Chapter 38

First class functions

The one idea in this chapter is letting us use variables for functions.

For example, suppose `f1` was a variable and we assigned it the round-up function: `f1=Mathf.Ceil;`. Then `f1(3.7f)` would be 4. Since it's a variable, we could change it to round-down: `f1=Mathf.Floor;`. Now the same call `f1(3.7f)` would be 3.

We could even go nuts and say `f1=Mathf.Sqrt;`. I don't know what the square-root of 3.7 is, but `f1(3.7f);` would compute it.

Here's how it looks in working code:

```
System.Func<float,float> f1; // declaring f1
f1 = Mathf.Ceil;
print( f1( 3.7f ) ); // 4 - it runs ceiling function

f1 = Mathf.Floor;
print( f1(3.7f) ); // 3 - runs the floor function

f1 = Mathf.Sqrt;
print( f1(3.7f) ); // 1.92
```

From the declaration, you might have figured out that `f1` is limited to `float` input and `float` output functions. But it's not limited to built-in functions. We can point it at one we wrote, and there's even a rule to make up one on the fly (which I'm not showing you, yet):

```
float randAround(float x) { return Random.Range(f/2, f*1.5f); }

f1=randAround;
float x = f1(10); // random from 5 to 15
x = f1(3); // random from 1.5 to 4.5
```

This is going to be a really useful trick. But, as usual, even though you probably guessed them, we should go over the rules first.

38.1 Rules and examples

These things act like pointers – they don't hold a function, they just aim at them. C# has a special name for them (which I'm saving for later.) Most people call them function pointers.

When you write `f1=Mathf.Ceil;`, you're aiming it at that function. That's why there aren't any `()` parens after `Ceil` – you're not calling the function. We've never left them out before, but we've never had this trick before.

Using them works like other pointers. You automatically follow the variable to the function. If you start with `f1=Mathf.Floor;` then `f1(3.7f);` automatically follows `f1` to the function `Floor`, then runs it the normal way.

38.1.1 Declaring function pointers

This is a longish and semi-boring section. Knowing that `System.Func<float, float, float>` can point to Min or Max, but not square root (since it only has one input) is 90% of what you need. Skim this, quit when you get bored, then come back when you need the details.

The technical term for the type of a function is the *signature*, which is just the input and output types. Like regular pointers, there's no such thing as just a generic function pointer. They need to know the signature of functions they can point to.

This example makes `f2` able to aim at functions with two float inputs and a float return. If you look at the sample call, it's obvious why we can't point it to other types of functions:

```
System.Func<float, float, float> f2; // aim at:  float A(float, float)
f2 = Mathf.Max; // legal
f2 = Random.Range; // legal (picks the (float,float) version
f2 = Mathf.Floor; // ERROR, only 1 input (signatures don't match)
float ff = f2(3.4f, 7.9f); // sample call
```

Another example of the same thing, `f3` is for functions with a string input and output:

```
string makeLonger(string w) { w="x"+w+"y"; return w; }
System.Func<string, string> f3;
f3=makeLonger; // legal
f3=Mathf.Max; // not even close to legal
```

The exact way C# makes signatures for these is a little funny, but not too bad. As we saw, you write `System.Func` with angle-brackets. Inside, you list the input types, in order, then the output type *last* (not first, like in real functions – the output goes last.)

For example, `showCat` takes a Cat and returns a string, so has signature flipped as `Func<Cat, string>`:

```
string showCat(Cat c) { return c.name+" is "+c.age+" years old"; }
System.Func<Cat, string> g1 = showCat; // legal. signatures match
```

For a more oddball example, suppose some functions search a `float` array, starting where you tell it, and return one item:

```
float largest(float[] N, int startIndex) { ... }
float nicest(float[] Nums, int startHere) { ... }
```

The signature (inputs then output last) is `Func<float[], int, float>`. We can use it to make a pointer for them:

```
System.Func<float[], int, float> searchFunc;
searchFunc = largest; // legal, signatures match
searchFunc = nicest; // also legal
float num = searchFunc(NList, 3); // sample call
```

This rule works as-is for functions with no inputs. Put the return type by itself. For example, these two die-rolling functions take no input and return an `int`, so have signature `Func<int>`:

```
int twoD6() { return Random.Range(1,7)+Random.Range(1,7); }
int cheatingCoinFlip() { if(Random.value<0.75f) return 0; else return 1; }

System.Func<int> rollerFunc = twoD6; // legal
rollerFunc = cheatingCoinFlip; // also legal
```

For an odder example. `System.Func<Cat[]>` is a no-input function that returns a Cat array.

