Chapter 6

If statements

So far, our programs can do math, but they can't really "think." They should be able to look at the variables and somehow decide what to do next. Using one very simple new rule, we can add the ability to make all sort of complex decisions.

I'll start out by just showing the first half of the rule, then the mechanics of how it works, then the second half. After that, we can program a crazy amount of stuff, just by being clever.

The basic decision statement is an **if**. Here's a sample `if` statement:

```
public int x; // input

void Start() {
  print( "A" );

  // the interesting part:
  if( x < 10 ) {
    print("x is small");
    print("It's less than 10");
  }

  print("B");
}
```

`x` is the input, using the usual trick – change `x` in the Inspector, then run. The middle chunk of Start is a single `if` statement. What it does is probably obvious: it only prints the middle two lines if you entered 9 or less for `x`.

If you just looked at that and guessed the rules, you'd probably get most of them right. But, just to be complete, here they are:

- The syntax (the things that just need to be there) of an `if` are: the word `if`, in lower-case; the `()`'s afterwards; the {}'s after that.

A basic if looks like:

```
if( trueOrFalse ) { stuffToDo }
```

The parens are like the required ones in a print statement. They aren't math parens. They just have to be there as part of the `if`.

- There's no semi-colon at the end. That's a special rule, since the close-curly brace already marks the end.
- The part in ()'s is a true/false value – the test. Rules for what you can write, mostly stolen from math, are later. `x<10`, in the example, does exactly what it looks like – true when `x` is less than 10.
- The {} curly-braces belong to the `if` (called the body.) Inside go normal program statements.

  This is a new use of {}'s, but not really. The old {}'s group statements together for Start or Update. This new version also groups statements, just for a different reason.

  Notice how each print still has it's own semicolon at the end. These aren't special `if` semicolons – they're just regular statement semicolons, for the prints.
- The semantics (what it does) are: check the test for true or false. If true, run the body (same as usual: run statements top to bottom.) False means to skip the entire body – jump to just after the ending curly-brace.
- The way we indent the body a little is just style. Like anything else, an `if` can have spaces and line-breaks however you like. But it looks nicer this way – it's easier to see that those two lines are inside of an `if`.
- The curly-braces are optional if the body has only one line (more later.)

Just in case, a run through of the example: the program always prints `A`, then it might print the things in the `if`, then it always prints `B`.

If you enter 14 for `x`, the program only prints `A` then `B`. The `if` is false, so it skips the lines in the middle. If you entered 6, it prints `A`, the `if` is true so it prints `small` and `it's less than 10`, then it prints `B`.

The program will always print one of those two ways: either only A then B; or else all four lines.

6.1 Comparison Operators

Less than, <, is the same as regular math. So is greater than, >. But the rest of the math symbols we use aren't written on our keyboard, so need rules for how to write them. Here are all of the compare symbols, and some rules/comments:

Computer math compare symbols are:

```
<  less than
>  greater than
<= less than or equal to
>= greater than or equal to
== equals
!= not equals
```

String compare symbols are the last two: equals and not-equals.

Less-than and greater look like regular math:

```
if(x<0) { print( "x is negative"); }
if(x>0) { print( "x is positive"); }
```

The last four take some explaining. In real math, we write "or equal to" by adding an extra line underneath the symbol. But we can't type that, so we add an equals after it. `<=` counts as one symbol – no spaces, must be in that order.

Flipping them to make `=<` is an error. I remember equals comes last because in school I learned to say it in that order: "less than or equal to". Ex's:

```
if( x <= -1 ) { print( "x is negative"); }
if( x >= 1 ) { print( "x is positive"); }
```

The second-to-last `==` is the compare-equals symbol. Together both ='s count as one symbol. The reason for this is that single = is already an assignment statement. We don't want `=` to mean different things in different places, so we invented a special double-equals to mean compare-equals. Ex's:

```
if(x==5) { print("you have five"); }
if(w=="cat") { print( "meiow" ); }
```

The compare for string `==` is case-sensitive. If `w` is `"cat"` then `if(w=="CAT")` is false.

Finally, `!=` is "not-equals". In regular math, we put a slash through an equals, but we can't type that. We could move it in front, like `/=`, but we used that up in the special divide-and-assign rule. It turns out, real mathematicians have been using an explanation point to mean **not** for a long time (they call it a bang,) so we stole it from them.

