Chapter 18

Function examples

18.1 Introduction

This chapter is all function examples and tricks, plus some new Unity functions that let us do neat things.

One style for writing short functions is to change the input into the answer. It saves you having to make an extra variable, and often looks fine. Here's the `clamp` function written in that style (recall if forces the first input to be between the other two.) `n` is the input and the output (after we fix it):

```
float clamp(float n, float min, float max) {
  if(n>max) n=max;
  else if(n<min) n=min;
  return n;
}
```

On longer functions it can be too confusing. You can forget that `n` isn't the original input any more.

Another style is putting all the math on the return statement. This is popular for short functions:

```
// recall lerp(4, 7, 0.3) is 30% of the way from 4 to 7:
float lerp(float begin, float end, float pct) {
  return begin+(begin-end)*pct;
}

// using the fancy ?: value-returning if for max:
float max(int a, int b) { return a>b?a:b; }
```

"Front-end" functions often use that trick. This uses the real clamp to do the work:

```
float clamp1to10(float n) { return clamp(n,1,10); }
```

Putting it on one line like that makes it easy to see that all this function does is run the real `clamp` with 1 and 10 already fllled in.

This uses the old `bool closeEnough` function to check for a number close to zero:

```
bool almostZero(float n) { return closeEnough(n,0); }
// ex:
if( almostZero(milkLeft)) print("buy milk, ma.");
```

The same trick works on made-up non-mathy functions. Here's a storytelling function, then a shortcut with the name filled in:

```
// two-input story function:
string adventure1(string name, string animal) {
  string ans = "One day "+name+" took the "+animal+" for a walk on the beach.";
  return ans;
}
```

We run that a lot with the name `"you"`, so we make a shortcut:

```
// short-cut using adventure1:
string youAdventure1(string ani) { return adventure1("you", ani); }
```

`youAdventure("cow");` is you and a cow on the beach.

We often mix the tricks. This `closeEnough` has the equation on one line, and uses function `abs` (absolute value) to do the work:

```
 bool closeEnough(float a, float b) {
   return abs(b-a)<0.01f;
 }
```

In the old days, I would have computed `abs(b-a)` into a variable, then used an if/else it check it. `return abs(b-a)<0.01f;` does the same thing, but shorter.

You're allowed to nest a function call in a function call. This uses 2-input `max` to find the max of 3 numbers:

```
 float max3(float a, float b, float c) {
   return max( max(a,b), c);
 }
```

It runs "inside out." The inside `max` finds the largest out of `a` or `b`, then the outside `max` compares that to `c`.

18.2 Change me type functions

A lot of value returning functions look like they change you, but, of course, they can't. For example:

```
int cows=14;
clamp(cows,0,10);
print(cows); // 14 ??  why wasn't this clamped to 10?
```

The confusion is, in our minds, `clamp(cows,0,10)` forces `cows` between 0 and 10. But what it really does is compute that number and return it. Like every other function, we have to "catch" the result:

```
int cows=14;
int c2=clamp(cows,0,10);
print( cows + " " + c2 ); // 14 10
```

To force cows between 1 and 10, use `cows=clamp(cows,0,10);`. Likewise, to make `n` positive, use `n=abs(n);`.

18.3 Dice functions

Functions using random feel different, so I'm putting them in their own section. None of these are all that special. Just examples of how, if you use `Random.Range` a lot in a certain way, you can write a function for it.

Rolling a 50% chance isn't that long, but if we use it a lot, we could turn it into a function. A funny thing is it takes no inputs. `heads()` is true half of the time:

```
bool heads() { return Random.Range(0.0f, 1.0f)>0.5f; }
```

Now we can say `if(heads())`.

Rolling a percent, like a 20% chance, is also short but awkward. Here's a common function to make that easier. It uses 1 to 100, so you can write `chance(20)` for a 20% chance:

```
bool chance(float pct100) { return Random.Range(0.0f, 100.0)<=pct100; }
```

`if(chance(20))` is true 1 time in 5. If we have a game with lots of percent-based odds, this will be useful.

Sometimes we want get a random plus/minus value, for example -0.1 to 0.1, or -2 to 2. We can make a shortcut function for that:

```
float randPM(float plusMinus) {
  return Random.Range(-plusMinus, plusMinus);
}
```

If we want about 50, we can use `50+randPM(2);` to get 48 to 52. To have red to be about 0.4 use `0.4f+randPM(0.1f);`.

