Chapter Three: Decisions II

Slides by Evan Gallagher & Nikolay Kirov

Lecture Goals

- To understand multiple alternatives and nested branches
- To understand the Boolean data type
- To develop strategies for validating user input

Multiple Alternatives

if it’s quicker to the candy mountain,

    we’ll go that way

else

    we go that way

but what about *<u>that</u>* *way?*

Multiple Alternatives

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

For example, consider a program that displays the effect of an earthquake, as measured by the Richter scale

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

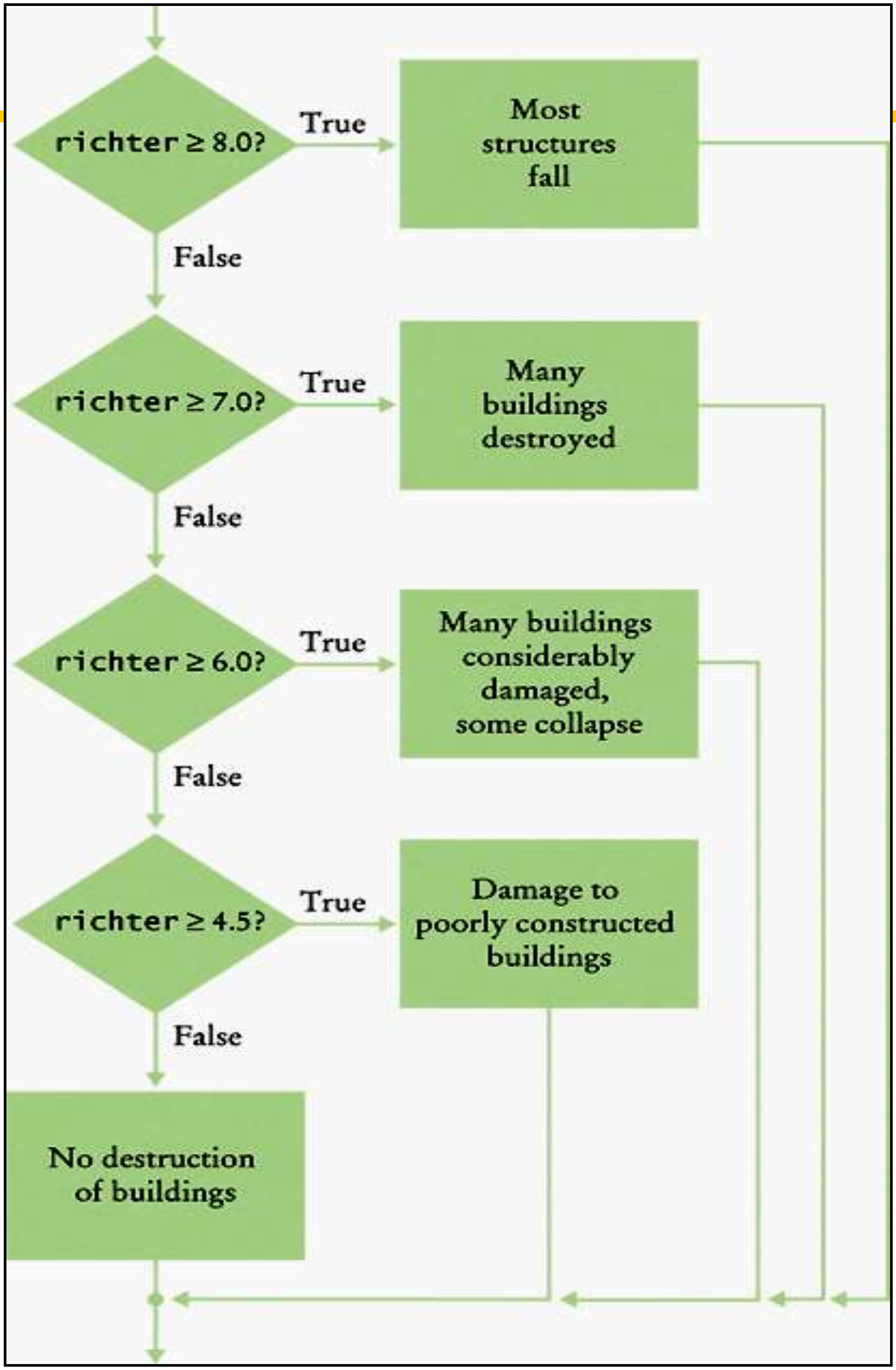
Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

and one for no destruction.

Multiple Alternatives

You use multiple **`if`** statements
to implement multiple alternatives.

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    )    ◄──── If a test is 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";
}
. . .
```

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";
}
. . .
```

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,

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`!

