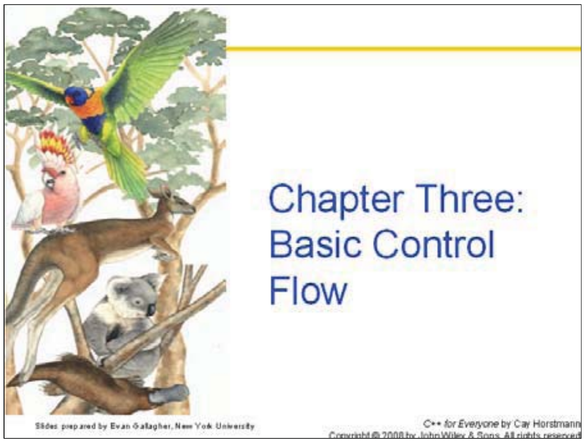
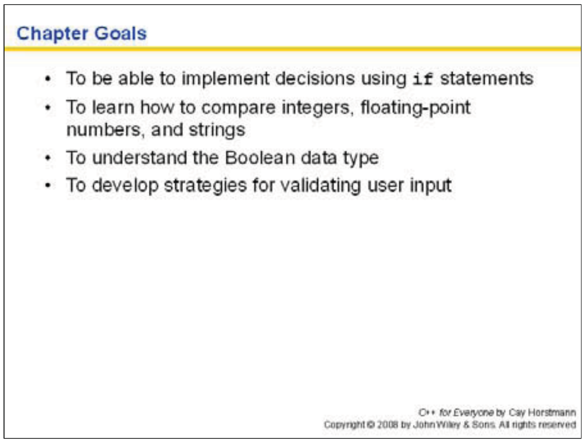


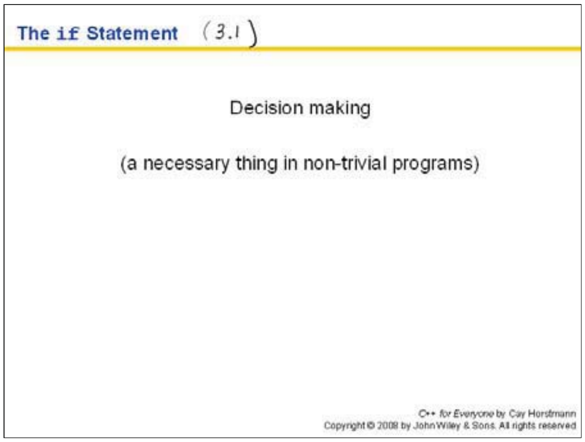
Chapter Three: Basic Control Flow

Slides prepared by Evan Gallagher, New York University



Chapter Goals

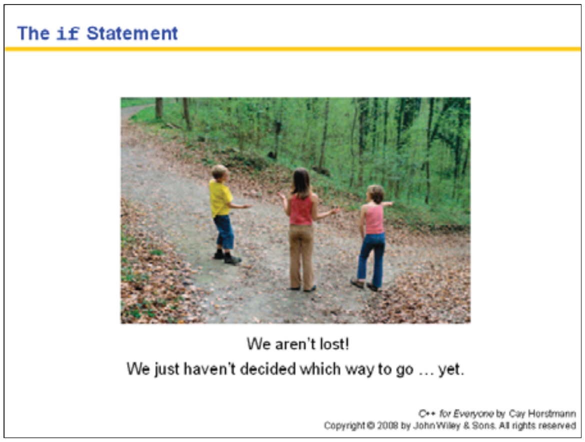
- To be able to implement decisions using `if` statements
- To learn how to compare integers, floating-point numbers, and strings
- To understand the Boolean data type
- To develop strategies for validating user input



The `if` Statement (3.1)

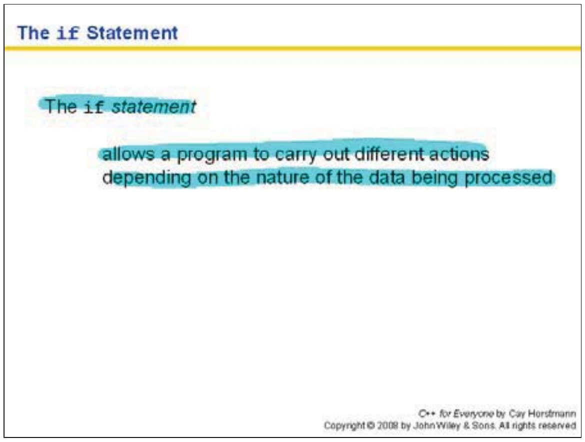
Decision making

(a necessary thing in non-trivial programs)



The `if` Statement

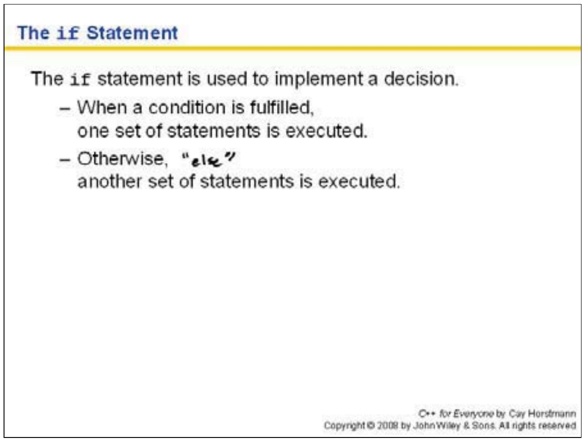
We aren't lost!
We just haven't decided which way to go … yet.



The `if` Statement

The `if` *statement*

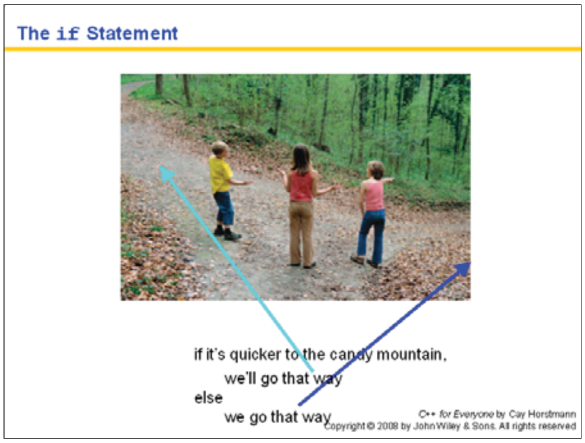
allows a program to carry out different actions depending on the nature of the data being processed



The `if` Statement

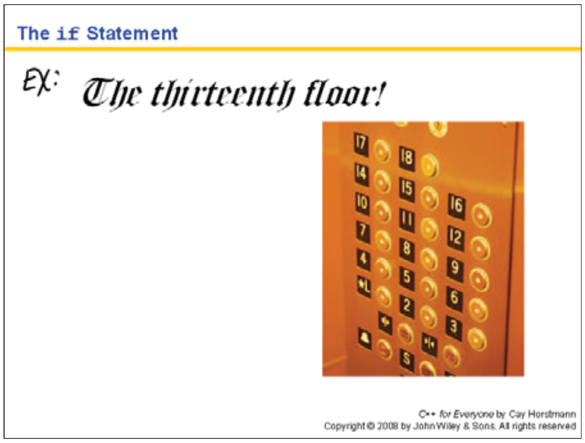
The `if` statement is used to implement a decision.

