Chapter 7

more if tricks

Now that we have ifs, there are lots of interesting problems we can solve, and tricks we can do. This section won't have any new rules, just lots of examples.

7.1 falling through

Sometimes you need a few `if`s to try to compute a value. One trick is, pick one of the values to start with. If all of the ifs are false, we say it "falls through" and keeps that starting value. It's like having a "none of the above" (except it's none-of-the-below.)

This is just a cutesy, not-really-useful way to rewrite an if-else, but it shows how the trick works:

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

Here's the same trick to to make a "which is larger" `if`:

```
// alternate way to find largest
largest=n1;
if(n2>n1) largest=n2;
```

A more interesting example is to rewrite finding positive, negative, or 0. In this next example, `"zero"` is the fall-through:

```
string w="zero";
if(n>0) w="positive";
if(n<0) w="negative";
```

Then, here's a made-up example, where most insurance costs $100, but helicopters and canoes are different:

```
float cost = 100;
if( vehicle == "helicopter" ) cost=250;
if( vehicle == "canoe") cost=85;
```

A trick to these is to think of it as a *process*. We're not saying that `cost` is definitely 100 – we're just starting it off as 100. The other trick is that `int`s aren't like mashed potatoes. Adding or subtracting to a pile of mashed potatoes makes a mess, but we can easily adjust `int`s all we want.

7.1.1 Cascading range-slicing ifs

We often want to program a lookup table with ranges and cut-offs. For example the standard grade 90=A, 80=B, 70=C rules. An obvious, awkward way is to hand-check each range, using this ugly, too-long code:

```
// bad, ugly grade-checking code:
if(score>=90) grade="A";
if(score>=80 && score<90) grade="B";
if(score>=70 && score<80) grade="C";
if(score>=60 && score<70) grade="D";
if(score<60) grade="F";
```

Notice how each breakpoint is repeated twice. 70 is in the 3rd `if` and 4th `if` (70-80 and 60-70.) And I had to be careful with `>` and `>=`. It would be easy to miss a number (like using `>80` and `<80` in both places, so we never give a grade to 80.)

A nicer way is using `if-else`'s and nested `if`s. This works, and follows the by-the-book rules for indenting and nesting `if`s, but is still long and confusing:

```
// better grade-checking, but still not done:
if(score>=90) grade="A";
else {
  if(score>=80) grade="B";
  else {
    if(score>=70) grade="C";
    else {
      if(score>=60) grade="D";
      else grade="F";
    }
  }
}
```

The `&&`'s are gone, which is nice, but there's a lot of confusing {}'s and indents. It's also harder to see that this is really just a table look-up.

For the final version, we'll re-arrange it to look better. This is the most common way people write a range-slicing cascading `if`:

```
// final version of grade-checking:
if(score>=90) grade="A";
else if(score>=80) grade="B";
else if(score>=70) grade="C";
else if(score>=60) grade="D";
else grade="F";
```

Again, this isn't a new rule, just a clever way to rewrite things. It's still one giant `if` (it's exactly like the thing above.) Moving those `if`'s to the line right after each else is just a way to make it look better.

It runs line-by-line, quitting when it sees one it likes. One way to figure it out is to imagine it running for every possible number. The first `if` "slices off" everything 90 or more for an A. The numbers that get to the second line are the left-overs, all less than 90. The second line slices off 80 or more for a B. 79 and less goes on to the third line, and so on.

Looking at it another way, the line `if(score>=70) grade="C";` would tell you 97 was a C *if it was by itself*. But it's not by itself. It's part of an assembly line. 97 never makes it past line 1.

This idea works going down or going up. Here's the grade-checker starting from F and going up:

```
if(score<60) grade="F";
else if(score<70) grade="D";
else if(score<80) grade="C";
else if(score<90) grade="B";
else grade="A";
```

We can analyze it the same way: line one slices off 0-59 for an F, sending 60+ on. Line two slices off 60-69 for a D, sending on 70+.

