Chapter Three: Decisions I

C++ for Everyone by Cay Horstmann

Slides by Evan Gallagher & Nikolay Kirov

Lecture Goals

- To be able to implement decisions using **`if`** statements
- To learn how to compare integers, floating-point numbers, and strings

The `if` Statement

Decision making

(a necessary thing in non-trivial programs)

The `if` Statement

We aren’t lost!

We just haven’t decided which way to go … yet.

The `if` Statement

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

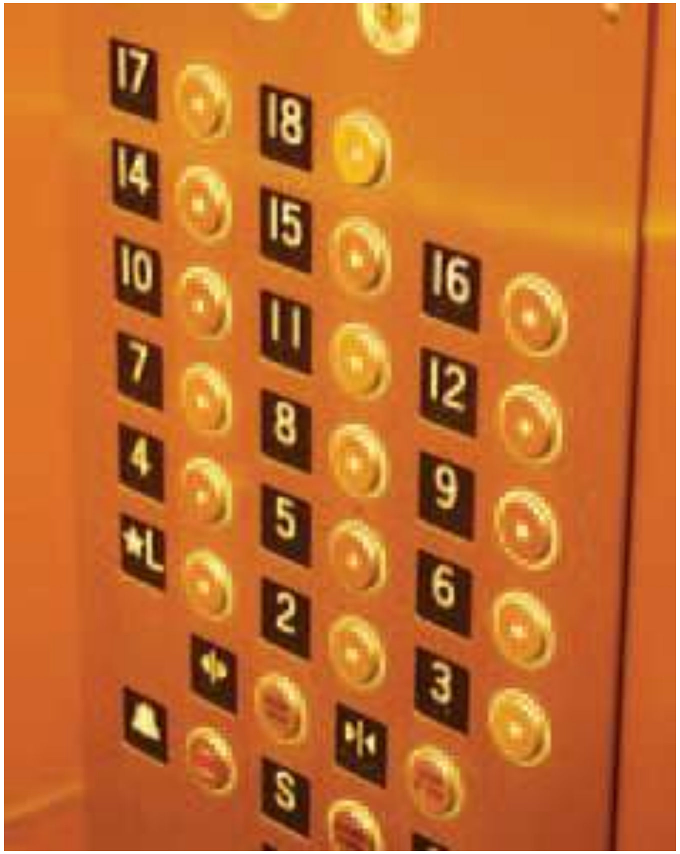
else

    we go that way

C++ for Everyone by Cay Horstmann

The `if` Statement

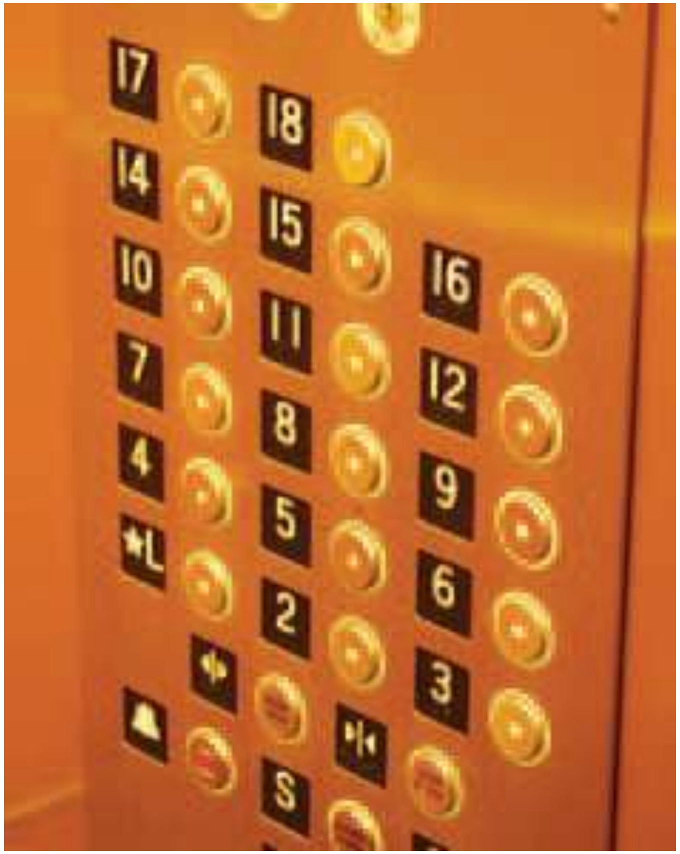
The thirteenth floor!



The `if` Statement

The thirteenth floor!

It's missing!

