Chapter 17

booleans

So far, our types are `int`, `string` and `float`. This whole section is adding a new one called **Boolean** for true/false variables (boolean is the math word for "true/false math.") They're really very simple, and we could have seen them back in the `if` chapters.

I waited this long since we didn't really need them so far. Plus, now that we know lots of other things, we can go back and see more tricks using `bool`s. For example, functions returning `bool`s, or using them to untangle some messy nested `if`s.

17.1 Bool rules

The official name for the type is `bool`. Like `int`, it's all lower-case. Since it's a **type**, you can declare variables with it: `bool a;`.

`bool` literals are `true` and `false`. Those are built-in computer words, like numbers, meaning you can say `a=true;`. Those are all there are, which is sort of a new thing. An `int` can be any of a billion different numbers, `string`s can be any word in the dictionary, plus lots more. `bool`s can only ever be `true` or `false`. Exs:

```
bool a;  a = true;
bool b = false;
a = b; // now a is false
```

It takes a little to get used to `true` and `false`. They aren't strings and they aren't variables. If it helps, `true` is stored as a 1 and `false` as a 0. But they officially count as type `bool`. For fun, some more examples, mostly errors:

```
string w; w="true"; // legal. Not a bool, just another string
w = true; // ERROR -- cannot assign a bool to a string
bool b; b="true"; // ERROR -- "true" is a string
```

```
int true; // ERROR -- "unexpected symbol."
// can't use a built-in C# word for an identifier

int n; n=false; // ERROR. false cannot be converted to an int
// even though false is really 0, it's still a bool

bool f; f=1; // ERROR. 1 cannot be converted to a bool
// even though true is 1, 1 counts as an int
```

The "math" you can do to a `bool` should be familiar: you can use `&&` and `||` in the exact way we did before. Can also use **not**(`!`) (explanation point) to flip true/false(this is also the same as before.) Some examples that do nothing useful except show rules:

```
bool a = true || false; // true
bool b; b = true && false; // false
bool c; c = !false; // c is true

a = b || c; // true (false||true is true)
a = b && c; // false (false&&true is false)
a = !b; // true (!false is true)
```

These look funny for two reasons: 1) we never used these outside of an `if` before, and 2) we always got true/false using compares (ex: `x==0`, `y>0`.) But, the computer doesn't care. Where-ever math is, the computer solves it. If something is `false` because we computed it, or because we just typed in `false`, that's all the same to the computer.

The answers to comparisons are `bools`, which means you can assign them. These set `bool` variables based on compares (they still don't do anything useful):

```
bool isPositive; isPositive = n>0;
bool is1to10= n>=1 && n<=10;
```

Those are easy if you cover up the first parts. We've seen `n>0` and `n>=1 && n<=10` plenty of times. We know how to convert them into true/false. If it helps, draw a little `if( )` in pencil around them.

If `n` is 12, `isPositive=n>0;` sets it to true.

These next two use `==` compares, which looks extra weird mixed with assignment `=`. The trick is the same: figure the true/false answer, then assign:

```
bool isCat; isCat= w=="cat";
bool isLucky= n==14 || n==77 || n=123;
```

If `w` is `"duck"`, then `isCat` will be false.

17.1.1 bool variables in `if` statements

Before, I said the rule for the test in an **if** was it had to be something true or false. The actual rule is it has to be a **bool**. Things like `n>0` or `x==1 || x==6` are officially `boolean expressions` and give a `bool` result.

But `if`'s don't especially like fancy expressions. Like everything else, you can give them an actual `bool` value (`true` or `false`,) or a `bool` variable, or a `bool` expression (what we've been doing.)

The simplest examples are using literals. These are useless, but they are legal:

```
if(true) print( "I always get printed" );

if(false) print( "I never get printed" );
```

These are confusing because they're so simple. The first one is like `if(1==1)`, but even simpler. The second is like `if(1>2)`, but even simpler.

