Chapter 14

Functions, part III - return

14.1 Introduction

One way to divide functions is into those that *do* something. and others that *compute* something. So far, we can write "do something" functions just fine, but the *compute something* ones have serious problems, which we need one new rule to fix.

"Do something" functions are things like `resetToLeft();` and `setGrayScale(0.5f);`. We use them to change something for us – usually some of our global variables (Color and position are pretty much global variables added by Unity3D.) They look and act like new commands, and seem fine the way they are.

`showAsWord(3);` is an example of a function that *computes* something. It turns 0,1,2 ... into "zero", "one", "two" ... which is great. But then it prints the answer to the ugly Console, which is terrible. It should have some way to tell us the answer, letting us use it how we want.

The plan is to steal from seventh grade math. Suppose we have this school problem:

```
define  f(x) = 2x+1

solve  f(5) + 9 = ______
```

We know the rule is to compute f(5) and replace it with the 11. Let me slow down since this is so obvious you can miss the rule: we jump up to `f` and work out f(5) is 2 times 5 plus 1, which is 11. Then we come back to the bottom line, replace f(5) with 11 and keep going (11+9 is twenty.)

The rule we're going to steal is that functions have one answer, which replaces then in the equation.

To start off, here's the old ice, water or steam function. I'm just changing it so it puts the answer in `w` before printing it:

```
void printWaterState(float tempInDegs) {
  string w;
  if(tempInDegs<=32) w="ice";
  else if(tempInDegs<212) w="water";
  else w="steam";

  print( w ); // ick. This is not a good way to give an answer
}
```

14.2 return values

Now we're ready for the rule that will let a function just tell us the answer. We'll be able to rewrite it like `string s; s=waterState(31);` and `s` will be `"ice"`. Then we can print it or whatever we want.

The technical term for a function's official answer is the **return value**. The things we need to add them:

- A way for the function to say it always has an answer.
- A way for it to say it never has an answer.
- A way to say what **type** the answer will be.
- A way for the function to say "this exact value is my answer."

The rule for the first three things on that list is the **return type**. It goes in front of the function name. `void` means the function has no answer (which is what we've been doing all this time) otherwise write any **type** in front of the name, and it means the function has an answer of that type.

For `waterState`, we replace `void` with `string` to say it will give us an official string answer

```
string waterState(int tempInDegrees) {
```

It might seem odd to put the word for the answer in front. The way I remember it is by thinking of `s=waterState(31);`. The answer goes into `s`, which is in front.

The second rule is the **return statement**. Anywhere inside the function, usually at the end, write `return` and whatever you want the answer to be. `return` tells the function to pop back, the usual way. Whatever you write after `return` is automatically the answer. Here's the complete new `waterState`:

```
string waterState(float tempInDegs) {
  string w;
  if(tempInDegs<=32) w="ice";
```

```
  else if(tempInDegs<212) w="water";
  else w="steam";

  return w;  // instead of printing it
}
```

The ending `print(w)` was replaced by `return w;`.

When we run `s=waterState(31);` it's still a function call. The usual things happen: we mark the spot, jump to `waterState`, run it, pop back.

But because of the `return`, the answer is `"ice"`. Our functions didn't have answers before. After running the function, the line turns into `s="ice";`.

Before showing the various ways we can and can't use value-returning functions, I want to give some more examples. My old `showAsWord` is better returning the word instead of printing it:

```
string intToString(int n) { // <- string return value
  string numWord="unknown";
  if(n==0) numWord="zero";
  if(n==1) numWord="one";
  if(n==2) numWord="two";
  // ... until 9
  return numWord;
}
```

`string ss=intToString(1);` makes `ss` into `"one";`.

I named my answer variable `numWord` just to be different. It's a regular local variable, so you can name it whatever you want.

The 90/80/70/60/50 letter grade `if` is good as compute-only function, since it has one letter answer:

```
string letterGrade(int n) {
  string gr;
  if(n>=90) gr="A";
  else if(n>=80) gr="B";
  else if(n>=70) gr="C";
  else if(n>=60) gr="D";
  else gr="F";

  return gr;
}
```

We could write `grade1 = letterGrade(score1);`.

We think of these as "pure" functions – basically like math. They get all input from parameters, and don't change anything in the program. They just give the answer back and let the main program use it however it likes. These are considered the best type of function (more on that later.)

A fun thing is to write out some pure math functions – things we already know, or seem like they might be in a calculator. These are built-in to many languages, but seeing how they work de-mystifies them:

Finding the largest of two numbers is usually called `max`. This function is just the same `if/else` from a few chapters ago, with a `return`:

```
float max(float a, float b) { // <- answer will be a float
  float ans;
  if(a>b) ans=a;
  else ans=b;
  return ans; // <- no printing. return the answer
}
```

If we have `float q=max(3,8);`, then `q` will be 8.

