ENGR 101 – Chapter 12

More C++ Basics and Branching

Demo: In-Class C++ Exercises (Lobster)

- Find a C++ visualization tool at lobster.eecs.umich.edu.
  - We'll use it for some in-class examples, and we encourage you to use it on your own to explore visually what your code is doing.
  - It works best in Google Chrome.
  - It doesn't support quite all of C++, but it should work for most 101 topics.

- Follow along with the starter code for "ENGR101_15_start"

Recall: Basic Types

- C++ supports many different types. Here are a few of the basics:

Type	Description	Example
`int`	A signed integer. (Can be negative)	`int x = 3;`
`double`	A floating point number. (i.e. has a fractional part)	`double y = 2.5;`
`bool`	A Boolean (i.e. logical) value. 1 – true, 0 – false.	`bool z = true;`
`char`	A single character.	`char c = 'w';`
`string`	A sequence of characters.	`string word = "hello";`

- A big difference from MATLAB… No more built-in matrices!

Recall: Type Errors

- There are two main kinds of type errors:
  - Invalid operations
  - Invalid conversions

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

int main() {
  int i = 5;
  double d = 3.5;
  string s = 7;
  i = s; // Invalid conversion (string to int)
  i + s; // Invalid operation (string + int)
  d = i; // Conversion allowed (int to double)
}
```

In most cases, we are not allowed to mix types. But there are some exceptions....

3 min

Exercise: Mixing Types

lobster.eecs.umich.edu
ENGR101_15_conversions

- What happens here?

```
int main() {

   int anInt = 7;
   double aDouble = 3.5;

   int a = aDouble;
   double b = anInt;

   int x = false;
   double y = true;

   bool b1 = 1;
   bool b2 = 0;
   bool b3 = 3.14;
   bool b4 = -1;
}
```

The compiler allows all of this code!

What rules does C++ use at runtime to convert one type to another?

Note: Lobster uses a pink box/outline to show an implicit conversion.

Implicit Conversion Between Numeric Types

- int double

```
double b = anInt;
```

  - No loss of information.
  - This is a **widening** conversion.
  - Generally safe.

We are going from a smaller set of values (possible `int`s) to a larger set (possible `double`s).

- double int

```
int a = aDouble;
```

  - Loss of information – the value is "truncated".
    - Only the integer part of the number is retained.
  - This is a **narrowing** conversion.
  - Dangerous!

Implicit Boolean Conversions

- bool anything
  - False turns into 0.
  - True turns into 1.

```
int x = false;
double y = true;
```

- anything bool
  - ONLY 0 turns into false.
  - Everything else is true.
    - Even negative numbers!

```
bool b1 = 1;
bool b2 = 0;
bool b3 = 3.14;
bool b4 = -1;
```

Exercise: Debugging

lobster.eecs.umich.edu
ENGR101_15_debug

2 min

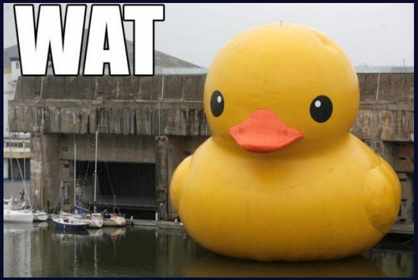
- The compiler won't warn you when it inserts an implicit conversion. Sometimes they can have surprising results!
- Why doesn't this code work?