A more useful trick is using `bool` variables inside `if`s. Some examples using a `bool` named `isReady`:

```
bool isReady; // pretend someone sets this

if( isReady ) print("ready!"); // most common use

if( isReady ) print("ready!"); // it also works with an else
else print("waiting...");

if( isReady == true ) print("B"); // longer version

// these are the same:
if( isReady==false ) print("waiting...");
if( !isReady ) print("waiting...");
```

The first one `if(isReady)` is the most common way to use bools and ifs. The `if` runs when `isReady` is true. The if-else below is nothing special. I just wanted to show that any kind of `if` can use a **bool** variable in the test.

The line with "B", `if(isReady==true)` is the same thing, and is also legal. It just works out that the `==true` isn't needed. Most people leave it out. For example, `if((n>0)==true)` is legal, but not needed. Use `==true` whenever you think it looks nicer.

The last two lines are the same – they run when `isReady` is false. `if(isReady==false)` should be obvious. `if(!isReady)` is a little more common. Most people read it as "if not isReady". Sometimes `if(b==false)` looks nicer, and sometimes `if(!b)` does. I use both about equally.

`bool`s with `&&` and `||` are nothing special. Assume we have `bool a,b,c;`. Here are some legal examples using and/or combinations:

```
if( a && b ) print( "A and B are both true" );
if( a==true && b==true ) print( "same as above" );

if( a && !b ) print( "A is true and B is false" );
if( a && b==false) print( "same as above" );
if( a==true && b==false ) print( "same as above" );

if( a || b ) print( "A, or B or both are true" );

if( a || !b ) print( "A true, or false B, or both" );
if( a==true || b==false ) print( "same as above" );

if( a || b&&c ) print( "tricky, but legal" );
if( a==true || (b==true && c==true) ) print( "same as above" );
```

We can also mix `bool` variables and regular compares:

```
if( isReady && n==0 ) print( "please select an action" );
if( !isReady || n<0) print( "waiting" );
```

17.2 Common bool tricks

17.2.1 Pre-computing if tests

One use for `bool`s is to compute and save an if-test, or part of one, then use it later. Sometimes that saves you from computing the same thing twice, and sometimes it makes the `if` easier to read.

Here's an example of an `if` spread out over a few lines:

```
// original 1-line version:
if( n>=100 ) print( "n is big" );

// same thing, in 2 steps:
bool big= n>=100;
if(big) print( "n is big" );
```

This is the same idea as using a temporary variable for math. The second version pre-computes `n>=100` and saves it. We couldn't save a true/false before, but now we can. Then `if(big)` uses the saved value. A longer example of the same thing:

```
bool oneDigit= n>=-9 && n<=9;
if(oneDigit) print( "n is a 1-digit number" );
```

The trick works with parts of if-tests. In this example, there are a lot of ways to win, and I use a temporary `bool` to save each one:

```
bool hasDoubles = n==33 || n==44 || n==55 || n==77;
bool zone1 = n>=150 && n<=170;
bool bigEnough = n>=1500;

if(hasDouble || zone1 || bigEnough) print( "winner" );
```

If we tried to write this as one big `if`, we'd have to spread it across lines anyway. This way we get to use helpful variable names, making it easier to read.

Precomputing tests can simplify my "which xy quadrant" example from back in the **if** chapter. Here's a nicer rewrite using `bool` variables (remember, positive x & y is quadrant 1, and so on):

```
bool xIsPos = x>=0, yIsPos = y>=0; // are x/y positive?

int quad=-1; // no quad, just in case
if(yIsPos) { // top part:
  if(xIsPos) quad=1;
  else quad=2;
}
else { // bottom part:
  if(xIsPos) quad=4;
  else quad=3;
}
```

This used `xIsPos` twice, so we saved some typing there. It only used `yIsPos` once, but I think it still looks nice, computing both together on top.

