Lecture 11: Boolean Variables and While Loops

Announcements and reminders

Submissions:

- HW 4 -- due Saturday at 6 PM
- project 1 due next sat.

Course reading to stay on track:

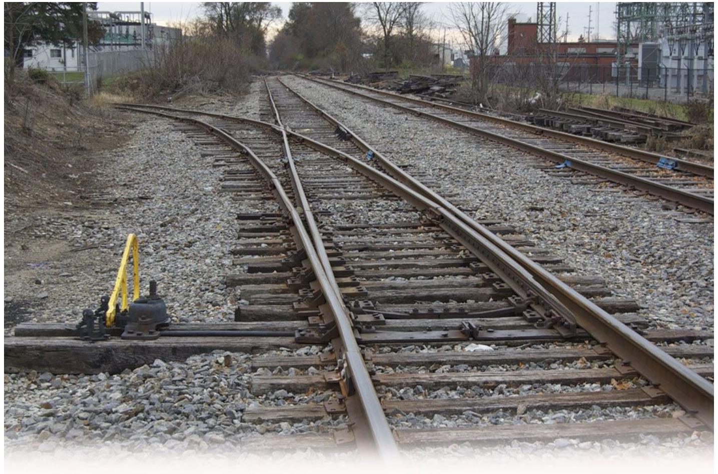
- 4.1-4.2 today
- 4.10 Monday (random numbers and simulation!)

Practicum 1

- Wednesday 20 Feb

Last time on *Intro Computing…*

- We learned about **if statements**!
- … and **else** statements!
- We learned about formatting **conventions for braces { } and indentation**!
- We learned about the **do-nothing statement**!
- We dipped our toes in the waters of **Boolean expressions**!
    - Either **true** or **false**