```
int main() {
  double x = 2.8;
  double y = 2.5;

  // find the maximum and store in z
  int z = x; // start with x
  if(y > z) {
    // if y was larger, replace
    cout << "y was larger" << endl;
    z = y;
  }

  cout << "max value is " << z << endl;
}
```

It claims y was larger and the max value is 2.



Binary Arithmetic Operations

- Most of these work similarly to MATLAB, except of course not with vectors and matrices anymore.

	Operator	Example	Result
Addition	`+`	`2 + 3`	`5`
Subtraction	`-`	`5 - 3`	`2`
Multiplication	`*`	`5 * 3`	`15`
Exponentiation[1]	None		
Division	`/`	`11 / 4`	`2.75`
Modulo (remainder)	`%`	`11 % 4`	`3`

1 Although there is no exponentiation operator, the C++ standard library contains functions you can use for exponentiation.

Floating-Point Division vs. Integer Division

- We often see two kinds of division in programming…
  - **Floating-point division**:
    11 divided by 4 yields 2.75
    - Use the / operator to get the quotient (there is no remainder)
  - **Integer division**:
    11 divided by 4 yields a quotient of 2, with remainder 3
    - Use the / operator to get the quotient
    - Use the % operator to get the remainder
- In C++, the kind of division depends on the type of the operands…

1.5 min

Exercise: Division

lobster.eecs.umich.edu
ENGR101_15_division

- What happens here?

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

int main() {

   int i1 = 3;
   int i2 = 4;

   double d1 = 3.0;
   double d2 = 4.0;

   cout << i1 / i2 << endl;
   cout << d1 / d2 << endl;
   cout << i1 / d2 << endl;
   cout << d1 / i2 << endl;
}
```

When do you get integer division?

When do you get floating point division?

Recall: Temperature Converter

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

int main() {
  cout << "Enter a temperature in Celsius: ";

  double c;
  cin >> c;

  ~~double f = 9 / 5 * c + 32;~~

  double f = 9.0 / 5. * c + 32;

  cout << f << " degrees Fahrenheit.";
}
```

Add either `.` or `.0` to the literals to ensure we get floating point division instead of integer division.

3 min

Exercise: Integer Division and Remainder

- Fill in the tables with the result of each operation.

Expression	Result
`0 / 3`	
`1 / 3`	
`2 / 3`	
`3 / 3`	
`4 / 3`	
`5 / 3`	
`6 / 3`	
`7 / 3`	

Expression	Result
`0 % 3`	
`1 % 3`	
`2 % 3`	
`3 % 3`	
`4 % 3`	
`5 % 3`	
`6 % 3`	
`7 % 3`	

Solution: Integer Division

- Fill in the tables with the result of each operation.

Expression	Result
`0 / 3`	`0`
`1 / 3`	`0`
`2 / 3`	`0`
`3 / 3`	`1`
`4 / 3`	`1`
`5 / 3`	`1`
`6 / 3`	`2`
`7 / 3`	`2`

Expression	Result
`0 % 3`	`0`
`1 % 3`	`1`
`2 % 3`	`2`
`3 % 3`	`0`
`4 % 3`	`1`
`5 % 3`	`2`
`6 % 3`	`0`
`7 % 3`	`1`

Using Integer Division and Modulo

- Why work with the quotient and remainder separately?
- Example: You're writing code for a stopwatch app, but the hardware only reports time in seconds. You want to display this in minutes/seconds instead.
- Goal: Convert x total seconds to m minutes and s seconds.

```
int main(){
  int x = 153; // total seconds
  int m = x / 60; // minutes
  int s = x % 60; // leftover seconds
}
```

We're doing math "mod 60" because there are 60 seconds per minute.

4 min

Exercise: Stopwatch

- Continue the stopwatch example, but now extend it to hours, minutes, and seconds.
- This is tricky, be creative!

lobster.eecs.umich.edu
ENGR101_15_stopwatch

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

int main() {

   int x = 3753; // total seconds


   // TODO: convert to hours, minutes, and seconds!
   // For example, 3753 seconds is: 1 hour, 2 minutes, 33 seconds.
}
```

Solution: Stopwatch

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

int main() {

  int x = 3753; // total seconds
  int h = x / 3600; // 3600 seconds per hour

  // update x to remainder not used for hours
  x = x % 3600;