This is a little new, since the return-type is a float. The return type can be any type we know about.

It takes two inputs, which is fine. If you have an idea for a function that needs 6 inputs and has one output, that's also legal.

Absolute value (`abs`) is even shorter, but it's still usually written as a function:

```
float abs(float a) {
  float ans=a;
  if(a<0) ans=a*-1;
  return ans;
}
```

`abs(6);` is 6 and `abs(-12)` is 12.

Just to be sure, this barely does anything. If `a` is 0 or more, the answer is just `a`. All it does is flip negative numbers. But, like previous functions, it's still shorter than writing out the `if` every time.

14.3 Using value-returning functions

Like I wrote in the introduction, we think of do-something and compute-something functions in different ways.

A function that does something will always be on a line by itself. Obviously, functions that compute something are no good by themselves – it would be like a line with just `4+3;`

Compute-something functions work like equations, so have to be part of a longer line. We also *like* how they do nothing by themselves – it's part of the idea "compute first, use later". For example:

```
string cw = intToString(2); // calculate the word for 2
// now we can use it however we want:
print( "There are " + cw + " cows.");
```

Even though we plan to print `"two"`, we prefer computing it first, so we can combine it with the rest of our sentence.

Or in this next example I really take advantage of being able to just calculate a letter grade without printing it yet. If the two letters are the same, I just print "A pair of":

```
string gw1=letterGrade(g1), gw2=letterGrade(g2);
if(gw1==gw2) print("A pair of "+gw1+"'s");
else print(gw1+" and "+gw2);
```

Functions returning numbers feel even more like equations. This uses `max` to figure out if anyone is old enough to rent a car:

```
int oldest = max(amyAge, bradAge);
if(oldest>=18) print("Can rent a car");
```

So far, I've been assigning function calls to variables. Nothing wrong with that, but you can use them inside math, as well.

We can use string-returning functions in string math:

```
print("Look, it's some " + waterState(10)); // Look, it's some ice

string w2 = waterState(100) +" world"; // water world
```

There aren't any special rules here. `waterState(10)` always becomes `"ice"`, no matter where it is.

It's more common to have `float` or `int` returning functions inside regular math. These aren't doing anything useful, but are legal:

```
print( max(3,7) + max(1,-6) + max(2,2) ); // 7+1+2 = 10

int n = abs(3) * abs(-5); // 15

n = max(10,2) * abs(-3) + abs(5); // 10*3 + 5 = 35
```

The computer has no problem using the same function, or multiple functions in the same equation. Each function call runs by itself, with it's own inputs, same as always.

Here's a picture of the steps in `max(10,2) * abs(-3) + abs(5)`:

```
n = max(10,2) * abs(-3) + abs(5);
// makes three function calls:
// n =  10    * abs(-3) + abs(5);
// n =  10    *    3   +  abs(5);
// n =  10    *    3   +    5;
```

To really check this out, we can add a `print` inside of `max`. That's a terrible idea for real, but good for testing:

```
float max( float a, float b ) {
  int ans;
  if(a>b) ans=a;  else ans=b;
  print("In max. a="+a+" b="+b+" answer="+ans); // testing!
  return ans;
}
```

Now we can run a line with two `max` function calls and watch them run:

```
print( max(2,6) + max(1,1) );
// In max. a=2 b=6 answer=6
// In max. a=1 b=1 answer=1
// 7
```

The output shows us how it really does jump to `max` twice, with those inputs.

A really interesting trick is using functions inside of an `if`. It's just another use of the "anything that counts as" rule. Some examples:

```
if( max(a,b)>10) ) print("both must be 10 or less");

if( waterState(temp) == "ice" ) print("can walk on it");

if( waterState(t1)=="ice" || waterState(t2)=="ice")
  print("at least one is ice");
```

The neat thing is, it's still not a new rule. Same as before, functions are called, run, and count as their answer. Then the line continues.

One trick to using a value-returning function is to forget how it works and just think about what it computes. For example, `max` turns into which ever one is larger. We know about all the details and how it really works, but when we use it, those are distractions – it turns into whichever is larger is all we need to know.