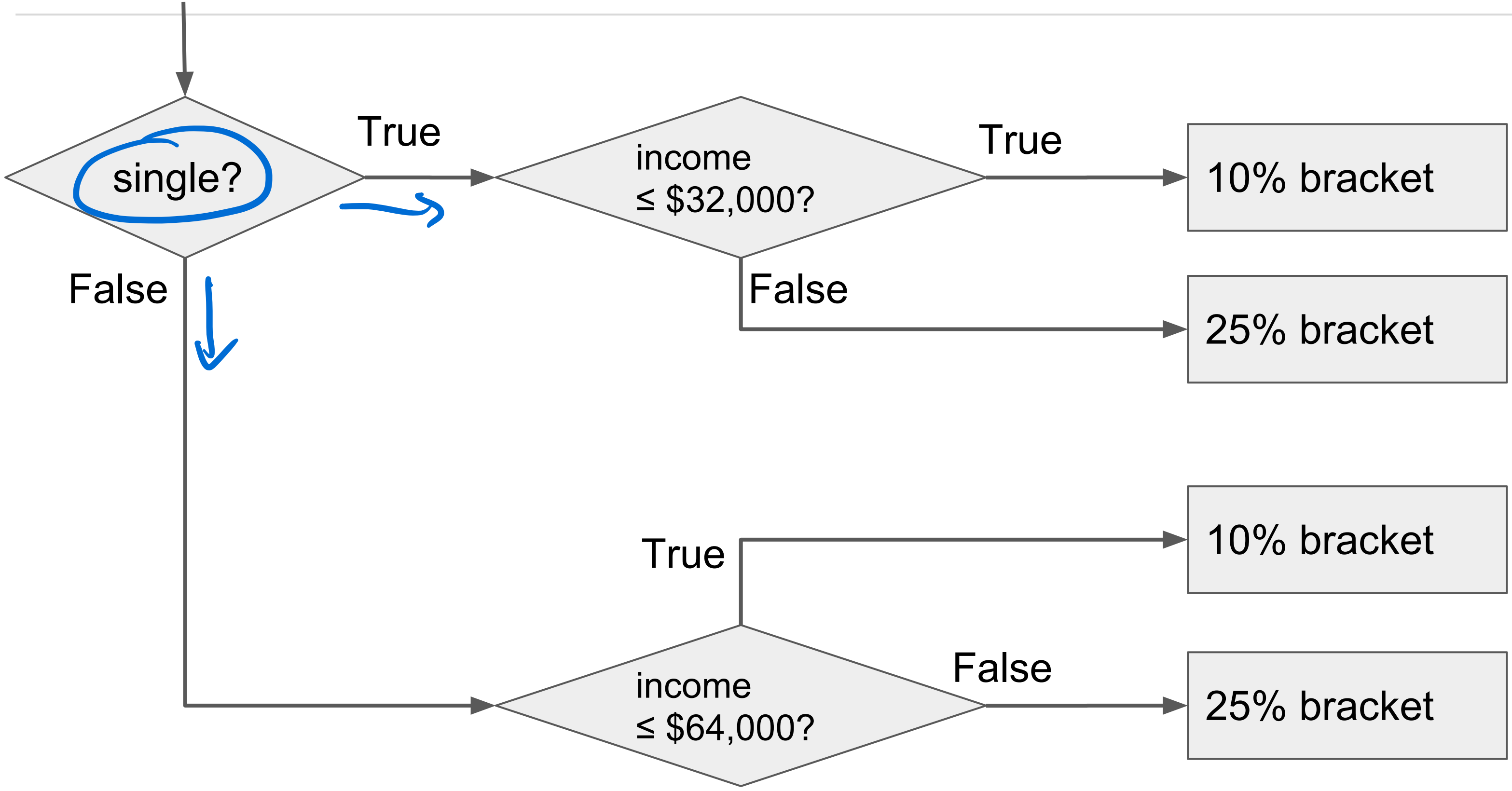
Last time on *Intro Computing…*

In the U.S., tax rates depend (among other things) on the taxpayer's marital status

- Single folks have higher tax rates
- Married taxpayers add their income together and pay taxes on the total
- From the IRS in a recent year:

Single and taxable income…	**… the tax rate is…**	**… for the amount over …**
≤ \$32,000	10%	\$0
> \$32,000	\$3,200 + 25%	\$32,000
Married and taxable income…	**… the tax rate is…**	**… for the amount over …**
≤ \$64,000	10%	\$0
> \$64,000	\$6,400 + 25%	\$64,000

Last time on *Intro Computing…*



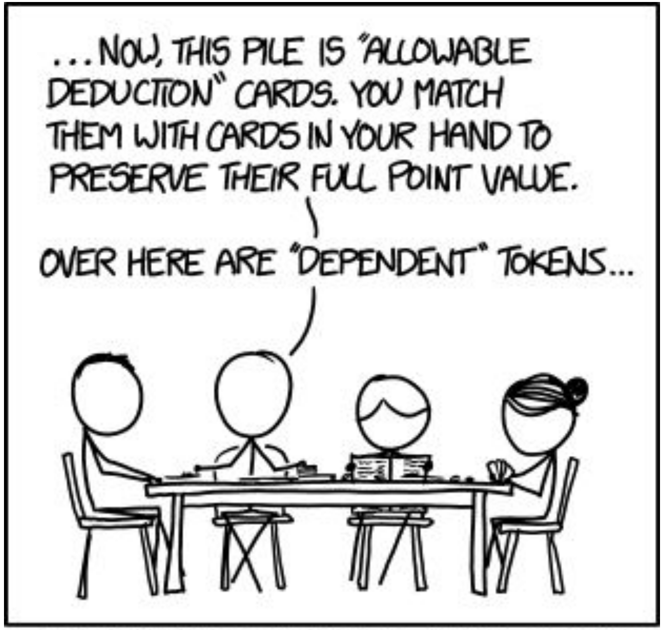
Inverting Conditions

if (income > 32000) is equivalent to if (!(income <= 32000))

not (income ≤ 32,000)

if (a > b && a > b) ....

if (!(a ≤ b) && !(a ≤ c))