  // now find minutes and seconds from the rest
  int m = x / 60;
  int s = x % 60;
}
```

Relational Operations

- These operations check for **equality** or perform **comparisons**.
- Those with different symbols than in MATLAB are highlighted.

	Operator	Example	Result
Equality	`==`	`2 == 3`	`false`
Inequality	`!=`	`2 != 3`	`true`
Less Than	`<`	`5 < 5`	`false`
Less Than or Equal	`<=`	`5 <= 5`	`true`
Greater Than	`>`	`'c' > 'd'`	`false`
Greater Than or Equal	`>=`	`4.5 >= 4.5`	`true`

The resulting type of all relational operations is `bool`.

Operators can be applied to different types.

These are also sometimes called **conditional** operators.

Logical Operations

- Essentially, combining two **truth values** in a particular way.
- Those with different symbols than in MATLAB are highlighted.

	Operator	Example	Result
Logical And	&&	`2 < 3 && 5 > 6`	`false`
Logical Or	||	`2 < 3 || 5 > 6`	`true`
Exclusive Or			
Not	!	`!('a' == 'b')`	`true`

All these operations also yield a `bool`.

We won't cover XOR in C++.

- C++ also includes "bitwise operators", which are &, |, ~, and ^.
   - These manipulate the binary representation of data. We won't use them for 101!

ercise: Logical Operators

5 min

- Are these expressions true or false?

lobster.eecs.umich.edu
ENGR101_15_logical

```
int main(){
  int a = 3;
  int b = 4;
  double c = 3.5;
  double d = 4.3;
  string e = "lizard";
  string f = "frog";
  bool g = true;

  cout << (a < b) << endl;

  cout << (c + 0.5 < d) << endl;

  cout << (a > 8 && 2 * a + 8 * b + 7 < 42) << endl;

  cout << (e < f || f < e) << endl;

  cout << (!g || 7 / 2 == 3) << endl;
}
```

After working through each by hand, check against the Lobster visualization.

Do you notice anything peculiar about the way some && and || expressions are evaluated in Lobster? (Hint: some code is "skipped".)

Short-Circuit Operators

- The && and || operators have **short-circuit** behavior.
- In any case that the result can be determined just from the left side, the right side is not run at all.

```
int main(){
  int a = 3;
  int b = 4;
  bool g = true;


  cout << (a > 8 && 2 * a + 8 * b + 7 < 42) << endl;

  cout << (!g || 7 / 2 == 3) << endl;
}
```

Because a > 8 works out to be `false`, the whole && will inevitably be `false` and the rest is not even evaluated.

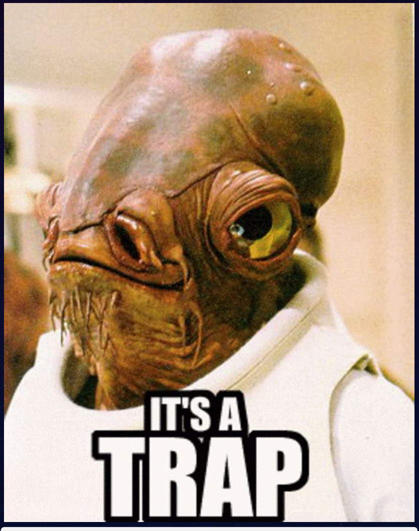
!g evaluates to be `false`, but since it's an || operation, we still have to check the rest.

Floating Point Precision

lobster.eecs.umich.edu
ENGR101_15_precision

- Computers can't perform floating-point math perfectly.
  - Limited memory means limited precision

```
int main() {
   double x = 0.1;
   double y = 0.2;
   if(x + y == 0.3) {
     cout << "equal" << endl;
   }
   else {
     cout << "not equal" << endl;
   }
}
```



What does this print?

Comparing Floating Point Numbers (i.e. doubles)

- It's not safe to use == or != with floating point numbers.
  - The results of computations that *should* be equal may not turn out to be *literally* equal, due to limited precision.
- Instead, check whether the numbers are very close…

```
bool double_eq(double x,
               double y) {
  double diff = x - y;
  if(diff < 0) {
    diff = -diff;
  }

  return diff < 0.0001;
}
```

This is often called an "epsilon value".

