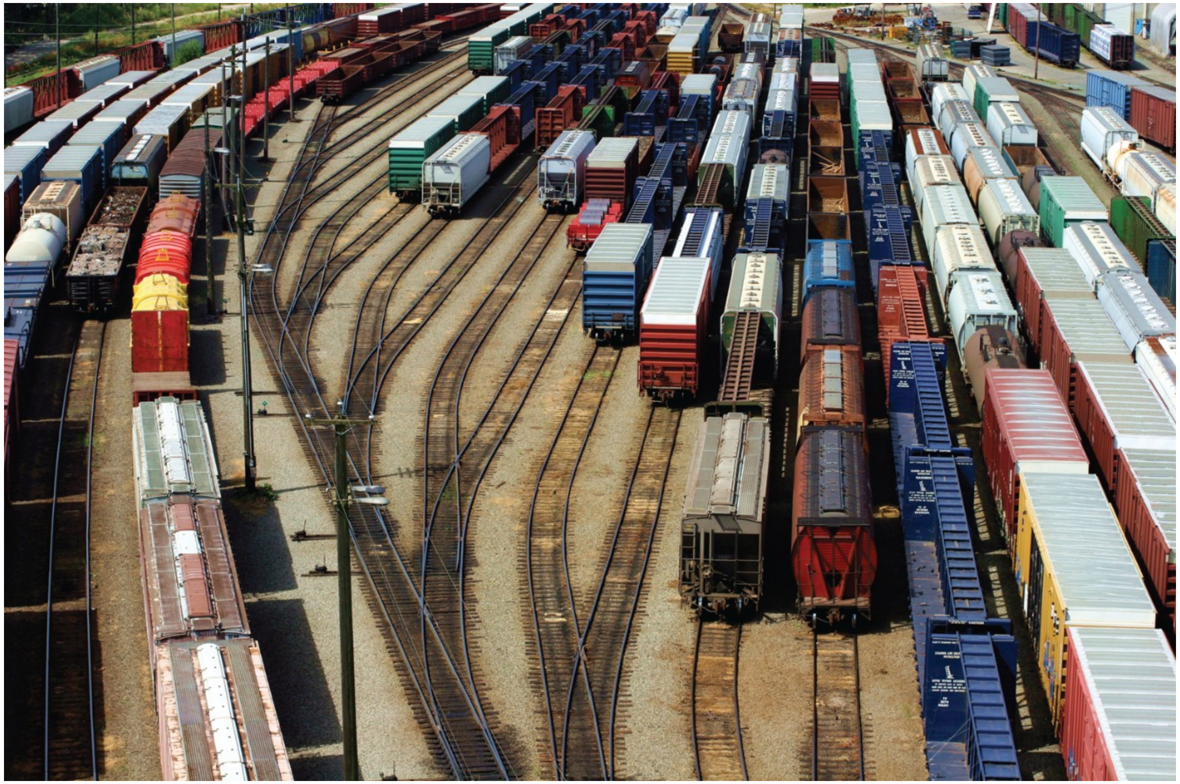


Chapter Three: Decisions

C++ for Everyone by Cay Horstmann

Slides by Evan Gallagher

Chapter Goals

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

The `if` Statement

Decision making

(a necessary thing in non-trivial programs)

The `if` Statement

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

  we’ll go that way

else

  we go that way

The `if` Statement

The thirteenth floor is missing.