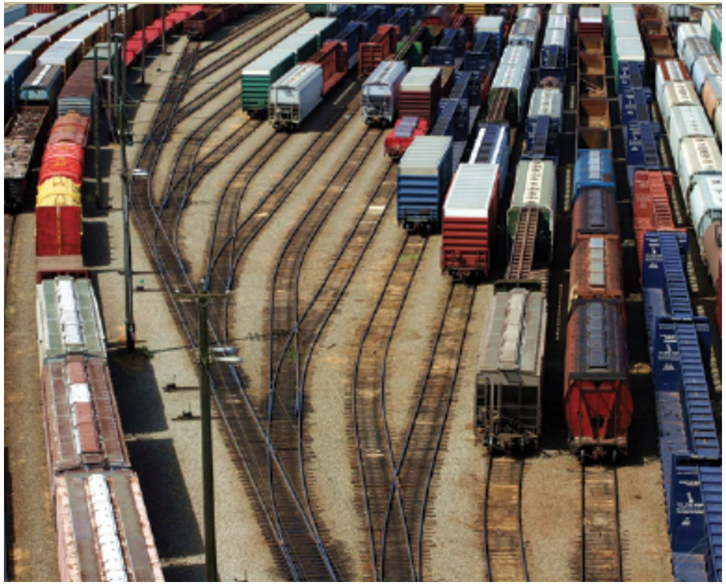


EVERY YEAR, I TRICK A LOCAL BOARD GAME CLUB INTO DOING MY TAXES.

Chapter 3: Decisions

Chapter Topics

- The `if` statement
- Comparing numbers and strings
- Multiple alternatives
- Nested branches
- Problem-solving: flowcharts
- Problem-solving: test cases
- **Boolean variables and operators**
- Application: input validation



Boolean Variables and Operators

- Sometimes you need to evaluate a logical condition in one part of a program and then use it elsewhere

≈ 0

- To store a variable that is either **true** or **false**, you use a **Boolean variable**

the word true & not a Boolean value

≈ 1

  - Not strings:
    - `bool failed = true` is ***not*** the same as `string failed = "true"`
  - Not integers:
    - `bool failed = true` is ***not quite*** the same as `int failed = 1`
    - But it's close:
      - **false** is 0, and any non-zero value is treated as true

Boolean Variables and Operators

Example:

```
bool failed = false;
// you could change the value of failed,
// depending on what's going on here
if (failed)
{
   cout << “We have failed :(” << endl;
}
else
{
   cout << “We did not fail!” << endl;
}
```

Boolean Variables and Operators

Example: S'pose you need to write a program to process temperature values, and tests whether a given temperature corresponds to liquid water or to solid ice.

At sea level, water freezes at 0 degrees Celsius and boils at 100 degrees Celsius.

→ Water is liquid ***IF*** the temperature is greater than 0 ***AND*** less than 100

Boolean Operators: And   &&

Example: S’pose you need to write a program to process temperature values, and tests whether a given temperature corresponds to liquid water or to solid ice.

At sea level, water freezes at 0 degrees Celsius and boils at 100 degrees Celsius.

→ Water is liquid ***IF*** the temperature is greater than 0 ***AND*** less than 100

In C++, the && operator (called “and”) yields **true** only when *both* conditions that it joins are true:

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

Boolean Operators: And &&

```
if ((temp > 0) && (temp < 100))
{
   cout << "Liquid" << endl;
}
else
{
   cout < "Not liquid" << endl;
}
```

condition is true only when all of the things chained together w/ && are true

- If temp is within the 0 to 100 range, then both the left-hand side and right-hand side are **true**, so the whole expression in parens ( ) has value = **true**
- In **all** other cases, the whole expression's value is **false**

Boolean Operators: Or ||

- The || operator (called ***or***) yields the result **true** if at least one of the conditions connected by it is **true**
- Written as two adjacent vertical bar symbols (above the Enter key)

```
if (temp <= 0 || temp >= 100)
{
   cout < "Not liquid" << endl;
}
```

- If ***either*** of the left-hand or right-hand side expressions is **true**, then the whole expression has value **true**
- **Question:** What is the only case in which "Not liquid" would appear?

Boolean Operators: Or ||

- The || operator (called ***or***) yields the result **true** if at least one of the conditions connected by it is **true**

- Written as two adjacent vertical bar symbols (above the Enter key)

```
if (temp <= 0 || temp >= 100)
{
   cout < "Not liquid" << endl;
}
```

- If ***either*** of the left-hand or right-hand side expressions is **true**, then the whole expression has value **true**

- **Question:** What is the only case in which "Not liquid" would *not* appear?
  Answer: If ***both*** of the expressions are **false**

  → temp > 0 and temp < 100

Boolean Operators: Not !