Multiple Alternatives – Wrong Order of Tests

```
if (   true   )    // 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`!

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!),

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!),

and we go…

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

In the United States, different tax rates are used depending on the taxpayer's marital status

Nested Branches – Taxes

Taxes…

Nested Branches – Taxes

What next after line 37?

Taxes…

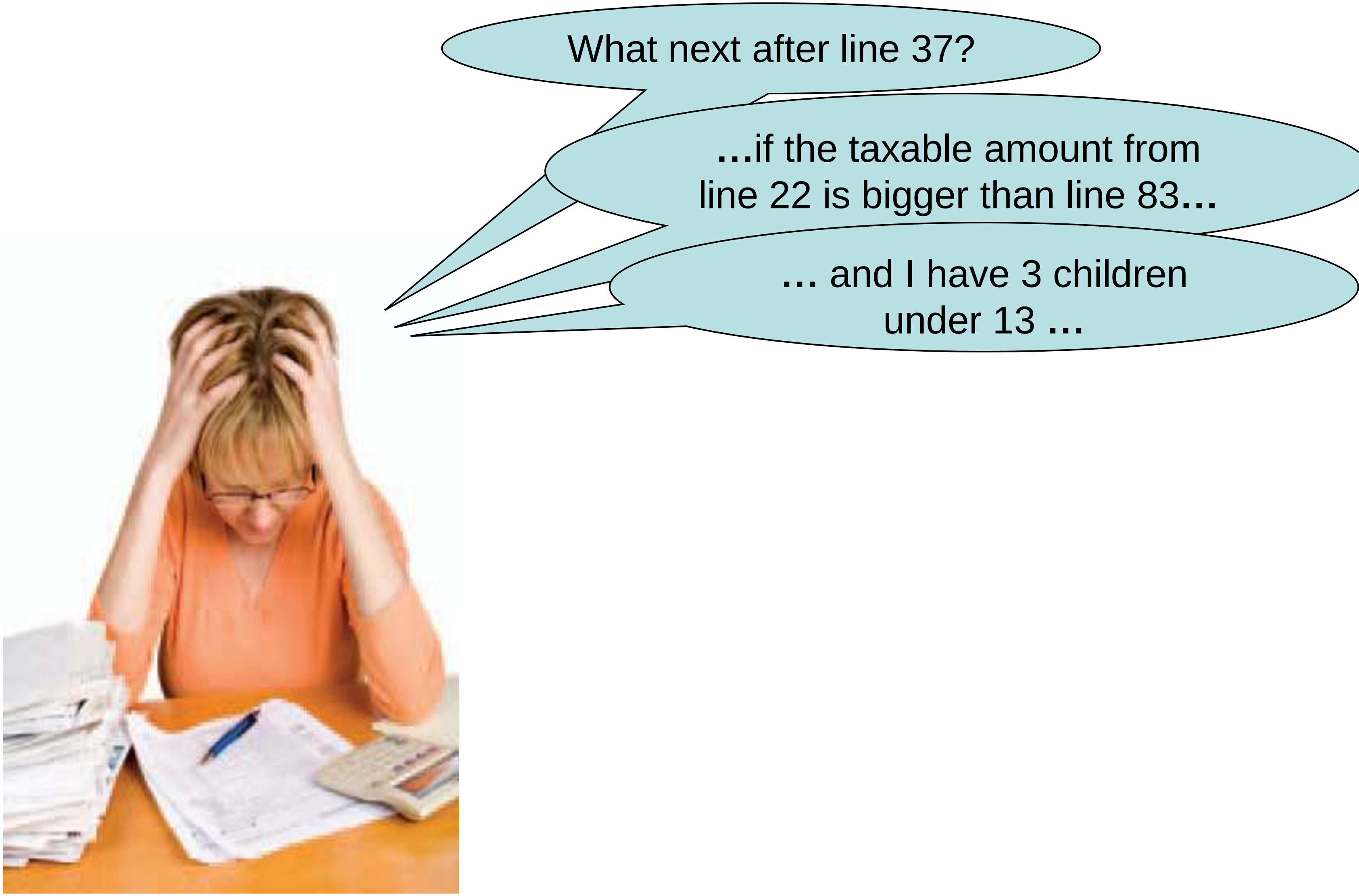
Nested Branches – Taxes

What next after line 37?

… if the taxable amount from line 22 is bigger than line 83 …

Taxes…

Nested Branches – Taxes



Taxes…

Nested Branches – Taxes

What next after line 37?

…if the taxable amount from line 22 is bigger than line 83…

…and I have 3 children under 13…

… unless I'm also married …

Taxes…

Nested Branches – Taxes

Taxes…

What next after line 37?

…if the taxable amount from line 22 is bigger than line 83**…**

…and I have 3 children under 13**…**

…unless I'm also married**…**

AM I STILL MARRIED?!

Nested Branches – Taxes

- In the United States different tax rates are used depending on the taxpayer’s marital status.
- There are different tax schedules for single and for married taxpayers.
- Married taxpayers add their income together and pay taxes on the total.

Nested Branches – Taxes

Let's write the code.

First, as always, we analyze the problem.

Nested Branches – Taxes

Nested branching analysis is aided by drawing tables showing the different criteria.

Thankfully, the I.R.S. has done this for us.

The Internal Revenue Service (I.R.S.) is the U.S. government agency responsible for tax collection and tax law enforcement.

