapse";
}
else if (richter >= 7.0)
{
   cout << "Many buildings destroyed";
}
else if (richter >= 8.0)
{
   cout << "Most structures fall";
}
. . .
```

Suppose the value of `richter` is 7.1, this test is true!

and that block is executed (Oh no!),

The `switch` Statement

- To implement sequence of if/else that compares a value against several constant alternatives.
- Every branch of switch must be terminated by a `break` instruction.
  - If missing, execution falls through the next branch.
- All branches test the same value.
- The controlling expression `switch` must always return either `bool` value, one of the integer data types or a character.

The `switch` Statement

```
int digit;
…
switch(digit)
{
        case 1: digit_name = “one”; break;
        case 2: digit_name = “two”; break;
        case 3: digit_name = “three”; break;
        default: digit_name = “”; break;

}
```

Nested Branches

It is often necessary to include an if statement inside another.

Such an arrangement is called a nested set of statements.

Nested Branches – Taxes

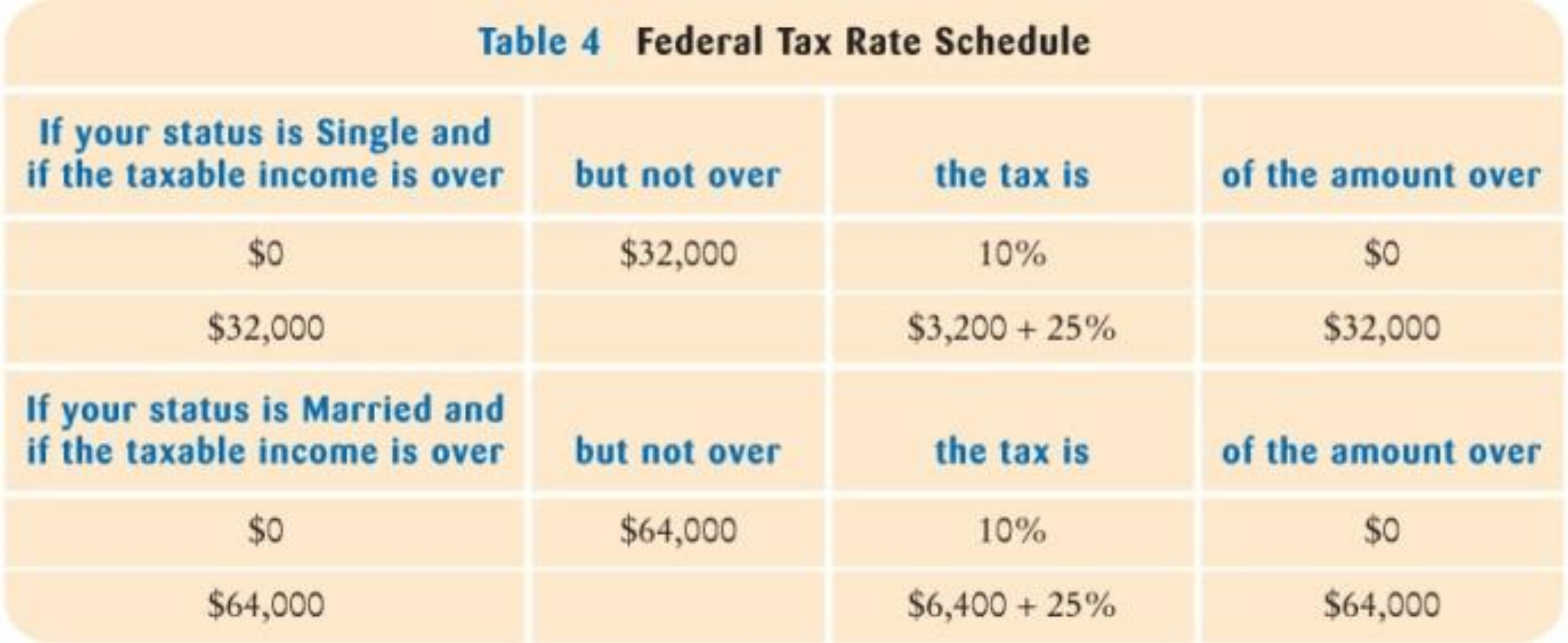


Table 4 Federal Tax Rate Schedule

If your status is Single and if the taxable income is over	but not over	the tax is	of the amount over