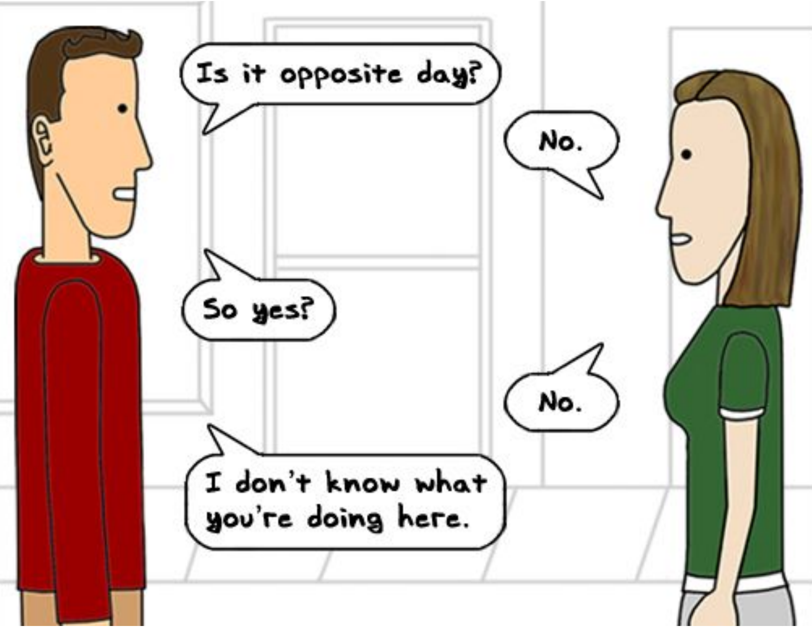
- Sometimes, you need to invert a condition with the logical ***not*** operator: !

- The ! operator takes a single condition and evaluates to **true** if the condition is **false**, and to **false** if the condition is **true**

```
if (!frozen)
{
   cout < "Not frozen" << endl;
}
```

- "Not frozen" will be written only when `frozen` contains the value **false**

- **Question:** What is the value of `!false` ?



Boolean Operators: Truth Tables

Definition: A **truth table** displays the value of a Boolean operator expression for all possible combinations of its constituent expressions.

(You'll look at truth tables a **lot** more in CSCI 2824 (Discrete) && it will be nice)

So if A and B denote bool variables or Boolean expressions, we have:

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	

Boolean Operators: Truth Tables

Definition: A **truth table** displays the value of a Boolean operator expression for all possible combinations of its constituent expressions.

(You'll look at truth tables a **lot** more in CSCI 2824 (Discrete))

So if A and B denote bool variables or Boolean expressions, we have:

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

Boolean Operators: Examples

<table>
<tr><th>Expression</th><th>Value</th><th>Comment</th></tr>
<tr><td><code>0 < 200 && 200 < 100</code></td><td><code>false</code></td><td></td></tr>
<tr><td><code>0 < 200 || 200 < 100</code></td><td><code>true</code></td><td></td></tr>
<tr><td><code>0 < 200 || 100 < 200</code></td><td><code>true</code></td><td>int ≠ 0 → True
true</td></tr>
<tr><td><code>0 < 200 < 100</code></td><td><code>true</code></td><td></td></tr>
<tr><td><code>-10 && 10 > 0</code></td><td><code>true</code></td><td>(-10) && (10 > 0)</td></tr>
<tr><td><code>0 < x && (x < 100 || x == -1)</code></td><td><code>(0 < x && x < 100) || x == -1</code></td><td>-10 > 0 && 10 > 0</td></tr>
<tr><td><code>!(0 < 200)</code></td><td><code>false</code></td><td></td></tr>
<tr><td><code>frozen == true</code></td><td><code>frozen</code></td><td></td></tr>
<tr><td><code>frozen == false</code></td><td><code>!frozen</code></td><td></td></tr>
</table>