Here's a different one. The old text-based dungeon MUDs didn't tell you the damage as a number – that was considered to be lazy and mood-breaking. Instead it turned the damage amount into a word. If you hit for 8, it told you "you cut the orc." Here's a damage-to-word table:

```
// damage-to-word table (damage is an int, and always 0 or more):
// 0-2: tickle
// 3-5: scratch
// 6-9: cut
// 10-15: hurt
// 16-25: wound
// 26+: mangle
```

This cascading `if` runs it (assume `hit` has the damage). Notice how each important number from the table occurs once:

```
string ouch;
```

```
if(hit<=2) ouch="tickle";
else if(hit<=5) ouch="scratch";
else if(hit<=9) ouch="cut";
else if(hit<=15) ouch="hurt";
else if(hit<=25) ouch="wound";
else ouch="mangle";

print("You "+ouch+" the "+monsterName);
```

A common error in a cascading-if is to forget an else in the middle. The program will still be legal, but it will run wrong. Here, I left out the else on the D line:

```
// bad code, missing an else:
if(score>=90) grade="A";
else if(score>=80) grade="B";
else if(score>=70) grade="C";
if(score>=60) grade="D"; // <-- oops, no else
else grade="F";
```

This is now *two* ifs. The first is a 3-line A/B/C if. The second is a simple if/else checking for D/F. Suppose grade is 93. The first `if` gives an A. But the second `if` gives a D. Forgetting the `else` makes it so everyone has a D or an F.

7.1.2 Do only one of these, none of the above

When you have several `if`'s in a row, each one could be true or false – all true, all false, or a mix could all happen. Often that's what you want. This example could say a number is only positive, or only 1-digit, or positive and 1-digit, or positive and close to 10, or all three:

```
// ex of how more than one if might be true:
if(n>0) print("positive");
if(n>-10 && n<10) print("one digit");
if(n>=8 && n<=12) print("close to 10");
```

But sometimes, it works out that only one of them can happen. Here's my "number to word" from before (written before we had `else`s):

```
if(n==1) w="one";
if(n==2) w="two";
if(n==3) w="three";
```

Logically only one of those is true, but you can't tell that right away. Rewriting it as a cascading `if` makes it more obvious that this is really a table look-up, and only one will be true:

```
if(n==1) w="one";
else if(n==2) w="two";
else if(n==3) w="three";
```

We didn't need to write it like that, but most people think it looks nicer.

Cascading `if`'s can also be used to make a "none of the above," by adding a final `else`. This example has a look-up table for what sound an animal makes, with a default sound "rrr-rrr":

```
if(ani=="cat") w="meio";
else if(ani=="crab) w="clickity click";
else if(ani=="sheep") w="baa-baa";
else w="rrr-rrr";
```

The other way to do this is with the fall-through trick, setting `w` to `"rrr-rrr"` at the top.

When most people see a cascading `if`, they know a final `else` is "none of the above," and no final `else` means the whole thing could do nothing.

Sometimes we use a cascading `if` to make sure that only one `if` can fire. This is a rare case, and is a little tricky, Here's a made-up example:

Suppose some formula adds 5 to numbers under 10, doubles numbers from 10-100, and divides larger numbers by 3. It's just a nonsense formula, but for real you sometimes have things like that.

My first try of putting that in the computer is to check each thing with an `if`. Here's the code, but it's got a sneaky logic error:

```
// wrong way to run the silly formula:
if(n<10) n += 5;
if(n>=10 && n<=100) n *= 2;
if(n>100) n = n/3;
```

The problem is that it might incorrectly do two of them. Say the number is 7. Line one adds 5 to get 12, like it should. But now stupid line two sees `n` is 12 and incorrectly doubles it! It doesn't know 12 isn't the input `n`, it's the answer `n`. We get the same problem with 60: line two correctly doubles to 120, then stupid line three incorrectly divides it!