If we have a board game, we might roll two 6-sided dice a lot. The integer version of 1-6 is `Random.Range(1,7)` (it won't roll the highest, just because.) We can write a function to roll that twice and add:

```
float roll2d6() {
  return Random.Range(1,7) + Random.Range(1,7);
}
```

Now we can use `int move = roll2d6();`. If you need an 8 or more to beat the orc, use `if(roll2d6()>=8)`.

18.4 More style; types of functions

We often break up a main program into slop-tastic chunks like this:

```
void Update() {
  doAllMoves();
  updateColorChange();
  if(difficulty>1) updateMoveMonsters();
  updateCheckWinLose();
}

void doAllMoves() { ... }
```

It's pretty much understood that `doAllMoves` will be a sad excuse for a function. It reads and sets a bunch of globals. The others will be the same way. All they do is move code from out of Update. But that's helpful. It makes things easier to find. We can use the editor collapse-code-block feature to hide or show only the functions we need to see now.

We love functions that only read from their inputs. We can see everything they use right there on the line calling them. But sometimes, rarely, we make a function that has a setting. Here's a standard set color with a special option to not be too dark:

```
void setColor(float r, float g, float b, bool notTooDark=false) {
  if(notTooDark) { // none can be less than 0.3:
    if(r<0.3f) r=0.3f; if(g<0.3f) g=0.3f; if(b<0.3f) b=0.3f;
  }
  GetComponent<Renderer>().material.color=new Color(r,g,b);
}
```

`setColor(r1,g1,b1);` runs it normally (since the 4th uses a Default setting). Otherwise run it with `setColor(r1,g1,b1,true);`.

Another way to do that is using a global for the setting:

```
// change this to change how setColor works
bool colorCantGetDark=false;

void setColor(float r, float g, float b) {
  if(colorCantGetDark) {
    if(r<0.3f) r=0.3f; if(g<0.3f) g=0.3f; if(b<0.3f) b=0.3f;
  }
  GetComponent<Renderer>().material.color=new Color(r,g,b);
}
```

This can be nice. Suppose we can't be too dark for the next 10 seconds. Set `colorCantGetDark=true;`, then `false` after 10 seconds. That's way easier than needing it as a 4th input.

We rarely do things this way. It's easy to forget about the setting – it's more hidden than if it was an input. And if we need this function in a new program we also have to copy the global variable. But Unity has at least one thing that works like this (there's a global setting for raycasts).

Some functions mostly do something, but return a minor value, usually saying if it worked. For example, this function's job is to move and wrap-around. It returns true when it wraps:

```
bool moveAndWrap(float xMove) {
  // true means we had to wrap-around
  x+=xMove;
  if(x>7) { x=-7; return true;}
  if(x<-7) { x=7; return true; }
  return false; // didn't wrap around
}
```

We can use it by itself: `moveAndWrap(0.1f);`. We ignore the return value. Or we can catch the answer and act on it:

```
bool didALap = moveAndWrap(0.1f);
if(didALap) {
  lapCounter++;
  // do more lap stuff
}
```

The first line is as if we're calling `moveAndWrap(0.1f)` to do things for us. But then we're asking it "so, how did it go? Was there a lap?"

We often us it directly inside the `if`:

```
if(moveAndWrap(0.1f)) { // do lap stuff
```

Inside of the `if`, the program actually moves us. Functions inside of `if`'s really, really shouldn't also change things. But in this case it's fine. We understand it's a "do some stuff then tell me how it went" situation.

When we have a function that mostly computes, but also does something, we call those *side effects*. For example, this checks whether we're off the edge, but also adjusts us:

```
bool isOffEdge() {
  // assume x is a global for our position
  if(x>7) { x=7; return true;}
  if(x<-7) { x=-7; return true; }
  return false;
}
```

This is confusing, since it feels like an answer-only function:

```
if(isOffEdge()) { // arrg. This also may have moved us
  // check how far off we are:
  // opps! We're in-bounds now, since isOffEdge fixed us
  ...
}
```

But if we renamed it `bool forceIntoBounds` it would feel like a do-something function that happened to have a minor return value.

18.5 More Unity Movement examples

We know enough now to use some new Unity built-ins and make some more complex, fun movement. I'm going to barely skim some of the set-up – I'm assuming at this point you either played with Unity enough to figure it out, or you're happy just reading these examples.

We need 3 new things. The Rotate function spins us. This spins us 2 degrees each frame, like a top:

```
void Update() {
  transform.Rotate(0,2,0);
}
```

Rotate has 3 inputs. The other 2 rotate us like a summersault, or rolling sideways. We won't use them.