Boolean Operators: Examples

<table>
<tr><th>Expression</th><th>Value</th><th>Comment</th></tr>
<tr><td><code>0 < 200 && 200 < 100</code></td><td><code>false</code></td><td>200 < 100 is false, so whole && thing is false</td></tr>
<tr><td><code>0 < 200 || 200 < 100</code></td><td><code>true</code></td><td>0 < 200 is true, so whole || thing is true</td></tr>
<tr><td><code>0 < 200 || 100 < 200</code></td><td><code>true</code></td><td>Both are true, so whole || is true (“inclusive or”)</td></tr>
<tr><td><code>0 < 200 < 100</code></td><td><code>true</code></td><td>0 < 200 done first, and is true, so converted to 1</td></tr>
<tr><td><code>-10 && 10 > 0</code></td><td><code>true</code></td><td>-10 is not 0, so it’s taken to be true (don’t do this!)</td></tr>
<tr><td><code>0 < x && x < 100</code>
<code>|| x == -1</code></td><td><code>(0 < x && x <</code>
<code>100)</code>
<code>|| x == -1</code></td><td>&& operator has higher precedence than ||</td></tr>
<tr><td><code>!(0 < 200)</code></td><td><code>false</code></td><td>0 < 200 is true, so !(0 < 200) is !(true), which is false</td></tr>
<tr><td><code>frozen == true</code></td><td><code>frozen</code></td><td>No need to compare Boolean variable with true</td></tr>
<tr><td><code>frozen == false</code></td><td><code>!frozen</code></td><td>Clearer to use ! than to compare a bool with false</td></tr>
</table>

Common Error: Combining Multiple Relational Operators

Consider the expression: `if (0 <= temp <= 100) { ... }`

Looks the same as the mathematical test: $0 \leq temp \leq 100$

... But it isn't the same: Say, `temp = 30`. Then... `0 <= temp <= 100`

$\underbrace{0 <= 30}_{\text{true}} <= 100$

$1 \leq 100$?

- It will compile fine, but will not run the way you expect
- Do **not** use that syntax in C++
- Instead, use the Boolean && operator to combine two pairwise comparisons:

```
if (0 <= temp && temp <= 100) { ... }
```

Common Error: Combining Multiple Relational Operators

Another similar error: `if (x && y > 0) { ... }`

Instead of: `if (x > 0 && y > 0) { ... }`

Top:

Bottom:

Common Error: Combining Multiple Relational Operators

Another similar error: `if (x && y > 0) { ... }`

Instead of: `if (x > 0 && y > 0) { ... }`

Top: will treat x as **true** if it is anything besides 0

→ whole thing is **true** if $x \neq 0$ and $y > 0$

Bottom: whole thing is **true** if $x > 0$ and $y > 0$

Short Circuit Evaluation

The question: When do we know if an expression is **true** or **false**?

(expression1 && expression2 && expression3 && ~~...~~)

False

→ with &&'s, we can stop after we find the first **false**

(expression1 || expression2 || expression3 || ...)

→ with ||'s, we can stop after we find the first **true**

Definition: Ceasing evaluation as soon as the truth value of the *entire* expression can be determined is called **short circuit evaluation**.

De Morgan's Laws

S'pose 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;
}
```

That looks pretty complicated, right?

Thankfully, De Morgan's Laws come to our rescue!

De Morgan's Laws

$-2(x+y)$

De Morgan's Laws allows us to rewrite complicated not/and/or expressions so they are easier to read:

!(A && B) is the same as !A || !B   and   !(A || B) is the same as !A && !B

(change || to &&, or change && to ||, and negate all the terms)

$-(-a) = a$

Fun fact: Also note that **double negation** holds:

!(!A) is the same as A

F

!(x>0 && y>0) "=" !(x>0) || !(y>0)

!(x>0 || y>0) "=" !(x>0) && !(y>0)

De Morgan's Laws

Armed with De Morgan's laws and Double Negation, let's simplify this beast:

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

… Turns into…

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

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

De Morgan’s Laws

Armed with De Morgan’s laws and Double Negation, let’s simplify this beast:

```
shipping_charge = 10.00;
if (!(country == “USA” && state != “AK” && state != “HI”))
{
    shipping_charge = 20.00;
}
```

… Turns into…