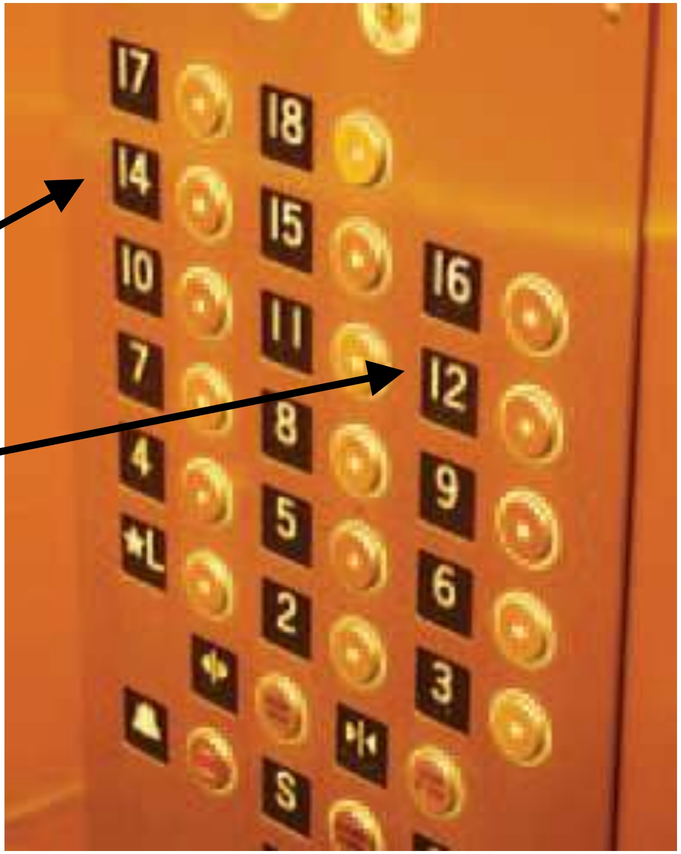


C++ for Everyone by Cay Horstmann

The `if` Statement

The thirteenth floor!

It's missing!



OH NO !!!

The `if` Statement

We must write the code to control the elevator.

How can we skip the 13th floor?

The `if` Statement

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

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

The `if` Statement

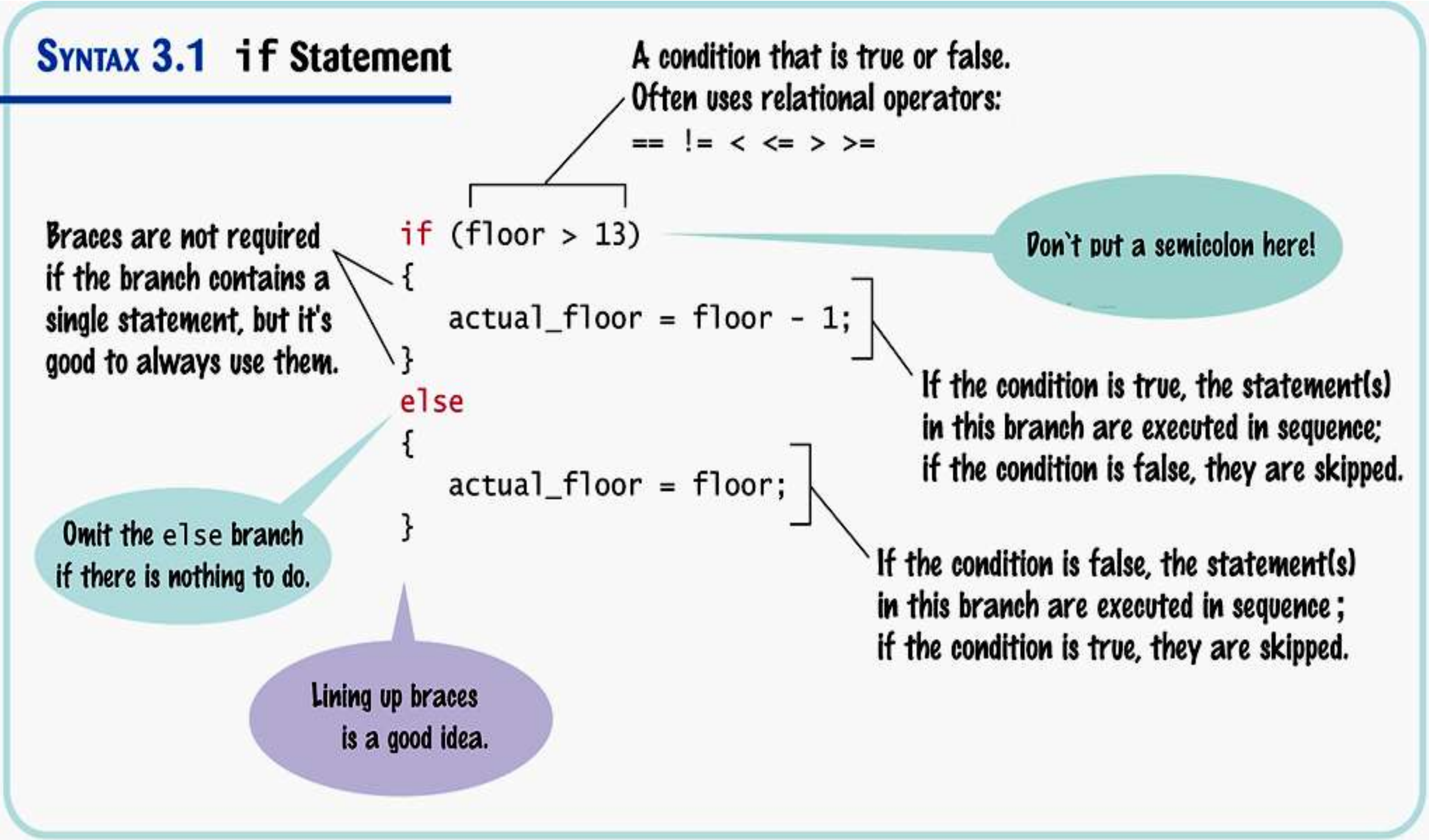
If the user inputs 20,
the program must set the actual floor to 19.

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.