$0	$32,000	10%	$0
$32,000		$3,200 + 25%	$32,000
If your status is Married and if the taxable income is over	**but not over**	**the tax is**	**of the amount over**
$0	$64,000	10%	$0
$64,000		$6,400 + 25%	$64,000

Tax brackets for single filers:
- from $0 to $32,000
- above $32,000

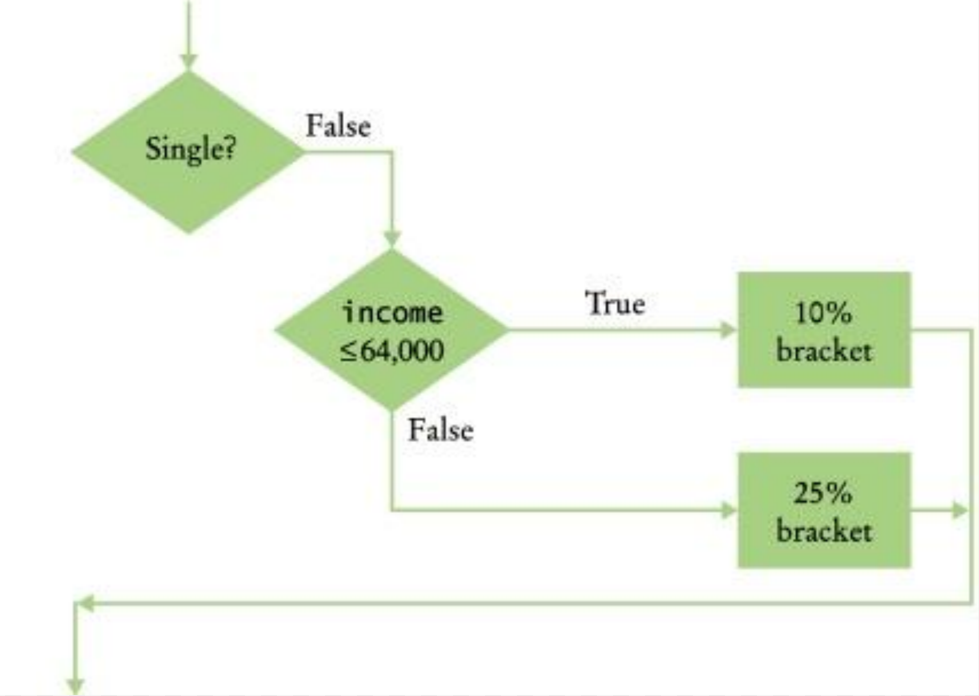
then tax depends on income

Tax brackets for married filers:
- from $0 to $64,000
- above $64,000

then tax depends on income

Nested Branches – Taxes

…a different *nested* **`if`** for using their figures.



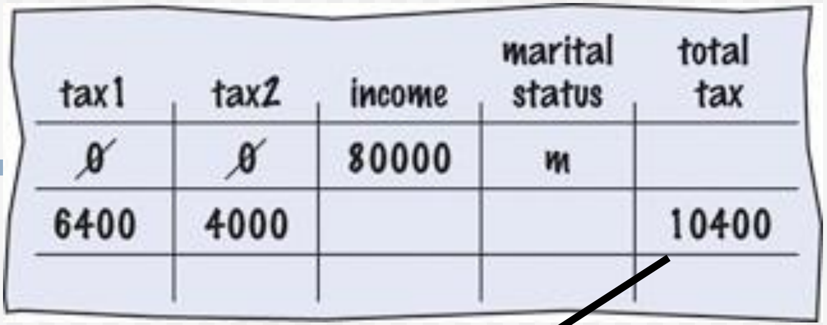
Hand Tracing/Desk Checking

A very useful technique for understanding whether a program works correctly is called hand-tracing.

You simulate the program’s activity on a sheet of paper.

You can use this method with pseudocode or C++ code.

Hand Tracing



tax1	tax2	income	marital status	total tax
~~0~~	~~0~~	80000	m	
6400	4000			10400

```
    double total_tax = tax1 + tax2;

    cout << "The tax is $" << total_tax << endl;
    return 0;
}
```

The Dangling `else` Problem

When an **`if`** statement is nested inside another **`if`** statement, the following error may occur.
Can you find the problem with the following?