```
int main() {
  double x = 0.1;
  double y = 0.2;
  if( double_eq(x + y,  0.3) ) {
    cout << "equal" << endl;
  }
  else {
    cout << "not equal" << endl;
  }
}
```

This is how we check your solutions for project 4 :)

Break Time

We'll start again in 5 minutes.

Recall: Control Flow

- **Branching** and **iteration** are techniques for managing **control flow** in our programs.
  - The line of code that is currently executing is said to have "control".

- In particular, flowcharts are an effective tool for mapping out the control flow of our program design.

- **Control flow structures** like `if`, `for`, and `while` allow us to structure our code to follow the desired control flow.

if Statements

- if statements allow **branching**
  - Also called "**selection**" statements

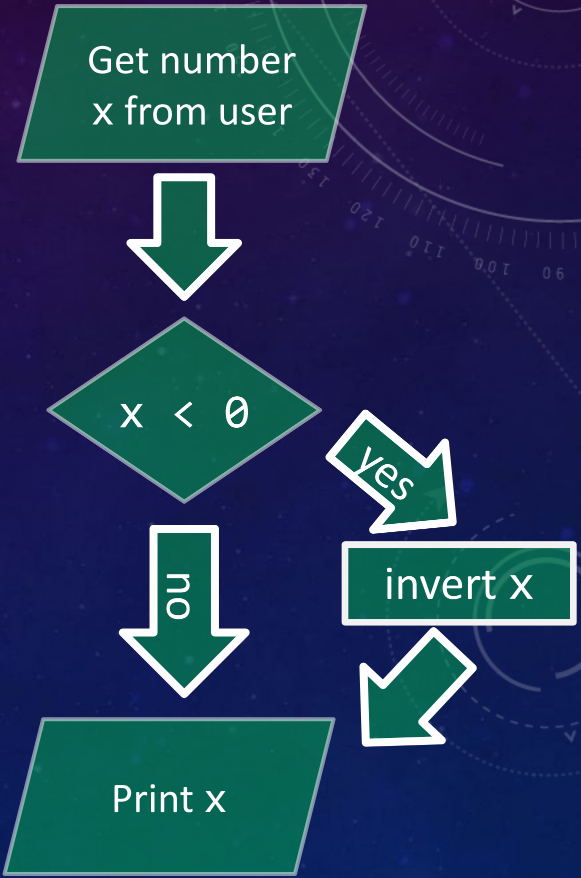
```
int main() {
  double x; // Declare x first

  cin >> x; // user inputs number

  if (x < 0) {
    x = -x; // invert if negative
  }

  cout << "abs value is" << x;

}
```



if Statement Syntax

condition
Any expression that can be converted to a bool. Written inside ( )

```
if (condition) {

   statement;
   statement;
   ...

}
```

braces
Always use these around the body.

body
A sequence of statements that will be executed if and only if the condition is true.

Why use the braces?

- It avoids confusion.

```
int main() {
  double x; // Declare x first

  cin >> x; // user inputs number

  if (x < 0)
    x = -x; // invert if negative
    cout << "inverting value" << endl;

  cout << "abs value is" << x;

}
```

No braces.

We decide to add another statement to the `if` body.

Oops. It always prints "inverting value". Only the first statement is actually inside the `if`.

Indentation can be misleading.

Why use the braces?

- Braces define a variable's **scope**.