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



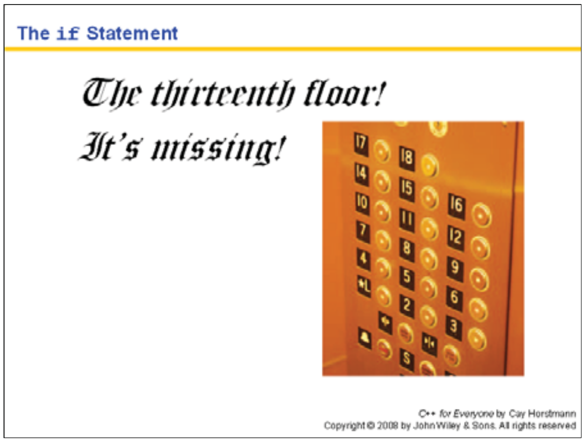
The `if` Statement

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



The `if` Statement

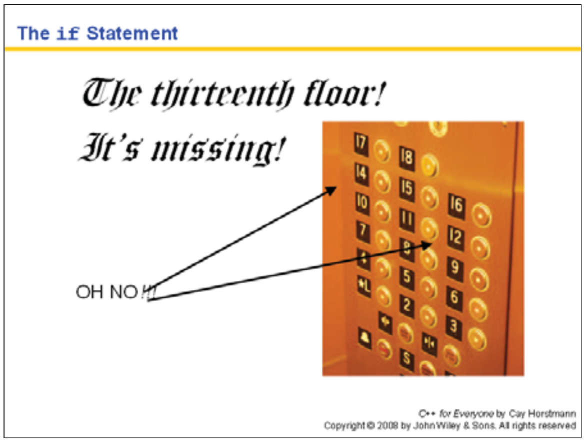
EX: The thirteenth floor!



The `if` Statement

The thirteenth floor!

It's missing!

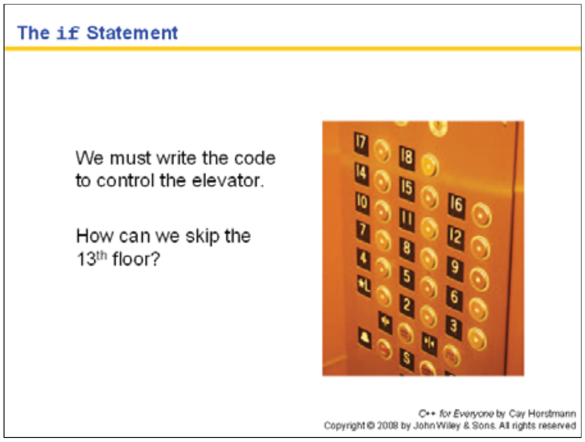


The `if` Statement

The thirteenth floor!

It's missing!

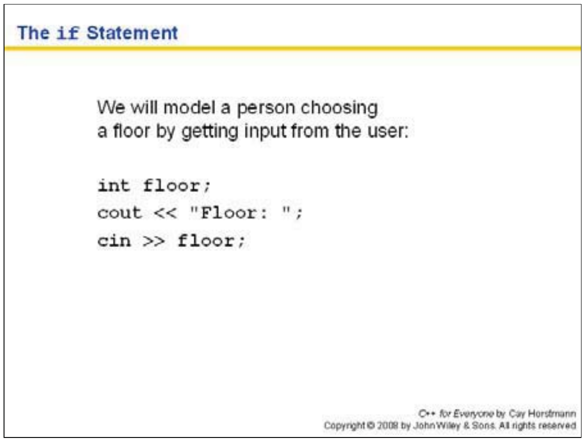
OH NO!!!



The `if` Statement

We must write the code to control the elevator.

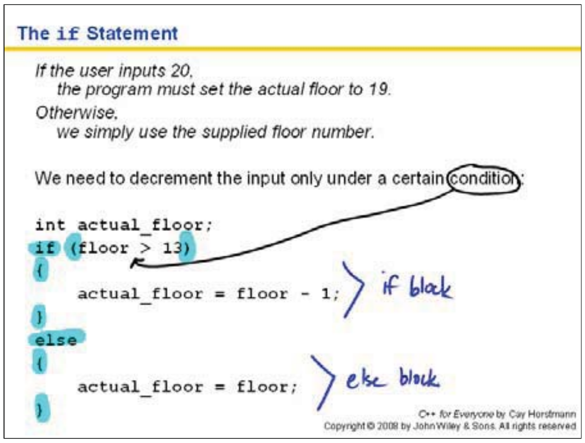
How can we skip the 13th floor?