A clever person could rewrite it to maybe avoid double-ifs. But an easy way is to turn them into a cascading `if`:

```
// correct silly formula, that never double-changes the input:
if(n<10) n += 5;
else if(n>=10 && n<=100) n *= 2;
else if(n>100) n = n/3;
```

7.2 Swapping

This isn't really an `if` problem, but it always happens inside of an `if`. Sometimes we want two variables, say `n1` and `n2`, to be in order. If `n1` is 5 and `n2` is 3, we want to switch them to be 3 and 5.

Switching the values of two variables is a little trickier than it seems. Suppose you write `n1=n2; n2=n1;`. That just makes two copies of `n2`.

To correctly switch variables, we need an extra spot, and a little dance. We usually call that a swap. This sorts `n1` and `n2` in order, if they weren't already:

```
int temp; // 3rd variable used for the swap
if(n1>n2) { // out of order
  // swap n1 and n2:
  temp=n1; // save n1
  n1=n2;
  n2=temp; // n2 gets saved copy of n1
}
```

You can hand-trace this for fun – make boxes for `n1`, `n2` and `temp`, and run the 3 lines. Changing the order usually makes it not work, but there is one other way to write it.

7.3 Incremental calculations

Often your instinct is to want a formula that just gives the answer. Like `n=` (insert equation here.) A lot of problems are easier to solve a little bit at a time.

Often we want to count up how many items fit some rule, like how many of `a`, `b` and `c` are positive. Here's my first try, which is long and not done:

```
if(a<=0 && b<=0 && c<=0) positiveCount=0;
if(a>0 && b>0 && c>0) positiveCount=3;
// check if only one is positive:
if(a>0 && b<=0 && c<=0 || a<=0 && b>0 && c<=0 || // I give up!!
```

A neat trick is, pretend you're counting on your fingers. Start the total at 0, look at each item, and add 1 if you like it:

```
int positiveCount=0; // we have 0, so far
if(a>0) positiveCount++; // add 1 shortcut
if(b>0) positiveCount++;
if(c>0) positiveCount++;

print( positiveCount + " are positive" );
```

We just push the answer a little at a time. By the time we finish, it's correct.

Here's the same idea, but it counts how many are between 40 and 60:

```
int inRange=0; // we have 0, so far
if(a>=40 && a<=60) inRange++;
if(b>=40 && b<=60) inRange++;
if(c>=40 && c<=60) inRange++;

print("There are "+inRange+" between 40 and 60");
```

Finding the largest of four numbers can also go a little at a time. It works like the "Price is Right" prize spin-off.

One person spins the wheel and steps into the winner's circle. They haven't won, but they're the winner so far. The next person spins and has to beat them. And so on. Each person only has to beat the previous best.

Here's how it looks in code, finding the largest of 4 numbers:

```
// find the largest out of a,b,c,d:
int largest=a; // a is the one to beat
if(b>largest) largest=b;
if(c>largest) largest=c;
if(d>largest) largest=d;
```

The trick is, `largest` is always the largest so far.

Just for fun, here's finding the largest of a,b,c,d using just a big if-else. It's longer and uglier:

```
if(a>b && a>c && a>d) largest=a;
else if(b>c && b>d) largest=b;
else if(c>d) largest=c;
else largest=d;
```

This version gets even worse with 5 numbers. In the incremental methods, we just need one more `if` for each extra item.

7.3.1 guessing game example

This is a more oddball example of computing something a little at a time.

Suppose we have some sort of darts or guessing game. 70 is the target, worth 5 points. We get 3 points for being within five, 2 points within ten, and 1 point within thirty (the exact numbers aren't really important.)

One way to solve it is with an ugly "hammer&tongs" approach, where we just brute figure the ranges and check them: "within 5" is 65-75, but not 70; "within 10 but not within 5" is 60-64 and 76-80; "within 30 but not 10 is" 40-59 and 81-100; and 0 points for "more than 30 away" can be the fall-through.