```
int main() {
  double x; // Declare x first

  cin >> x; // user inputs number

  if (x < 0) {
    x = -x; // invert if negative
    string message = "inverting value";
    cout << message << endl;
  }
  cout << "abs value is" << x;

}
```

Here's an opening brace; it starts a local scope.

Here's another opening brace; it starts *another* local scope.

…but what is this "scope" ??

Scope

- A variable can only be used…
  - …after its declaration
  - …within its **scope**.

- If you try to use a variable before its declaration or outside its scope, you'll get a compiler error!

Local Scope / Block Scope

- Many variables have **local** scope, also known as **block scope**.
- A **block** is a chunk of code enclosed by curly braces { }.
  - Technically, "chunk of code" means a sequence of statements.

```
int main() {
  int x = 5;
  if( x % 2 == 0 ) { // if x is even
    int y = x / 2;
  }
  cout << x << endl;
  cout << y << endl;
}
```

These curly braces define a block. The variable y lives inside this block.

Error! y used out of scope.

Local Scope / Block Scope

- Block scope applies to any block of code, including the bodies of control flow structures like `if`, `for`, and `while`.

```
int main() {
  int a = 0;
  while(a < 10) {
    int b = a + 1;

    if( b % 2 == 0 ) { // if b is even
      int x = 2 * b;
      cout << x << endl;
    }
    a += x;
  }
  cout << b << endl;
}
```

General rule: A variable is allowed to "enter" a nested block, but it can't leave its own block.

Error! x used out of scope.

Error! b used out of scope.

Why use the braces?

- Braces define a variable’s **scope**.

```
int main() {
  double x; // Declare x first

  cin >> x; // user inputs number

  if (x < 0) {
    x = -x; // invert if negative
    string message = "inverting value";
    cout << message << endl;
  }
  cout << "abs value is" << x;