Like `<=`, `!=` counts as one symbol, and must be in that order. The way I remember is `!` means not, so `!=` means not equal. Exs:

```
if(x!=0) { print( "not zero" ); }
if(w!="abc123") { print( "password incorrect" ); }
```

6.2 Some semi-useful ifs

There are a few ways to think of an `if`, and a few tricks we can do with them. One thing `if`'s can do is keep numbers in range:

```
// x can't be more than 10:
if(x>10) { x=10; }
```

This uses the rule that any normal statements can be inside of an `if`. Assignment statements are normal statements, so they can be in the body.

We can grow that idea to make sure `x` is between 1 and 10:

```
// make sure x is 1-10:
if(x<1) { x=1; } // fix too small
if(x>10) { x=10; } // fix too big
```

In this case, we can think of the `if` as detecting and fixing bad things. True means the `if` is unhappy. False means we "passed" and don't need to do anything.

The same idea can be used fix bad strings:

```
// fix common spelling errors
if(w=="teh") { w="the"; }
if(w=="thier") { w="their"; }

// if they leave the name blank, use jane:
if(name=="") { name="jane"; }
```

Notice how this mixes `==` and `=`. They can be easy to confuse. Just remember that `==` means compare and `=` means change. It works out the `==` goes inside the parens, since that's where you do the comparing.

I want to add a note here: `if`'s follow the same "do it and move on" rules as everything else. They aren't sticky. For example, this code ends with `x` as 11, even though it has a "can't go past 10" `if` in it:

```
int x=17;
if(x>10) { x=10; } // fixed to not be past 10
x=x+1; // it's 11. The if was a 1-time thing
```

We can also use an `if` to make tweaks. The way it works is the same as fixing bad values, but it feels like a different thing. In this example, `"Beth"` and `"Dick"` aren't bad, but we prefer the long versions:

```
// undo common nicknames:
if(name=="Beth") { name="Elisabeth"; }
if(name=="Dick") { name="Richard"; }
```

`if`'s can be used to make table lookups. For example numbers to number words. I'm using the shortcut where we can leave out the body curly-braces if we only have one line:

```
// get words for numbers:
string numWord = "unknown num";
if(num==1) numWord="one";
if(num==2) numWord="two";
if(num==3) numWord="three";
if(num==4) numWord="four";
if(num>=5) numWord="many"; // <- everything else
```

We can use this trick for any made-up tables, like hurricane names:

```
// get hurricane names:
if(num==1) hName="Ana";
if(num==2) hName="Bill";
if(num==3) hName="Claudette";
 ...
```

Or, this is a completely made-up table of monsters and weapons they use, which looks up on strings:

```
public string monster; // enter this

  string weapon = "";
  if(monster=="pirate") weapon="rusty cutlass";
  if(monster=="ninja") weapon="poisoned shuriken";
  if(monster=="alien") weapon="zap ray";
```

Then there are some common miscellaneous uses of an `if`:

If we have a word, like `duck`, and a count, we can add an "s" for plurals:

```
if( howMany != 1 ) animal=animal+"s"; // ex: duck becomes ducks
print(howMany+" "+animal); // 1 duck, or 4 ducks
```

The last print is a little interesting. In something like `"The "+w` we sneak the space into the end of `The`. In `howMany+animal` there's nowhere to sneak in that space, but we can still add it using `+" "+`.

We can "take off the minus sign" from a number by flipping negatives to positive:

```
if(x<0) x=x*-1; // make negative #'s positive
```

This turns -6 into 6, but leaves +4 alone. `-x` is a shortcut for `-1*x`, so `x=-x;` also would have worked.

We sometimes use `if`s as a guard. In our minds, this adds 1 to `x`, but not if it would bring us past 10:

```
// add 1 to the score, but not if it would go past 10:
if(x<10) { x=x+1; }
```

We can also add first, then fix it later:

```
x=x+1;
if(x>10) x=10;
```

In real life, you wouldn't want to add 11 gallons to a 10-gallon bucket, then take out a gallon. But in computers it's often easier to go past, then fix it.