That's a lot of numbers. I'd probably make a number-line and a little chart, to keep track of them. Then we write an `if` for each. This works, but is hard to read:

```
// ugly range example:
int points=0;
if(n==70) points=5;  // exact
if( (n>=65 && n<=75) && n!=70) points=3;  // within 5, but not exact
if(n>=60 && n<65 || n>75 && n<=80) points=2; // 6-10 lower or higher
if(n>=40 && n<60 || n>80 && n<=100) points=1; // 11-30 lower or higher
```

One obvious way it seems like we could improve this is with a range-slicing cascading if. The trick to making a good one is figuring out what to slice off first. Usually you go low to high, or high to low. But for this one, it makes more sense to start at the target, and to go away from it. I think this looks not-too-bad:

```
if(n==70) points=5;  // exact
else if(n>=65 && n<=75) points=3;  // within 5
else if(n>=60 && n<=80) points=2; // within 10
else if(n>=40 && n<=100) points=1; // within 30
else points=0; // too far
```

This uses the same slicing principles. The second `if` is really "65 to 75, but not 70," because 70 was sliced off in the first line.

Finally, here's what I think is the best way to solve this problem. The really neat thing about some computer code is you can think "how would a human solve this?" and then program the computer to do that.

As a human, I'd first see that the points table is written as "how much away." So, I'd figure out how far `n` was away from 70, as a positive number. Then I'd look up that number in the "points for how far away" table.

Here's how that plan looks in the computer:

```
// figure out how far we are away from the target:
int missedBy = n-70;
if(missedBy<0) missedBy = missedBy * -1; // always positive

// table look-up:
if(missedBy==0) points=5;
else if(missedBy<=5) points=3;
else if(missedBy<=10) points=2;
else if(missedBy<=30) points=1;
else points=0;
```

It's easier to see the plan, and the cascading `if` is easier to read (it's the very first way I described it, before I confused myself with all the math.)

7.3.2 Gold mine problem

We have several gold mines. Each can have 1 or 2 workers. With 1 worker, they produce 2 gold. With 2 workers make 3 gold. We want to know how much total gold we can mine.

There are plenty of computer science word problems like this. They might be math, or might be about using `if`'s. They're probably not difficult, once you figure them out.

My first thoughts are to list facts: if we have more mines than workers, then mines don't matter – everyone works by themself, getting 2 gold. Workers past double the mines won't matter – every mine produces 3 gold. We can never have a mine with 2 workers, and another empty – we want a worker in each mine before we start to double-up.

From the darts example, I know writing code based on that will turn into a mess. An old trick is to pick some "typical" numbers, get the answers by hand, and see it that leads to a formula:

```
Mines  Workers  empty one two
  7      5        2    5   0
  7     34        0    0   7
  7     12        0    2   5

  5      6        0    4   1
 20     32        0    8  12
```

Without thinking about it, I computed only the workers in each mine. That sounds about right. We can easily compute the gold from that. It also seems as if there are 3 cases. Workers <= mines; workers between mines and double mines, and workers >= double the mines.

This partly done cascading `if` should properly select between the cases:

```
public int mines, workers; // input

  int m1=0, m2=0; // how many mines have 1 or 2 workers
  // NOTE: we don't care about mines with 0 workers, since they make no gold

  // testing to choose category::
  if(workers<mines) print("everyone works, alone");
  else if(workers<mines*2) print("all mines used, some are doubled-up");
  else print("all mines have 2 workers");

  // normal mines make 2 gold, doubled-up mines make 3:
  int totalGold = m1*2 + m2*3;
```

I should test it (maybe just in my head) with some of those sample numbers.