}
```

Here’s an opening brace; it starts a local scope.

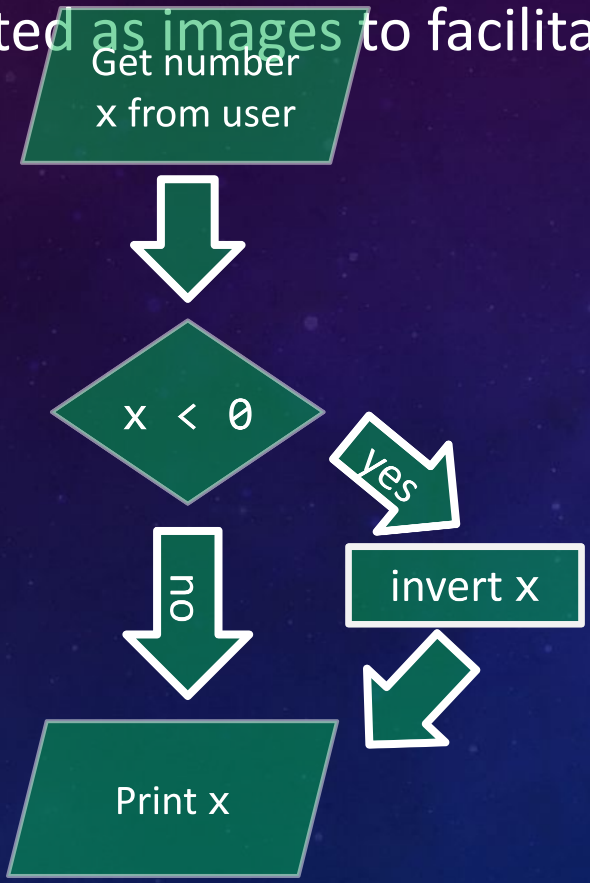
Here’s another opening brace; it starts *another* local scope. x was previously declared and can enter this nested block/local scope

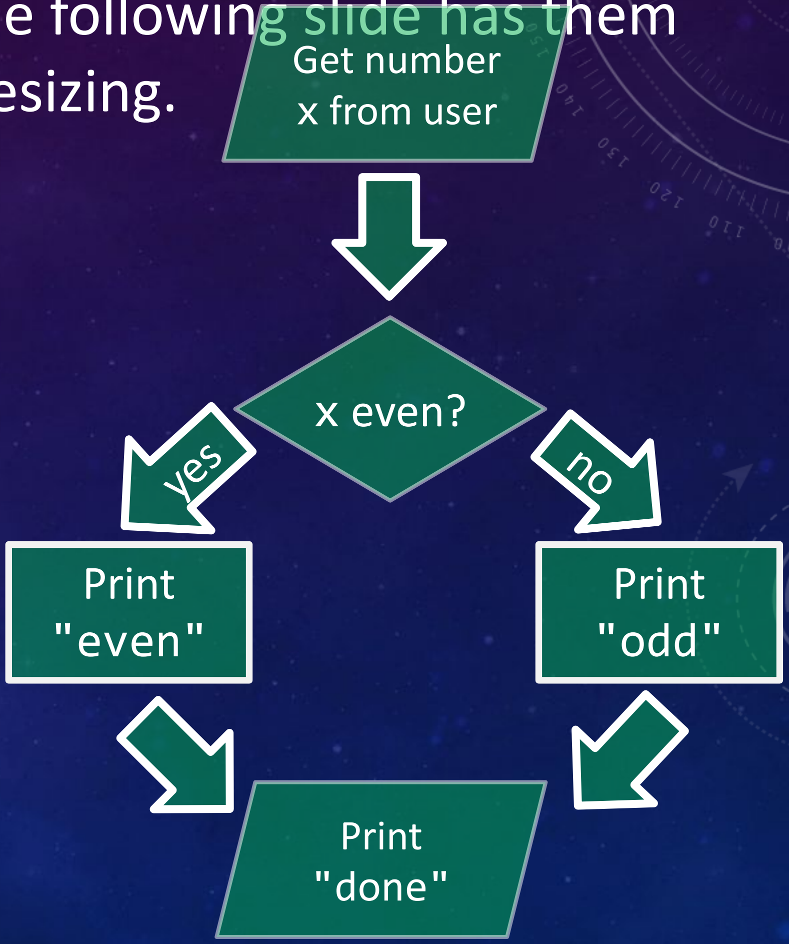
message is declared here, so it can only be used within this set of curly braces

x can be used here, but message cannot be used

else

- These are the raw diagrams, the following slide has them pasted as images to facilitate resizing.





else

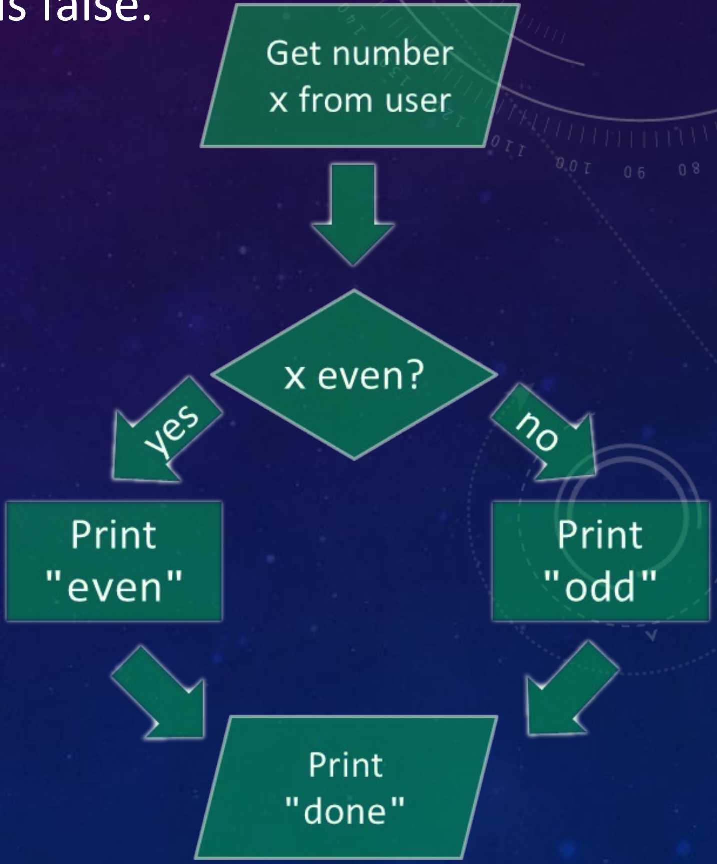
- An `if` statement may have two branches
  - A "then" branch – executed if the condition is true.
  - An "else" branch – executed if the condition is false.
- Guarantee: Only one branch is chosen.