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

Now that you understand,
given a filing status and an income figure,
compute the taxes due.

Nested Branches – Taxes

ARGHHHH!!!!

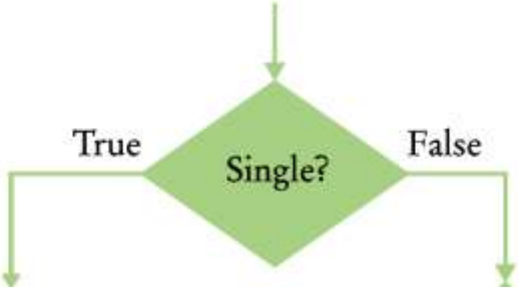
Nested Branches – Taxes

- The key point is that there are two levels of decision making.

Really, only two (at this level).

Nested Branches – Taxes

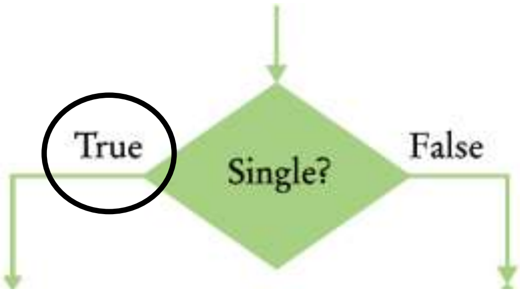
First, you must branch on the marital status.



Nested Branches – Taxes

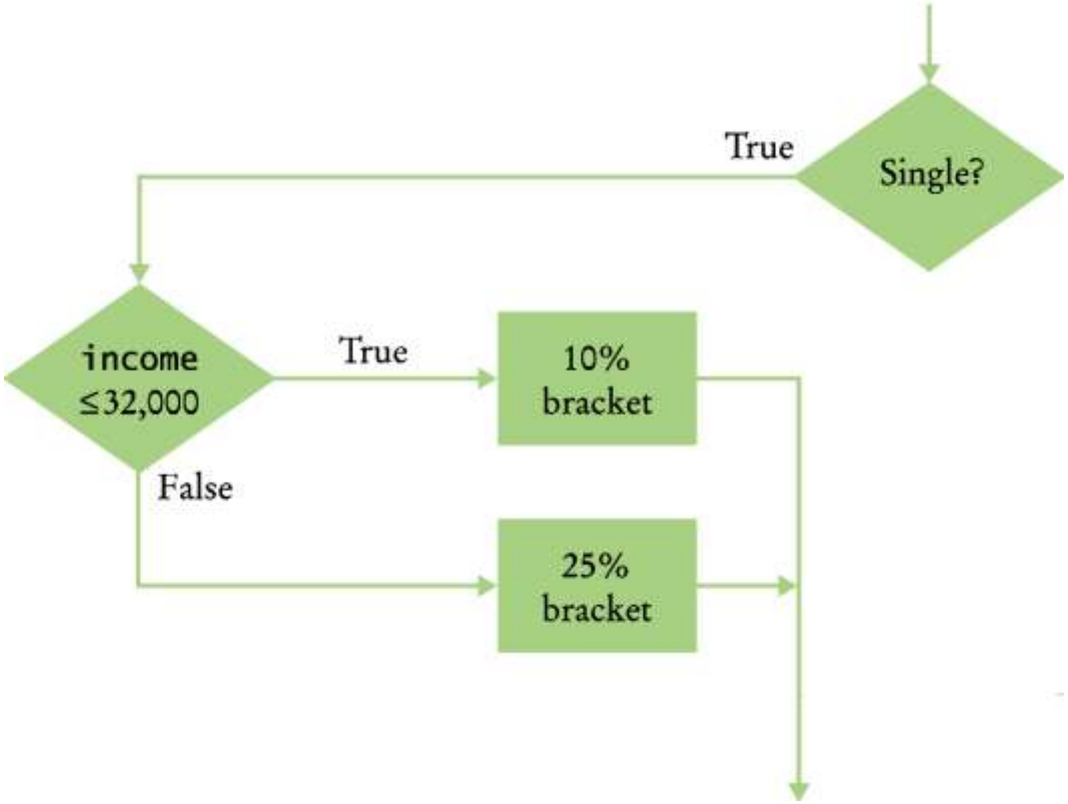
Then, for each filing status,
you must have another branch on income level.

The single filers …



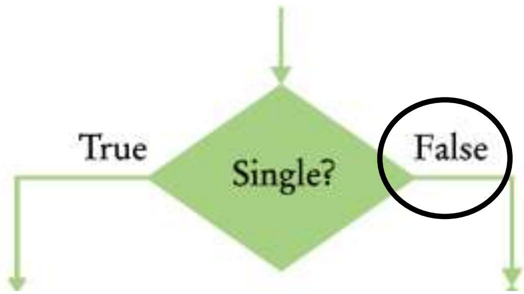
Nested Branches – Taxes

…have their own *nested* **`if`** statement
with the single filer figures.



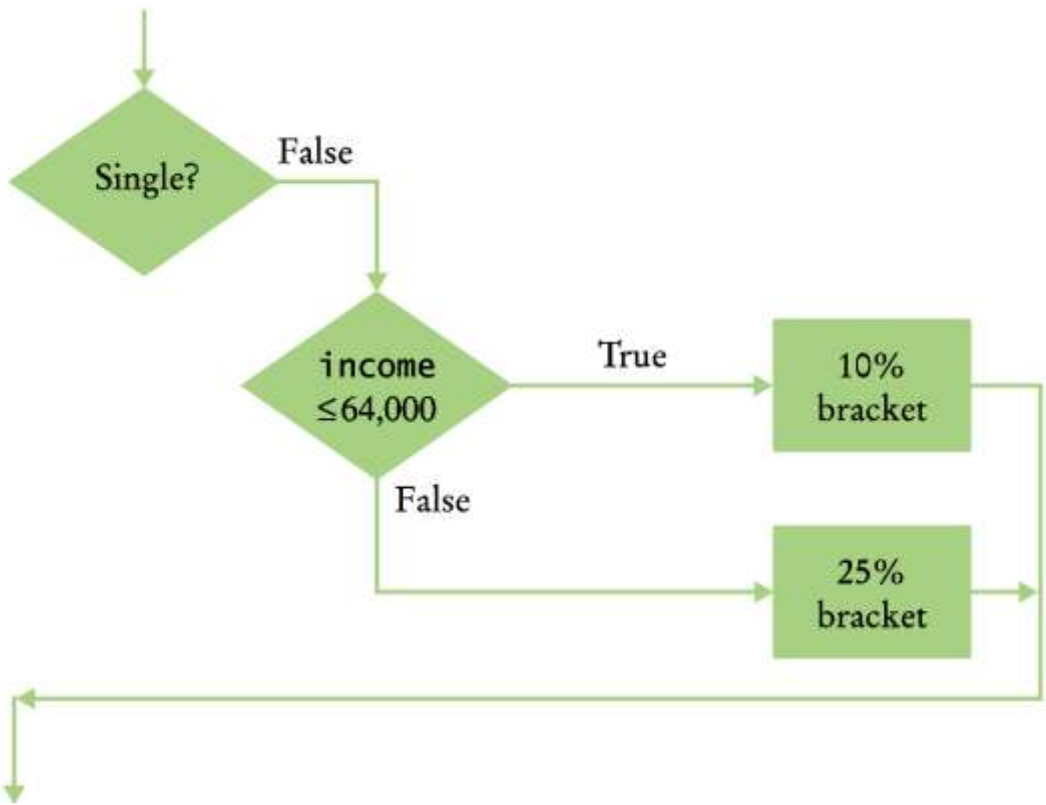
Nested Branches – Taxes

For those with spouses (spice?) …



Nested Branches – Taxes

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



Nested Branches – Taxes

In theory you can have even deeper levels of nesting.

Consider:

- first by state
- then by filing status
- then by income level

This situation requires three levels of nesting.

Nested Branches – Taxes

ch03/tax.cpp

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   const double RATE1 = 0.10;
   const double RATE2 = 0.25;
   const double RATE1_SINGLE_LIMIT = 32000;
   const double RATE1_MARRIED_LIMIT = 64000;

   double tax1 = 0;
   double tax2 = 0;

   double income;
   cout << "Please enter your income: ";
   cin >> income;

   cout << "Please enter s for single, m for married: ";
   string marital_status;
   cin >> marital_status;
```