Then I'll have to work out the equations for each one. The middle is the most difficult. With 7 mines and 12 workers, the first 7 workers go to each mine, then the next 5 double-up. 5 came from 12-7. So `workers-mines` is how many doubled-up mines we have. All three equations, inside the `if`:

```
  // compute m1=mines w/1 worker, m2=mines w/2 workers
```

```
if(workers<=mines) { m1=workers; m2=0; }
else if(workers<mines*2) { m2=workers-mines; m1=mines-m2; }
else { m1=0; m2=mines; }

//print("m1="+m1+" m2="+m2); // testing
```

7.4 Tricks with modulo

Integers have a remainder operation named **modulo** (we stole the idea and the name from math.) It can be used in some sneaky ways. First, how it works.

We know integer division drops the fraction. 16/5 is 3, not 3.2. If you think of it this way, division really has two answers – the number of times it goes in, and the remainder. From second grade, 16/5 is 3, remainder 1. Both answers are perfectly good numbers.

The / symbol means to divide and give me the first answer – how many times it goes in. % means to divide and give me the second answer – the remainder. Obviously, if you divide by 5, the remainder is between 0 and 4. It looks like this:

```
print( 87%5 ); // 2. Fives up to 85, remainder 2

print( 35%5 ); // 0. Five goes evenly into 35, No remainder

print( 13%3 ); // 1. Threes up to 12, one left over

print( 7%10 ); // 7. Tricky. Ten goes in zero times. Everything is the remainder
```

No one ever uses modulo in big equations, like x+y%5*2. But, just so you know, modulo counts as division, so has the same precedence as `*` and `/` (it happens before plus and minus.) Also, modulo with negative numbers works funny (does 5 go into -8 minus one or minus two times?) People rarely use modulo on negative numbers.

Here are some common tricks with modulo:

If you divide by 2, the remainder is 0 or 1. Remainder 0 meant that 2 went in evenly, 1 means it didn't. So, checking even/odd integers works like this:

```
if(n%2==0) print("even");
else print("odd");
```

This is also a fun example of math inside an `if`. Try it with 7: Two goes into 7 three times, remainder 1. So `7%2==0` is false.

You can think of the `if` as "if 2 goes into `n` with no remainder."

Another fun trick with modulo is breaking a number into digits. `n%10` is always the one's place, just because of how base-10 numbers work. Try it: take `569%10`. Ten goes into 569 fifty-six times, with 9 left over.

We can get the 10's and 100's places by shifting them into the 1's place. Since integers drop the fraction, dividing by 10 merely shifts: `569/10` is 56. The 10's place was moved into the 1's place. In use:

```
print("Ones place is " + n%10 );
print("Tens place is " + (n/10)%10);
print("Hundreds place is " + (n/100)%10);
```

Now we can do some fun modulo examples, where we grow the answer.

A fun one is converting pennies into quarters, dimes, nickels and left over pennies. As a human, you take out the quarters, then take dimes out of the remainder, and so on. We can do that in a program. For simplicity, say we always have 99 or less pennies. The first part:

```
public int pennies; // input, in Inspector
public quarters, dimes, nickels; // output (we will see them change)
```

Finding quarters is just dividing by 25 and dropping the remainder, which is exactly how integer math works (you knew there was a reason.) The left-over pennies are the remainder, which **modulo** will get:

```
void Start() {
  quarters = pennies/25;
  pennies = pennies%25; // always 0-24 left over pennies
}
```

Testing just that, if we enter 93 for pennies, running gives us 3 quarters and 18 pennies (in the Inspector.) Not completely correct, but correct so far. Using 12 for pennies gives 0 quarters and 12 pennies, which is also correct, so far. The pennies variable is now the 0-24 left-over pennies.

