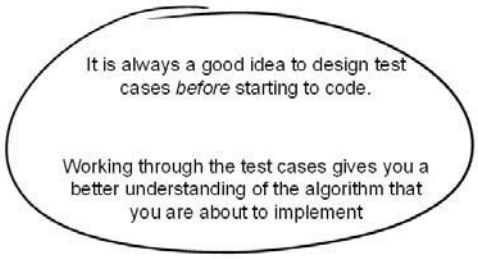
Prepare Test Cases Ahead of Time



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 // Pitfall!
   shipping_charge = 20.00;
                         // As are foreign shipments
```

The Dangling `else` Problem

This is what the code actually is.
And this not what you want.

```
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

This is what the code actually is.
And this not what you want.

And it has a name:

```
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

And it has a name: "the dangling else problem"

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 (3.5)

- 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
(1815–1864),
a pioneer in the study of logic.

He invented an algebra based on only two values.

Boolean Variables and Operators

Two values, eh?

like true and false

like on and off
– like electricity!

In essence he invented the computer!

Boolean Variables and Operators

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.

data types we know: int, double, string, bool

Boolean Variables

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

EX:

```
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

EX:

- 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.

not C++ syntax: $0. < temp < 100.$

$0. < temp$ (and) $temp < 100.$??

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`.

expression	yields
T && T	true
T && F	false
F && T	
F && F	

The Boolean Operator || (or)

double pipe symbol

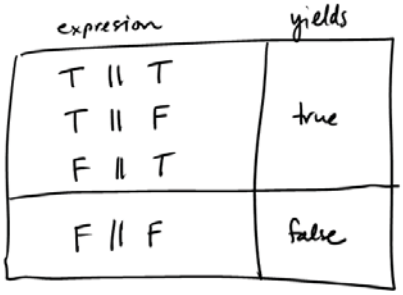
The || operator (called *or*) yields the result `true` if at least one of the conditions is `true`.

- 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`.

! true = false

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` (T, F)	`false`	Only the first condition is true.
`0 < 200 || 200 < 100` (T, F)	`true`	The first condition is true.
`0 < 200 || 100 < 200` (T, T)	`true`	The || is not a test for "either-or". If both conditions are true, the result is true.
`0 < 200 < 100` error — compiler will not catch!	`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

nonzero = true
zero = false

Table 5 Boolean Operators (continued)

Expression	Value	Comment
`-10 && 10 > 0` error but compiler will not catch	`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` ← has precedence	`(0 < x && x < 100) || x == -1`	The && operator binds more strongly than the || operator.
`!(0 < 200)` (T)	`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:

$$0 \leq \text{temp} \leq 100$$

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

```
if ( true  <= 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

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

Is `true` larger than 100 or not?

Combining Multiple Relational Operators

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

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

Combining Multiple Relational Operators

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

(x and y are ints)

Combining Multiple Relational Operators

Naturally, that computation makes no sense.

(But it was a good attempt at translating:
"both x and y must be greater than 0" into
a C++ expression!).

Again, the compiler would not issue an error message.
It would use the C conversions.

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 (3.6)

Input validation is an important part of working with live human beings.

It has been found to be true that, unfortunately, all human beings can mistke makez.

Input Validation with `if` Statements

Let's return to the elevator program and consider input validation.

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 ← non integers
- 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;
}
```

print an error message

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;
}
```

error msg

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.

Stopped here