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 = 0;
printf("Floor: ");
scanf("%d", &floor);
```

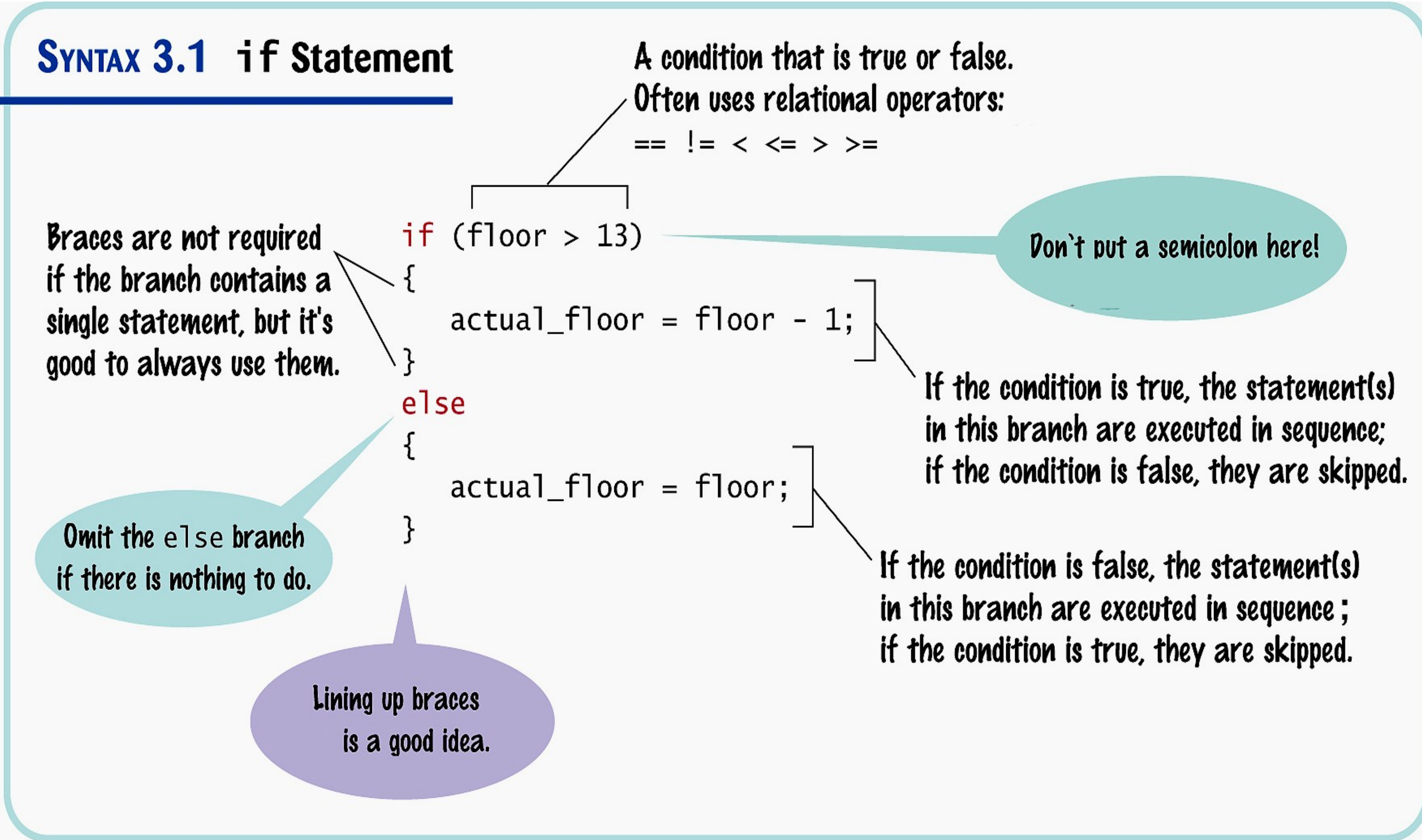
The `if` Statement

If the user inputs 20,
the program must set the actual floor to 19.
Otherwise,
we simply use the supplied floor number.

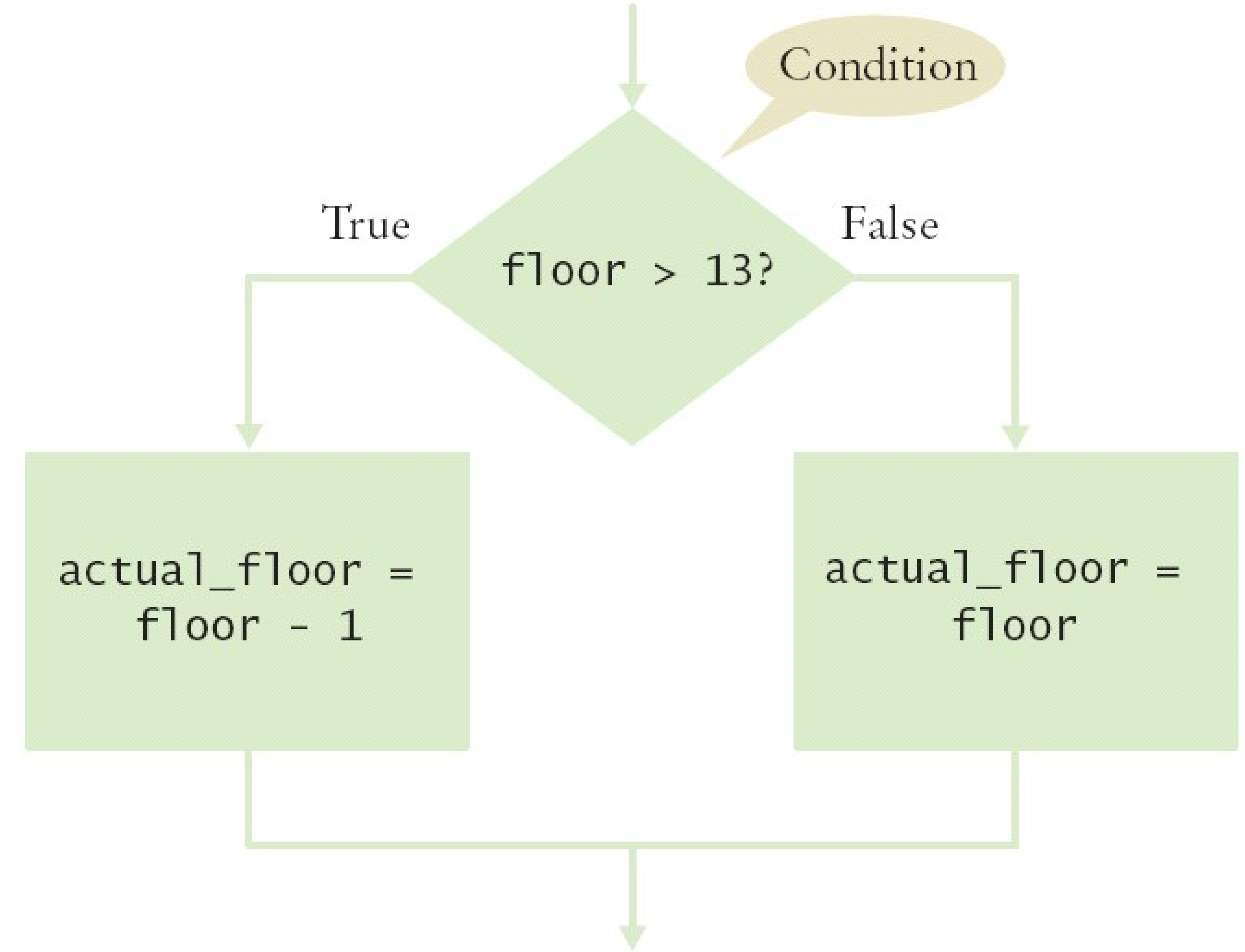
We need to decrement the input only under a certain condition:

```
int actual_floor = 0;
if (floor > 13)
{
   actual_floor = floor - 1;
}
else
{
   actual_floor = floor;
}
```

The `if` Statement



The `if` Statement – The Flowchart



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:

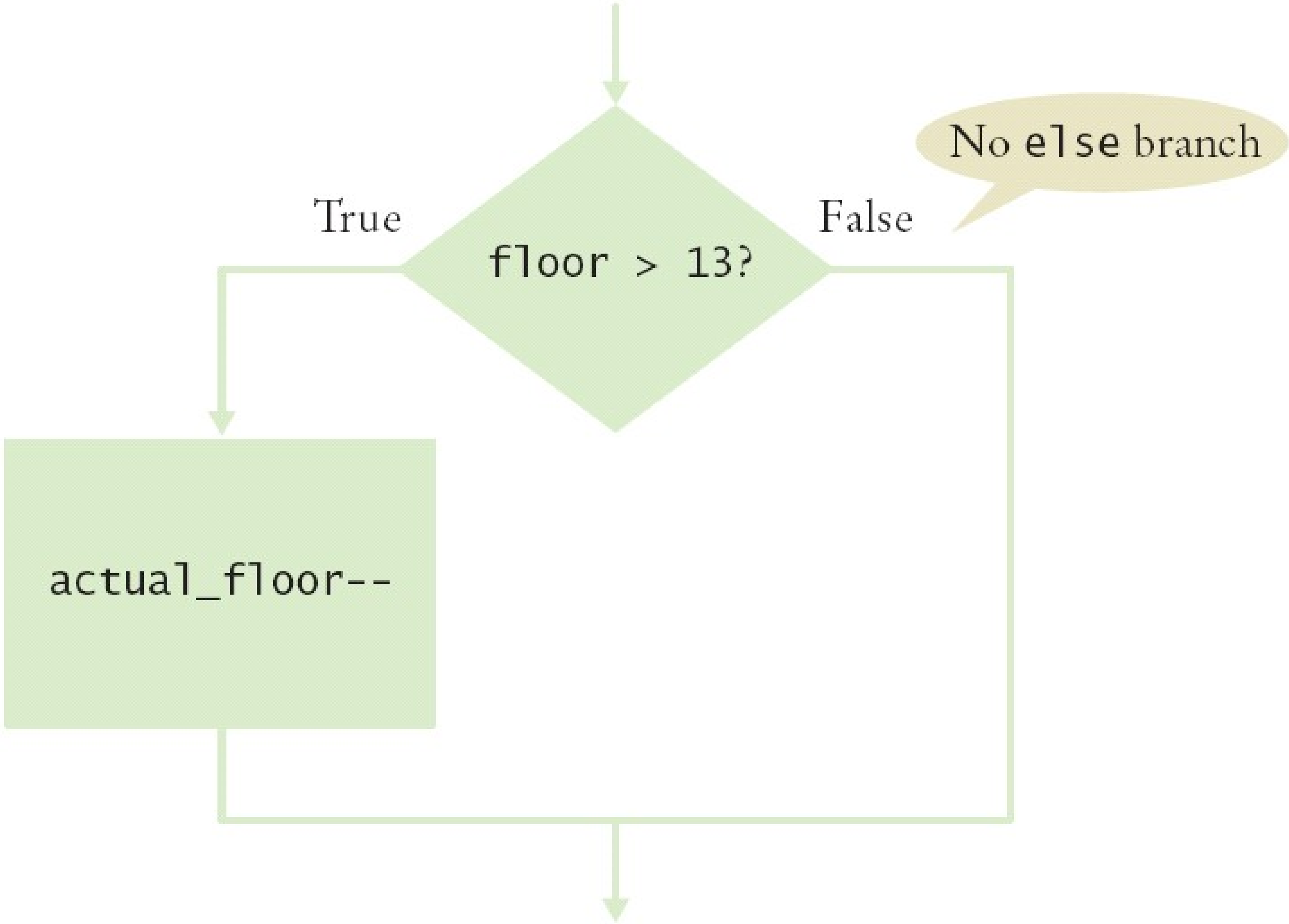
We only need to decrement
when the floor is greater than 13.

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

```
#include <stdio.h>

int main()
{
   int floor = 0;
   printf("Floor: ");
   scanf("%d", &floor);
   int actual_floor = 0;
   if (floor > 13)
   {
      actual_floor = floor - 1;
   }
   else
   {
      actual_floor = floor;
   }

   printf("The elevator will travel to the floor %d.\n",
          actual_floor);

   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) ;  ERROR
{
   floor--;
}
```

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

It is not compiler error. But it *is* a computation error.

The `if` Statement – Indent when Nesting

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

```
int main()
{
    int floor;
    ...
    if (floor > 13)
    {
        floor--;
    }
    ...
    return 0;
}
0   1   2
```

Indentation level

The `if` Statement – 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

Sometimes you might find yourself wanting to do this:

```
printf("%d", 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:

```
printf("Actual floor: %d\n",
       floor > 13 ? floor - 1 : floor);
```

The `if` Statement – Removing Duplication

```
if (floor > 13)
{
   actual_floor = floor - 1;
   printf("Actual floor: %d\n", actual_floor);
}
else
{
   actual_floor = floor;
   printf("Actual floor: %d\n", actual_floor);
}
```

Do you find anything curious in this code?

The `if` Statement – Removing Duplication

```
if (floor > 13)
{
   actual_floor = floor - 1;
   printf("Actual floor: %d\n", actual_floor);
}
else
{
   actual_floor = floor;
   printf("Actual floor: %d\n", actual_floor);
}
```

Do these depend
on the test?

The `if` Statement – Removing Duplication

```
if (floor > 13)
{
   actual_floor = floor - 1;
}
else
{
   actual_floor = floor;
}
printf("Actual floor: %d\n", actual_floor);
```

You should remove
this duplication.

Relational Operators

Which way *is* quicker?

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
`>=`	$\geq$	Greater than or equal
`<`	$<$	Less than
`<=`	$\leq$	Less than or equal
`==`	$=$	Equal
`!=`	$\neq$	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.

C++ for Everyone by Cay Horstmann

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

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)
```

Common Error – Confusing = and ==

The C language allows the use of = inside tests.

Earlier versions of 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.

Common Error – Confusing = and ==

Furthermore, in 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.

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

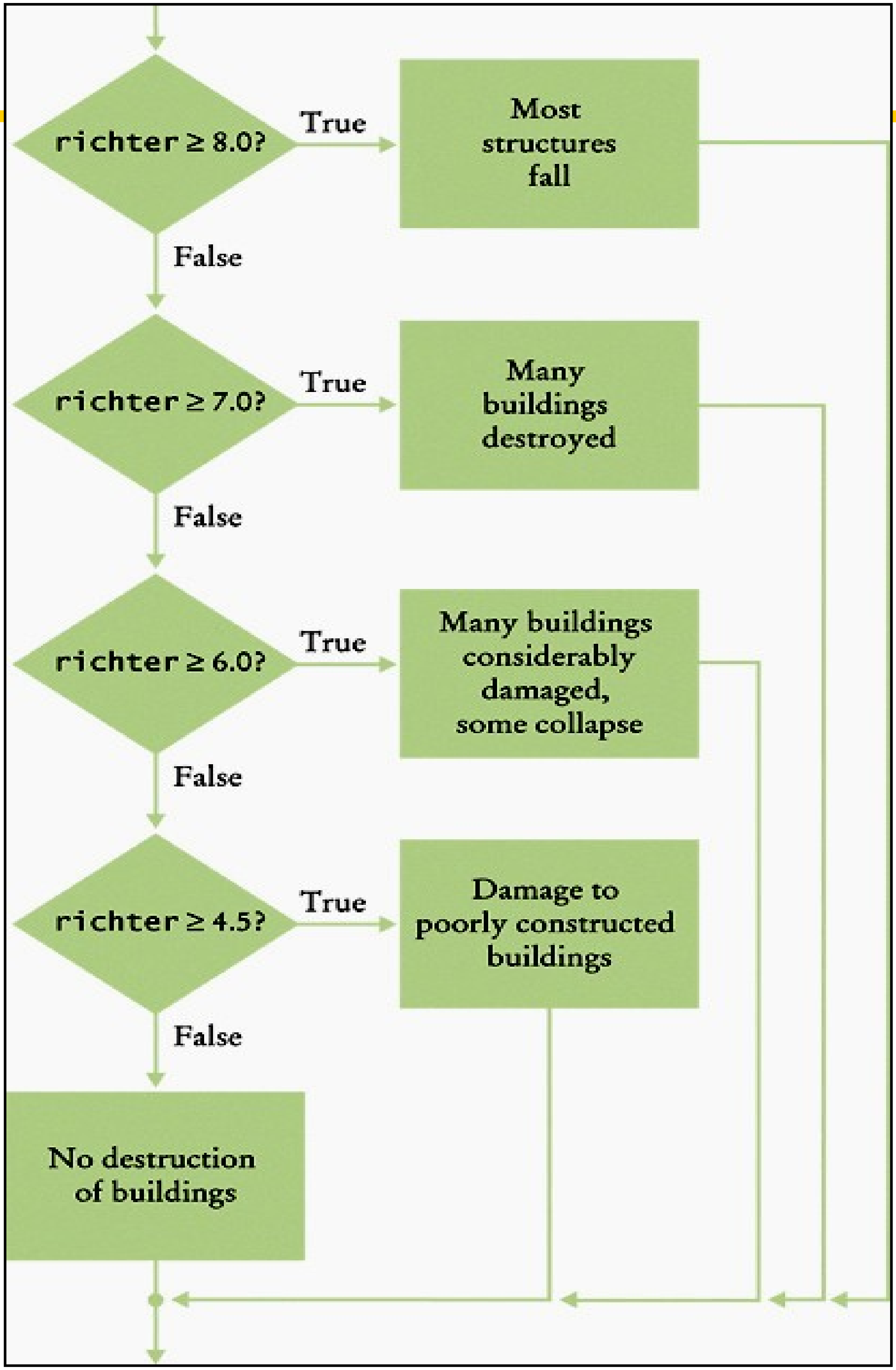
Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

and one for no destruction.

Multiple Alternatives

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

Richter flowchart



C++ for Everyone by Cay Horstmann

Multiple Alternatives