```
double shipping_charge = 5.00; //$5 inside continental U.S.

if (country == "USA")
   if (state == "HI")
      shipping_charge = 10.00; // Hawaii is more expensive
else      // Pitfall!
   shipping_charge = 20.00;  // As are foreign shipments
```

The Dangling `else` Problem

The indentation level seems to suggest that the **`else`** is grouped with the test **`country == "USA"`**.

Unfortunately, that is not the case.

The compiler ignores all indentation and matches the **`else`** with the preceding **`if`**.

```
double shipping_charge = 5.00;       // $5 inside continental U.S.

if (country == "USA")
   if (state == "HI")
      shipping_charge = 10.00;       // Hawaii is more expensive
else
   shipping_charge = 20.00;          // As are foreign shipments
```

The Dangling `else` Problem – The Solution

So, is there a solution to the dangling **`else`** problem.

Of, course.

You can put one statement in a block. (Aha!)

The Dangling `else` Problem – The Solution

```
double shipping_charge = 5.00;
                        // $5 inside continental
  U.S.
if (country == "USA")
{
   if (state == "HI")
      shipping_charge = 10.00;
                        // Hawaii is more expensive
}
else
   shipping_charge = 20.00;
                        // As are foreign shipments
```

Boolean Variables and Operators

- Sometimes you need to evaluate a logical condition in one part of a program and use it elsewhere.
- To store a condition that can be **`true`** or **`false`**, you use a Boolean variable.

Boolean Variables and Operators

Boolean variables are named after the mathematician George Boole.

Two values, eh?

like “yes” and “no”

Boolean Variables and Operators

- In C++, the **`bool`** *data type* represents the Boolean type.
- Variables of type **`bool`** can hold exactly two values, denoted **`false`** and **`true`**.
- These values are **not** strings.
- There values are *definitely* **not** integers;

  they are special values, just for Boolean variables.

Boolean Variables

Here is a definition of a Boolean variable, initialized to `false`:

```
bool failed = false;
```

It can be set by an intervening statement so that you can use the value later in your program to make a decision:

```
// Only executed if failed has
// been set to true
if (failed)
{
        ...
}
```

Boolean Operators

At this geyser in Iceland, you can see ice, liquid water, and steam.

Boolean Operators

- Suppose you need to write a program that processes temperature values, and you want to test whether a given temperature corresponds to liquid water.
  - At sea level, water freezes at 0 degrees Celsius and boils at 100 degrees.
- Water is liquid if the temperature is greater than zero and less than 100.
- This not a simple test condition.

Boolean Operators

- When you make complex decisions, you often need to combine Boolean values.
- An operator that combines Boolean conditions is called a Boolean operator.
- Boolean operators take one or two Boolean values or expressions and combine them into a resultant Boolean value.

The Boolean Operator && (and)

In C++, the **`&&`** operator (called *and*) yields **`true`** only when *both* conditions are **`true`**.

```
if (temp > 0 && temp < 100)
{
   cout << "Liquid";
}
```

If **`temp`** is within the range, then both the left-hand side *and* the right-hand side are **`true`**, making the whole expression’s value **`true`**.

In all other cases, the whole expression’s value is **`false`**.

The Boolean Operator || (or)

The **||** operator (called *or*) yields the result **true** if at least one of the conditions is **true**.

- This is written as two adjacent vertical bar symbols.