Functions with no return values are a little different. Instead of `Func` you write `Action` and just list the input types. Two examples of output-less functions and pointers for them:

```
void move(int x, int y) { ... }
System.Action<int, int> f1 = move;  f1(2,-4);

void makeAllCatsThisOld(Cat[] C, int newAge) {
  for(int i=0;i<C.Length;i++) C[i].age=newAge;
}
System.Action<Cat[], int> catChangeFunc = makeAllCatsThisOld;
```

There's one more special case: no inputs and no outputs, like `void doMove();`. For that, you write Action all by itself, with no angle-brackets: `System.Action f1; f1=doMove; f1();`.

Just so you know, there's nothing special about functions with no return values and the Func/Action split. It just worked out that way. You'd normally write `Func<int,void>` for an `int` input and no output, but `void` isn't legal there. Instead of making it be, they decided to create `Action` for no-output signatures. There's nothing special about the word `Action`. Functions with no return probably have side-effects, which make them action-y.

38.1.2 Using function pointers

Once you have a function pointer, you can do three things with it: point it somewhere, call it, or compare it.

Built-in functions and ones we wrote aren't any different to the computer. One pointer can flip between any functions, as long as the signatures match. The rules for changing where they point are the same as regular pointers. `f1=f2;`, makes `f1` point at the same function as `f2` does. We can even use `f1=null;`

The most interesting thing about using them to call a function is how it looks like a real function call. When you see `func1(5);`, you can't tell right away whether `func1` is a real function or a function pointer. It's similar to how in `c1.age` you don't know if `c1` is a struct or a class. In both cases, C# invisibly follows the pointer for you.

The rest of this is just examples of those rules, and little notes:

This is the standard pointer example, showing how we can aim these wherever we want. `average` is one of our functions, which has the same `float(float, float)` signature as Min and Max:

```
float average(float n1, float n2) { return (n1+n2)/2.0f; }

System<float, float, float> f1, funcPointer2;

void Start() {
  f1 = Mathf.Max;
  funcPointer2 = f1; // <- copy pointer to pointer
  f1 = Mathf.Min;
  float num = f1(5,8); // runs min, 8
  num = funcPointer2(5,8); // 5, f2 didn't follow f1 when we changed it

  f1=average;
```

```
num = f1(5,8); // 6.5
```

One thing to note is the variables aren't "smart" beyond the signature. What I mean is, Min, Max and Average feel like the same kind of math, but we're allowed to aim at any arbitrary function fitting the signature.

For example, we can aim `f1` at Random.Range, or even a useless "always -1" function:

```
float alwaysNeg1(float dummy1, float dummy2) { return -1; }

f1=Mathf.Max; // can go here
f1=Random.Range; // but also here
print( f1(6,9) ); // maybe 7.83?
f1=alwaysNeg1; // useless, but legal
print( f1(6,9) ); // -1
```

The `Random.Range` part is a little interesting, since we know there's a version with two `int`s and another with two `float`s. The trick is, `f1=Random.Range;` has to pick which one we use, using the signature of `f1`. When we come to `f1(6,9)`, we've already chosen the `float,float` version. That's not a big advantage, just kind of neat.

Compare really is the same as with regular pointers. This sets two function pointers and compares them to each other:

```
System.Func<float, float, float> f1, f2;
f1=Mathf.Max; f2=Mathf.Min;
if(f1==f2) print("point to same function"); // false
if(f1==Mathf.Max) print("f1 points to Max"); // true
if(f2==Mathf.Max) print("f2 points to Max"); // false
```

The last two lines are something new. We can compare function pointers directly to functions. The same as with single-equals, `if(f2==Mathf.Max)` isn't running the function. It's just checking where `f2` is aimed.