```
if (richter >= 8.0) {
   printf("most fall\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 4.5) {
   printf("damage to weak\n");
} else {
   printf("no destruction\n");
}
. . .
```

Multiple Alternatives

```
if (richter >= 8.0) {
   printf("most fall\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 4.5) {
   printf("damage to weak\n");
} else {
   printf("no destruction\n");
}
```

If a test is false,

Multiple Alternatives

If a test is false,

```
if (    false      ) {
   printf("most fall\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 4.5) {
   printf("damage to weak\n");
} else {
   printf("no destruction\n");
}
```

Multiple Alternatives

```
if (richter >= 8.0) {
   printf("most fall\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 4.5) {
   printf("damage to weak\n");
} else {
   printf("no destruction\n");
}
```

If a test is false, that block is skipped

Multiple Alternatives

```
if (richter >= 8.0) {
   printf("most fall\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 4.5) {
   printf("damage to weak\n");
} else {
   printf("no destruction\n");
}
```

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

Multiple Alternatives

```
if (richter >= 8.0) {
    printf("most fall\n");
} else if (richter >= 7.0) {
    printf("many destroyed\n");
} else if (richter >= 6.0) {
    printf("many damaged, some collapse\n");
} else if (richter >= 4.5) {
    printf("damage to weak\n");
} else {
    printf("no destruction\n");
}
. . .
```

As soon as one of the four tests succeeds,

Multiple Alternatives

As soon as one of the four tests succeeds,

```
if (richter >= 8.0) {
   printf("most fall\n");
} else if (    true        ) {
   printf("many destroyed\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 4.5) {
   printf("damage to weak\n");
} else {
   printf("no destruction\n");
}
. . .
```

Multiple Alternatives

```
if (richter >= 8.0) {
   printf("most fall\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 4.5) {
   printf("damage to weak\n");
} else {
   printf("no destruction\n");
}
. . .
```

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

Multiple Alternatives

```
if (richter >= 8.0) {
   printf("most fall\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 4.5) {
   printf("damage to weak\n");
} else {
   printf("no destruction\n");
}
. . .
```

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

```
// Tests in wrong order
if (richter >= 4.5) {
   printf("damage to weak\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 8.0) {
   printf("most fall\n");
}
...
```

Multiple Alternatives – Wrong Order of Tests

```
// Tests in wrong order
if (richter >= 4.5) {
   printf("damage to weak\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 8.0) {
   printf("most fall\n");
}
...
```

Suppose the value of `richter` is 7.1,

Multiple Alternatives – Wrong Order of Tests

```
// Tests in wrong order
if (richter >= 4.5) {
   printf("damage to weak\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 8.0) {
   printf("most fall\n");
}
...
```

Suppose the value of **richter** is 7.1, this test is **true**

Multiple Alternatives – Wrong Order of Tests

```
// Tests in wrong order
if (    true        ) {
   printf("damage to weak\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 8.0) {
   printf("most fall\n");
}
...
```

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

Multiple Alternatives – Wrong Order of Tests

```
// Tests in wrong order
if (richter >= 4.5) {
   printf("damage to weak\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 8.0) {
   printf("most fall\n");
}
...
```

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

and that block is executed,

Multiple Alternatives – Wrong Order of Tests

```
// Tests in wrong order
if (richter >= 4.5) {
   printf("damage to weak\n");
} else if (richter >= 6.0) {
   printf("many damaged, some collapse\n");
} else if (richter >= 7.0) {
   printf("many destroyed\n");
} else if (richter >= 8.0) {
   printf("most fall\n");
}
...
```

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

and that block is executed,

and we go…

The `switch` Statement

This is a bit of a mess to read.

```
int digit;
...
if (digit == 1) {
    digit_name = "one";
} else if (digit == 2) {
    digit_name = "two";
} … {
} else if (digit == 9) {
    digit_name = "nine";
} else {
    digit_name = "";
}
```

The `switch` Statement

C has a statement that helps a bit with the readability of situations like this:

The **switch** statement.

ONLY a sequence of **if** statements that compares a single integer value against several constant alternatives can be implemented as a **switch** statement.

The `switch` Statement

```
switch (digit) {
    case 1:
        digit_name = "one";
        break;
    case 2:
        digit_name = "two";
        break;
    …:
    case 9:
        digit_name = "nine";
        break;
    default:
        digit_name = "";
        break;
}
```

Nested Branches

It is possible to have multiple case clauses for a branch:

```
case 1:
case 3:
case 5:
case 7:
case 9:
    odd = true;
    break;
```

The **default:** branch is chosen if none of the case clauses match.

Nested Branches

Every branch of the switch must be terminated by a **break** statement.

If the **break** is missing, execution falls through to the next branch, and so on, until finally a **break** or the end of the switch is reached.

In practice, this fall-through behavior is rarely useful, and it is a common cause of errors.

If you accidentally forget the **break** statement, your program compiles but executes unwanted code.

Nested Branches

Many programmers consider the **switch** statement somewhat dangerous and prefer the **if** statement.

It certainly is not needed and if you can't write your code using **if**, you can't even think about using **switch**.

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

What next after line 37?

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

... and I have 3 children under 13 ...

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

- 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.

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

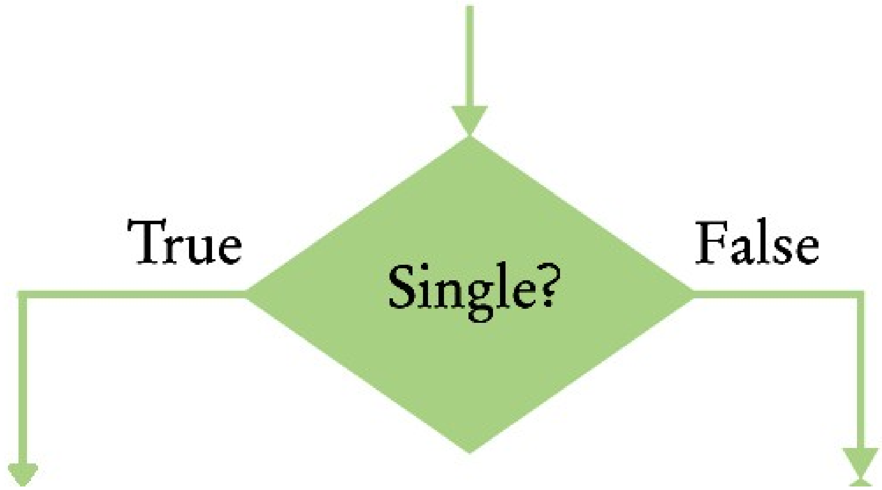
Now that you understand,
given a filing status and an income figure,
compute the taxes due.

Nested Branches – Taxes

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

Nested Branches – Taxes

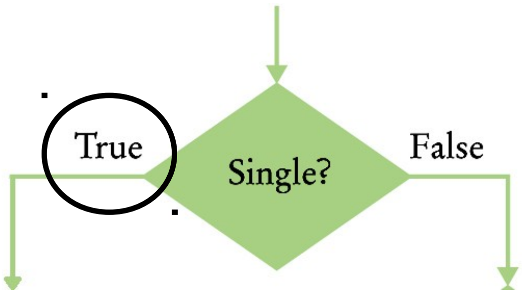
First, you must branch on the marital status.



Nested Branches – Taxes

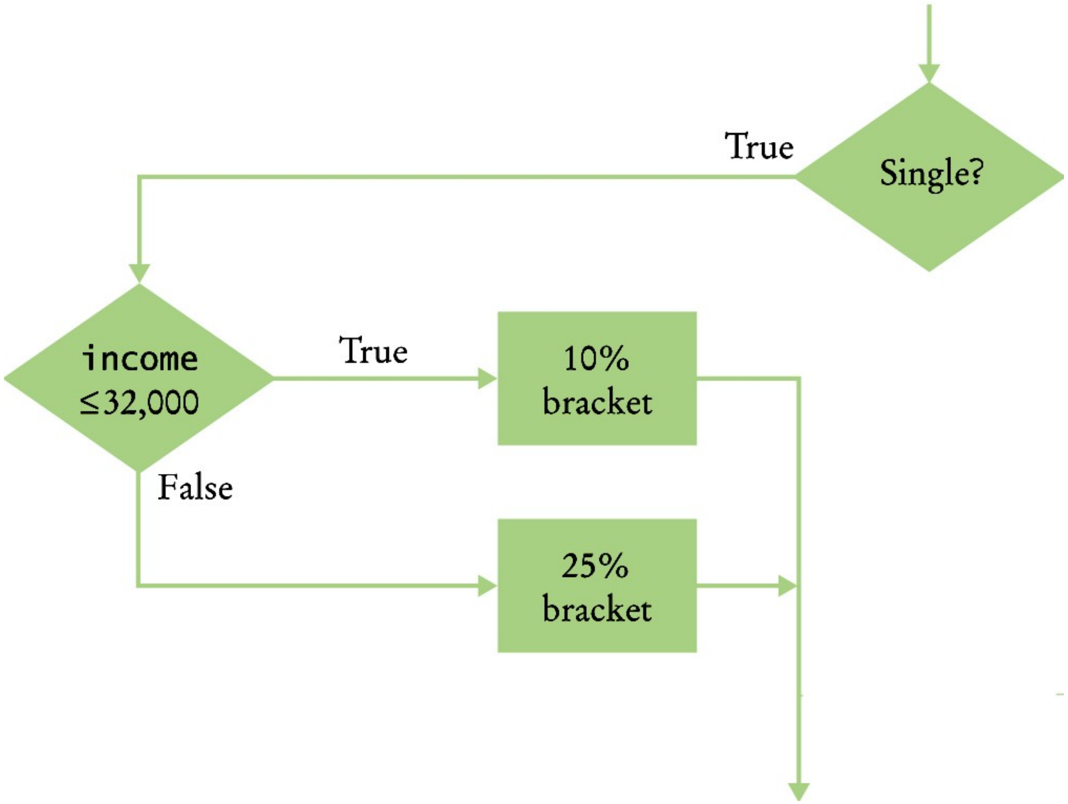
Then, for each filing status,
you must have another branch on income level.

The single filers …



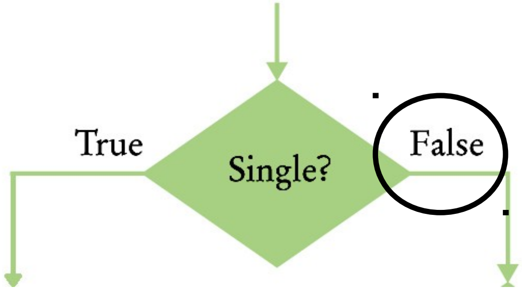
Nested Branches – Taxes

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



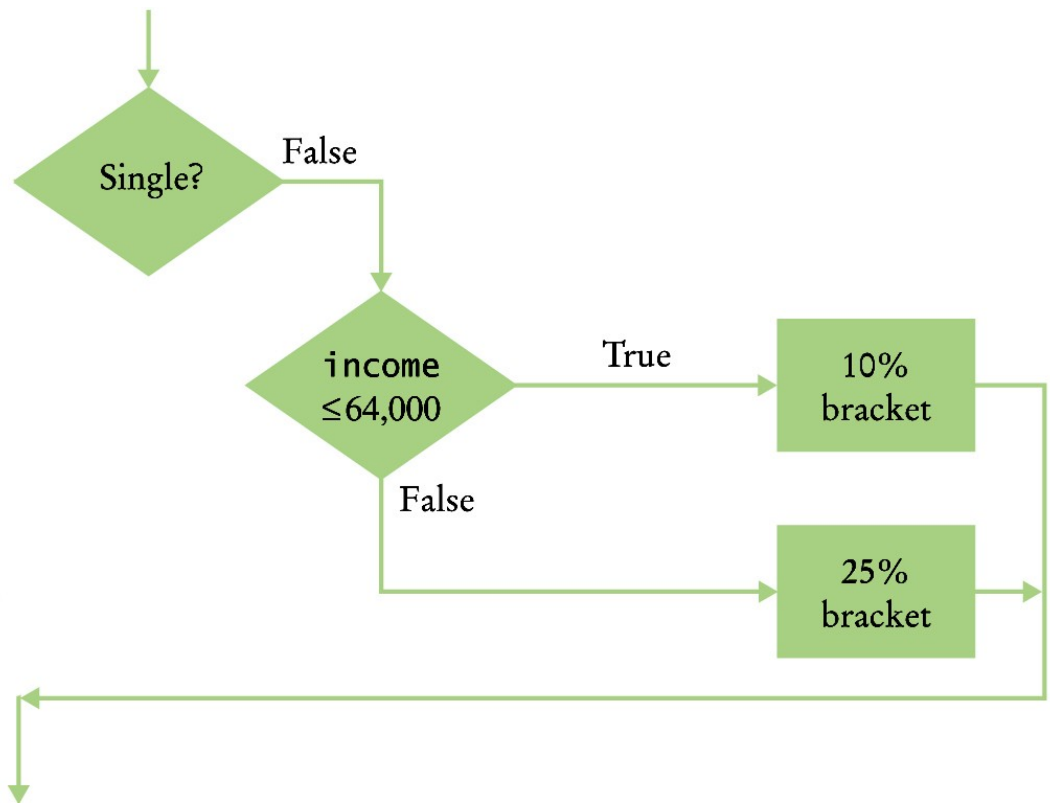
Nested Branches – Taxes

For those with spouses …



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