The Translate function moves us on our personal forward. Running this with a tilted Cube will go whichever way it's facing. The speed of 0.1 is about the same speed we were using before – takes 140 updates to cross from -7 to 7 at that speed:

```
void Update() {
  transform.Translate(0,0,0.1f);
}
```

We won't use the other 2 input slots. They move us sideways and up.

Combined, the two new functions can move us in a circle, spinning and moving forward:

```
void Update() {
  transform.Rotate(0,2,0);
  transform.Translate(0,0,0.1f);
}
```

The last new rule is asking the cube where it is. `transform.position.x`. gives us our current x position, which is probably -7 to 7. As a test, this snaps us to the center whenever `x` goes past 3:

```
void Update() {
  transform.Rotate(0,1,0); // slow spin for bigger circles
  transform.Translate(0,0,0.2f); // faster movement

  // checking our position:
  if(transform.position.x>4)
    transform.position = new Vector3(0,0,0); // snap to center
}
```

To see this we'll need to be in top view. We can put the camera 10 units over the center, at (0,10,0), Then tilted looking down with rotation (90,0,0). I put a little ball as a child, just in front of it, so I could see which way it was aimed.

Now we're ready to write some code. Whatever we do, we want to stay in bounds. We'll write a function to check for it:

```
bool outOfBounds() {
  float x=transform.position.x;
  float z=transform.position.z;
  if(x<-7 || x>7 || z<-5 || z>5) return true;
  return false;
}
```

If we slow down our spinning, we can rotate in circle big enough that we'll go out of bounds. The code below will spin us 180 degrees when that happens:

```
void Start() {
  // We start with a random spin:
  float ySpinAny=Random.Range(0,360);
  transform.Rotate(0, ySpinAny, 0);
}

void Update() {
  // spin slowly and move forward (same code as before):
  transform.Rotate(0,1,0);
  transform.Translate(0,0,0.1f);

  if(outOfBounds()) // turn around when out-of-bounds:
    transform.Rotate(0,180,0);
}
```

This is nice, but gives us boring, predictable curves after watching it for a while.

We can change it up with the usual tricks. Our turn speed can be a variable and we can add a little randomness to the 180 degree flip:

```
float spin=1; // our turn speed

void Start() {
  transform.Rotate(0, Random.Range(0,360), 0); // random facing
}

void Update() {
  transform.Rotate(0,spin,0);
  transform.Translate(0,0,0.1f);

  // 4% chance to change the spin:
  if(Random.Range(0.0f, 1.0f)<0.04f)
    spin=Random.Range(-1.0f, 1.0f);

  if(outOfBounds()) {
    // spin about 180 degrees, 150 to 210 degrees:
    float turnAround = Random.Range(180-30, 180+30);
    transform.Rotate(0,turnAround,0);
  }
}
```

This moves in semi-interesting curves, with fun bounces.

This has the same rare stuck-out-of-bounds bug as the old back-and-forth code. If it somehow gets too far out it will just spin in place. We can make an improved outOfBounds function to force it in. The name is changed so we remember it also can move us:

```
bool keepInBounds() {
  float x=transform.position.x;
  float y=transform.position.y;
  float z=transform.position.z;
  bool wasOut=false;
  if(x<-7) { x=-7; wasOut=true; }
  if(x>7) { x=7; wasOut=true; }
  if(z<-5) { z=-5; wasOut=true; }
  if(z>5) { z=5; wasOut=true; }

  // if it was out of bounds, move us to the fixed position:
  if(wasOut) transform.position=new Vector3(x,y,z);

  return wasOut;
}
```

I think that's a neat use of a bool variable.

We can try other things. If we move sideways and forward, we're going diagonally. If we use a counter we can move left 10 times, right 10 times, left 10 times in a zig-zag:

```
float zigSpd=+0.05f; // sideways movement. Will flips +/-
public int zigLen=14; // how long each zig lasts
int zigTimer; // counts up to zigLen as we move

void Start() {
  transform.Rotate(0, Random.Range(0,360), 0); // random facing
  zigTimer=zigLen/2; // start with 1/2 a zig
}

void Update() {
  // forward and left or right:
  transform.Translate(zigSpd,0,0.1f); // forward and L or R

  zigTimer--;
  if(zigTimer<0) {
    zigSpd*=-1; // flip left/right
    zigTimer=zigLen; // restart the count
  }

  if(keepInBounds()) {
    float turnAround = Random.Range(180-30, 180+30+1);
    transform.Rotate(0,turnAround,0);
    zigTimer=zigLen/2; // another 1/2-zig
  }
}
```