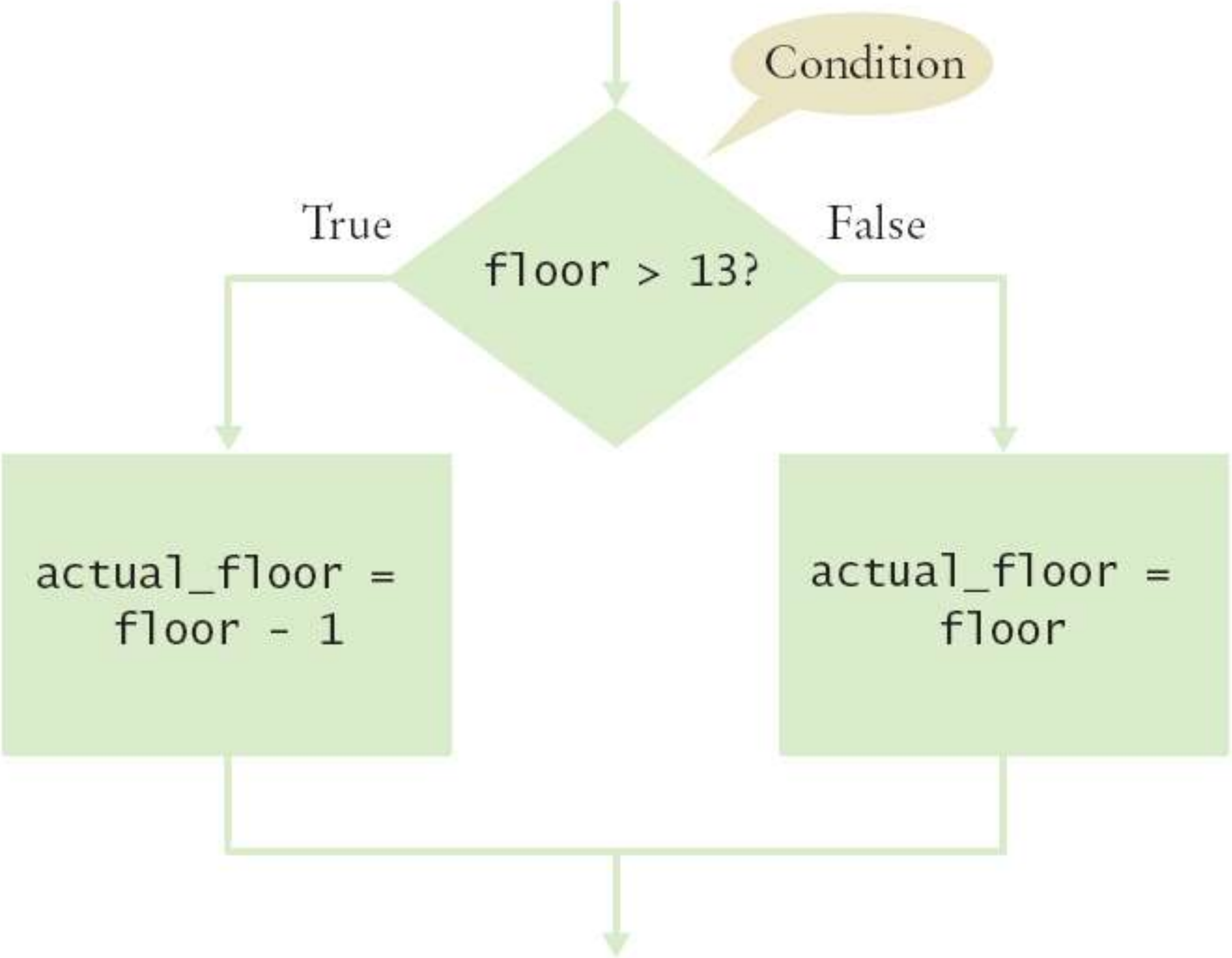
If the condition is true, the statement(s) in this branch are executed in sequence; if the condition is false, they are skipped.

Omit the `else` branch if there is nothing to do.

If the condition is false, the statement(s) in this branch are executed in sequence ; if the condition is true, they are skipped.

Lining up braces is a good idea.

The `if` Statement – The Flowchart



C++ for Everyone by Cay Horstmann

The `if` Statement

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

So don’t write it.

The `if` Statement

Here is another way to write this code:

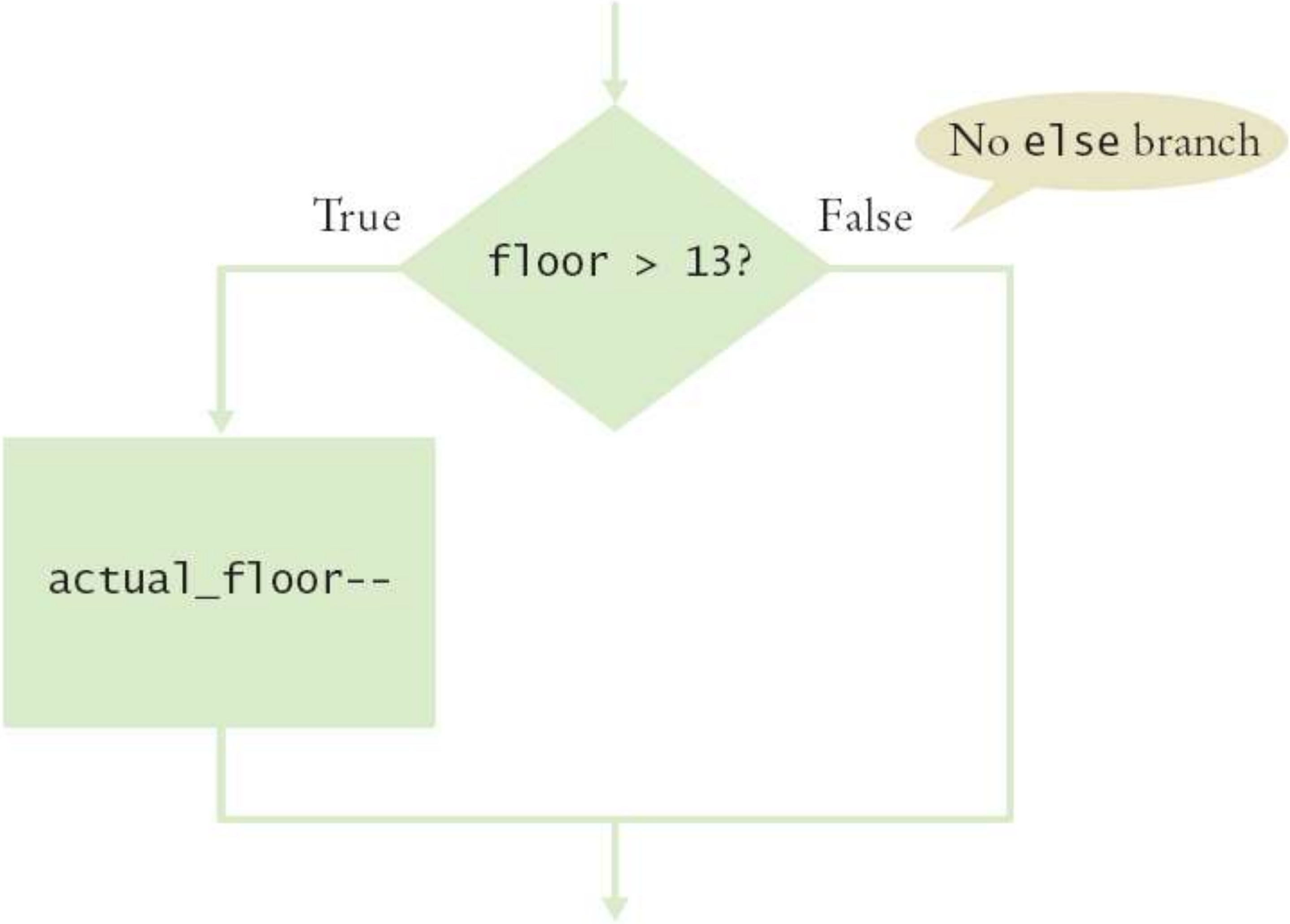
We only need to decrement
when the floor is greater than 13.

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

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

(And you’ll notice we used the decrement operator this time.)

The `if` Statement – The Flowchart



C++ for Everyone by Cay Horstmann

The `if` Statement – A Complete Elevator Program

ch03/elevator1.cpp

```
#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--;
}
```

The `if` Statement – Brace Layout

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

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

Some programmers prefer this style

—it saves a physical line in the code.

The `if` Statement – Brace Layout

This is a passionate and ongoing argument,
but it is about style, not substance.

The `if` Statement – Brace Layout

It is important that you pick a layout scheme and stick with it consistently within a given programming project.

Which scheme you choose may depend on

- your personal preference
- a coding style guide that you need to follow

(that would be your boss' style)

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--;
```

The `if` Statement – Always Use Braces

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) ;
{
   floor--;
}
```

ERROR

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

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

This is *not* a compiler error.
The compiler does not complain.
It interprets this **`if`** statement as follows:

If floor is greater than 13, execute the *do-nothing statement*.
(semicolon by itself is the do nothing statement)

Then *after that* execute the code enclosed in the braces.
Any statements enclosed in the braces are no longer a part of the **`if`** statement.

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

Can you see the error?

This one should be easy now!

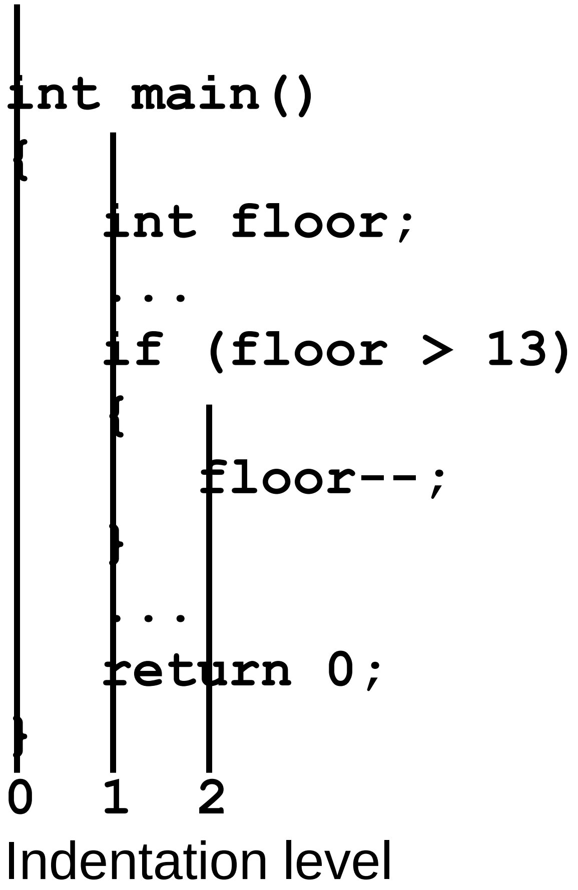
```
if (floor > 13)
 {
     actual_floor = floor - 1;
 }
 else ;
 {
     actual_floor = floor;
 }
```

ERROR

And it really *is* an error this time.