Nested Branches – Taxes

```
if (marital_status == "s")
   {
      if (income <= RATE1_SINGLE_LIMIT)
      {
         tax1 = RATE1 * income;
      }
      else
      {
         tax1 = RATE1 * RATE1_SINGLE_LIMIT;
         tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
      }
   }
else
```

Nested Branches – Taxes

```
    {
        if (income <= RATE1_MARRIED_LIMIT)
        {
            tax1 = RATE1 * income;
        }
        else
        {
            tax1 = RATE1 * RATE1_MARRIED_LIMIT;
            tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
        }
    }

    double total_tax = tax1 + tax2;

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

Nested Branches – Taxes

In practice two levels of nesting should be enough.

Beyond that you should be calling your own functions.

– But, you don't know to write functions…

…yet

Hand-Tracing

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

- Depending on where you normally work, get:

Hand-Tracing

- Depending on where you normally work, get:
  - an index card

Hand-Tracing

- Depending on where you normally work, get:
  - an index card
  - an envelope

Hand-Tracing

- Depending on where you normally work, get:
  - an index card
  - an envelope (use the back)

Hand-Tracing

- Depending on where you normally work, get:
  - an index card
  - an envelope (use the back)
  - a cocktail napkin

Hand-Tracing

- Depending on where you normally work, get:
  - an index card
  - an envelope (use the back)
  - a cocktail napkin

(!)

Hand-Tracing

Looking at your pseudocode or C++ code,

- Use a marker, such as a paper clip, (or toothpick from an olive) to mark the current statement.
- "Execute" the statements one at a time.
- Every time the value of a variable changes, cross out the old value, and write the new value below the old one.

Hand-Tracing

Let's do this with the tax program.

(take those cocktail napkins out of your pockets and get started!)

Hand-Tracing

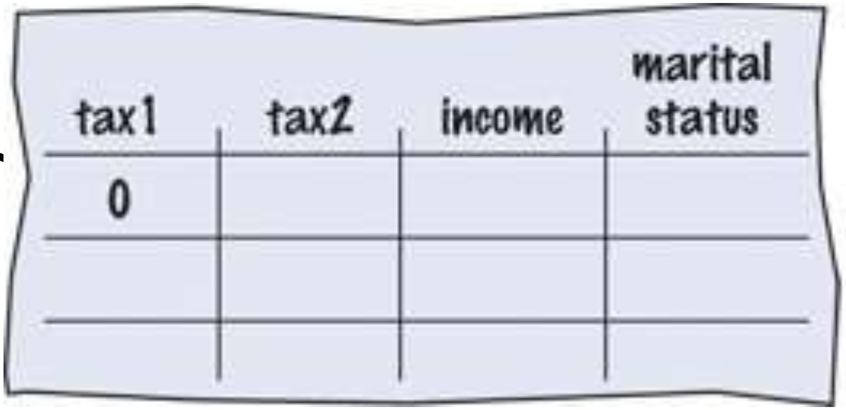
```
int main()
{
   const double RATE1 = 0.10;
   const double RATE2 = 0.25;
   const double RATE1_SINGLE_LIMIT = 32000;
   const double RATE1_MARRIED_LIMIT = 64000;
```

Constants aren't "changes" during execution.

They were created and initialized earlier
so we don't write them in our trace.

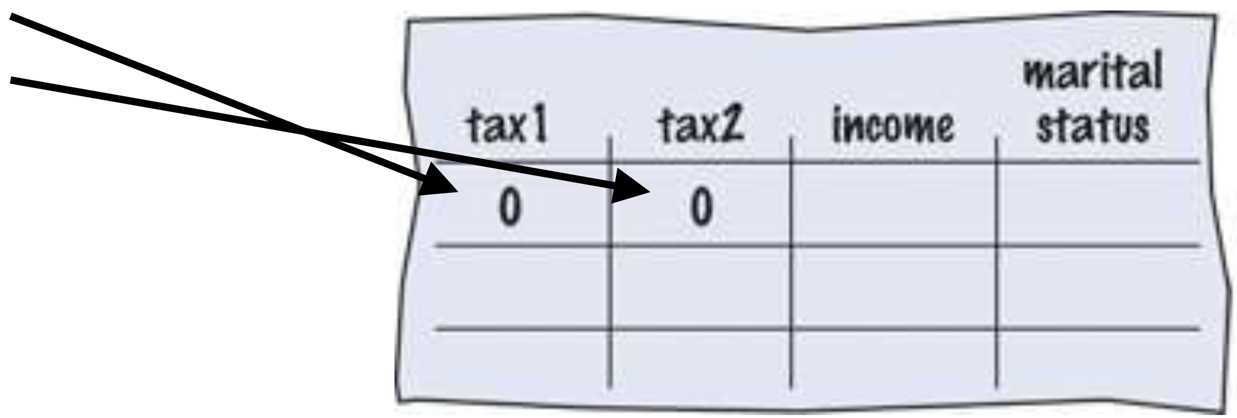
Hand-Tracing