The `if` Statement

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

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



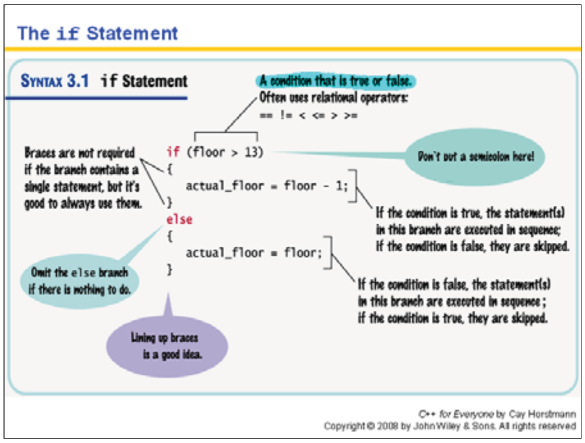
The if Statement

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;
}
```

if block
else block



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!

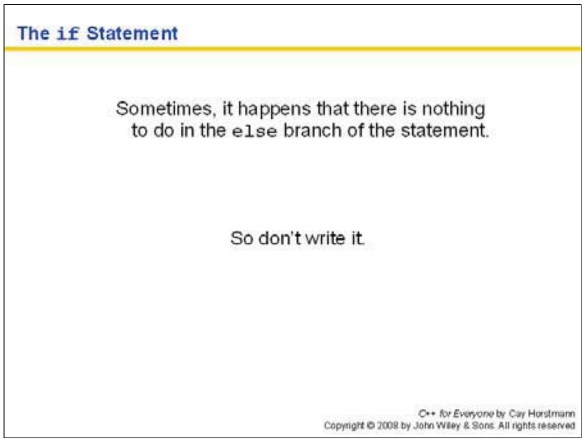
Braces are not required if the branch contains a single statement, but it's good to always use them.

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.

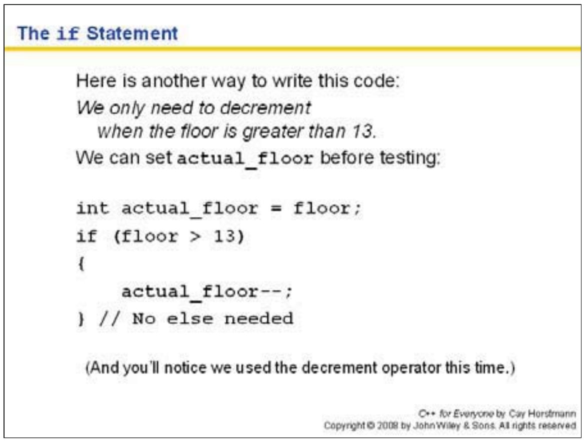
Lining up braces is a good idea.



The if Statement

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

So don't write it.



The if Statement

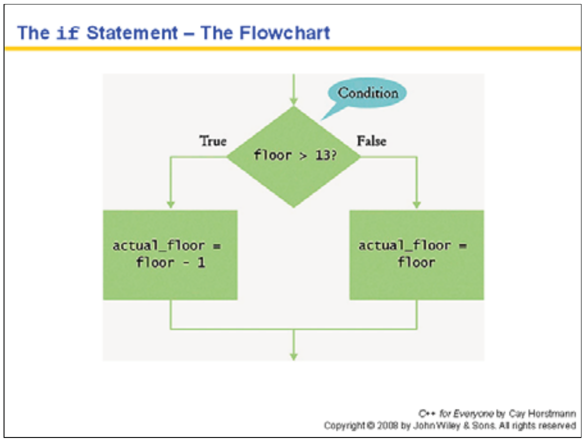
Here is another way to write this code:

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

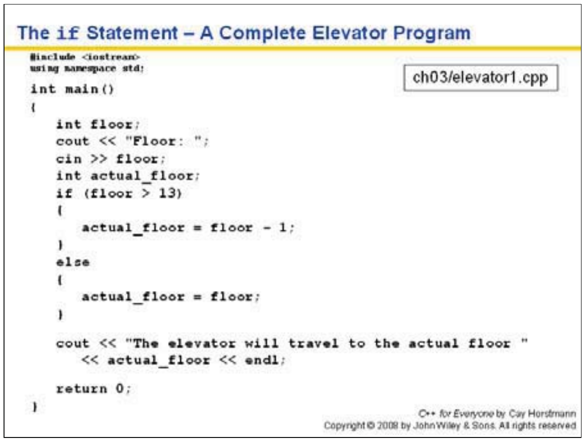
We can set actual_floor before testing:

```
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



The if Statement – A Complete Elevator Program

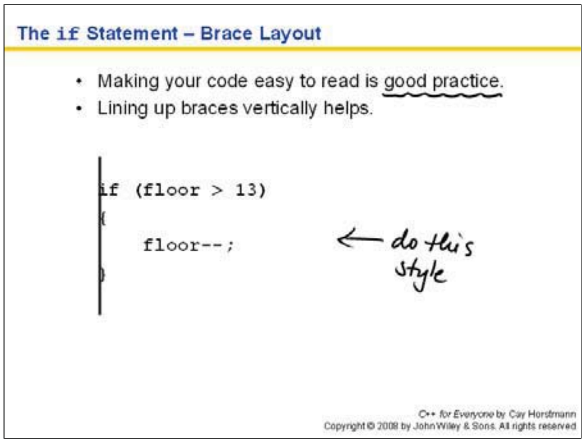
ch03/elevator1.cpp

```
#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;
}
```

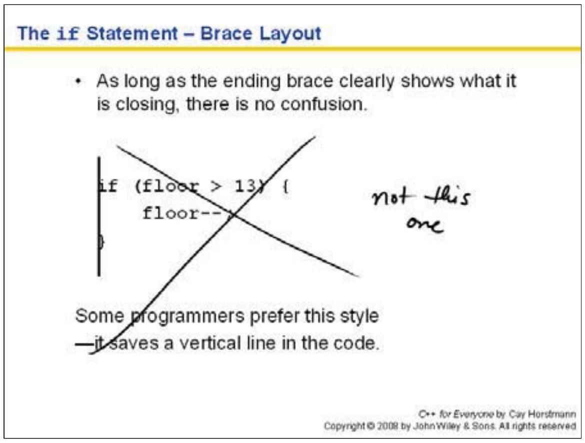


The if Statement – Brace Layout

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

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

← do this style



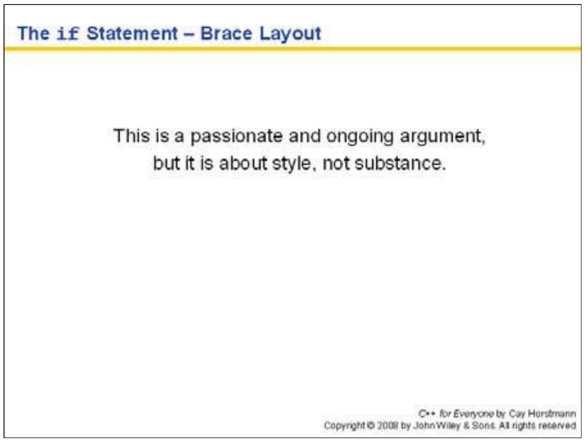
The if Statement – Brace Layout

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

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

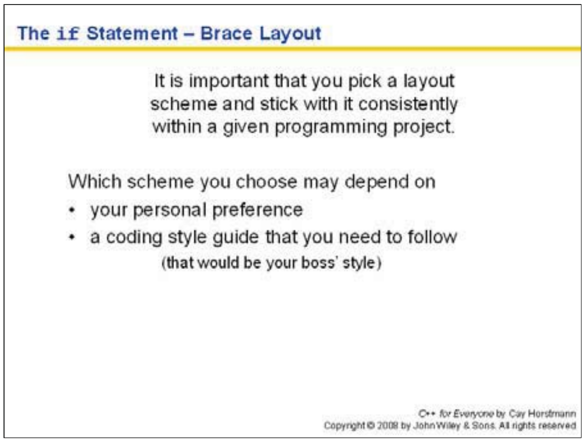
not this one

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



The if Statement – Brace Layout

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

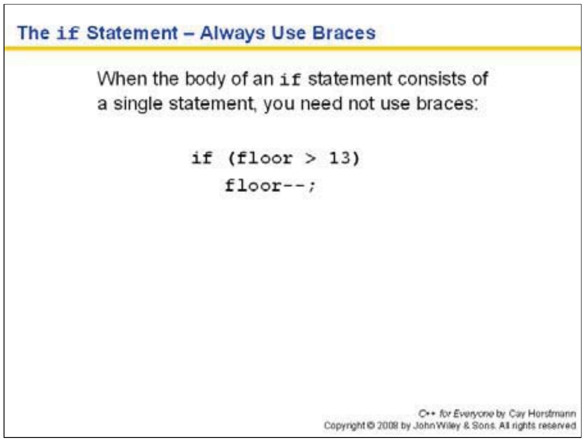


The if Statement – Brace Layout

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

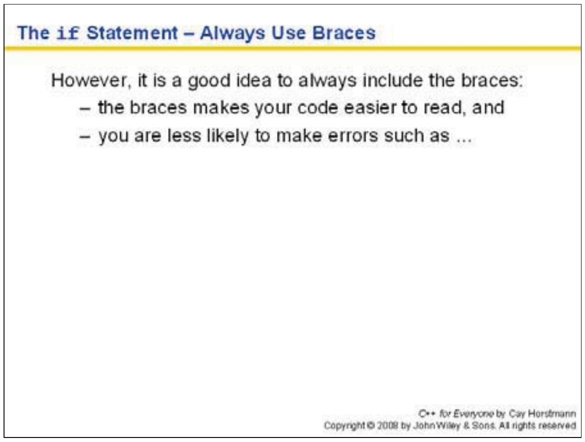
- your personal preference
- a coding style guide that you need to follow
  (that would be your boss' style)