```
if (country != “USA” || state == “AK” || state == “HI”))
{
    shipping_charge = 20.00;
}
```

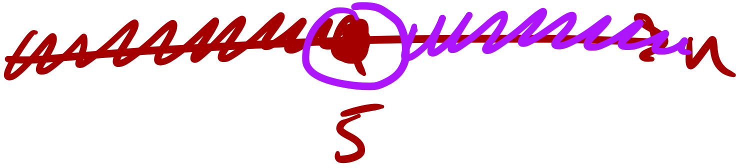
… Ah! Much nicer!

Expression Simplification

int n; bool b;

Examples: Simplify the following logical conditions.

everything that isn't ≤5



n < 5 || n == 5 ____________________

n <= 5 && n != 5 ____________________

n <= 5 && n >= 5 ____________________

n <= 5 || n >= 5 ____________________

!(n <= 5) → n > 5

!!b ____________________

b == true ____________________

b == false ____________________

Chapter 4: Loops

1. **The `while` loop**
2. Problem solving: hand-tracing
3. The `for` loop
4. The `do` loop
5. Processing input
6. Problem-solving: storyboards
7. Common loop algorithms
8. Nested loops
9. Problem solving: solve a simpler problem first
10. Random numbers and simulations

Why loops?

Definition: A **loop** is a statement that is used to execute one or more statements repeatedly until some goal is reached. Sometimes these statements will not be executed at all, if that is how the goal can be reached.

C++ has three looping statements:

```
while ()

for ()

do {} while ()
```

The while loop

HERE

Example: Our pseudocode from Chapter 1 for making a fruit salad included a while loop:

```
while (largest piece of fruit is bigger than bite-size)
{
    cut the largest piece of fruit in half
}
```

while loop template:

```
while (condition)
{
    Statements
}
```

The *condition* is some kind of a test. It is a **Boolean expression**.

The *statements* will be repeatedly executed until the *condition* is **false**.

Each time before the *statements* are executed, the truth of *condition* is checked.

The while loop

Example: Starting with $10,000, how many years until we have at least $20,000, at 5% interest? (compounded annually)

The algorithm: (pseudocode)

The while loop

Example: Starting with $10,000, how many years until we have at least $20,000, at 5% interest? (compounded annually)

The algorithm: (pseudocode)

1. Start with a year value of 0 and a balance of $10,000
2. ***Repeat*** the following steps ***while*** the balance is less than $20,000:
   a. Add 1 to the year value
   b. Compute the interest by multiplying the current balance value by 0.05 (5%, use a `const`, of course!)
   c. Add the interest to the balance
3. Report the final year value as the answer

Let's code this up…

The while loop

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
   const double RATE = 0.05;
   const double INITIAL_BALANCE = 10000.0;
   const double TARGET_BALANCE = 2 * INITIAL_BALANCE;

   double balance = INITIAL_BALANCE;
   int year = 0;

   while (balance < TARGET_BALANCE)
   {
      year++;
      balance = (1.0 + RATE) * balance;
      cout << "Year " << setw(2) << year << ": balance = " << balance << endl;
   }

   return 0;
}
```

The while loop

Example (continued): As the program runs, the values for balance are updated for 15 ***iterations*** of the **while** loop…

… until the balance is > $20,000 and the **while** loop test condition (`balance < TARGET_BALANCE`) becomes **false**