```
int main()
{
    const double RATE1 = 0.10;
    const double RATE2 = 0.25;
    const double RATE1_SINGLE_LIMIT = 32000;
    const double RATE1_MARRIED_LIMIT = 64000;

    double tax1 = 0;
    double tax2 = 0;
```



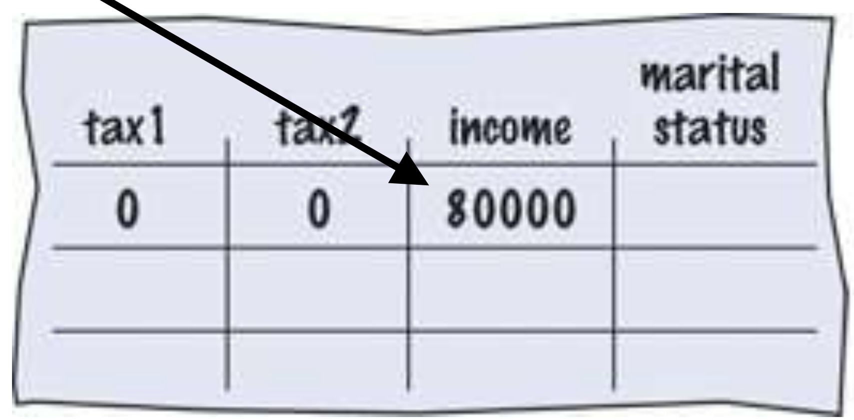
Hand-Tracing