The if Statement – Always Use Braces

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

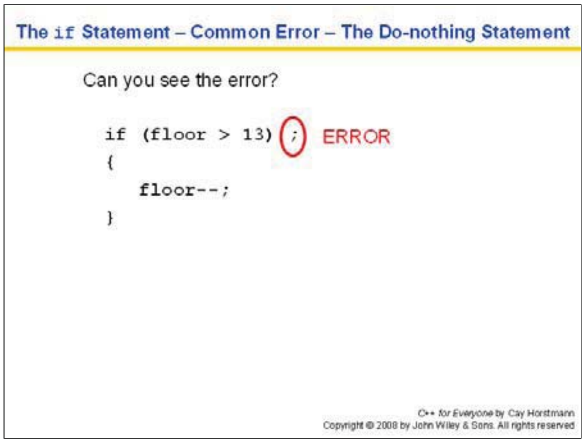
```
if (floor > 13)
   floor--;
```



The if Statement – Always Use Braces

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

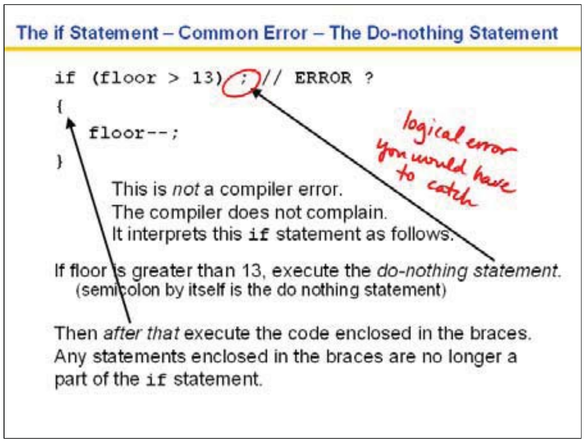
- the braces makes your code easier to read, and
- you are less likely to make errors such as …



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

Can you see the error?

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



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

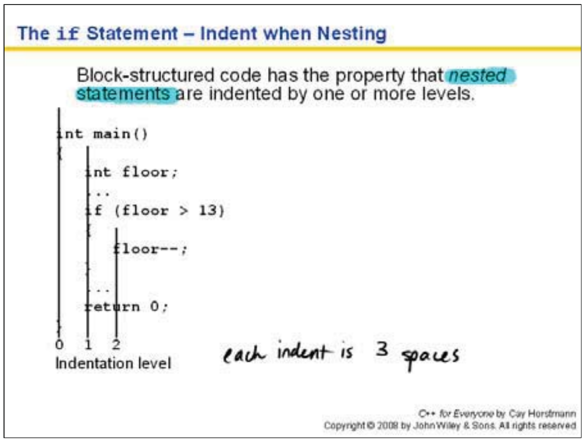
```
if (floor > 13) ; // ERROR ?
{
   floor--;
}
```

logical error
you would have to catch

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.



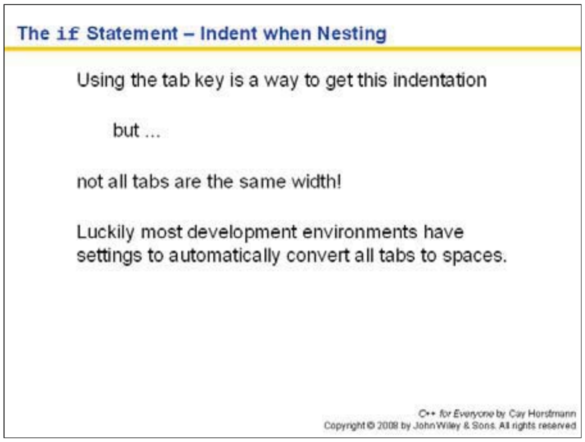
The if Statement – Indent when Nesting

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

each indent is 3 spaces



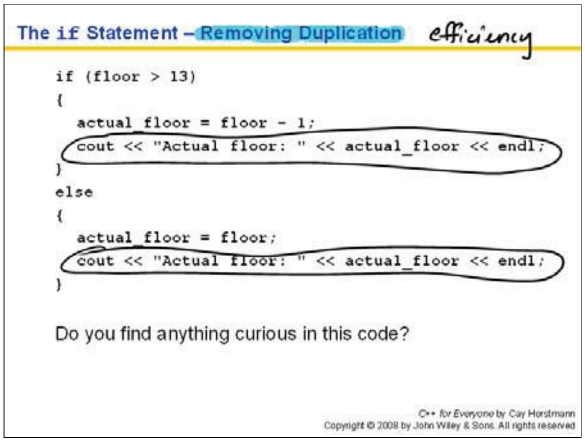
The if Statement – Indent when Nesting

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.

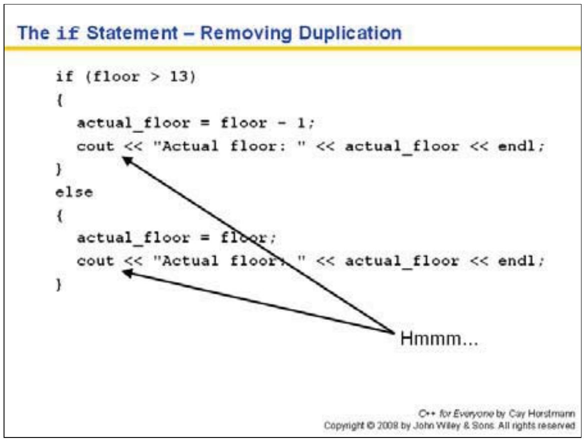


The if Statement – Removing Duplication

efficiency

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

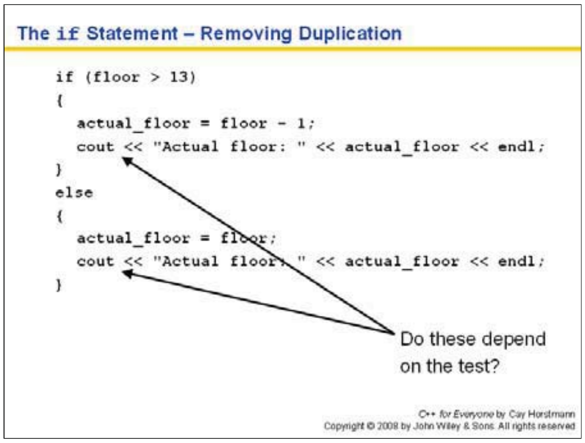
Do you find anything curious in this code?