Here's a very long example using `bool`s to precompute possible answers. We're making a game that rolls three dice, and you get points for things like 3-of-a-kind, a pair, a straight (like 234 or 345,) or a wrap-around straight (561 or 612). The first part rolls the dice and sorts them. Sorting three numbers has nothing to do with `bool`s, but is fun:

```
int a=Random.Range(1,7), b=Random.Range(1,7), c=Random.Range(1,7);
// uses the int-only version. rolls 1,2,3,4,5 or 6

// Sort them: very sneaky way:
if(a>b) { int temp=a; a=b; b=temp; }
if(b>c) { int temp=b; b=c; c=temp; }
if(a>b) { int temp=a; a=b; b=temp; } // same as 1st if
```

Sorting like this is very, very sneaky, but it's a neat example of `if`s. If you want, try it with combinations of `a,b,c=` 321, 213, 312 ... . But all you have to know for the rest is that this really does sort them.

Now for the interesting part. I'll use a `bool` to hold each "fact." I don't want to worry right now about how many points a pair or straight is worth. For this part, I just want to compute & save each fact:

```
bool triples= a==b && b==c;
bool pair= triples==false && (a==b || b==c);
bool straight= b==a+1 && c==b+1;
bool wrapStraight= a==1 && b==5 && c==6 || a==1 && b==2 && c==6;
```

I think that looks really nice: the left side with `triples`, `pair` ... makes it easy to see we're computing "do we have any of these special rolls?" And having each one on a line by itself, after the name, makes it easy to see which math is computing which thing (imagine instead all that math was in a very long `if`. Ugg.)

The math is also somewhat interesting, because they're sorted. Some comments: `triples` doesn't need to compare `a` and `c`. `pair`, can also skip comparing `a` and `c` (we can't have 323.) `pair` also checks `triples==false` so we don't double-count 333 as a triple and pair.

`straight` seems obvious, but I still had to check it three times to be sure it said "`b` is one more than `a`, `c` is one more than `b`." For `wrapStraight`, I just listed both possibilities (561 or 612.) Cheesy, but not too long and easy to read.

To make this next part really interesting, pretend there are three versions of the game where different rolls give different points (I just made up some rules and what amounts you win.) This checks the game version, then uses the `bool`s to figure results:

```
int winChips=0; // how many chips we win from this roll

if(version==1) { // big payout on triples, everything else is nothing:
  if(triples) winChips=50;
}
else if(version==2) { // only triples and pairs pay out:
  if(triples) winChips=30;
  if(pair) winChips=8;
}
else if(version==3) { // trying for straights, get a little for pairs/triples:
  if(straight) winChips=25;
  if(wrapStraight) winChips=20; // not quite as good as a real straight
  if(triples || pair) winChips=5;
}
```

Because we computed `triples` and so on ahead of time, and gave them nice names, the lines in this part are easy to read. `if(triples) winChips=30;` says exactly what it does.

For real, I only do this for long, ugly `if`s. This dice game is an example of when I would, but probably nothing simpler.

17.2.2 bool return values

Now that `bool` is a type, functions can return them. Their main use is for putting a function inside of an `if`. Suppose there's some long compare that we use a lot – why wouldn't we want to write a function for it?

A good one is checking whether two numbers are close together. We already know how to do that with `if`'s. We only need to wrap a function around them. I'll name it `isCloseEnough`:

```
bool isCloseEnough(float a, float b) { // <- bool return
  float diff = b-a;
  if(diff<-0.01f || diff>0.01f) return false;
  else return true;
}
```

As a simple use, `b=isCloseEnough(5,9);` makes `b` false. But the best uses are inside of an `if`:

```
if( isCloseEnough(x1, x2) )
  print("objects are lined up!");
```

The inside of the `if` jumps to the function, runs it, and returns true if they were close together. It's a normal function call. Then the `if` continues. There aren't any new rules here, but it's certainly a new trick.

Notice how it ends with two close-parens. I left some space, but most people smush them together.

Two more sample uses:

```
if( isCloseEnough(x,0) )
  print("x is centered");

if( isCloseEnough(x,0) && isCloseEnough(y,0) )
  print("almost at (0,0)");
```

A fun trick is to add `print("comparing "+a+" and "+b);` as the first line in `isCloseEnough`. In the last example, you'd see it print maybe "comparing 1 and 0" then "comparing 2 and 0."