```
int main()
{
    const double RATE1 = 0.10;
    const double RATE2 = 0.25;
    const double RATE1_SINGLE_LIMIT = 32000;
    const double RATE1_MARRIED_LIMIT = 64000;

    double tax1 = 0;
    double tax2 = 0;
```



Hand-Tracing

```
double income;
cout << "Please enter your income: ";
cin >> income;
```



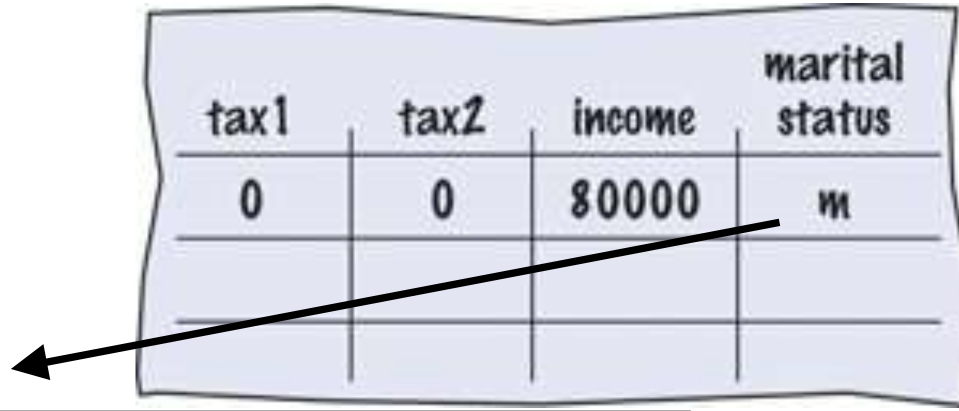
The user typed 80000.

Hand-Tracing

```
double income;
cout << "Please enter your income: ";
cin >> income;

cout << "Please enter s for single, m for married: ";
string marital_status;
cin >> marital_status;
```

The user typed **`m`**

Hand-Tracing




```
if (marital_status == "s")
{
   if (income <= RATE1_SINGLE_LIMIT)
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_SINGLE_LIMIT;
      tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
   }
}
else
```

Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

```
if (         false         )
{
   if (income <= RATE1_SINGLE_LIMIT)
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_SINGLE_LIMIT;
      tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
   }
}
else
```

Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

```
if (marital_status == "s")
{
   if (income <= RATE1_SINGLE_LIMIT)
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_SINGLE_LIMIT;
      tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
   }
}
else
```

Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

```
else
{
   if (income <= RATE1_MARRIED_LIMIT)
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_MARRIED_LIMIT;
      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
   }
}
double total_tax = tax1 + tax2;
```

Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

```
else
{
   if (income <=      64000      )
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_MARRIED_LIMIT;
      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
   }
}
double total_tax = tax1 + tax2;
```

Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

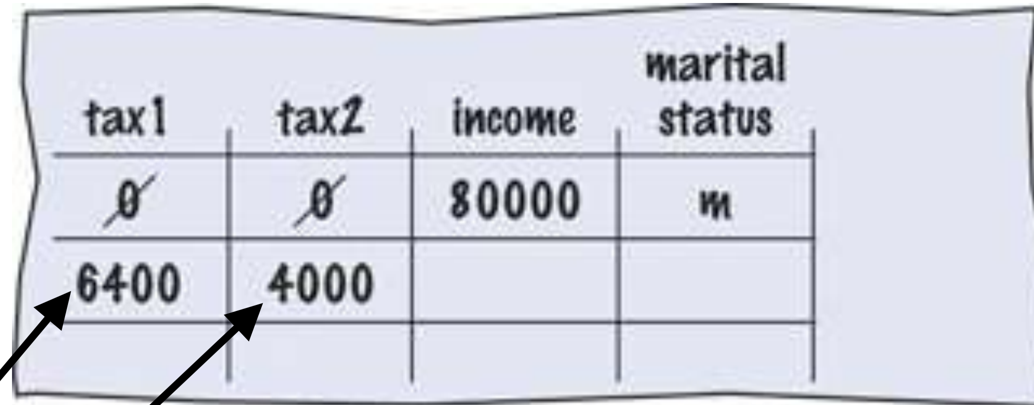
```
else
{
   if (         false          )
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_MARRIED_LIMIT;
      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
   }
}
double total_tax = tax1 + tax2;
```

Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