Before entering body of while loop

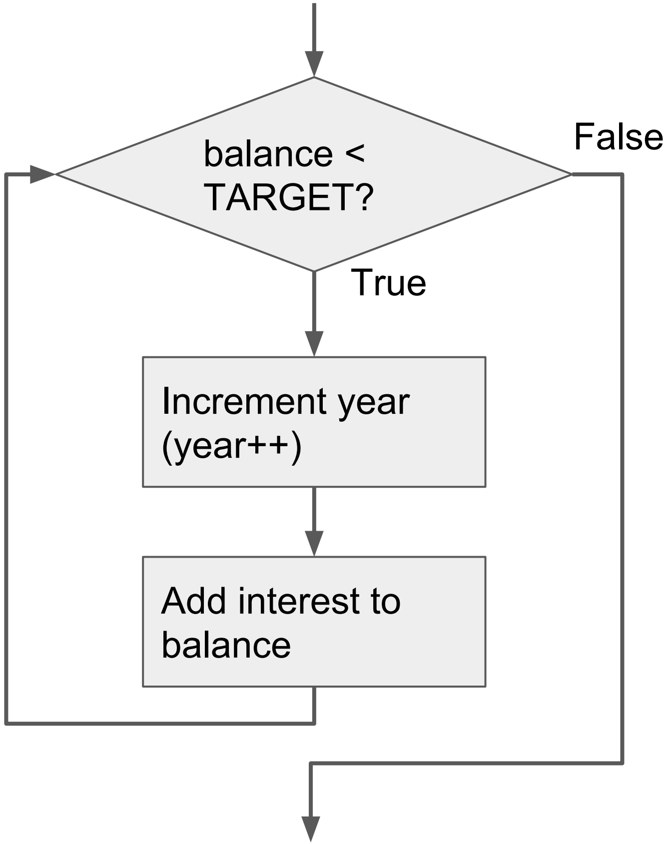
```
Year =  0, balance = 10000
Year =  1, balance = 10500
Year =  2, balance = 11025
Year =  3, balance = 11576.25
Year =  4, balance = 12155.06
Year =  5, balance = 12762.82
Year =  6, balance = 13400.96
Year =  7, balance = 14071
Year =  8, balance = 14774.55
Year =  9, balance = 15513.28
Year = 10, balance = 16288.95
Year = 11, balance = 17103.39
Year = 12, balance = 17958.56
Year = 13, balance = 18856.49
Year = 14, balance = 19799.32
```

At the end of while loop

```
Year =  1, balance = 10500
Year =  2, balance = 11025
Year =  3, balance = 11576.25
Year =  4, balance = 12155.06
Year =  5, balance = 12762.82
Year =  6, balance = 13400.96
Year =  7, balance = 14071
Year =  8, balance = 14774.55
Year =  9, balance = 15513.28
Year = 10, balance = 16288.95
Year = 11, balance = 17103.39
Year = 12, balance = 17958.56
Year = 13, balance = 18856.49
Year = 14, balance = 19799.32
Year = 15, balance = 20789.28
```

The while loop

Example (continued): Flowchart of the investment calculation while loop



The while loop

Example: What will happen here?

```
const double RATE = 0.05;
const double INITIAL_BALANCE = 10000.0;
const double TARGET_BALANCE = 2 * INITIAL_BALANCE;

double balance = INITIAL_BALANCE;
int year = 0;

while (balance < TARGET_BALANCE)
{
   year++;
   balance = (1.0 - RATE) * balance;
   cout << "Year " << setw(2) << year << ": balance = " << balance << endl;
}

return 0;
```

The while loop

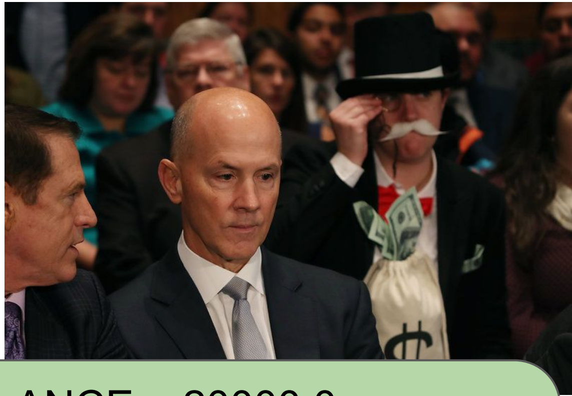
Example: What will happen here?

```
const double RATE = 0.05;
const double INITIAL_BALANCE = 10000.0;
const double TARGET_BALANCE = 2 * INITIAL_BALANCE;

double balance = INITIAL_BALANCE;
int year = 0;

while (balance < TARGET_BALANCE)
{
   year++;
   balance = (1.0 - RATE) * balance;
   cout << "Year " << setw(2) << year << ": bc
}

return 0;
```