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;
}
```

The `if` Statement – Indent when Nesting

Using the tab key is a way to get this indentation

but …

not all tabs are the same width!

Luckily most development environments have settings to automatically convert all tabs to spaces.

The Conditional Operator

The Conditional Operator

Sometimes you might find yourself wanting to do this:

```
cout << if (floor > 13)
        {
            floor - 1;
        }
        else
        {
            floor;
        }
```

Statements don’t have any value so they can’t be output. But it’s a nice idea.

The Conditional Operator

C++ has the conditional operator of the form

```
condition ? value1 : value2
```

The value of that expression is either **`value1`** if the test passes or **`value2`** if it fails.

The Conditional Operator

For example, we can compute the actual floor number as

```
actual_floor = floor > 13 ? floor - 1 : floor;
```

which is equivalent to

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

The Conditional Operator

You can use the conditional operator anywhere that a value is expected, for example:

```
cout << "Actual floor: " << (floor > 13 ? floor - 1 : floor);
```

We don’t use the conditional operator in this book, but it is a convenient construct that you will find in many C++ programs.

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;
   cout << "Actual floor: " << actual_floor << endl;
}
else
{
   actual_floor = floor;
   cout << "Actual floor: " << actual_floor << endl;
}
```

Hmmm…

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 these depend on the test?

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

Which way *is* quicker to the candy mountain?

Relational Operators

Let’s compare the distances.

Relational Operators

Relational operators

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

are used to compare numbers and strings.

Relational Operators

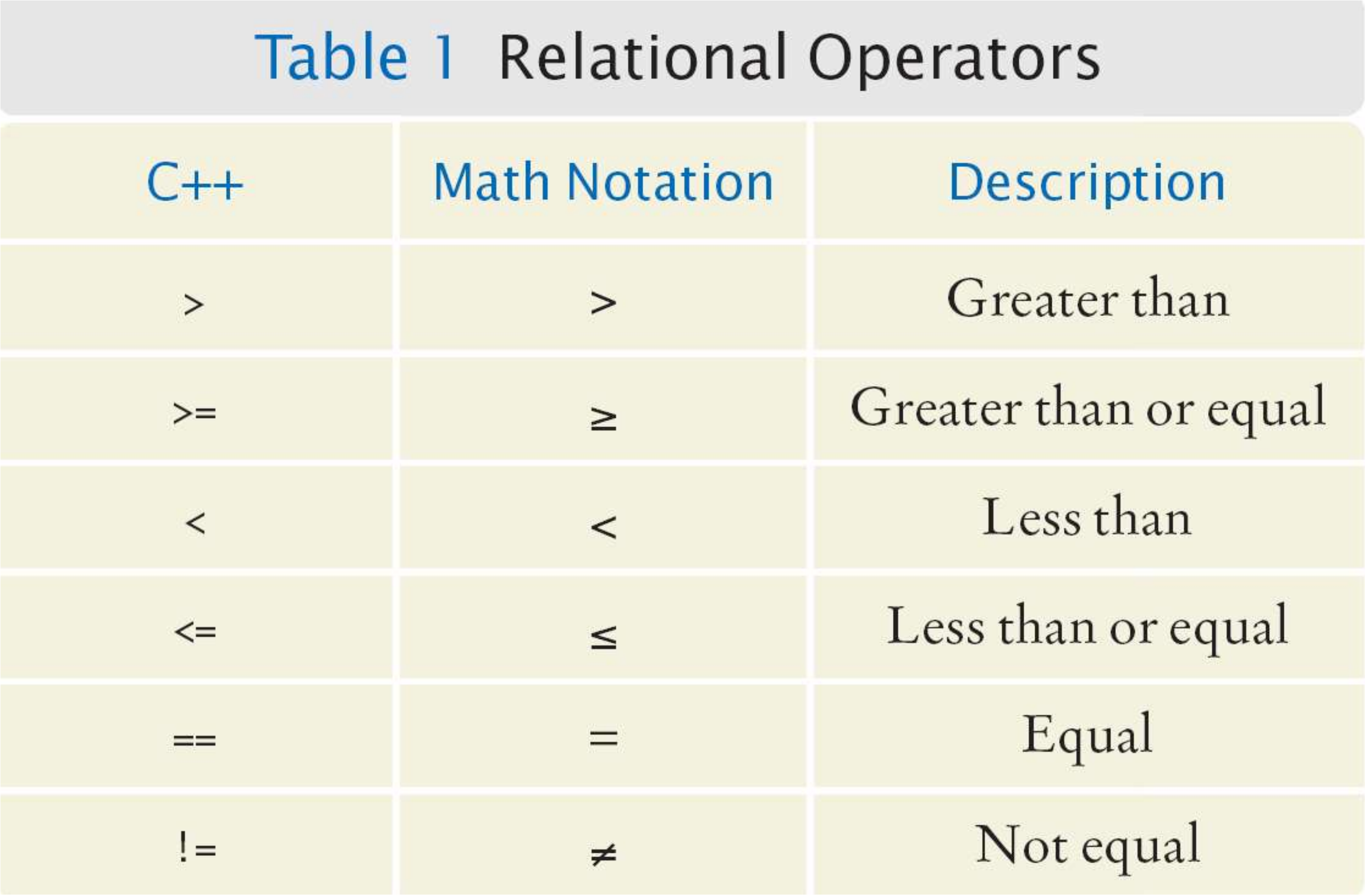
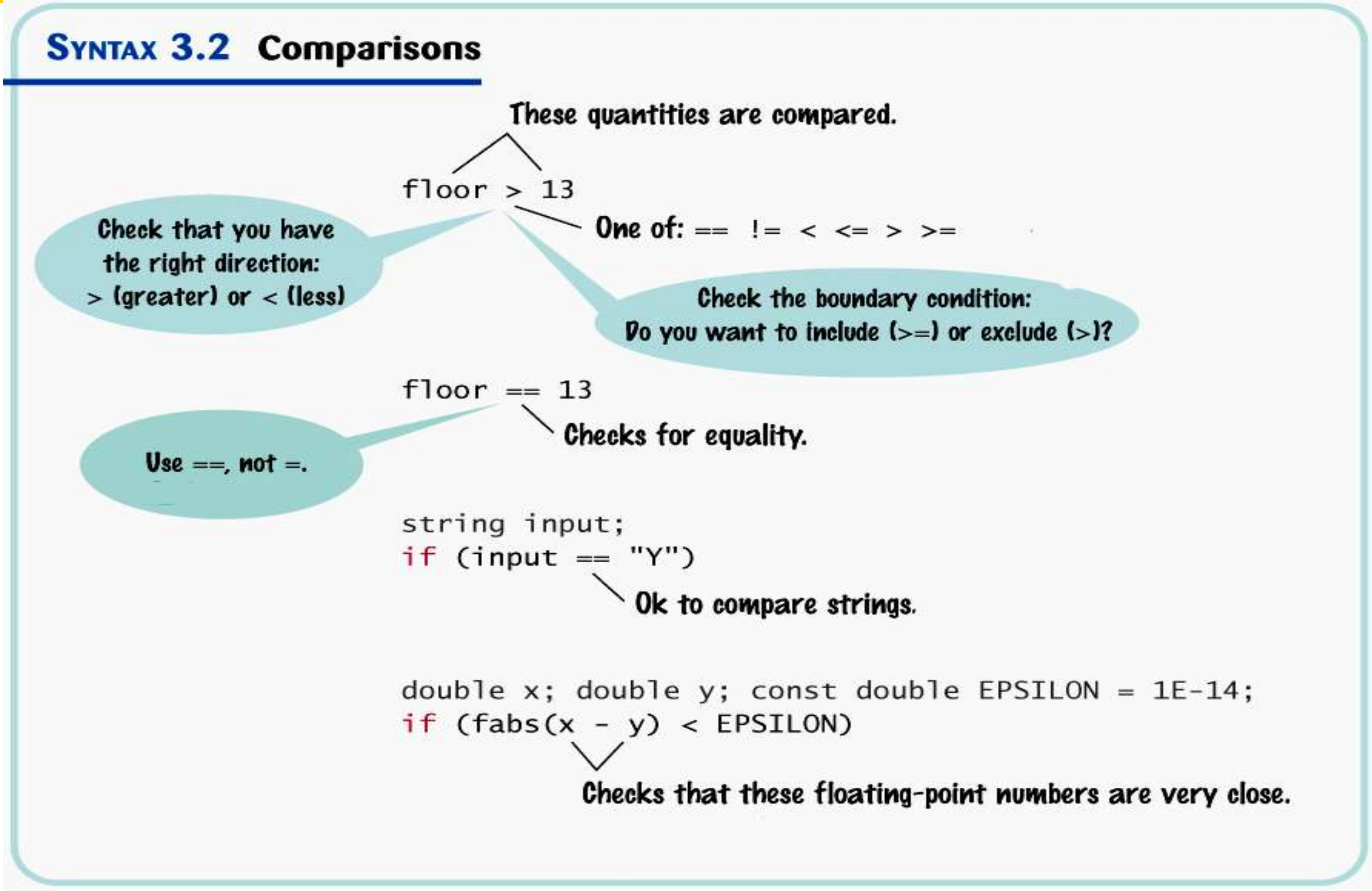