```
int main() {
  int x;
  cin >> x;

  if (x % 2 == 0) { // is x even?
    cout << "even" << endl;
  }
  else { // otherwise must be odd
    cout << "odd" << endl;
  }

  cout << "done" << endl;
}
```

A common trick! If the remainder is 0, x is divisible by 2.



Nested `if` statements

- Control flow structures can be nested within each other.
- Let's write code to check if 0 <= x < 5…

```
// Version 1: nested if
if (0 <= x) {
  if (x < 5) {
    cout << "within range" << endl;
  }
}

// Version 2: compound condition
if (0 <= x && x < 5) {
  cout << "within range" << endl;
}
```

To get here, the code must take both branches.

Just as in MATLAB, `0 <= x < 5` doesn't work! You need to use &&.

In this case, version 2 is probably better style and nesting is unnecessary.

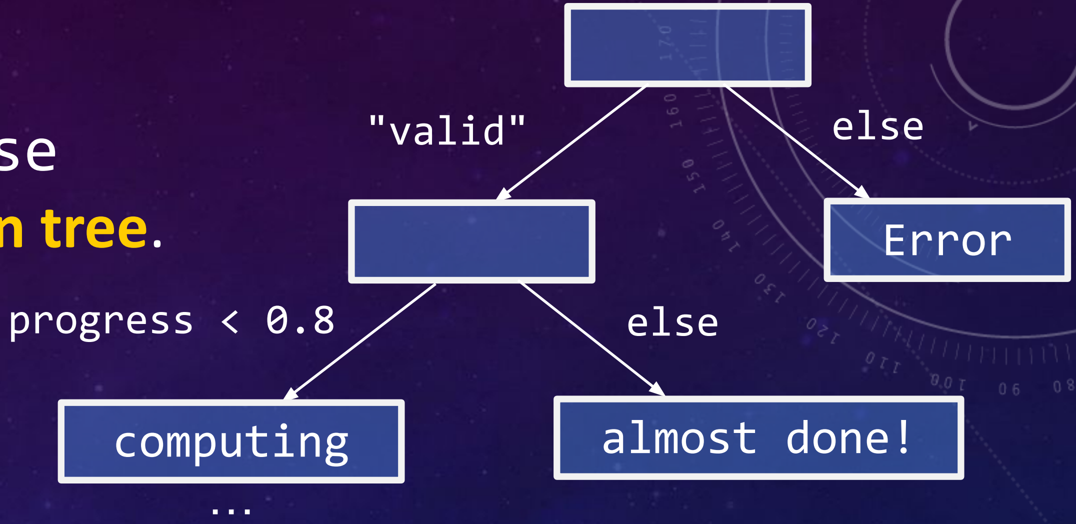
Nested `if` statements

- Sometimes nesting definitely makes your code cleaner:
- e.g. Print a report based on `status` and `progress` variables:

```
if (status == "valid") {
  if (progress < 0.8) {
    cout << "computing..." << endl;
  }
  else {
    cout << "almost done!" << endl;
  }
}
else {
  cout << "Error: invalid status" << endl;
}
```

Decision Trees

- We can model an `if/else` structure using a **decision tree**.