```
if (temp <= 0 || temp >= 100)
{
   cout << "Not liquid";
}
```

If *either* of the expression is **true**,
the whole expression is **true**.

The only way “Not liquid” won’t appear is if *both* of the expressions are **false**.

The Boolean Operator ! (not)

Sometimes you need to invert a condition with the logical *not* operator.

The `!` operator takes a single condition and evaluates to **`true`** if that condition is **`false`** and to **`false`** if the condition is **`true`**.

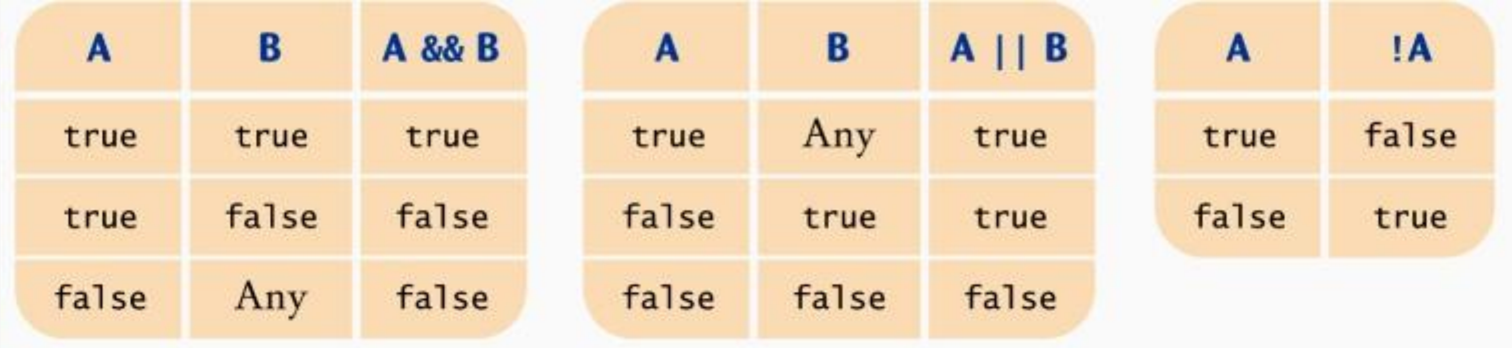
```
if (!frozen) { cout << "Not frozen"; }
```

“Not frozen” will be written only when frozen contains the value **`false`**.

`!false` is **`true`**.

Boolean Operators

This information is traditionally collected into a table called a *truth table*:



A	B	A && B
true	true	true
true	false	false
false	Any	false

A	B	A || B
true	Any	true
false	true	true
false	false	false

A	!A
true	false
false	true

where A and B denote **`bool`** variables or Boolean expressions.

Boolean Operators – Some Examples

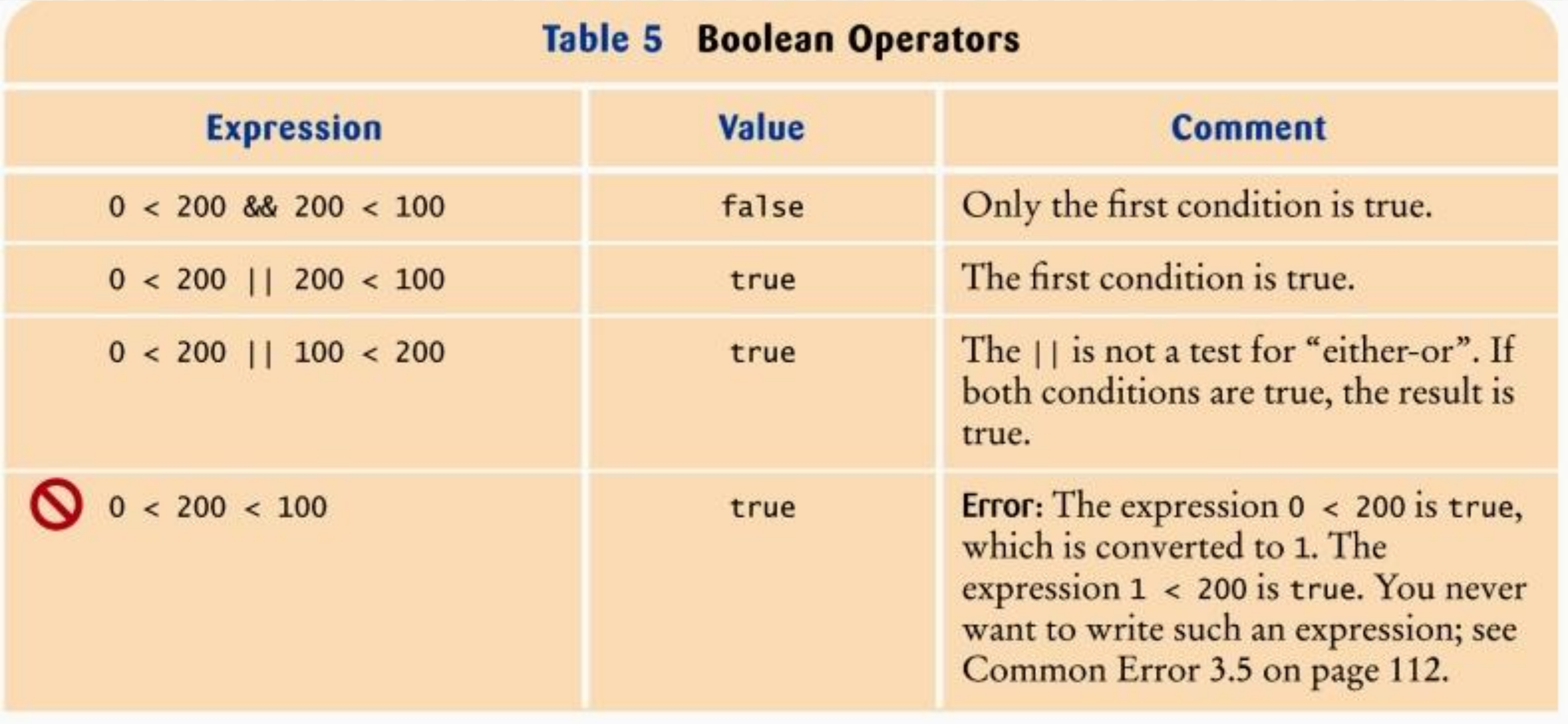


Table 5 Boolean Operators

Expression	Value	Comment
`0 < 200 && 200 < 100`	`false`	Only the first condition is true.
`0 < 200 || 200 < 100`	`true`	The first condition is true.
`0 < 200 || 100 < 200`	`true`	The `||` is not a test for "either-or". If both conditions are true, the result is true.
🚫 `0 < 200 < 100`	`true`	**Error:** The expression `0 < 200` is `true`, which is converted to 1. The expression `1 < 200` is `true`. You never want to write such an expression; see Common Error 3.5 on page 112.