Table 1 Relational Operators

C++	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

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

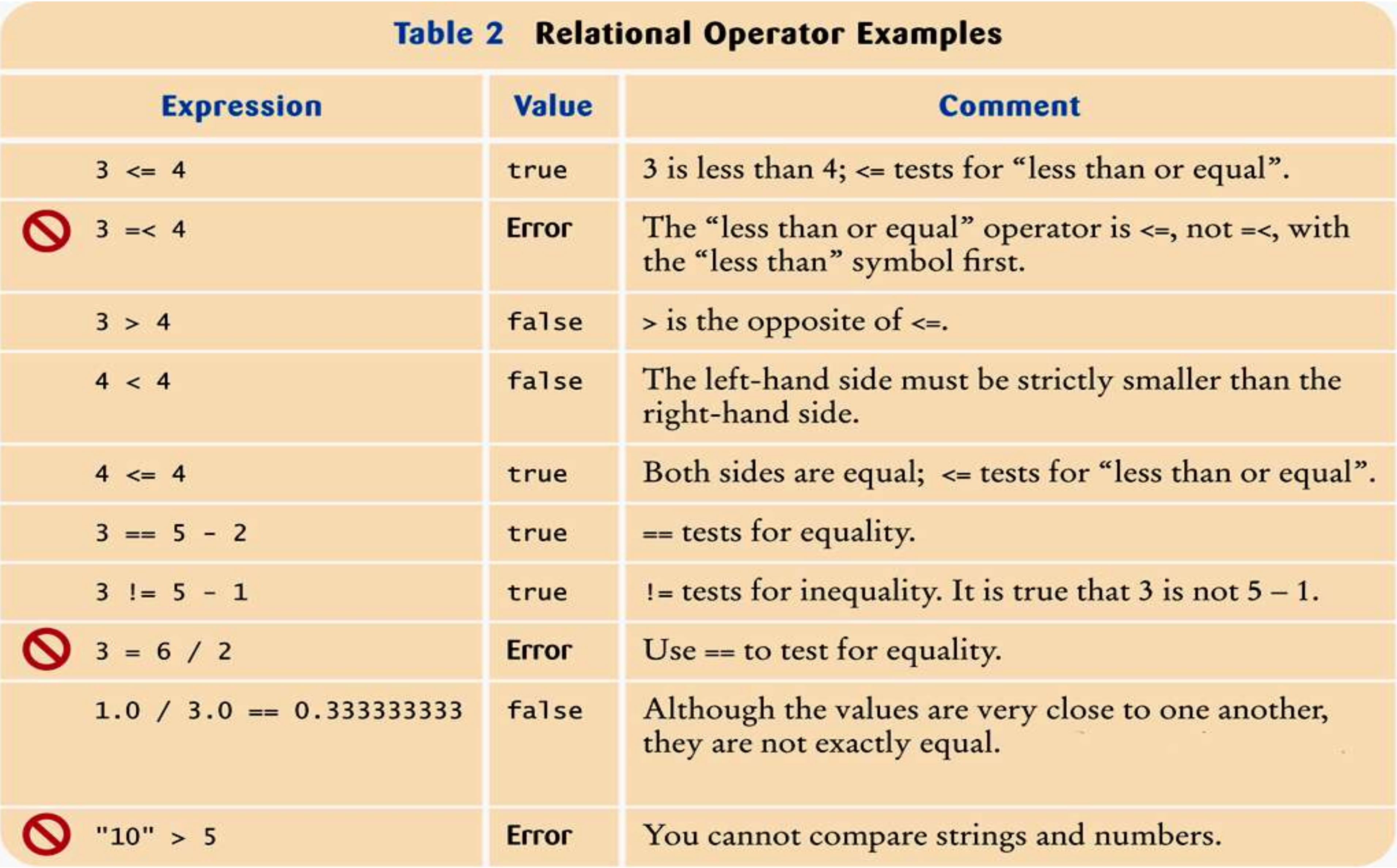


Table 2 Relational Operator Examples

	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).

C++ for Everyone by Cay Horstmann

Relational Operators – Some Notes

The **==** operator is initially confusing to beginners.

In C++, **=** already has a meaning, namely assignment

The **==** operator denotes equality testing:

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

You can compare strings as well:

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

Common Error – Confusing = and ==

The C++ language allows the use of = inside tests.

To understand this, we have to go back in time.

The creators of C, the predecessor to C++, were very frugal thus C did not have true and false values.

Instead, they allowed any numeric value inside a condition with this interpretation:

> 0 denotes false
> any non-0 value denotes true.

In C++ you should use the `bool` values `true` and `false`

Common Error – Confusing = and ==

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++.

Common Error – Confusing = and ==

The code sets **`floor`** to 13,
and since that value is not zero,
the condition of the **`if`** statement is *always* **`true`**.

```
if (floor = 13) …
```

(and it’s really hard to find this error at 3:00am
when you’ve been coding for 13 hours straight)

Common Error – Confusing = and ==

Don’t be shell-shocked by this
and go completely the other way:

```
    floor == floor - 1; // ERROR
```

This statement tests whether `floor` equals `floor - 1`.

It doesn’t do anything with the outcome of the test,
but that is not a compiler error.

Nothing really happens
(which is probably not what you meant to do
– so that’s the error).

Common Error – Confusing = and ==

You must remember:

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

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

Kinds of Error Messages

There are two kinds of errors:

Warnings

Errors

Kinds of Error Messages

- Error messages are fatal.
  - The compiler will not translate a program with one or more errors.
- Warning messages are advisory.
  - The compiler will translate the program, but there is a good chance that the program will not do what you expect it to do.

Kinds of Error Messages

It is a good idea to learn how to activate warnings in your compiler.

It as a great idea to write code that emits no warnings at all.

Kinds of Error Messages

We stated there are two kinds of errors.

Actually there’s only one kind:

The ones you must read

(that’s all of them!)

Kinds of Error Messages

Read all comments and deal with them.

If you understand a warning, and understand why it is happening, and you don’t care about that reason

– Then, and only then, should you ignore a warning.

and, of course,
you can’t ignore an error message!

Common Error – Exact Comparison of Floating-Point Numbers

Round off errors

Floating-point numbers have only a limited precision.

Calculations can introduce roundoff errors.

Common Error – Exact Comparison of Floating-Point Numbers

Roundoff errors

Does $\left(\sqrt{r}\right)^{2} == 2$?

Let’s see (by writing code, of course) …

Common Error – Exact Comparison of Floating-Point Numbers

roundoff error

```
double r = sqrt(2.0);
if (r * r == 2)
{
   cout << "sqrt(2) squared is 2" << endl;
}
else
{
   cout << "sqrt(2) squared is not 2 but "
      << setprecision(18) << r * r << endl;
}
```

This program displays:

```
sqrt(2) squared is not 2 but 2.000000000000000044
```

Common Error – Exact Comparison of Floating-Point Numbers

Roundoff errors – a solution

Close enough will do.

$$|x - y| < \varepsilon$$

Common Error – Exact Comparison of Floating-Point Numbers

Mathematically, we would write that *x* and *y* are close enough if for a very small number, *ε*:

$$|x - y| < \varepsilon$$

ε is the Greek letter epsilon, a letter used to denote a very small quantity.