Then we do the same thing for dimes and nickels:

```
void Start() {
  quarters = pennies/25;
  pennies = pennies%25;
  // pennies is now 0-24, after taking out quarters

  //print("after Q, pennies="+pennies); // debug

  dimes = pennies/10; // will be 0, 1 or 2
  pennies = pennies%10; // will be 0-9, after taking out dimes

 // print("after D, pennies="+pennies); // debug

  // only makes 0 or 1 nickel, but shorter than an if!!
  nickels=pennies/5;
  pennies=pennies%5; // 0-4 left over pennies
}
```

Those two prints labelled *debug* are typical left-over testing lines, commented out instead of deleted until we're completely sure this is working correctly. With it all done, 63 for pennies, should give quarters=2, dimes=1, nickles=0 and pennies=3.

Here's a similar example, converting seconds to hours, minute & seconds like "00:05:32". It starts with the modulo trick:

```
public int seconds;
public int minutes, hours;

void Start() {
  hours=seconds/(60*60); // 60 times 60 seconds in an hour
  seconds=seconds%(60*60); // left-over seconds is 0 to 3599

  minutes=seconds/60;
  seconds=seconds%60; // 0-59 left-over seconds
}
```

Leaving seconds-per-hour as `(60*60)` is a common trick to make programs easier to read (3600 looks funny to me, but I can remember there are 60 seconds per minute and 60 minutes per hour.) It feels like you should pre-do as much math as possible, but the compiler does that for you. Trying to keep formulas readable is more important.

For more fun with ifs, we can try to print it as 00:00:00. We turn each number into a string, then add a `"0"` in front if needed. This would come at the end of `Start`:

```
  string wHours, wMinutes, wSecs; // will be things like "00" "45" "04"

  wHours=""+hours; // using the int-to-string trick
  if(hours<10) wHours="0"+wHours; // turn "9" into "09"

  wMinutes=""+minutes;
  if(minutes<10) wMinutes="0"+wMinutes;

  wSecs=""+seconds;
  if(seconds<10) wSecs="0"+wSecs;

  // combine, with colons:
  string wTime = wHours+":"+wMinutes+":"+wSecs;

  print("time left is "+wTime);
```

This would print 8 hours, 31 minutes and 4 seconds as 08:31:04.

7.5 Mixed and/or

Long expressions using AND and OR mixed-up can be difficult to read. Like everything else, they usually make sense when you know what they're for. But sometimes trying to figure that out can be a pain.

Suppose I like any color cats and white dogs. This `if` correctly checks that:

```
// any cat, white dog
if(a=="cat" || a=="dog" && color="white") print("good pet!");
```

It works because `&&` has higher precedence than `||`. In other words, the computer puts invisible parens around the "dog and white" part. It's like `4+3*5`. You could put parens around them yourself, if you think that looks better.

Here's a longer one that likes cats, white dogs, goats or red skunks. For real, I might add parens on white dog and red skunk, and break it over two lines, as in the second one:

```
// any cat, white dog, goat, red skunk
if(a=="cat" || a=="dog" && col=="white" || a=="goat || a=="skunk && col=="red")

//maybe nicer way?:
if(a=="cat" || (a=="dog" && color=="white") ||
   a=="goat" || (a=="skunk" && color=="red") )
```

Back to just dogs and cats, if you wanted white no matter what, and a dog or cat, you could write it like this, with parens around "cat or dog":

```
if((a=="cat" || a=="dog") && color=="white")
```

We need the parens. Without them, it would be a cat or a white dog.

Some longer ones; this is "white or black" and "cat or dog" (so white dogs, white cats, black dogs and black cats). The parens around the OR groups are required:

```
if( ( (color=="white" || color=="black") && (a=="cat" || a=="dog") )
```

For fun, if we forgot the parens, we'd get something completely different:

```
if( ( color=="white" || color=="black" && a=="cat" || a=="dog" )
// what the computer sees:
if( ( color=="white" || (color=="black" && a=="cat") || a=="dog" )
```

That would accept lots of things it shouldn't: anything white and any color dog.

7.5.1 not, DeMorgan's law

This is a little tricky, and not vital. I just think you should see it. It's about tricks using not (`!`).