For fun, we can use both ways to add 3 to the score, but not past 10. Here's the method where we check before adding:

```
// add 3, not past 10:
if(x<10) x+=3;
```

There's a problem. This will add 3 to 8, giving 11; or 3 to 9, giving 12. We could fix that with `if(x<8)`, but now it won't add anything to 8 or 9. Gah! We can do it correctly by using several `if`s:

```
// add 3, not past 10, 2nd try:
// handle overshoot cases. They go straight to 10:
if(x==8) x=10;
if(x==9) x=10;
// now handle non-overshoot:
if(x<=7) x+=3;
```

That's a bit long. The "fix afterwards" method works with no changes:

```
x+=3;
if(x>10) x=10; // 7,8,9 all become 10
```

Moving on, we can use a pair of `if`s to find the largest of two variables:

```
// find the largest:
if(n1>=n2) largest=n1;
if(n2>=n1) largest=n2;
```

The two `if`'s work together – one of them has to be true.

An interesting thing is that this gives the largest value, but doesn't tell us which variable it came from. Usually it won't matter – just knowing the number is good enough. Like if we order pizza and meatballs. Maybe the pizza takes 10 minutes and the meatballs take 15 – they tell us it will be a 15 minute wait, and that's all we want to know most of the time.

Another interesting thing is if the numbers are equal, both `if`'s are true, and that's fine.

6.3 Compare precedence

At some point, you might put some math inside of an `if`, and it might start to look funny. For examples:

```
if( x>y+1 ) ...
if( x + y<10 ) ...
```

The rule is: math goes first, compares go last. The official rule is that compares count as math, but all have lower **precedence** than plus, minus, multiply and divide.

This is the rule you want. It means you almost never have to put extra grouping parens inside of an `if`. But, same as before, you can if you want:

```
if( x > (y+1) ) ...
if( (x+y) < 10 ) ...
```

As with `print`, the outer parens are required, and the inner ones are just math parens.

As usual, you can still use parens to change the order of `+` and `*`. This adds 1 to `a` first:

```
if( (a+1)*b < 10 ) ...
```

This is really just the rule that things work the same, no matter where they are in the program. Any place you have math, you can use all the regular math rules.

6.4 else

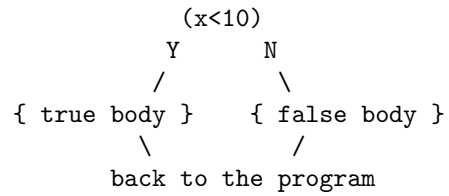
A regular `if` either runs the lines, or does nothing. A little different version can choose between two things. It adds the word **else**, and is usually called an `if-else` statement. An example:

```
if(x<10) {
  print( "less than 10" );
}
else {
  print( "10 or more" );
}
```

It's just a regular `if`, with `else { doThisWhenFalse }` jammed at the end. Notice how there are still no special `if` semicolons. All six lines count as one giant if-else statement.

Also notice there isn't a test on the else. It's just `else`, then a curly brace. We usually think of it as two bodies with a test to pick one.

Sometimes we call them **branches**. Here's a picture of how we sometimes think about it flowing through an if-else, picking a branch:



You can also think of an if-else as a shortcut for two "opposite" `if`s in a row. Here are two ways to write small or big. The if-else version is shorter, and it's easier to see what it does:

```
// 2 opposite ifs version:
if(n<100) { w="small"; }
if(n>=100) { w="big"; }

// else version:
if(n<100) { w="small"; }
else { w="big"; }
```

Here's "find the largest" both ways. I think the if-else version makes it more obvious we're picking between `n1` and `n2`:

```
// 2 opposite ifs largest:
if(n1>n2) largest=n1;
if(n2>=n1) largest=n2;

// if-else:
if(n1>n2) largest=n1;
else largest=n2;
```

Choosing between two equally good options is one way to use an if-else. Another way is just to tack-on a little extra. Two old examples with an extra `else` added:

```
if(x<10) x=x+1;
else { print("sorry - can't add past 10"); }

if(password != "abc123") print("bad password");
else print("password accepted");
```

More `else` notes:

- There's no test in the else. I wrote this before, but it's a common mistake. Besides not needing it, putting one there is an error. Ex. of an extra else-test:

  ```
  // common error putting a test with the else:
  string sound;
  if(w=="cat") { sound="meio"; }
  else (w!="cat") { sound="unknown noise"; } // this is wrong!!
  ```

- The `else` body is usually indented the same as the true body. This goes with the idea that true and false are just two branches of the same `if`. Here's a silly example where both have two lines and are indented the official way:

```
if(x>=0) {
  print("positive");
  n=+1;
}
else {
  print("negative");
  n=-1;
}
```

- The leave-out-{}s-for-one-line rule can also be used for the `else`. Examples:

```
// both can be 1-line and skip {}:
if(x<100) w="too small";
else w="too big";

// 1-line true skips, else doesn't:
if(z==0) z=1;
else {
  print("you picked a good z");
  print("thanks");
}

// true uses {}, else skips:
if(w=="cat") {
  print("changed it to kitty for you");
  w="kitty";
}
else print("not changing it");
```

- An `if-else` can flip the test and flip the true and false parts. Sometimes that makes it look nicer. These are the same:

```
if(password != "abc123") print("bad password");
else print("password accepted");

if(password == "abc123") print("password accepted");
else print("bad password");
```

  Both versions only accept `abc123`.

This comes in handy when you're having trouble writing the test – maybe the opposite way is easier. Another example:

```
if(n<100) { w="small"; }
else { w="big"; }

if(n>=100) { w="big"; }
else { w="small"; }
```

The second version might look nicer if we "like" big numbers.

- It's possible to put "nothing" after an `if` by writing {} (an empty body.) Here's a strange "don't let `x` get past 10":

```
if(x<=10) {} // everything is fine, don't do anything
else { x=10; } // too big, lower it
```

The `else` can have an empty body, which is the same as not having it at all:

```
if(x>10) { x=10; }
else {} // oops! don't need this. But legal
```

I do it rarely, but sometimes empty bodies work as placeholders:

```
if(animal=="dog") print("arf");
if(animal=="elephant") {} // nothing, yet
```

6.5 And, Or

The test is allowed to use combiners *and* and *or*. This `if` is true for yellow dogs:

```
if(animal=="dog" && color=="yellow")
  print("You should name your yellow dog Sparky.");
```

I'm just going to write all the rules, then examples for them later:

- The symbol for *and* is `&&`. The symbol for *or* is `||`. Both are two of the same thing in a row, no spaces, and count as one symbol. The *or* bar is the line above the backslash on most keyboards.
- They go between legal if-compares (things that can be in an `if` by themselves.) In the example above, `animal=="dog"`, and `color=="yellow"` could both be in regular `if`s.

- `&&` means they both have to be true. `||` means either can be true, or both.

  The "or both" rule seems like it might be confusing, but it's not once you start using it for real things. A computer `or` is always something like "13 years old or at least 4 feet tall to go on this ride." Being both is clearly fine.

- You can use `&&` or `||` multiple times. It means they all have to be true, or at least one has to be true (examples below.)

- If you mix `&&` and `||` in a big compare, `&&` has higher precedence. You can think of `&&` being like multiply, and `||` like plus. You can use parens to group. This rule can be pretty tricky. We won't use it right away. I just wanted to get all the rules in one place.

- `&&` and `||` have lower precedence than compares do. In other words, they work the way you think. It does all the math, then all the compares, then the and/or checks. You don't have to add parens, but you can.

Suppose we want to know if `animal` is a cat or a dog. We have to write a legal check for a cat, then a legal check for a dog, and combine them:

```
if(animal=="cat" || animal=="dog")  print("common housepet");
```

You can think of this as the "each has to be legal" rule, or as the "and/or always goes last" rule. Suppose we wrote `if(animal=="cat" || "dog")`. The computer sees 2 things joined by an `or`, and the second one is `if("dog")`, which makes no sense (it gives a compile error.)

Suppose we want to check for discount tickets (very old or very young.) We have to write each part out, like this:

```
if(age<=12 || age>=65)  print("discount tickets");
```

If we tried the shortcut `if(age<=12 || >=65)` the computer would see `if(>=65)` for the second part (obviously an error.)

If we need `x` and `y` to both be 0 or more, we also have to write it the long way:

```
if(x>=0 && y>=0) print("both 0 or more");
```

The shortcut `if(x && y>=0)` would see `if(x)` for the first part (also a compile error.)