```
if (status == "valid") {
  if (progress < 0.8) {
    cout << "computing..." << endl;
  }
  else {
    cout << "almost done!" << endl;
  }
}
else {
  cout << "Error: invalid status" << endl;
}
```

else if

- In some cases, we want to split into more than two branches.
- Use the `else if` pattern to accomplish this:

```
// Print the phase of water based on temperature
if (temp <= 0) {
  cout << "solid" << endl;
}
else if (temp <= 100) {
  cout << "liquid" << endl;
}
else if (temp <= 11727) {
  cout << "gas" << endl;
}
else {
  cout << "plasma" << endl;
}
```

This branch will run if temp is between 0 and 100. Why?

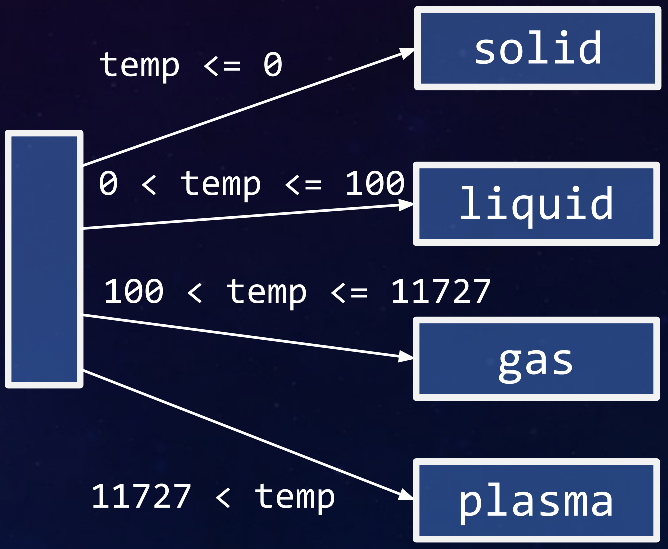
The use of else before each new branch ensures we stop as soon as any branch is entered. Again, only one branch is chosen.

It is common, although not required, to end with a plain `else`.

else if

- In some cases, we want to split into more than two branches.
- Use the `else if` pattern to accomplish this:

```
// Print the phase of water based on temperature
if (temp <= 0) {
  cout << "solid" << endl;
}
else if (temp <= 100) {
  cout << "liquid" << endl;
}
else if (temp <= 11727) {
  cout << "gas" << endl;
}
else {
  cout << "plasma" << endl;
}
```



6 min

Exercise: Branching

lobster.eecs.umich.edu
ENGR101_15_temperature

```
int main(){
  int temp = 45;
  string season = "summer";

}
```

- Write a program to print messages based on the weather:
  - Temperature less than 0: "Warning: Very cold!"
    - Otherwise, print "At least it's not below 0."
  - Temperature between 29 and 34: "Watch out for freezing rain!"
  - If the season is "summer" AND the temperature is negative:

    "Error. Please check the thermometer."

- Sometimes, more than one message may be printed!

Solution: Branching

- Write a program to print a message based on the weather.

```
int main(){
  int temp = 45;
  string season = "summer";

  if(temp < 0) {
    cout << "Warning: Very cold! << endl;
    if(season == "summer") {
      cout << "Error. Please check the thermometer." << endl;
    }
  }
  else {
    cout << "At least it's not below 0." << endl;
    if(29 <= temp && temp <= 34) {
      cout << "Watch out for freezing rain!" << endl;
    }
  }
}
```

Bad Solution: Branching

- Write a program to print a message based on the weather.

```
int main(){
int temp = 45;
string season = "summer";

if(temp < 0) {
cout << "Warning: Very Cold!" << endl;
if(season == "summer") {
cout << "Error. Please check the thermometer." << endl;
}
}
else {
cout << "At least it's not below 0." << endl;
if(29 <= temp && temp <= 34) {
cout << "Watch out for freezing rain!" << endl;
}
}
}
```

Indentation is no joke. Without it, code becomes nearly impossible to read.