You could also use `if(isCloseEnough(x,0)==true)`, but don't need to. If you want to check whether two things aren't close, can use `if( isCloseEnough(n1,n2) == false)` or `if(!isCloseEnough(n1,n2) )`.

`bool` functions aren't limited to standard math. A really nice use of a `bool`-returning function is to simplify some of our personal tests.

Suppose we're using 0-6 for day of the week, where Sunday is 0. Here are some functions that "know" the days, and use them:

```
bool isWeekend(int dayNum) { return dayNum==0 || dayNum==6; }
bool isWeekday(int dayNum) { return dayNum>=1 && dayNum<=5; }
bool isPartyday(int dayNum) { return dayNum>=5; }
```

Even though these are short, you can see how nice this trick is by comparing with and without a function:

```
if(d1==0 || d1==6) { dogHours=2; } // what??
if(isWeekend(d1)) { dogHours=2; } // ah ... checking for weekend
```

In the second one, it's really obvious that you're checking for the weekend. The first is also easier to write the wrong way – maybe you forget whether Sunday is 0 or 1.

You can also assign a `bool` function to a variable. It might be helpful if you reuse that value a lot:

```
bool xIs5 = isCloseEnough(x, 5.0f); // we use this twice

if( xIs5 && y<5 ) { ... }
if( xIs5==false && y>10) { ... }
```

17.2.3 Bools for function settings

A `bool` parameter can be used to add an option to a function. For example, my very old `resetToLeft` function might have an option to give `y` a random value:

```
void resetToLeft(bool randomY) {
  x=-7; xSpd=0; ySpd=0;  // the usual stuff

  if(randomY) y=Random.Range(-3.0f, 3.0f);
  else y=0;

  transform.position = new Vector3(x,y,0);
}
```

`resetToLeft(false);` has `y` centered, at 0. `resetToLeft(true);` tells it to roll a random value for `y`.

Suppose we want to randomize `y` when we have 10 or more laps, we could call `resetToLeft(laps>=10);` (yikes! Cover up everything except the inside of the parens, solve it, then call the function with the result).

We often use this with default parameters. If we rarely randomize we'd write `void resetToLeft(bool randomY=false)`. That lets us use just `resetToLeft();` most places, with a true stuck in for the few times we need to randomize.

17.2.4 flags

Like any other variable, `bool`s can be created to remember things. They can't count, but they are good for remembering "did that happen yet?" things. We usually call that a **flag**.

In this example, I want to take the wrap-around moving Cube code, and make it "bend" halfway across by rolling a random y-speed. My first attempt has a bug: when we get halfway across, it rerolls `ySpd`, but does it every step, making us twitch as we move:

```
// standard Cube wrap-around code, with midway bend (works wrong)
float x=-7, y=0;
float ySpd=0;

void Update() {
  x+=0.1f; y+=ySpd;

  if(x>0) { // midway bend:
    ySpd=Random.Range(-0.06f, 0.06f);
  }

  if(x>7) { // standard wrap-around:
    x=-7; y=0; ySpd=0;
  }
  transform.position = new Vector3(x,y,0);
}
```

The problem is, it rolls a random speed for `y` when it passes midway, at 0, But during the next move, it doesn't remember that it's already done it, so does it again, and again.

We can use an extra `bool` to remember we've already done it. I'll name it `bent`. False means we haven't bent yet, true means we have:

```
// Working Cube wrap-around code, with midway bend
float x=-7, y=0;
float ySpd=0;

public bool bent = false; // <- new
// public, in case we want to watch in the Inspector

void Update() {
  x+=0.1f; y+=ySpd;

  if(x>0 && bent==false) { // <- new
    ySpd=Random.Range(-0.06f, 0.06f);
    bent=true; // <- new
```

```
  }

  if(x>7) {
    x=-7; y=0; ySpd=0;
    bent=false; // <- new
  }
  transform.position = new Vector3(x,y,0);
}
```

The first `if` says "if we're past the middle and haven't bent yet." Then inside, `bent=true;` makes sure we don't do it again. When we restart a lap, we also reset `bent` back to false or this round.