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;
}
```

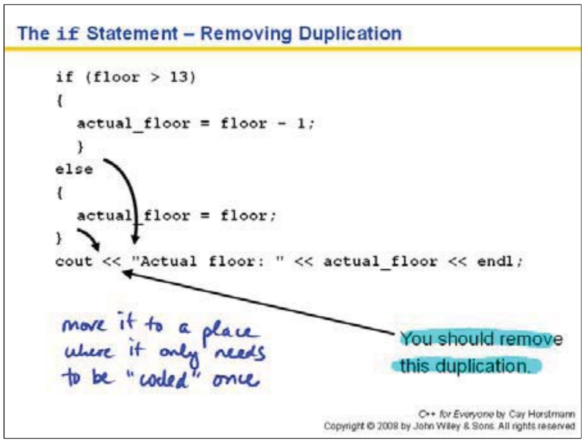
Hmmm...



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?

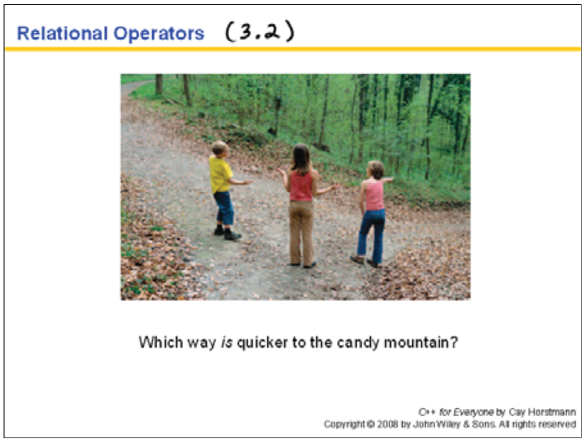


The `if` Statement – Removing Duplication

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

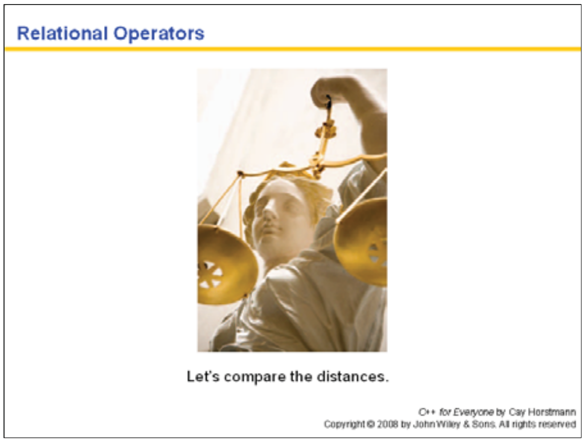
move it to a place where it only needs to be "coded" once

You should remove this duplication.



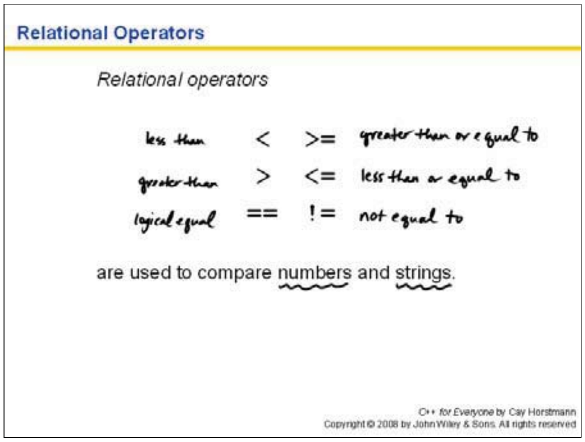
Relational Operators (3.2)

Which way *is* quicker to the candy mountain?



Relational Operators

Let's compare the distances.

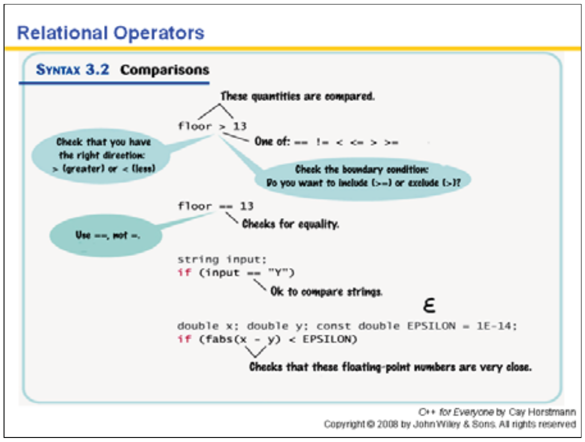


Relational Operators

Relational operators

less than	<	>=	greater than or equal to
greater than	>	<=	less than or equal to
logical equal	==	!=	not equal to

are used to compare numbers and strings.



Relational Operators

Syntax 3.2 Comparisons

```
floor > 13
```

These quantities are compared.

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 =.

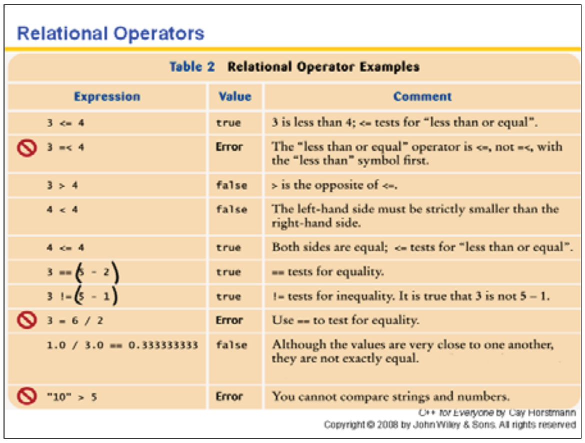
```
string input;
if (input == "Y")
```

Ok to compare strings.

ε

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

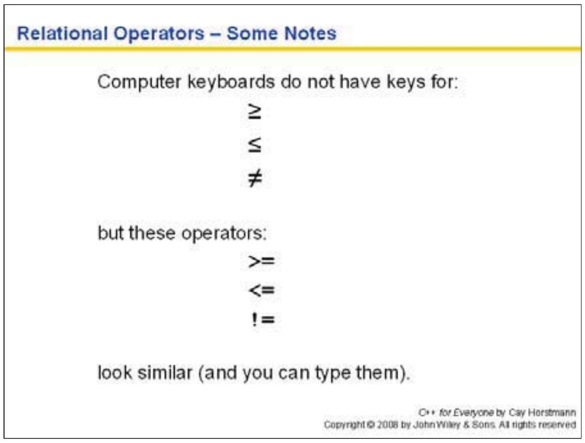
Checks that these floating-point numbers are very close.



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

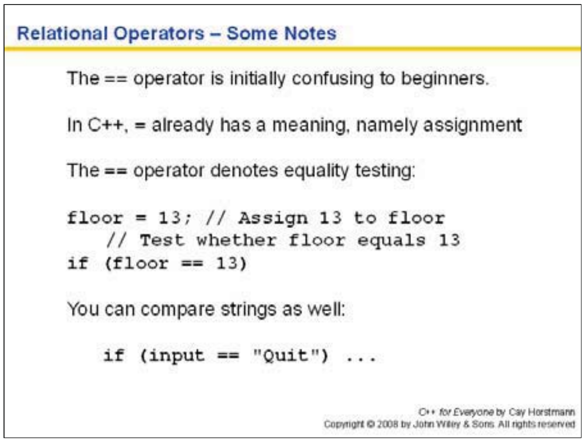
Computer keyboards do not have keys for:

≥
≤
≠

but these operators:

>=
<=
!=

look similar (and you can type them).



Relational Operators – Some Notes

The == operator is initially confusing to beginners.

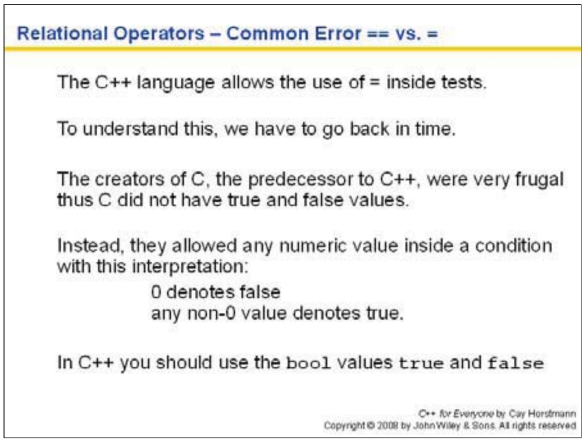
In C++, = already has a meaning, namely assignment

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") ...
```