```
else
{
   if (income <= RATE1_MARRIED_LIMIT)
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_MARRIED_LIMIT;
      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
   }
}
double total_tax = tax1 + tax2;
```

Hand-Tracing




```
else
{
   if (income <= RATE1_MARRIED_LIMIT)
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_MARRIED_LIMIT;
      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
   }
}
double total_tax = tax1 + tax2;
```

Hand-Tracing

tax1	tax2	income	marital status
~~0~~	~~0~~	80000	m
6400	4000		

```
else
{
   if (income <= RATE1_MARRIED_LIMIT)
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_MARRIED_LIMIT;
      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
   }
}
double total_tax = tax1 + tax2;
```

Hand-Tracing

tax1	tax2	income	marital status
~~0~~	~~0~~	80000	m
6400	4000		

```
else
{
   if (income <= RATE1_MARRIED_LIMIT)
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_MARRIED_LIMIT;
      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
   }
}
double total_tax = tax1 + tax2;
```

Hand-Tracing

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

```
else
{
   if (income <= RATE1_MARRIED_LIMIT)
   {
      tax1 = RATE1 * income;
   }
   else
   {
      tax1 = RATE1 * RATE1_MARRIED_LIMIT;
      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
   }
}
double total_tax = tax1 + tax2;
```

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;
}
```

Prepare Test Cases Ahead of Time

Consider how to *test* the tax computation program.

Of course, you cannot try out all possible inputs of filing status and income level.

Even if you could, there would be no point in trying them all.

Prepare Test Cases Ahead of Time

If the program correctly computes one or two tax amounts in a given bracket, then we have a good reason to believe that all amounts will be correct.

You should also test on the *boundary conditions*, at the endpoints of each bracket

this tests the < vs. <= situations.

Prepare Test Cases Ahead of Time

There are two possibilities for the filing status and two tax brackets for each status, yielding four test cases.

- Test a handful of boundary conditions, such as an income that is at the boundary between two brackets, and a zero income.
- If you are responsible for error checking, also test an invalid input, such as a negative income.

Prepare Test Cases Ahead of Time

Here are some possible test cases for the tax program:

Test Case	**Expected**	**Output Comment**
30,000 s	**3,000**	**10% bracket**
72,000 s	**13,200**	**3,200 + 25% of 40,000**
50,000 m	**5,000**	**10% bracket**
10,400 m	**16,400**	**6,400 + 25% of 40,000**
32,000 m	**3,200**	**boundary case**
0		**0 boundary case**

Prepare Test Cases Ahead of Time

It is always a good idea to design test cases *before* starting to code.

Working through the test cases gives you a better understanding of the algorithm that you are about to implement.

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 is 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
      shipping_charge = 20.00;
                       // As are foreign shipments
```

The Dangling `else` Problem

This is what the code actually is.

And this is not what you want.

And it has a name: “the dangling **else** problem”

```
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

Will we remember next time?
I wish I could put the way to go in my pocket!

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 are named after the mathematician George Boole (1815–1864), a pioneer in the study of logic.

Boolean Variables and Operators

He invented an algebra based on only two values.

Two values, eh?

like true and false

like on and off

– like electricity!

In essence he invented the computer!

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 **<u>not</u>** strings.
- There values are *definitely* **<u>not</u>** 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 Variables

Sometimes bool variables are called “flag” variables.

The flag is either up or down.

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,* conjunction) 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*, disjunction) 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 expressions 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	true	false
false	false	false

A	B	A || B
true	true	true
true	false	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 6 Boolean Operators

Expression	Value	Comment
`0 < 200 && 200 < 100`	`false`	Only the first condition is true. Note that the < operator has a higher precedence than the && operator.
`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 < 100` is true. You never want to write such an expression; see Common Error 3.5 on page 107.

Boolean Operators – Some Examples

`-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 on page 107.
`0 < x && x < 100 || x == -1`	`(0 < x && x < 100) || x == -1`	The && operator has a higher precedence 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`.

Common Error – Combining Multiple Relational Operators

Consider the expression

```
if (0 <= temp <= 100)…
```

This looks just like the mathematical test:

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

Unfortunately, it is not.

Common Error – 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`**.

Common Error – 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 integer numbers?

Common Error – Combining Multiple Relational Operators

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

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

Common Error – 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.

Common Error – 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`**.

Common Error – 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 **`int`**s)

Common Error – 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.

Common Error – Confusing `&&` and `||` Conditions

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.

Common Error – Confusing `&&` and `||` Conditions

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.

Common Error – Confusing `&&` and `||` Conditions

Elsewhere, the same instructions:

You may use the status of married filing jointly
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 **`&&`**.