When someone says "use a flag," this is all they're saying – declare an extra `bool` variable. Setting it true or false is called setting or resetting the flag. The term comes from mailbox flags, which can be up or down.

17.2.5 Incremental bool calculations

Sometimes it's nice to compute a `bool` a bit little at a time. That seems funny – normal variables can get bigger, but bool's are just true/false. But the a-little-at-a-time trick is so good it even works for bools.

Here's a somewhat silly example of a game with three ways to win. I compute `win` in three steps, then check it:

```
bool win=false; // so far, but we're not done...

// check all three win conditions (we win if any of these happen):
if(n==7) win=true;
if(n>20) { // some pretend fake extra math to compute this one:
  int frogs=toads+tadpoles;
  if(frogs<=2) win=true;
}
if(toads>=10 && tadpoles==0) win=true;

if(win) {
  print("winner");
  print("pretend we do other win stuff here");
}
```

This is really un-rolling a 3-part OR: it starts false, if none of the `if`'s happen it stays false. If one or more fire, `win` will be true. It's similar to computing `win1`, `win2` and `win3` and then using `win=win1||win2||win3`.

Here's another very made-up example that checks a little bit at a time whether we can go to the movies. If we have homework, or it's a weekday or if we went recently, we're stuck at home:

```
bool stuckAtHome=false;

if(homework>0) stuckAtHome=true;
if(isWeekday(dd))  stuckAtHome=true;
int daysSinceLastMovie=dayNow-lastMovieDay;
if(daysSinceLastMovie<4) stuckAtHome=true;

if(!stuckAtHome) print("Here's $20. Have fun at the movies" );
else print( "where are you going, young lady?" );
```

Or we could flip that, using a `canGoToMovies` variable: start true and any problems make it false:

```
bool canGoToMovies=true; // so far

if(homework>0) canGoToMovies=false;
if(isWeekday(dd))  canGoToMovies=false;
int daysSinceLastMovie=dayNow-lastMovieDay;
if(daysSinceLastMovie<4) canGoToMovies=false;

if(canGoToMovies) ...
```

This exact trick isn't so important. It's good to know when you have a giant `if`, there are options for how to break it into simpler parts.

17.3 Testing tricks

Sometimes you want to temporarily enable or disable an `if`. Putting a `true` or `false` is a cheap way to do that. For example, you're testing a mini-game and want to disable a 10 second time-out:

```
//if(totalSecs>10.0f) { // original line
if(false) { // testing temp line. Give me all the time I need
  // time-out code. Never runs for testing
}
```

You can also use `if(true)` as a placeholder when you're writing the program. Suppose you have plans for being out-of-ammo (but aren't even close to writing it yet.) You might do this in the firing code:

```
if(true) { // add ammo check here, later
  // make bullet
  // aim bullet, etc ...
}
else {} // add out-of-ammo stuff
```

All that does is give you a code-reminder, and make it a little easier to add the real ammo-test later. I only use this trick when I'm sure I'll be adding something, and I know this is where it should go, and I'm pretty sure I'll forget it later.

Pulling out a test can help with debugging. Suppose we have `if(x*y<z/q)` (I just made up some random, complicated-looking equation) and it's working funny. We could rewrite it like this:

```
bool check1 = x*y<z/q;
if(check1) { doComplicatedThing }
```

No change, so far, but now we can test by adding a new line, all by itself:

```
bool check1 = x*y<z/q;
check1 = x<10; //  TESTING, temporarily change the condition
if(check1) { doComplicatedThing }
```

Simple debugging involves mucking around inside of a line, then having to remember to change it back. This way, we still might forget to change it back, but when we do it's one whole line to remove. Way less error-prone.