Relational Operators – Common Error == vs. =

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

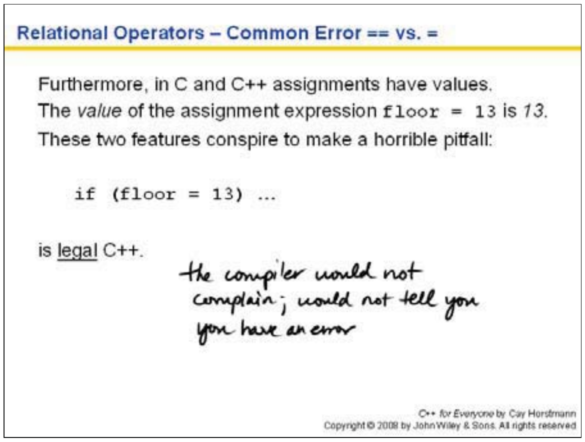
To understand this, we have to go back in time.

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`



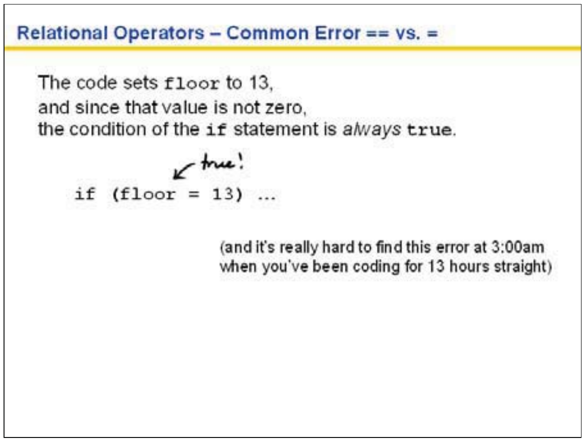
Relational Operators – Common Error == vs. =

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++.

the compiler would not
complain; would not tell you
you have an error



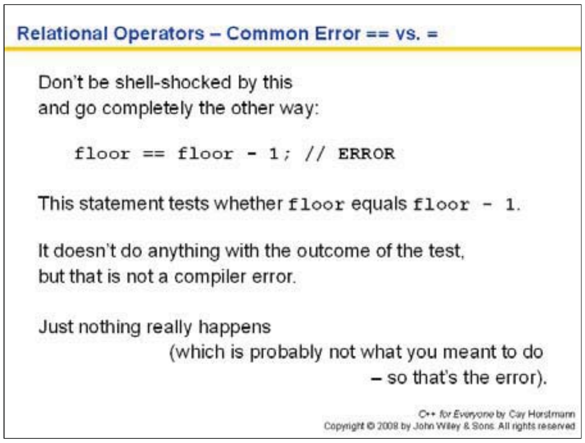
Relational Operators – Common Error == vs. =

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

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)