14.4 More math functions

Here are some more examples of math-like functions and some uses. There aren't any new rules in this section. Feel free to skip any examples you don't

like:

`sign` returns -1, 0 or +1. The idea is, it really says negative, positive or zero, but -1/0/+1 is the best way to say that: As usual, the inside is completely boring, except for the `return`:

```
int sign(float a) {
  int ans;
  if(a<0) ans=-1;
  else if(a>0) ans=+1;
  else ans=0;
  return ans;
}
```

For examples `sign(6)` is 1, `sign(-0.43f)` is -1. Here's a use for it (which is really, really tricky – it checks whether `num+change` goes towards `endNum`)
`if(sign(change) != sign(endNum-num)) print("wrong way");`

Squaring a number is super-easy, and just one line:

```
float square(float n) { return n*n; }
```

You might think writing `cow*cow` yourself is easier, but suppose you want `a+b` squared. Written out that's `(a+b)*(a+b)`. Or call `square(a+b)`. That's easier to read and less chance to make a mistake.

We aren't limited to real math. Suppose your magical manna points are your Intelligence stat or Wisdom stat, which-ever is higher, plus half of the other one, but at least 5. We can write that:

```
int mannaPoints(int intell, int wis) {
  int ans;
  if(intell>wis) ans=intell+wis/2;
  else ans=wis+intell/2;
  if(ans<5) ans=5;
  return ans;
}
```

Even though this is totally made up, it works the same as any other math function. `int myManna = mannaPoints(10,5);` gives 12. `mannaPoints(1,2);` is 5.

Making sure a number is between 1 and 10 (or any min/max) is common enough that there's usually a function written for it, named `clamp`. It's interesting because it takes 3 inputs, standing for different things. It's also one of the built-in Unity functions which tend to confuse people.

If you don't find that interesting, just skip ahead. Otherwise, here's `clamp`:

```
float clamp(int n, int min, int max) {
  float ans=n; // so far
  if(n<min) ans=min;
  else if(n>max) ans=max;
  return ans;
}
```

Of course, the inside is the same pair of range-fixing `if`s from way back, but that's the point. We put them in a function and gave them a nice name. It's usually used something like: `b=clamp(a,1,10)`. In our minds, that means "`b` is `a`, except forced to be 1-10."

Another interesting 3-input function, also often causing confusion in Unity, is `lerp`. Suppose we want to know what 10% of the way from 3 to 7 is. We'd figure 7-3 is 4, 10% of 4 is 0.4, so the answer is 3+0.4 = 3.4. The big word for that is linear interpolation, abbreviated as lerp (yes, lin-erp might be better, but everyone calls it lerp.)

Here's that same math written as a function. The last input is always given as a 0-1 percent – 0.25 if you want 25%:

```
float lerp(float start, float end, float pct) {
  float gap=end-start;
  float distFromStart = gap*pct;
  float ans = start+distFromStart;
  return ans;
}
```

Suppose I want to place a ball 1/4th of the way across my screen. The edges are -7 and 7, so I can use: `float xx=lerp(-7, 7, 0.25f);`. That reads as "from -7 to 7, 25% of the way." It would give us -3.5.

14.4.1 Extra return rules

I've been using one style for functions: make a variable for the answer, compute it in the middle of the function, and use a `return` at the end to send it back. You can also use `return` as soon as you know the answer.

The `return` statement jumps out of the function even if it's not at the end. The function quits right then – any lines after the `return` are skipped.

You're allowed to have as many `return` statements as you need, and they can have any kind of constant or equation after them. Here's `intTostring` rewritten using the "return as soon as you know it" method:

```
string intToString(int n) {
  if(n==0) return "zero";
  if(n==1) return "one";
  if(n==2) return "two";
```

```
  return "three";
}
```

That `return "three"` at the end looks mighty suspicious. There's no `else` in front of it, so it looks like the answer is always `"three"`. But the function works by abusing the `return` rules. If `n` is 0, the function quits right then with answer `"zero"`, and never gets past line one. It only gets to `return "three"` if the `if`s were all false.

Here's `max` rewritten to have return-abuse:

```
float max(float a, float b) {
  if(a>b) return a;
  else return b;
}
```

It doesn't need the `else`, but I thought it looked nice with it.

You can use the return-early trick for testing. This will temporarily make `intToString` always say NUMBER_WORD. It always runs the first line and quits:

```
// abusing return for testing:
string intToString(int n) {
  return "NUMBER_WORD"; // temporary for testing
  // this is all skipped:
  if(n==0) return "zero";
  if(n==1) return "one";
  if(n==2) return "two";
  return "three";
}
```

14.4.2 Optional empty return

If you have a function with no return type (it has `void` in front,) you're allowed to use just `return;` (with nothing after it) to jump back. You can put it at the end, for fun, and can also use return-from-middle.