Boolean Operators – Some Examples

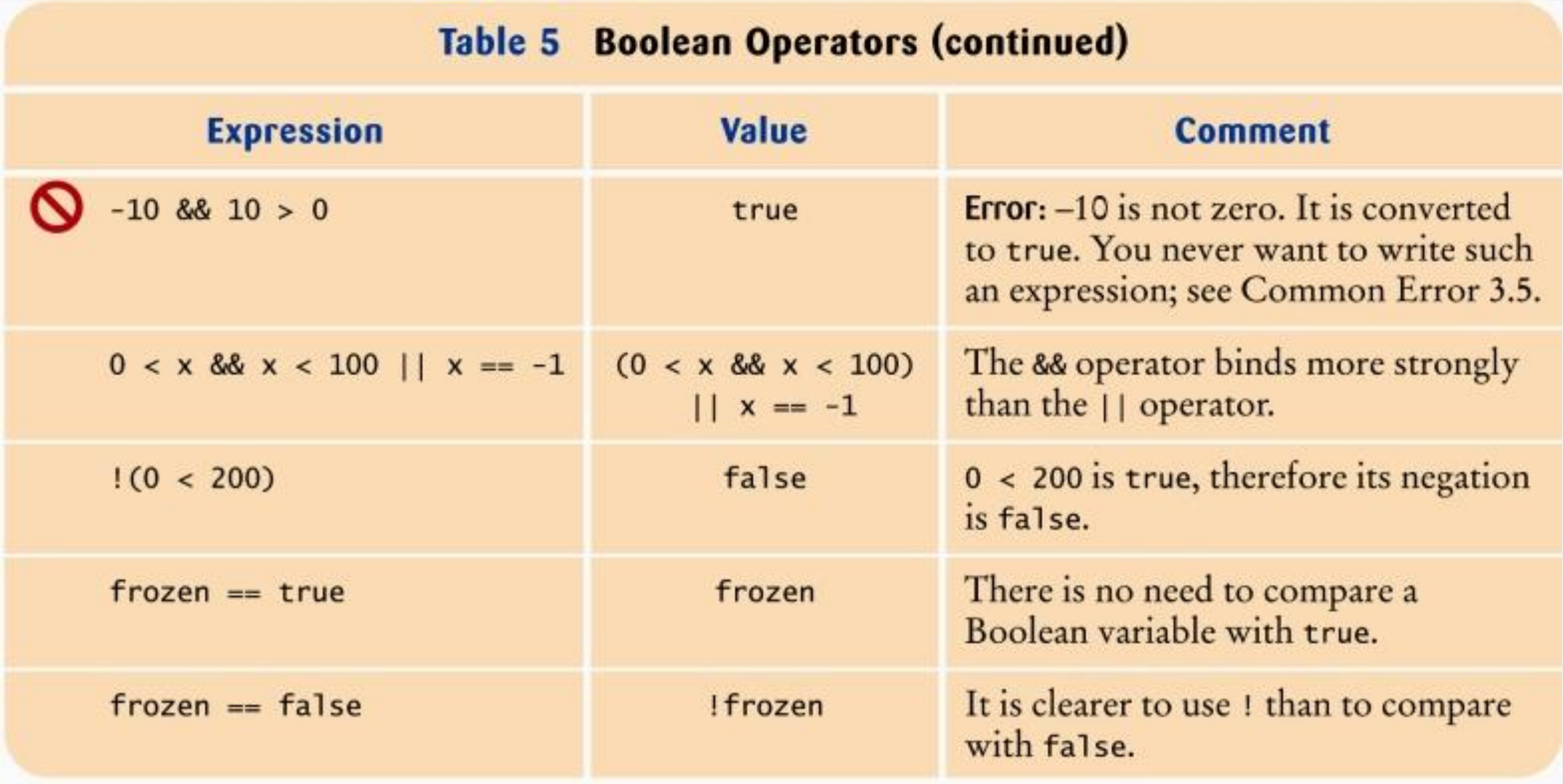


Table 5 Boolean Operators (continued)

Expression	Value	Comment
⊘ `-10 && 10 > 0`	`true`	**Error:** –10 is not zero. It is converted to `true`. You never want to write such an expression; see Common Error 3.5.
`0 < x && x < 100 || x == -1`	`(0 < x && x < 100) || x == -1`	The `&&` operator binds more strongly than the `||` operator.
`!(0 < 200)`	`false`	`0 < 200` is `true`, therefore its negation is `false`.
`frozen == true`	`frozen`	There is no need to compare a Boolean variable with `true`.
`frozen == false`	`!frozen`	It is clearer to use `!` than to compare with `false`.

Combining Multiple Relational Operators

Consider the expression