A good thing about each part being separate is you can mix and match any kind of variables and tests. For example, a string and int check:

```
// young cats are kittens:
if(animal=="cat" && age<=3)  print("kitten");
```

The way the computer sees it, `animal=="cat"` is true or false. Then `age<=3` is true or false. By the time the `&&` happens, everything is down to only true's and false's.

With string comparing, it's fine to mix `==` for one and `!=` for the other:

```
// non-brown cows are fancy:
if(animal=="cow" && color!="brown") print("fancy cow");
```

The multiple compare rule says we can check for baby fancy cows by adding one more:

```
if(animal=="cow" && color!="brown" && age<=3) print("young fancy cow");
```

This is a little bit like `3+7+4`. There's no special 3-way rule. Technically you're adding a pair at a time: `(3+7)+4`. Also technically, the computer *and*'s the animal and color first. But it works out the same either way - they all have to be true.

It can be helpful to look at some broken code. These are legal, but pointless. Just say them out loud and it's obvious way:

```
if(age>=18 && age>=21)  print("same as if(age>=21)");
if(age>=18 || age>=21)  print("same as if(age>=18)");
```

For the second one, `age` of 21 or more seems like it might be "double-true," but there's no such thing. Everything inside of an `if` always comes out to be just regular true or false, no matter how complicated it is.

We can make some other wrong example `if`s that are always false. A number can't be less than 5 and more than 10 at the same time, and a word can't be `"cow"` and `"horse"` at the same time:

```
if(num<5 && num>10)  print("always false");
if(animal=="cow" && animal=="horse") print("always false");
```

Here are some useless always true `if`s using an *or*. They're always either true or "double true":

```
if(num>7 || num<15)  print("always true");
if(animal!="cow" || animal!="horse") print("always true");
```

In the first, all small numbers are less than 15, all larger numbers are greater than 7, and 8-14 are "double true" (which is the same as normal true). In the second, cow is true since it's not horse, horse is true since it's not cow, and everything else is true since it's not cow or horse.

This next example shows how precedence normally does the right thing (the math is made-up. Just think about the grouping):

```
if( x + 3 < y || 3 < x || y == 9 ) ...
// how the computer sees it:
if( ((x+3) < y) || (3<x) || (y==9) ) ...
```

It looks a little like the computer might try `y || 3`, or maybe `x || y`, but it knows *and*'s and *or*'s go last.

6.5.1 Useful and/or's

Now onto tricks and examples people really use.

We check "if it's one of these" by using `==`'s chained together by *or*'s:

```
if(n==2 || n==4 || n==6 || n==8)
  print("single digit even");

if(w=="bill" || w=="willy" || w=="billy")
  print("william");
```

It might seem like there would be a shortcut where we didn't have to write `n==` or `w==` over and over. But there isn't.

Range-checking is common. In math, we'd write 1 to 10 as: $1 \leq n \leq 10$. That's really a shorthand for two compares. In a program, we have to write them out:

```
if(n>=1 && n<=10) print("n between 1 and 10");
```

A way to think of this is it can't be too small (`n>=1`) and can't be too big (`n<=10`). Both parts have to be true, so it's an *and*. It we used an *or* by mistake, the whole thing would always be true (everything is either bigger than 1 or less than 10. Some numbers are both.)

We can do some cute things with a range check. Say we want to check how many digits a number has. There isn't a special way, but 0-9 is 1-digit, 10-99 is two digits, and so on (pretend we know the number isn't too big):

```
// how many digits does n have:
int digits=0;
if(n>=0 && n<=9) digits=1;
if(n>=10 && n<=99) digits=2;
if(n>=100 && n<=999) digits=3;
```

We can mix ranges and exact numbers. Suppose we win if we get 7 or 10-15 or 22. That's like checking for three exact numbers, using `||`'s. Except the middle one is a range:

```
if( n==7 || (n>=10 && n<=15) || n==22 ) { print("You win"); }
```

Suppose instead we win on 5-9 or 18-21. That's "this range or that range":

```
if( (n>=5 && n<=9) || (n>=18 && n<=21) ) { print("you win differently"); }
```