```
#include <stdio.h>

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 = 0.0;
    double tax2 = 0.0;

    double income;
    printf("Please enter your income: ");
    scanf("%lf", &income);

    int marital_status;
    printf("Please enter 1 for single, 2 for married: ");
    scanf("%d", &marital_status);
```

Nested Branches – Taxes

```
if (marital_status == 1) {
    if (income <= RATE1_SINGLE_LIMIT) {
        tax1 = RATE1 * income;
    } else {
        tax1 = RATE1 * RATE1_SINGLE_LIMIT;
        tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
    }
} else {
    if (income <= RATE1_MARRIED_LIMIT) {
        tax1 = RATE1 * income;
    } else {
        tax1 = RATE1 * RATE1_MARRIED_LIMIT;
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
    }
}
```

Nested Branches – Taxes

```
    double total_tax = tax1 + tax2;

    printf("The tax is $%f\n", total_tax);
    return 0;
}
```

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

Looking at your pseudocode or C code,

- ***Use a marker 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

```
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.

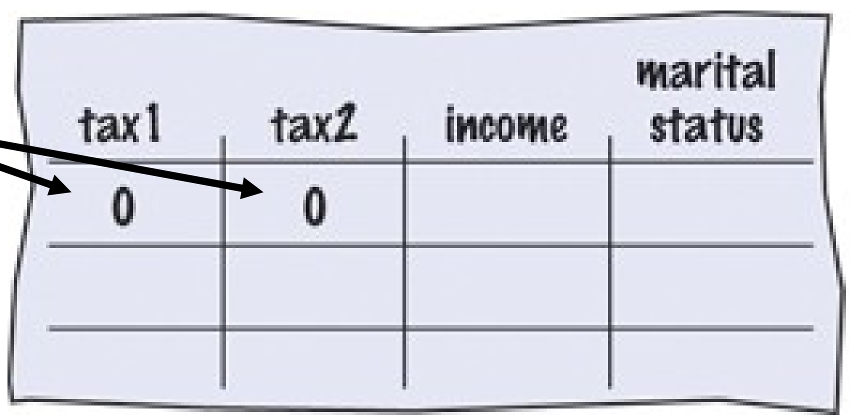
Han-Tracing

```
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;
```

tax1	tax2	income	marital status
0			

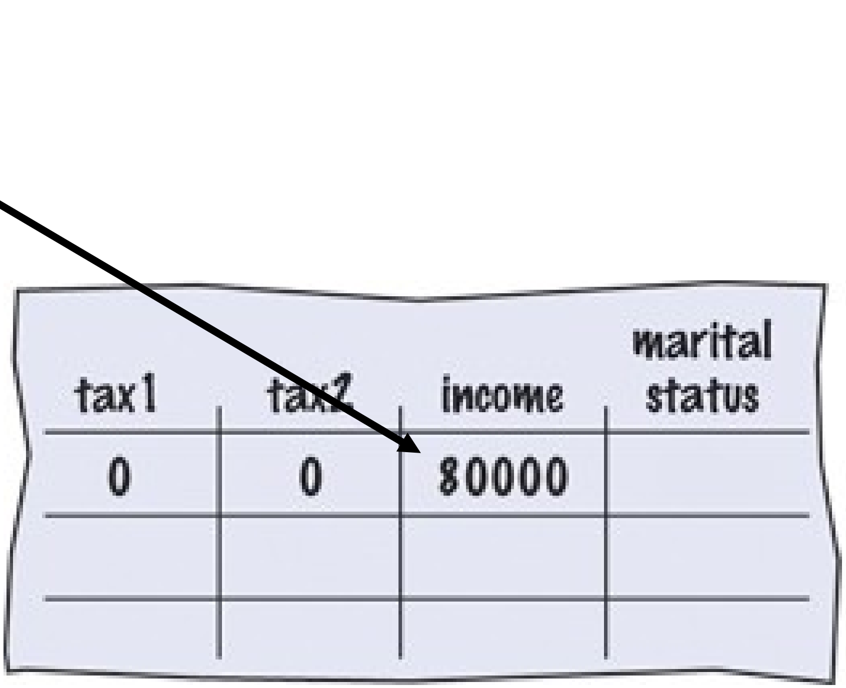
Hand-Tracing

```
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

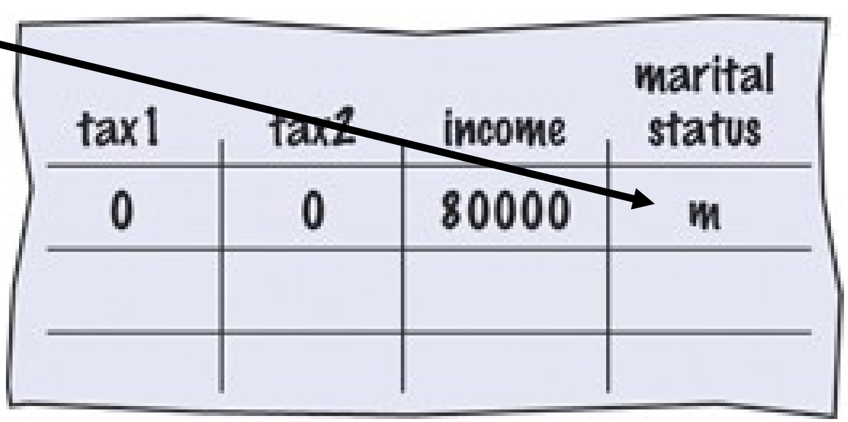
```
double income = 0.0;
printf("Please enter your income: ");
scanf("%lf", &income);
```



The user typed 80000.

Hand-Tracing