Nested Branches –Taxes

Wait, I *am* still married

Taxes…

Nested Branches –Taxes

Wait, I *am* still married
—according to the IRS?!

Taxes…

Short Circuit Evaluation

When does an expression become **`true`** or **`false`**?
And once sure, why keep doing anything?

`expression && expression && expression &&` …

In an expression involving a series of &&'s,
we can stop after finding the first **`false`**.

Due to the way the truth table works,
anything and **`&& false`** is **`false`**.

`expression || expression || expression ||` …

In an expression involving a series of **`||`**'s,
we can stop after finding the first **`true`**.

Due to the way the truth table works,
anything and **`|| true`** is **`true`**.

Short Circuit Evaluation

C++ does stop when it is sure of the value.

This is called *short circuit evaluation*.

But not the shocking kind.

DeMorgan’s Law

Suppose we want to charge a higher shipping rate
if we don’t ship within the continental United States.

```
shipping_charge = 10.00;
if (!(country == "USA"
      && state != "AK"
      && state != "HI"))
   shipping_charge = 20.00;
```

This test is a little bit complicated.

DeMorgan’s Law to the rescue!

DeMorgan’s Law

DeMorgan’s Law allows us to rewrite complicated *not/and/or* messes so that they are more clearly read.

```
shipping_charge = 10.00;
if (country != "USA"
     || state == "AK"
     || state == "HI")
   shipping_charge = 20.00;
```

Ah, much nicer.

But how did they do that?

DeMorgan's Law

DeMorgan's Law:

`!(A && B)` is the same as **`!A || !B`**

(change the **`&&`** to **`||`** and negate all the terms)

`!(A || B)` is the same as **`!A && !B`**

(change the **`||`** to **`&&`** and negate all the terms)

DeMorgan’s Law

So

!(country == "USA“ && state != "AK" && state != "HI")

becomes:

!(country == "USA“) || !(state != "AK“) || !(state != "HI")

and then we make those silly !(...==...)’s and !(...!=...)’s better by making !(==) be just != and !(!=) be just ==.

country != "USA“ || state == "AK“ || state == "HI"

Input Validation with `if` Statements

You, the C++ programmer, doing Quality Assurance

(by hand!)

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”
- 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.

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;
}
```

Input Validation with `if` Statements

Later you will learn more robust ways to deal with bad input, but for now just exiting `main` with an error report is enough.

Here's the whole program with validity testing:

Input Validation with `if` Statements – Elevator Program

ch03/elevator2.cpp

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

int main()
{
   int floor;
   cout << "Floor: ";
   cin >> floor;

   // The following statements check various input errors
   if (cin.fail())
   {
      cout << "Error: Not an integer." << endl;
      return 1;
   }
   if (floor == 13)
   {
      cout << "Error: There is no thirteenth floor." << endl;
      return 1;
   }
   if (floor <= 0 || floor > 20)
   {
      cout << "Error: The floor must be between 1 and 20." << endl;
      return 1;
   }
```

Input Validation with `if` Statements – Elevator Program

```
    // Now we know that the input is valid
    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;
 }
```

Chapter Summary

Use the `if` statement to implement a decision.

- The **`if`** statement allows a program to carry out different actions depending on the nature of the data to be processed.

Implement comparisons of numbers and objects.

- Relational operators (**`< <= > >= == !=`**) are used to compare numbers and strings.
- Lexicographic order is used to compare strings.

Implement complex decisions that require multiple if statements.

- Multiple alternatives are required for decisions that have more than two cases.
- When using multiple **`if`** statements, pay attention to the order of the conditions.

Chapter Summary

Implement decisions whose branches require further decisions.

- When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.
- Nested decisions are required for problems that have two levels of decision making.

Draw flowcharts for visualizing the control flow of a program.

- Flow charts are made up of elements for tasks, input/ outputs, and decisions.
- Each branch of a decision can contain tasks and further decisions.
- Never point an arrow inside another branch.

Design test cases for your programs.

- Each branch of your program should be tested.
- It is a good idea to design test cases before implementing a program.

Use the `bool` data type to store and combine conditions that can be `true` or `false`.

- The **`bool`** type bool has two values, **`false`** and **`true`**.
- C++ has two Boolean operators that combine conditions: **`&&`** (*and*) and **`||`** (*or*).
- To invert a condition, use the **`!`** (*not*) operator.
- The **`&&`** and **`||`** operators use *short-circuit evaluation*: As soon as the truth value is determined, no further conditions are evaluated.
- De Morgan's law tells you how to negate **`&&`** and **`||`** conditions.

Chapter Summary

Apply `if` statements to detect whether user input is valid.

- When reading a value, check that it is within the required range.
- Use the fail function to test whether the input stream has failed.

End Decisions II

Slides by Evan Gallagher & Nikolay Kirov

C++ for Everyone by Cay Horstmann