Mixed and's/or's can be confusing. Sometimes you can just see what they do. Other times you have to trace them. Convert each part to T/F, then combine them a little at a time:

```
// suppose n is 9
if( n==7 || (n>=10 && n<=15) || n==22) { print("You win"); }
1)   F   || (  F   &&   T  ) ||   F
2)   F   ||        F         ||   F
3)               F
```

Often we want to check whether two `float`s are within 0.01 of each other. If we're sneaky, we can find the difference, then use a range-check to see it that's small:

```
float diff = n2-n1;
if(diff>-0.01f && diff<0.01f) print("close enough");
```

But it's fun to try to write it directly. We can do a range check comparing `n1` to "a little less than `n2`" and "a little more than `n2`":

```
if( (n1 > n2-0.01f) && (n1 < n2+0.01f) ) print("close enough");
```

Checking not-in-range isn't as common. It looks like a range-check, but it has to be too low *or* too high:

```
if( n<1 || n>10 ) print("n is NOT 1-10");
```

If you use an *and* by mistake it will always be false (nothing can be less than 1 and more than 10 at the same time.)

6.6 Nested ifs

`if` and `else` bodies can have more `if`s inside. We usually call them **nested** `if`'s (after those nested Russian dolls.) We don't need any new rules for this, but we do need practice thinking about how to use this trick.

Suppose we have a truck fee: 1000 pounds and under is free, up to 5000 pounds is $10, and past that is $20. This nested `if` prints the fee:

```
int fee=0;
if(wt<=1000) { fee=0; print("No fee"); }
else {
  if(wt<=5000) fee=10;
  else fee=20;

  print("Truck fee is $" + fee );
}
```

I think this seems pretty natural. The first `if` divides it into free and pay. If it's free, we print that and are done. Inside the pay part, we need one more `if` to check the cost. Notice how the inside `if` is indented to look nice in the `else` body.

A trick is to check it by "exercising" each part. Run it in your head with a number from each category, like 500, 1500 and 7000, which should give every possible result, and make sure it does.

Here's a longer one that finds the prices for animals. Most animals cost 3, except for cats and dogs, and red dogs and white and black cats are exceptions:

```
int cost=3; // "other" animals all cost 3

if(animal=="dog") {
  cost=8; // most dogs are 8
  if(color=="red") cost=16; // red dogs are rare
}

if(animal=="cat") {
  cost=10; // most cats are 10
  if(color=="white") {
     cost=14; // white cats are valuable
  }
  if(color=="black")
    cost=5; // no one likes black cats
}
print("Cost of " + color + " " + animal + " is $" + cost);
```

Hopefully this reads pretty well. The dog `if` says dogs cost 8, then does one extra check for red dogs. Notice how it only checks the color – we already know it's a dog. The cat `if` is the same, but longer (and I changed-up the paren style for fun.)

If you remember from school, the x/y graph is broken into quadrants around (0,0) (like the picture below.) To find the quadrant of an (x,y), we could brute force with four lines like `if(x>=0 && y>=0) quadrant=1;`.

But a nicer way is to make a plan: divide it into top/bottom half, and go from there:

```
// Standard quadrants (0,0) in the middle:
// 2 | 1
// -----
// 3 | 4

if(y>=0) { // top half
  if(x>=0) quadrant=1;
```

```
  else quadrant=2;
}
else { // bottom
  if(x>=0) quadrant=4;
  else quadrant=3;
}
```

It could also be written starting with a left/right check.

We can make several plans to find the largest of three numbers. This one checks whether A is the largest. If not, it must be B or C:

```
// find largest of a,b,c
if(a>b && a>c) largest=a;
else {
  // it wasn't A, so check B and C:
  if(b>c) largest=b;
  else largest=c;
}
```

A different plan, trickier than the first way, starts by comparing A and B. The smaller one can't be the answer. Compare the larger one to C:

```
// find largest of a,b,c (version 2)
if(a<b) {
  // A is too small, must be B or C:
  if(b>c) largest=b;
  else largest=c;
}
else {
  // B is too small. Must be A or C:
  if(a>c) largest=a;
  else largest=c;
}
```

A simple way to test is using every combination of 1,2,3 (there are 6 of them.) Not that it matters, but this way is a little faster. It always uses two compares. The first way sometimes uses three compares.