```
double income = 0.0;
printf("Please enter your income: ");
scanf("%lf", &income);

int marital_status = 0;
printf("Please enter 1 for single, 2 for married: ");
scanf("%d", &marital_status);
```



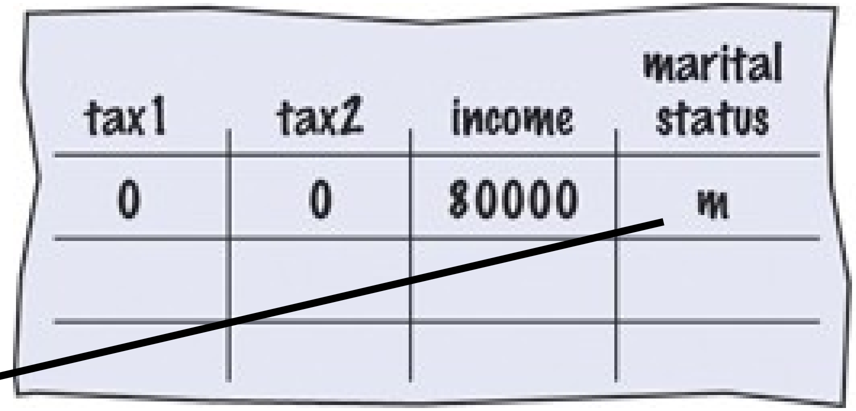
The user typed **2 (m)**

Hand-Tracing

tax1	tax2	income	marital status
0	0	80000	m

```
if (marital_status == 1) {
    if (income <= RATE1_SINGLE_LIMIT) {
        tax1 = RATE1 * income;
    } else {
        tax1 = RATE1 * RATE1_SINGLE_LIMIT;
        tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
    }
} else {
```

Hand-Tracing




```
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

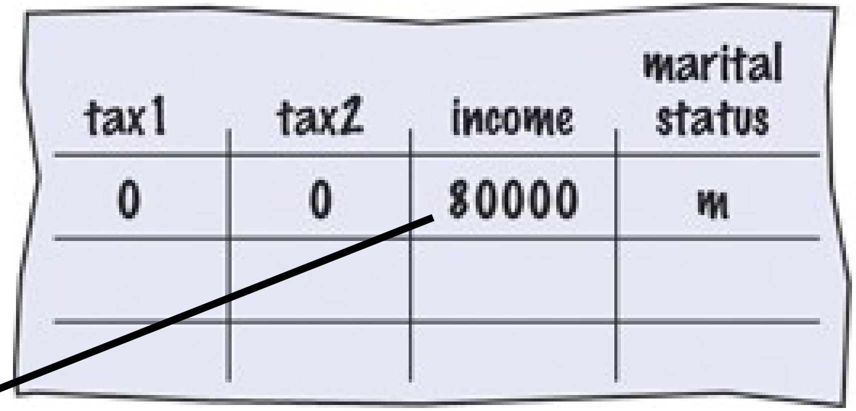
```
if (marital_status == 1) {
    if (income <= RATE1_SINGLE_LIMIT) {
        tax1 = RATE1 * income;
    } else {
        tax1 = RATE1 * RATE1_SINGLE_LIMIT;
        tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
    }
} else {
```

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




```
} 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

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

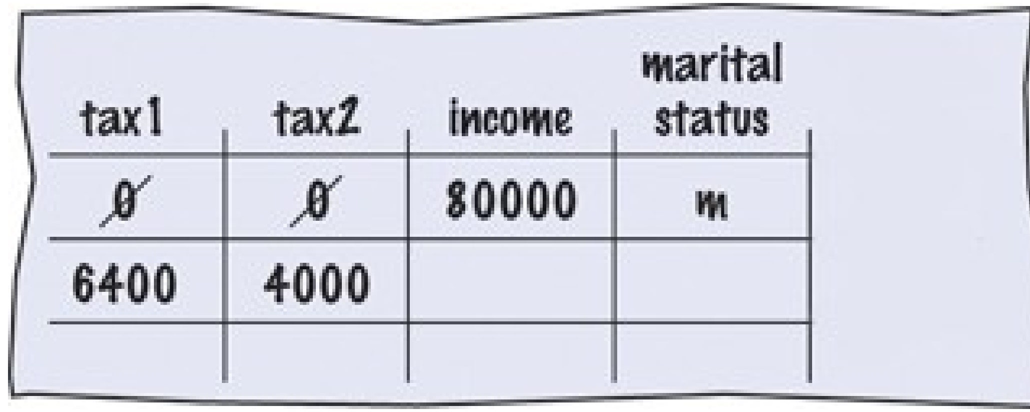
```
} else {
    if (income <= RATE1_MARRIED_LIMIT) {
        tax1 = RATE1 * income;
    } else {
        tax1 = RATE1 * RATE1_MARRIED_LIMIT;
        tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
    }
}