Relational Operators – Common Error == vs. =

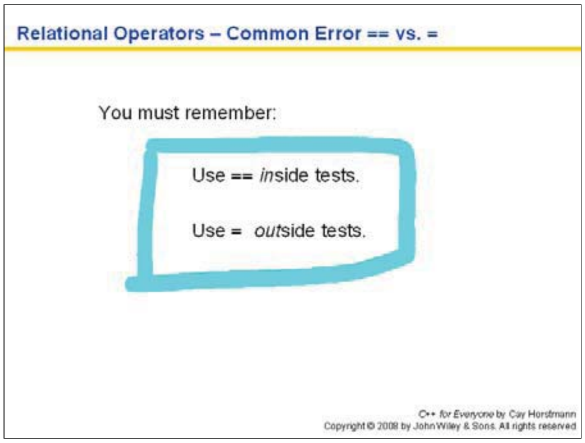
Don't be shell-shocked by this
and go completely the other way:

```
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.

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

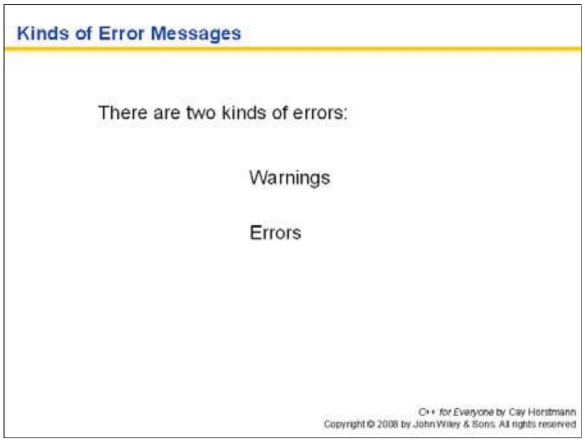


Relational Operators – Common Error == vs. =

You must remember:

Use == *in*side tests.

Use = *out*side tests.

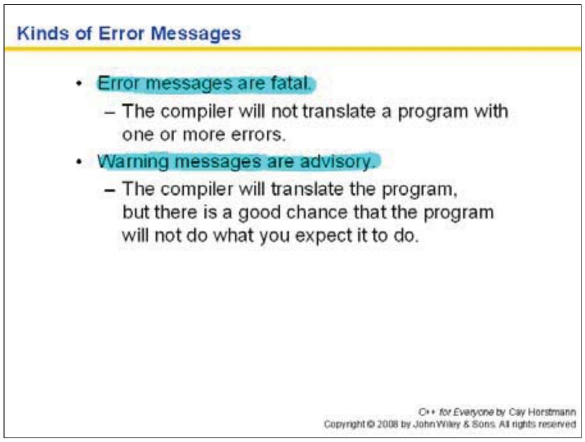


Kinds of Error Messages

There are two kinds of errors:

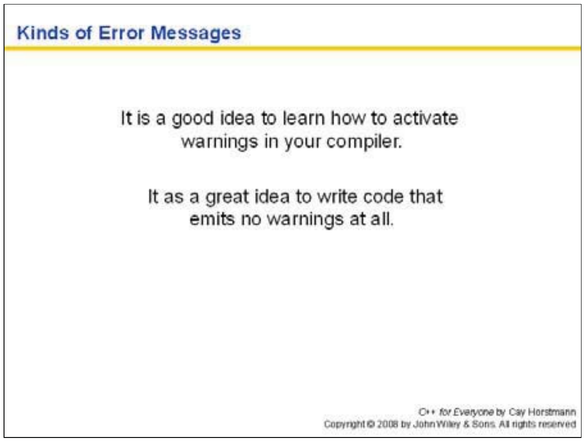
Warnings

Errors



Kinds of Error Messages

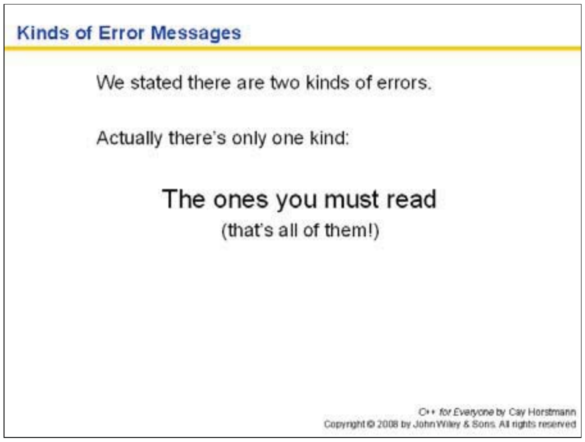
- Error messages are fatal.
  - The compiler will not translate a program with one or more errors.
- Warning messages are advisory.
  - The compiler will translate the program, but there is a good chance that the program will not do what you expect it to do.



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.

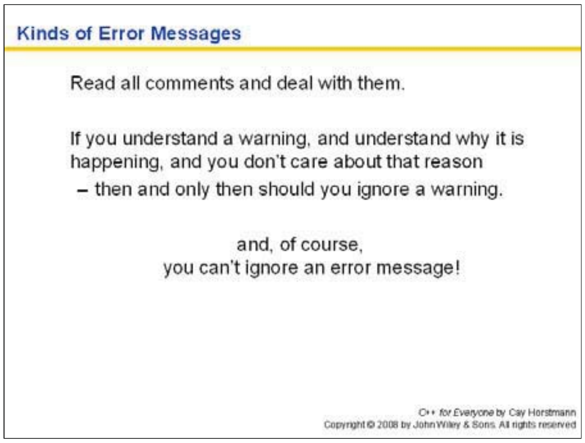


Kinds of Error Messages

We stated there are two kinds of errors.

Actually there's only one kind:

The ones you must read
(that's all of them!)



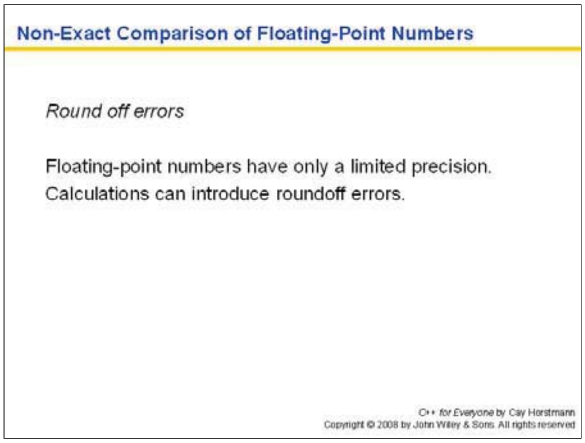
Kinds of Error Messages

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.

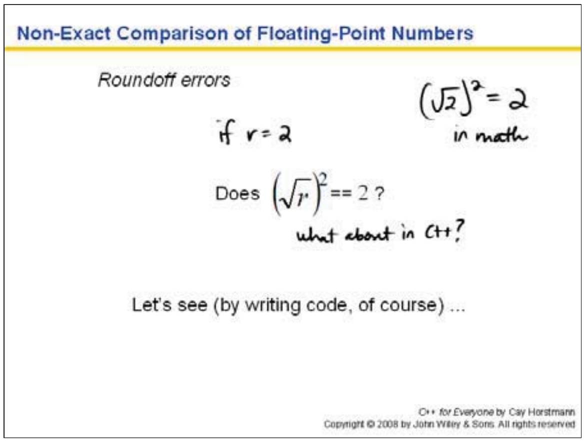
and, of course,
you can't ignore an error message!



Non-Exact Comparison of Floating-Point Numbers

Round off errors

Floating-point numbers have only a limited precision.
Calculations can introduce roundoff errors.



Non-Exact Comparison of Floating-Point Numbers

Roundoff errors

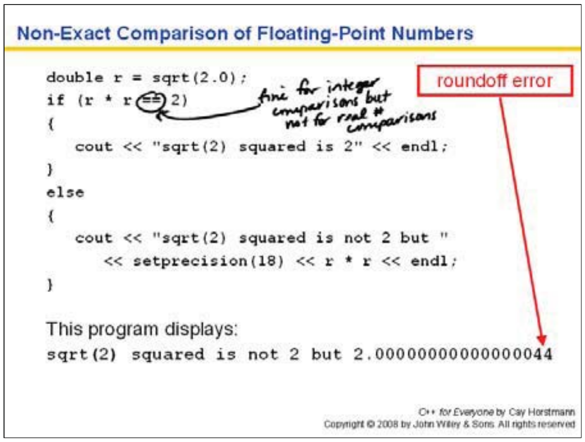
$\left(\sqrt{2}\right)^2 = 2$
in math

if $r = 2$

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

what about in C++?

Let's see (by writing code, of course) ...



Non-Exact Comparison of Floating-Point Numbers

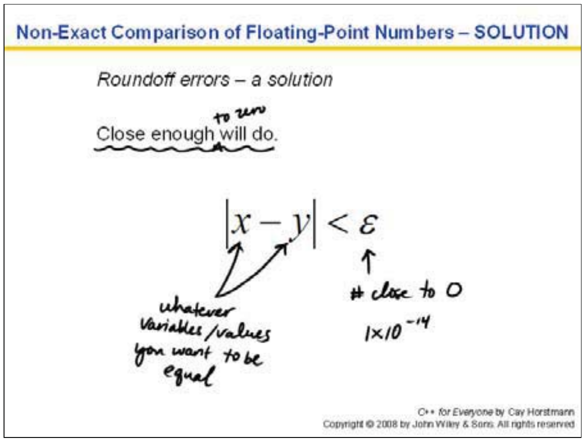
```
double r = sqrt(2.0);
if (r * r == 2)
{
   cout << "sqrt(2) squared is 2" << endl;
}
else
{
   cout << "sqrt(2) squared is not 2 but "
      << setprecision(18) << r * r << endl;
}
```

fine for integer comparisons but not for real # comparisons

roundoff error

This program displays:

```
sqrt(2) squared is not 2 but 2.00000000000000044
```



Non-Exact Comparison of Floating-Point Numbers – SOLUTION

Roundoff errors – a solution

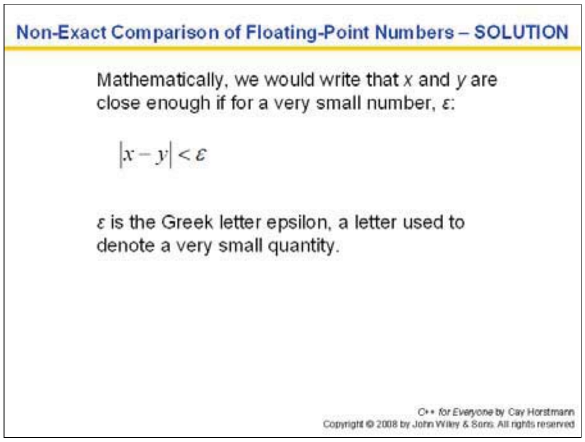
Close enough to zero will do.

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

whatever variables/values you want to be equal

close to 0

1×10^{-14}

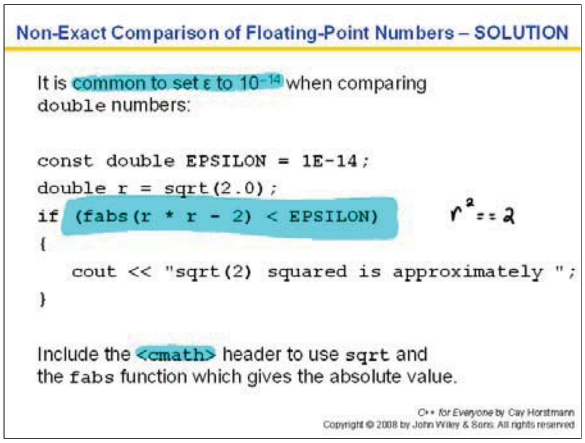


Non-Exact Comparison of Floating-Point Numbers – SOLUTION

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.



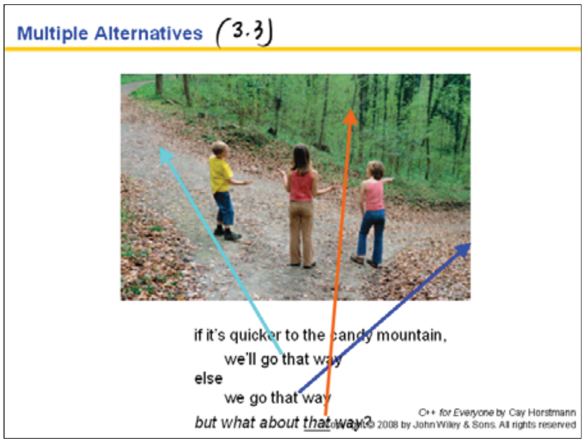
Non-Exact Comparison of Floating-Point Numbers – SOLUTION

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 ";
}
```

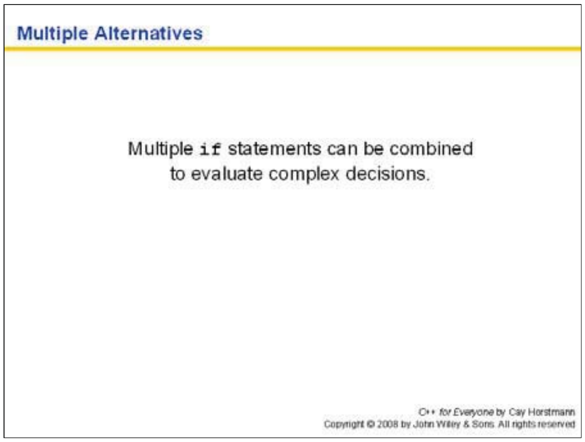
$r^2 == 2$

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



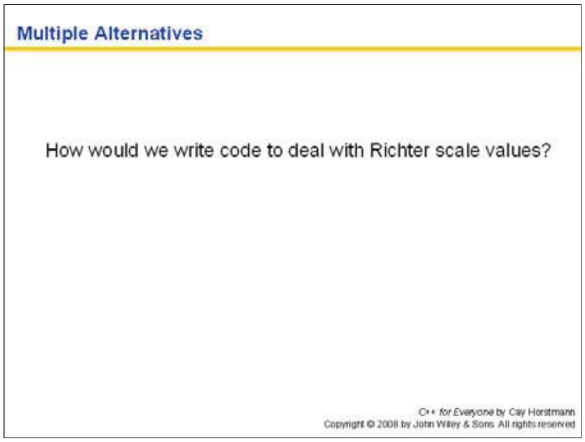
Multiple Alternatives (3.3)

if it's quicker to the candy mountain,
    we'll go that way
else
    we go that way
but what about that way?



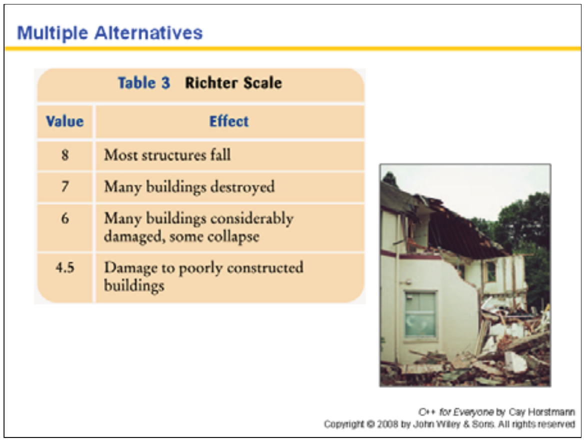
Multiple Alternatives

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



Multiple Alternatives

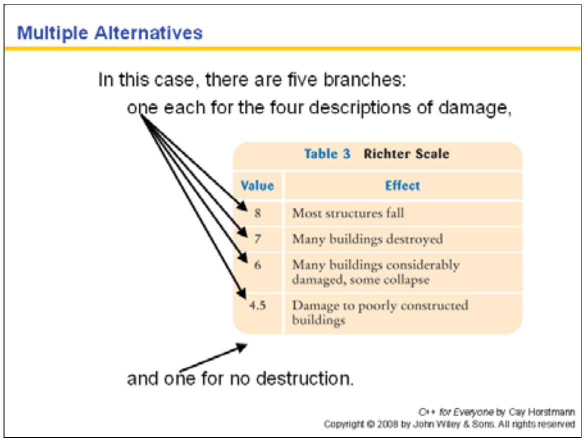
How would we write code to deal with Richter scale values?



Multiple Alternatives

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.