Here's just a decorative end `return`. It's a little like writing The End:

```
void reset() {
  x=-7; y=0; speed=0.05f;
  return; // not needed, but looks pretty
}
```

Here's the early-return trick on a wrap-around function. It uses the `return` to quit and not wrap if it doesn't need to:

```
void wrapAround() {
  // assuming x is a global
  if(x<7) return; // not too big, so just quit
  x=-7;
}
```

You can even use return-from-middle in `Start` or `Update` – they're functions with `void` return values. Here's working move&wrap code with return-from-middle:

```
public float x=-7;

void Update() {
  x=x+0.1f;
  transform.position = new Vector3(x,0,0);

  if(x<7) return;
  x=-7;
}
```

Normally, `Update` quits when it gets to the end. Quitting early does the same thing, only faster. Either way, the "auto-run over and over" rule still happens.

Return-early can cause tricky bugs. You know I'm going to grow that Update by adding some color-change code to the end. That was fine before, but now won't work. The color will mysteriously only change on the wrap-around. Otherwise it will be like it doesn't feel like running it. Huh. Then finally, after wasting an hour testing, I'll see that stupid `return` and realize it really wasn't running it.

I was taught, for that reason, to never use return-early. But it's fine for short functions where it makes them easier to read.

14.4.3 `return` can use math

Sometimes it looks nicer to put equations after a `return`. Like anything else, the computer works them out first – `return 1+3;` is the same as `return 4;`.

Some examples. This `abs` use two returns, one of them using math:

```
float abs(float a) {
  if(a<0) return -1*a; // could also use -a
  return a;
}
```

This redo of `lerp` does the math in the return line. As usual, it does the math first, then the `return`:

```
float lerp(float start, float end, float pct) {
  float distBetween = end-start;
  return start + distBetween*pct;
}
```

This is a slightly fakey mad-lib sentence maker, using a giant string add in the `return`:

```
string walk1(string animal, string place) {
  string action="walk";
  if(animal=="fish" || animal=="dolphin") action="swim";
  return "My "+animal+" and I went for a "+action+" in the "+place;
}
```

14.4.4 Errors

If you say you're going to return something of a certain type, you have to. And it you say you won't return anything (by using `void`,) you can't. That seems pretty obvious – a function breaking that rule won't even make any sense. But there are some funny cases and seeing the error messages is helpful:

It's easy to write a function that usually returns something, but for some values it just reaches the end without ever hitting a `return`. For example this gives compiler error: *not all code paths return a value.* because of 10 through 100:

```
string badFunction(int num) {
  if(num>100) return "big";
  if(num<10) return "small";
  // return ""; // this fixes the error
}
```

It's a compile error – you can't even run the program. Maybe you just forgot. Or maybe you know the input can never be 10-100. The fix is adding a dummy return at the end. Sometimes I'll use something like `return "BAD VAL";`

This one's a little tricker. *We* can see it always `return`s something, but the computer can't tell. It incorrectly gives the *not all code paths return a value* error:

```
string sayNum2(int num) {
  if(num<0 || num>1) return "bad";
  if(num==0) return "zero";
  if(num==1) return "one";
}
```

Adding `return ""; // can't get here` as the last line will make the compiler happy.

For functions with no return value (`void`) it's easy to use `return 0;` by mistake (instead of simply `return;`.) Zero is nothing, but it's an integer that means nothing. It's not actually nothing. `return 0;` gives error: *A return keyword must not be followed by any expression when method returns void*:

```
void A(int n) {
  if(n>10) return 0; // ERROR, use "return;"
  print("something about n");
}
```

Here's the error the other way, with a return-value function trying to return nothing. It gives us error: *An object of a type convertible to int is required for the return statement*:

```
int B(int n) {
  if(n>10) return; // ERROR must put something after the return
  return -1;
}
```

This error makes sense because if you say you have an answer, you have to say what it is. If the inputs are junk and you just want to quit, use `return -1;` or something like that.

Returning the wrong type is also an error:

```
int C(int n) {
  if(n>10) return 1.5f; // ERROR float
  return -1;
}
```

It says *Constant value 1.5 cannot be converted to a int.* It's a little different wording than we're use to, and doesn't mention the `return`, but we can double-click to the line and figure it out.

Trying to use a `void` function inside of math gives an error. This function prints a string, but it has no official answer, so trying to assign from it won't work:

```
void AA() { print("mooo"); } // <- void == can't use in math

string w = AA(); // ERROR. AA() isn't anything
```

It says: *Cannot implicitly convert type void to int.* That's a little misleading, since `void` isn't a type – it's nothing at all – but it gives the idea.

The function we were trying to use is `string AA() { return "mooo"; }`.

Oddly, it's *not* an error to ignore a return value. This is legal, but useless:

```
max(3,15); // not an error. Answer is 15, but we don't use it for anything
```

This could happen if you meant to print it, but forgot to write the print part.