Common Error – Exact Comparison of Floating-Point Numbers

It is common to set ε to 10^{-14} when comparing `double` numbers:

```
const double EPSILON = 1E-14;
double r = sqrt(2.0);
if (fabs(r * r - 2) < EPSILON)
{
   cout << "sqrt(2) squared is approximately ";
}
```

Include the `<cmath>` header to use `sqrt` and the `fabs` function which gives the absolute value.

Try `round.cpp`.

Lexicographical Ordering of Strings

Comparing strings uses “lexicographical” order to decide which is larger or smaller or if two strings are equal.

“Dictionary order” is another way to think about “lexicographical” (and it’s a little bit easier to pronounce).

```
string name = "Tom";
if (name < "Dick")...
```

The test is false because “Dick”

Lexicographical Ordering of Strings

Comparing strings uses “lexicographical” order to decide which is larger or smaller or if two strings are equal.

“Dictionary order” is another way to think about “lexicographical” (and it’s a little bit easier to pronounce).

```
string name = "Tom";
if (name < "Dick")...
```

The test is false because “Dick” would come before “Tom” if they were words in a dictionary.

(not to be confused with dicktionary – if there is such a word)

Lexicographical Ordering of Strings

- All uppercase letters come before the lowercase letters.
  For example, "Z" comes before "a".
- The space character comes before all printable characters.
- Numbers come before letters.
- The punctuation marks are ordered but we won’t go into that now.

Lexicographical Ordering of Strings

ASCII Table

(American Standard Code for Information Interchange)

0-31 are control codes, for example **"/n"** (newline) has ASCII code 10.

32:	33:!	34:"	35:#	36:$	37:%	38:&	39:'	40:(	41:)
42:*	43:+	44:,	45:-	46:.	47:/	48:0	49:1	50:2	51:3
52:4	53:5	54:6	55:7	56:8	57:9	58::	59:;	60:<	61:=
62:>	63:?	64:@	65:A	66:B	67:C	68:D	69:E	70:F	71:G
72:H	73:I	74:J	75:K	76:L	77:M	78:N	79:O	80:P	81:Q
82:R	83:S	84:T	85:U	86:V	87:W	88:X	89:Y	90:Z	91:[
92:\	93:]	94:^	95:_	96:`	97:a	98:b	99:c	100:d	101:e
102:f	103:g	104:h	105:i	106:j	107:k	108:l	109:m	110:n	111:o
112:p	113:q	114:r	115:s	116:t	117:u	118:v	119:w	120:x	121:y
122:z	123:{	124:|	125:}	126:~	127:□	128:Ђ	129:Ѓ	130:‚	131:ѓ
132:„	133:…	134:†	135:‡	136:€	137:‰	138:Љ	139:‹	140:Њ	141:Ќ
142:Ћ	143:Џ	144:ђ	145:‘	146:’	147:“	148:”	149:•	150:–	151:—
152:	153:™	154:љ	155:›	156:њ	157:ќ	158:ћ	159:џ	160:	161:Ў
162:ў	163:Ј	164:¤	165:Ґ	166:¦	167:§	168:Ё	169:©	170:Є	171:«
172:¬	173:-	174:®	175:Ї	176:°	177:±	178:І	179:і	180:ґ	181:µ
182:¶	183:·	184:ё	185:№	186:є	187:»	188:ј	189:Ѕ	190:ѕ	191:ї
192:А	193:Б	194:В	195:Г	196:Д	197:Е	198:Ж	199:З	200:И	201:Й
202:К	203:Л	204:М	205:Н	206:О	207:П	208:Р	209:С	210:Т	211:У
212:Ф	213:Х	214:Ц	215:Ч	216:Ш	217:Щ	218:Ъ	219:Ы	220:Ь	221:Э
222:Ю	223:Я	224:а	225:б	226:в	227:г	228:д	229:е	230:ж	231:з
232:и	233:й	234:к	235:л	236:м	237:н	238:о	239:п	240:р	241:с
242:т	243:у	244:ф	245:х	246:ц	247:ч	248:ш	249:щ	250:ъ	251:ы
252:ь	253:э	254:ю	255:□						

This is Windows-1251 encoding table.

Lexicographical Ordering of Strings

When comparing two strings,

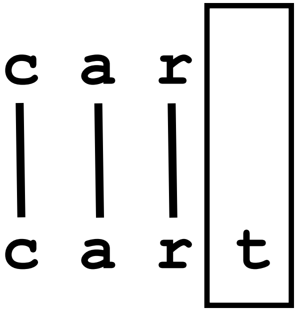
you compare the first letters of each word, then the second letters, and so on, until:

- one of the strings ends
- you find the first letter pair that doesn't match.

If one of the strings ends, the longer string is considered the "larger" one.

Lexicographical Ordering of Strings

For example, compare "car" with "cart".



c a r

c a r t

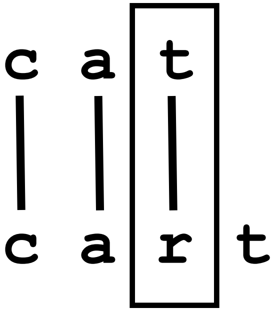
The first three letters match, and we reach the end of the first string – making it less than the second.

Therefore "car" comes before "cart" in lexicographic ordering.

Lexicographical Ordering of Strings

When you reach a mismatch, the string containing the “larger” character is considered “larger”.

For example, let’s compare "cat" with "cart".



The first two letters match.

Since **`t`** comes after **`r`**, the string "cat" comes after "cart“ in the lexicographic ordering.

End Decisions I

Slides by Evan Gallagher & Nikolay Kirov

C++ for Everyone by Cay Horstmann