TARGET_BALANCE = 20000.0

and each time through while loop,

balance = 0.95 * balance

- balance is ***decreasing***
- balance will ***always*** be less than TARGET
- So the while loop goes on ***forever!***

The while loop -- a summary

We often **define variables outside** the loop, then **update them inside** the loop

If the condition never becomes false, an **infinite loop** will occur

Beware of “off-by-one” errors in the loop condition

```
const double RATE = 0.05
double balance = 0;
int year = 0;
...
while (balance < TARGET_BALANCE)
{
   year++;
   double interest = balance * RATE;
   balance = balance + interest;
}
```

No semicolon after conditions

Statements within curly brackets { } are executed repeatedly ***while*** the condition is true

Lining up braces is good practice

Braces not required for a single statement, but always a good idea

Variables are **created** and/or **updated** each time through the loop

The while loop

Examples: Before each loop, s'pose we set `int i = 5;`

loop	output	explanation

`while (i > 0) {`
`   cout << i << " "; i--;`
`}`		

`while (i > 0) {`
`   cout << i << " "; i++;`
`}`		

`while (i > 5) {`
`   cout << i << " "; i--;`
`}`		

`while (i < 0) {`
`   cout << i << " "; i--;`
`}`		

`while (i > 0); {`
`   cout << i << " "; i--;`
`}`		

The while loop

Examples: Before each loop, s'pose we set `int i = 5;`

loop	output	explanation

`while (i > 0) {`
`   cout << i << " "; i--;`
`}`
	5 4 3 2 1	When `i` is 0, the loop condition is **false**, so the loop ends after `i` starts at 5, then is 4, then…

`while (i > 0) {`
`   cout << i << " "; i++;`
`}`
	5 6 7 8 9 10 11 …	The `i++` statement is likely an error causing an **infinite loop**

`while (i > 5) {`
`   cout << i << " "; i--;`
`}`
	(no output)	The statement `i > 5` starts out as **false**, so the loop is never executed

`while (i < 0) {`
`   cout << i << " "; i--;`
`}`
	(no output)	Programmer likely wanted to ***stop*** the loop once `i < 0`, but the **condition says when to execute, not when to stop**

`while (i > 0); {`
`   cout << i << " "; i--;`
`}`
	(no output)	Semicolon after the condition leads to a **do-nothing** loop. Runs forever, repeatedly checking if `i > 0` (`i` never changes)

The while loop

Example: Hand-trace this **while** loop. What is the output?


```
const double POUR = 0.02;
const double INITIAL_VOLUME = 0.85;
const double TARGET_VOLUME = 1;
double volume = INITIAL_VOLUME;

while (volume < TARGET_VOLUME)
{
   volume = volume + POUR;
}

cout << “We measured ” << volume << “ cups of tasty oil” << endl;
return 0;
```

The while loop

Example: Hand-trace this **while** loop. What is the output?

```
const double RATE = 0.05;
const double INITIAL_BALANCE = 10000.0;
const double TARGET_BALANCE = 2 * INITIAL_BALANCE;
double balance = INITIAL_BALANCE;
int year = 1;

while (year <= 20)
{

   balance = (1.0 + RATE) * balance;
}
cout << "After 20 years, we have a balance of " << balance << endl;
return 0;
```

The while loop

Example: Hand-trace this **while** loop. What is the output?

```
const double RATE = 0.05;
const double INITIAL_BALANCE = 10000.0;
const double TARGET_BALANCE = 2 * INITIAL_BALANCE;
double balance = INITIAL_BALANCE;
int year = 1;

while (year <= 20)
{

   balance = (1.0 + RATE) * balance;
}
cout << "After 20 years, we have a balance of " << balance << endl;
return 0;
```

The variable *year* is not updated within the while loop statements, so it is ***always*** <= 20.

→ Get an **infinite** while loop

What just happened?

- We learned about **Boolean variables and operators**!

    && (and) || (or) ! (not)

- We learned **De Morgan's Laws!**
    - … a way to simplify expressions like
      !(A && B) or !(A || B)

- We learned about **while** loops!
    - … how to repeat a set of instructions until some kind of a condition is met
    - … and we learned about avoiding common errors with while loops