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




```
} 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;

    printf("The tax is $%f\n", total_tax);
    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

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

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.

- include “stdbool.h”:
- `bool` data type
- `true` and `false`

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*) yields **`true`** only when *both* conditions are **`true`**.

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

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

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

The Boolean Operator || (or)

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

– ***This is written as two adjacent vertical bar symbols.***

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

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

<table>
<tr><td><code>-10 && 10 > 0</code></td><td><code>true</code></td><td>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.</td></tr>
<tr><td><code>0 < x && x < 100 || x == -1</code></td><td><code>(0 < x && x < 100)
|| x == -1</code></td><td>The && operator has a higher precedence than the <code>||</code> operator.</td></tr>
<tr><td><code>!(0 < 200)</code></td><td><code>false</code></td><td><code>0 < 200</code> is <code>true</code>, therefore its negation is <code>false</code>.</td></tr>
<tr><td><code>frozen == true</code></td><td><code>frozen</code></td><td>There is no need to compare a Boolean variable with <code>true</code>.</td></tr>
<tr><td><code>frozen == false</code></td><td><code>!frozen</code></td><td>It is clearer to use <code>!</code> than to compare with <code>false</code>.</td></tr>
</table>

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 floating-point numbers?

Common Error – Combining Multiple Relational Operators

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

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

Common Error – Combining Multiple Relational Operators

`if (` $\begin{array}{c}\boxed{\mathbf{1}}\\ \boxed{\mathbf{0}}\end{array}$ **`<= 100)`**...

Unfortunately, **`false`** is 0 and **`true`** is 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 **&&**.

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 || …

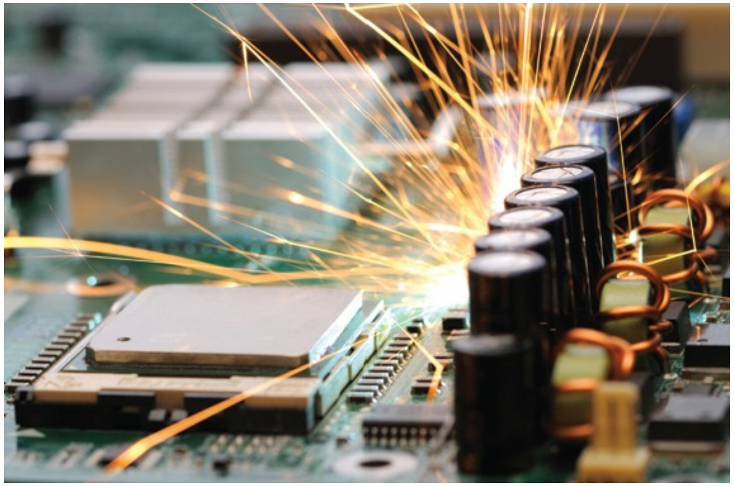
In an expression involving a series of **||**'s,
we can stop after finding the first **true**.

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.

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) {
   printf("Error: There is no thirteenth floor.\n");
   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.

```
return EXIT_FAILURE;
```

Input Validation with `if` Statements

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

```
if (floor <= 0 || floor > 20) {
   printf("Error: The floor must be between 1 and 20.\n");
   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

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int floor = 0;
   printf("Floor: ");
   scanf("%d", &floor);

   // The following statements check various input errors
   if (floor == 13) {
      printf("Error: There is no thirteenth floor.\n");
      return EXIT_FAILURE;
   }
   if (floor <= 0 || floor > 20) {
      printf("Error: The floor must be between 1 and 20.\n");
      return EXIT_FAILURE;
   }
```

Input Validation with `if` Statements – Elevator Program

```
    // Now we know that the input is valid
    int actual_floor = 0;
    if (floor > 13) {
       actual_floor = floor - 1;
    } else {
       actual_floor = floor;
    }

    printf("The elevator will travel to the actual floor %d.\n",
                actual_floor);

    return EXIT_SUCCESS;
 }
```

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.

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.

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.

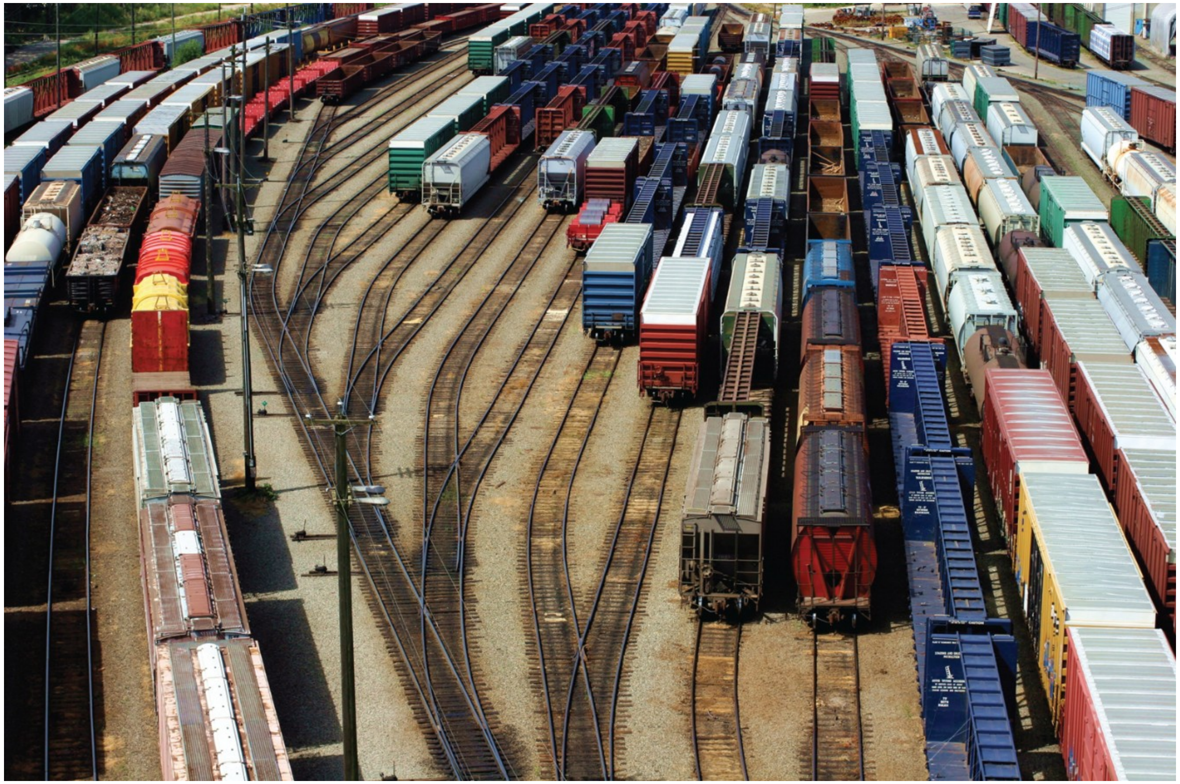
Use the `bool` data type to store and combine conditions that can be `true` or `false`.

- 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.

Chapter Summary

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

- When reading a value, check that it is within the required range.



End Chapter Three

Slides by Evan Gallagher