This works – it moves in short, straight lines making a zig-zag – but I just don't like the way it looks. My second try at zig-zagging is to always move forward, and change our facing. We'll snap 30 degrees left and right as we move:

```
public int zigLen=35; // how long each zig lasts
public float zigDegs=30.0f; // degrees of each zig
int zigTimer; // counts up to zigLen


void Start() {
  transform.Rotate(0, Random.Range(0,360), 0); // random facing
  zigTimer=zigLen/2; // start with 1/2 a zig
}

void Update() {
  transform.Translate(0,0,0.1f); // only move forward

  zigTimer--;
  if(zigTimer<0) {
    zigTimer=zigLen; // reset the count
    transform.Rotate(0,zigDegs,0); // do a zig
    zigDegs*=-1; // zig the other way, next time.
  }

  if(keepInBounds()) {
    float turnAround = Random.Range(180-30, 180+30+1);
    transform.Rotate(0,turnAround,0);
    zigTimer=zigLen/2; // another 1/2-zig
  }
}
```

18.6 Reading keys

We finally know enough rules to decipher a built-in keyboard reading function. This uses the A and D keys to move back-and-forth:

```
float x=0;

void Update() {
  if( Input.GetKey(KeyCode.A) ){
    x-=0.2f;
    if(x<-7) x=-7; // stay in bounds
  }
  if( Input.GetKey(KeyCode.D) ){
    x+=0.2f;
    if(x>7) x=7; // stay in bounds
  }

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

You can probably figure out that `Input.GetKey(KeyCode.A)` is true when A is pressed. Here are some notes, mostly obvious, about it:

- The name of the function is `GetKey`. It's in the namespace `Input`. So, altogether: `Input.GetKey`.
- It returns a `bool` – true means that key is down. Like any `bool` function, you can use it inside of an `if`.
- The input is which key you're checking for. `KeyCode` is just an enumerated type, specially made to give names to all the keys. For example, `KeyCode.DownArrow`.
- There's also an overload which takes a `string` as input. For example, `Input.GetKey("a")`.
- In general there are three ways to check keys: just pressed, currently being held, or just let go. In Unity, these are `GetKeyDown`, `GetKey` and `GetKeyUp`. Most sorts of keys or virtual buttons have those three things.

Here's a different version, which also moves back&forth, but using `GetKeyDown`. Tapping A or D starts you moving left or right. Tapping S stops you. I don't like this as much, but it's fun to program:

```
float x=0;
float spd=0;
// set to 0, +0.2 or -0.2 by keypresses, stays the way it was set

void Update() {
  if( Input.GetKeyDown(KeyCode.A) ) spd=-0.2f;
  if( Input.GetKeyDown(KeyCode.D) ) spd=0.2f;
  if( Input.GetKeyDown(KeyCode.S) ) spd=0;

   x+=spd;
   if(x>7) { x=7; spd=0; }
   if(x<-7) { x=-7; spd=0; }

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

We can tweak that so holding a key gradually speeds you up:

```
float x=0;
float spd=0; // changed by holding down a key
public float accel=0.005; // how much speed changes in one tick

void Update() {
  if( Input.GetKey(KeyCode.A) ) spd -= accel;
  else if( Input.GetKey(KeyCode.D) ) spd += accel;
  else {
    // if not holding a key, slow down, then stop:
    spd *= 0.98f; // slow down (0.98 is trial and error)
    if( spd<0.02f && spd>-0.02f) spd=0; // near 0 should stop now
  }

   x+=spd;
   if(x>7) { x=7; spd=0; } // kill our speed when we hit an edge
   if(x<-7) { x=-7; spd=0; }

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

The first two ifs don't limit the speed, but we'll hit an edge before we get too fast. The part where it slows down is an old movement trick. We want to make the speed go towards zero, whether it's positive or negative. Multiplying by 0.98 makes 1 and -1 both go towards zero. It's only 2% smaller, but it adds up fast. The last slowdown line says "when speed gets really small, just be 0."

For something completely different, we can have A and D rotate, while W moves forward:

```
void Update() {
  if( Input.GetKey(KeyCode.A) ) transform.Rotate(0,-2,0);
  else if( Input.GetKey(KeyCode.D) ) transform.Rotate(0,2,0);

  if( Input.GetKey(KeyCode.W) ) transform.Translate(0,0,0.1f);
}
```

We could change this version so the W key is acceleration instead of movement:

```
float spd=0; // forwards speed. always positive

void Update() {
  if( Input.GetKey(KeyCode.A) ) transform.Rotate(0,-2,0);
  else if( Input.GetKey(KeyCode.D) ) transform.Rotate(0,2,0);

 // speed up if W held, slow down if not:
  if( Input.GetKey(KeyCode.W) ) {
    spd+=0.01f;
    if(spd>0.14f) spd=0.14f; // maximum speed of 0.14
  }
  else { // slow down if W not held:
    spd-=0.005f;
    if(spd<0) spd=0;
  }
  transform.Translate(0,0,spd);
}
```

Moving onto another completely different movement, we want the Cube to crawls around the sides of the screen. A moves clockwise and D moves counter clockwise.

The basic plan is using a 0-3 `int` to remember which side we're on. I'll use an enumerated type, written specially for this:

```
enum Wall {top, right, bottom, left}; // Wall vars can be 0,1,2,3
Wall wall = Wall.bottom; // sample use
```

`Wall wall;` looks funny but it's common. The type is `Wall` and the variable is lower-case `wall` since I couldn't think of a better name.

`wall` will act as a state-variable. Each wall handles movement on itself, passing you to the next wall when you hit an edge:

```
float x=0, z=-5; // middle of bottom wall

void Update() {
  // get direction of move:
  int mv=0; // -1 0 or 1. Direction based on bottom wall
  if(Input.GetKey(KeyCode.A)) mv=-1; // A=clockwise
  else if(Input.GetKey(KeyCode.D)) mv=1; // D=counter-clockwise

  // handle move for each wall:
  if(mv!=0) {
    float mvAmt=0.1f*mv;

    if(wall==Wall.bottom) {
      x+=mvAmt; // note: + for D, - for A
      if(x>7) { x=7; wall=Wall.right; }
      else if(x<-7) { x=-7; wall=Wall.left; }
    }
    else if(wall==Wall.top) {
      x-=mvAmt; // top moves moves backwards from bottom
      if(x>7) { x=7; wall=Wall.right; }
      else if(x<-7) { x=-7; wall=Wall.left; }
    }
    else if(wall==Wall.right) {
      z+=mvAmt;
      if(z>5) { z=5; wall=Wall.top; }
      else if(z<-5) { z=-5; wall=Wall.bottom; }
    }
    else if(wall==Wall.left) {
      z-=mvAmt; // left wall: forwards=down
      if(z>5) { z=5; wall=Wall.top; }
      else if(z<-5) { z=-5; wall=Wall.bottom; }
    }
  }
  transform.position = new Vector3(x,0,z);
}
```

The enumerated type really was helpful here. While writing this, I messed up some of the numbers and directions. Seeing lines like `if(wall==Wall.right)` made it easy to see I was in the correct section.

18.7 Making a real function library

So far, I've been suggesting that all of your useful functions should be pasted into each new script. That will work, and we did it in the old days, but there's a nicer way. Almost all languages and environments let you spread your program over several files. Common, similar functions are usually placed in a file by themselves.

The steps to putting functions by themselves are:

- Make a new C# script the usual way (`Project -> CreateC#script`; double-click to open in the editor.) The name doesn't matter.
- Delete everything past the first two `using` lines.
  All that stuff is Unity set-up for things that go on a gameObject. we're making something different.
- Pick what you want to call the namespace. I'm calling mine `rand`. This does not have to match the file name.
- Set up the namespace like below. You might notice it's a simplified version of what we had before:

  ```
  public class rand {

  }
  ```

- Write your pure functions inside of it, with `public static` in front of each one (ex below).
  By pure, I mean they take inputs, and return values. They can't use any globals (which should be obvious, since how would they know what globals your other scripts have) and they also can't change position, color or size.

- Don't put it onto a gameObject (the system won't even let you.) Just having it be created is enough. The system will find it.
- Now any other script can run them by putting `rand`-dot in front.

A sample file with some random functions:

```
using UnityEngine;
using System.Collections;

class rand {
  public static int twoD6() { return Random.Range(1,7)+Random.Range(1,7); }

  public static bool heads() { return Random.Range(0.0f, 0.1f)>0.5f; }

  public static bool chance(int pct100) {
    return Random.Range(0.0f,100.0f)<pct100;
  }
}
```

Anyone can run `if(rand.head()` for a coin flip.

Unity uses this trick in a few places. `Random` holds several randomizing functions. `Mathf` has lots: `Mathf.Approximately(a,b)` is the close-enough function. `Mathf.Abs`, `Mathf.Clamp` and `Mathf.Max` are also there.