```
    if (0 <= temp <= 100)…
```

This looks just like the mathematical test:

$$\mathbf{0 \le temp \le 100}$$

Unfortunately, it is not.

Combining Multiple Relational Operators

`if (0 <= temp <= 100)`…

The first half, **`0 <= temp`**, is a *test*.

The outcome **`true`** or **`false`**,
depending on the value of **`temp`**.

Combining Multiple Relational Operators

```
          true
if (                <= 100)...
          false
```

The outcome of that test (**`true`** or **`false`**) is then compared against 100.

This seems to make no sense.

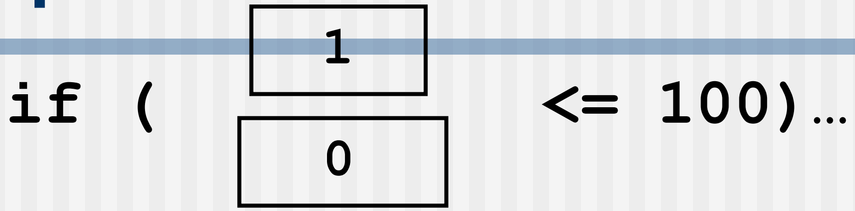
Can one compare truth values and floating-point numbers?

Combining Multiple Relational Operators

```
          true
if (                <= 100)…
          false
```

Is **`true`** larger than 100 or not?

Combining Multiple Relational Operators



```
          1
if (              <= 100)…
          0
```

Unfortunately, to stay compatible with the C language, C++ converts **`false`** to 0 and **`true`** to 1.

Combining Multiple Relational Operators

```
            1
if (                <= 100)...
            0
```

Unfortunately, to stay compatible with the C language, C++ converts **`false`** to 0 and **`true`** to 1.

Therefore, the expression will always evaluate to **`true`**.

Combining Multiple Relational Operators

Another common error, along the same lines, is to write

```
if (x && y > 0) ... // Error
```

instead of

```
if (x > 0 && y > 0) ...//correct
```

(x and y are ints)

An && or an ||?

It is quite common that the individual conditions are nicely set apart in a bulleted list, but with little indication of how they should be combined.

Our tax code is a good example of this.

An && or an ||?

Consider these instructions for filing a tax return.

You are of single filing status if any one of the following is true:

- *You were never married.*
- *You were legally separated or divorced on the last day of the tax year.*
- *You were widowed, and did not remarry.*

Is this an `&&` or an `||` situation?

Since the test passes if any one of the conditions is `true`, you must combine the conditions with the `or` operator.

An && or an ||?

Elsewhere, the same instructions:

You may use the status of married filing jointl
if all five of the following conditions are true:

- *Your spouse died less than two years ago and you did not remarry.*
- *You have a child whom you can claim as dependent.*
- *That child lived in your home for all of the tax year.*
- *You paid over half the cost of keeping up your home for this child.*
- *You filed a joint return with your spouse the year he or she died.*

`&&` or an **`||`?**

Because all of the conditions must be **`true`** for the test to pass, you must combine them with an and.

Input Validation with `if` Statements

You, the C++ programmer, doing Quality Assurance

(by hand!)

Input Validation with `if` Statements

- Assume that the elevator panel has buttons labeled 1 through 20 (*but not 13!*).
- The following are illegal inputs:
  - The number 13
  - Zero or a negative number
  - A number larger than 20
  - A value that is not a sequence of digits, such as five
- In each of these cases, we will want to give an error message and exit the program.

Input Validation with `if` Statements

It is simple to guard against an input of 13:

```
if (floor == 13)
{
   cout << "Error: "
      << " There is no thirteenth floor."
        << endl;
   return 1;
}
```

Input Validation with `if` Statements

The statement:

```
    return 1;
```

immediately exits the **`main`** function and therefore terminates the program.

It is a convention to return with the value 0 if the program completes normally, and with a non-zero value when an error is encountered.

Input Validation with `if` Statements

To ensure that the user doesn’t enter a number outside the valid range:

```
if (floor <= 0 || floor > 20)
{
   cout << "Error: "
      << " The floor must be between 1 and 20."
      << endl;
   return 1;
}
```

Input Validation with `if` Statements

Dealing with input that is not a valid integer is a more difficult problem.

What if the user does not type a number in response to the prompt?

‘F’ ‘o’ ‘u’ ‘r’ is not an integer response.

Input Validation with `if` Statements

When

```
    cin >> floor;
```

is executed, and the user types in a bad input, the integer variable **`floor`** is not set.

Instead, the input stream **`cin`** is set to a failed state.

Input Validation with `if` Statements

You can call the **`fail`** member function to test for that failed state.

So you can test for bad user input this way:

```
if (cin.fail())
{
   cout << "Error: Not an integer." <<
  endl;
   return 1;
}
```

Chapter Summary

1. The **`if`** statement allows a program to carry out different actions depending on the nature of the data to be processed.
2. Relational operators (**`< <= > >= ==!=`**) are used to compare numbers and strings.
3. Multiple **`if`** statements can be combined to evaluate complex decisions.F
4. When using multiple **`if`** statements, pay attention to the order of the conditions.
5. The Boolean type **`bool`** has two values, **`false`** and **`true`**.
6. C++ has two Boolean operators that combine conditions: **`&&`** (and) and **`||`** (or).
7. To invert a condition, use the **`!`** (not) operator.
8. Use the **`fail`** function to test whether stream input has failed.