This next one is simpler. It checks for positive, negative or 0. I thought it made sense to "knock out" 0 first, then check for +/-:

```
if(x==0) print("zero");
else {
  if(x>0) print("positive");
  else print("negative");
}
```

6.7 Common errors

- It's very easy to forget/mistype one of the =='s in a compare:

  ```
  if(n=3) { print("perfect number of cats"); } // ERROR
  // Cannot implicitly convert type int to bool.
  ```

  The error is the computer way of saying that assigning `n=3` isn't a true/false. But even if you didn't know that, double-clicking will give you the line. When you look it over, you'll spot the bad single-equals eventually.

- It's easy to become "`==`-happy" and accidentally use them for assignment:

  ```
  x==5; // ERROR. Wanted x=5;
  // Only assignment, call, increment, ... can be used as a statement.

  if(animal=="cat") // this == is correct
    color=="red"; // ERROR (same message) Wanted =
  ```

  It's the general purpose error, but at least it gives you the line. The computer is really telling you that comparing `x` to 5, just by itself, is useless, so you must not have meant to do it.

- Adding an extra semi-colon in the middle makes a very hard-to-find non-error error. This will always print `"n is positive"`, no matter what `n` is:

  ```
  if(n>0); // <- oh no!!
    {
      print("n is positive");
    }
  ```

  The semicolon legally ends the `if`. There's no error, but there's also nothing inside the `if`. Here's how the computer sees it:

  ```
  if(n>0) {} // empty body. 100% useless if
  print("n is positive"); // out of the if, runs every time
  ```

  This can cause hours of frustration if you don't know to look for that extra semi-colon.

  Putting an extra semi-colon after the {} isn't a problem. Don't do it on purpose, but it's fine if you do:

  ```
  if(n>0) { print("moo"); }; // useless ending, but runs the same
  ```

- The rule where you can leave out the {}'s for one statement can cause some serious problems if you aren't careful. What happens is, you start with this, which is fine:

```
if(n>10)
  n=10;
```

  But then you add another line to the body, which causes a problem:

```
if(n>10)
  print("too big");
  n=10;
```

  The indenting makes it look like both are in the body (and that's what you meant.) But the missing{} rule takes only 1 line. The computer reads it like this:

```
if(n>10)
  { print("too big"); }
n=10; // uh-oh. out of the if. n is always 10!
```

  Non-error errors caused this way can also cause really frustrating, hard-to-find wrong code. Here's a version of the same thing, on one line:

```
if(a=="cow") sound="moo"; eats="grass"; // Not what it looks like
```

  In our minds, the whole line is the `if`. But the short-cut is only for one statement. The computer sees this:

```
if(a=="cow") { sound="moo"; }
eats="grass"; // always does this line
```

  We'll drive ourselves crazy trying to figure out why the stupid computer thinks tigers and bears also eat grass. Of course, writing it on one line with curly-braces is fine:

```
if(a=="cow") { sound="moo"; eats="grass"; }
```

  Some people think it's just easier to always put {}s, even for very short bodies. That way you can never have this sort of error. That's partly why some editors always add {}'s after you type the `if`.

- When comparing the same thing several times, it can be tempting to leave out the variable, like this:

```
if(n==3 || 6 || 8) // ERROR
// Operator ‘||’ cannot be applied to operands of type ‘bool’ and ‘int’

if(animal=="goat" || "pig" || "cow") // same error
```

There's no shortcut like this. You have to write out each part.

- Nested if-else-ifs and missing {}'s can trick you into not knowing which `if` goes with the `else`. Suppose you have this:

```
// runs wrong:
if(n<10)
  if(n>0) print("between 1 and 9");
else
  print("big");
```

Indenting makes it look like the `else` goes with the first `if`. But it really goes with the second one. The computer sees this:

```
if(n<10)
  if(n>0) print("between 1 and 9");
  else print("big");
```

It will print `big` for negative numbers and do nothing for ten or more.

Writing out the curly-braces can fix these problems (another reason some people always write them):

```
if(x<10) {
  if(n>0) print("between 1 and 9");
  // an else for the inner if would need to be here
}
else
  print("big");
```