The not symbol (`!`) flips between true and false. For example `if(!(x>0))` is "if `x` is not greater than 0." It counts as a real math symbol, so it goes inside the official `if` parens, with all the other math. It has higher precedence than almost everything, so you usually have to use parens with it.

It can often make things easier to write. For example, suppose I want to check if `x` is *not* between 10 and 20. Here are two ways to write that:

```
if( x<10 || x>20) print("not 10-20"); // old "too small or too big" way

if( !(x>=10 && x<=20) ) print("not 10-20");
```

The second one is longer, but I'm used to seeing "between 10 and 20", and the `!`-bang means not, so I can directly read it as "not 10 to 20."

Suppose I *don't* want a white dog or cat. Writing it directly is tricky (I'll do it later.) Writing "white dog-or-cat" and not-ing it might be easier:

```
// anything but a white cat or white dog:
if(( !( (a=="cat" || a=="dog") && color=="white") )
```

The inside is "either a cat or a dog, has to be white," which is a white cat or dog. Then the `!` in front make it the opposite. This might look terrible to you now, but once you get more practice using `if`'s it starts being a neat trick.

In general, the trick is: whenever you're writing the inside of an `if`, see if writing the opposite might be easier. If it is, write that, and put a `!` in front of it.

You can also use the `else`. This looks silly, but it works:

```
// very silly "anything but a white cat or white dog":
if( (a=="cat" || a=="dog") && color=="white" ) {} // do nothing
else print("not white cat or dog");
```

For fun, here's a direct "not a white cat or dog." Notice the `||`'s and `!=`'s:

```
if( a!="cat" && a!="dog" || color!="white")
```

The official rule for bringing a *not* through an AND/OR grouping is **DeMorgan's Law**. In algebra, you can take `a(b+c)` and distribute the `a` to get `ab+ac`. DeMorgan's law says you can do something similar with true/false and *not*. Except it's more complicated. You have to `!`(not) each part *and* flip the AND/OR. Here's an example:

```
// not 1 to 10, converted using DeMorgan's:
if( ! (a>=1 && a<=10) )

// bring the not through:
if( !(a>=1) || !(a<=10) ) // && flipped to an ||

// turn "not >=" into <; turn "not <=" into >:
if( a<1 || a>10)
```

Here's an example with `||`, showing that it flips to `&&`. The inside part is "5, 9 or 16" and then I not it, to say "anything except 5, 9 and 16", then I work out it using DeMorgan's:

```
// not 5, 9 or 16:
if( !( a==5 || a==9 || a==16 ) )

// Convert using DeMorgan's. Bring the not through, flipping || to &&:
if( !(a==5) && !(a==9) && !(a==16) )

// ! == is !=:
if( a!=5 && a!=9 && a!=16) print("a is not 5, 9 or 16");
```

We can test this one – the `&&`'s really are the correct symbol (`||` would always be true – 5 isn't 9 or 16.)

The most useful things to remember about this are 1) slapping a big `!()` around everything is fine, but moving the `!`s into the parts is extra tricky, 2) if you really, really need to convert true/false formulas using `!`s, there's an equation for it. If you only remember "French pirate sounding name," that should be enough for you to find it years from now.

7.6 Switch Statements

There's a limited version of an `if` that isn't useful for anything, and is kind of complicated to learn. But it's semi-common, so you should at least see it.

Here's an example `if` and then the replacement `switch`

```
if(n==5) { a="cow"; }
else if(n==12) { a="dog"; }
else if(n==19) { a="goat"; }
// and so on
```

A `switch` statement to do the same thing looks like this:

```
switch(n) {
```

```
  case 5: a="cow";
    break;
  case 12: a="dog";
    break;
  case 19: a="goat";
}
```

It looks so strange that you figure it must run faster, or be better in some way, but they aren't. In the old days, we tried a lot of crazy ways to write computer languages. If you're curious, reading about a `switch` is like going into computer language history.