As usual, one way of using pointers is to always assign them, so you never need to use `null`. But if you want, you can use `null` to mean not assigned yet, empty or invalid. A typical semi-useful example:

```
System.Func<float, float> fixerFunc=null;
if(mode==0) fixerFunc=Mathf.Floor;
else if(mode==1) fixerFunc=Mathf.Ceil;
if(fixerFunc!=null) result=fixerFunc(result);
```

As usual, having `fixerFunc` be `null` isn't an error. Trying to follow a `null` pointer causes the problem. Running `fixerFunc(result)` on a null would give the standard run-time crash with nullReferenceException.

38.2 Uses

This whole section has no new rules – just common ways function-pointers are used.

38.2.1 As a regular variable

Any time you're thinking about making an `int` where 1 stands for function A, 2 for function B, you could make a function pointer instead.

Here's part of a program where our special weapon is either a laser, sonic cannon, plasma blob or nothing. This first version uses an integer to say which one it is (no function pointers, yet):

```
void fireLaser() { ... } // pretend these are written
void fireSonicCannon() { ... }
void firePlasmaBlob() { ... }

int specWeapon = -1; // 0=laser, 1=sonic, 2=plasma, -1=none

void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    if(specWeapon==0) fireLaser();
    else if(specWeapon==1) fireSonicCannon();
    else if(specWeapon==2) firePlasmaBlob();
  }

  void loadNextLevel(int lNum) {
    if(lNum==2) {
      specWeapon=0; // laser
      ...
    }
  }
}
```

The key messy part is having to remember the 0,1,2 weapon table. When we press a space, we need `if`s to decode the table. Then, when we change levels, we need to look at the table to assign the correct number for a laser.

We can simplify this by changing `specWeapon` from a look-up `int` to a function pointer:

```
System.Action specWeaponFunc = null; // no special weapon yet
// remember System.Action is the special syntax for no inputs or output

void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
```

```
    if(specWeaponFunc!=null) specWeaponFunc(); // <- easy call
  }

  void loadNextLevel(int lNum) {
    if(lNum==2) {
      specWeaponFunc=fireLaser; // <= nicer than a 0, and activates pop-up
      ...
    }
  }
}
```

The `if` is gone, and I think `=fireLaser` down below is easier to read than `=0`. Another advantage is when we type `specWeaponFunc=fire`, auto-complete will show us the three possible functions.

38.2.2 Arrays of pointers

If we have several functions, putting them in an array makes it easier to pick one. That's just a standard array trick, but it's worth seeing how we use it.

Some game background: when a character hasn't been moving for a while, we usually play an "idle" animation, like stretching. In this example, we have three of them, and want to pick one at random. Pretend each needs a different function to set it up:

```
void beginStretch() { ... } // pretend this starts stretching
void beginHandsOnHips() { ... }
void beginFlex() { ... }
```

We can put them in an array in the usual way:

```
System.Action[] Idles; // array of function pointers

void Start() {
  // set up the array:
  Idles = new System.Action[3];
  Idles[0]=beginStretch;
  Idles[1]=beginHandsOnHips;
  Idles[2]=beginFlex;
}
```

Now we can run `beginStretch` using `Idles[0]()`. That looks funny, but it's just array rules: `Idles[0]` is a function-pointer, so putting `()` after will run the function.

We already know how to pick a random item from a regular array. This picks a random item and runs it:

```
  IdleActions[Random.Range(0, IdleActions.Length)]();
  // runs beginStretch or one of it's friends, at random
```

It might look nicer broken over three lines:

```
  int iaIndex = Random.Range(0, IdleActions.Length);
  System.Action nextAct = IdleActions[iaIndex];
  nextAct(); // run it
```

You're even allowed to use the array-creation shortcut with functions. Start could just make the array in one line:
`Idles = {beginStretch, beginHandsOnHips, beginFlex};`.

I used simple void/void functions since it was shorter, but it might be nice to see what it looks like when we need the angle-brackets. Suppose the idle functions took a float input and returned a bool. We'd do it like this:

```
System.Func<float, bool>[] Idles;

void Start() {
  Idles = new System.Func<float, bool>[3];
  ...
  bool b = Idles[0](3); // run the 1st idle function, with input "3"
```

38.2.3 User-defined keys

This is another example of an array of function pointers. The idea is, we have three actions, drop, pickup and throw, and we want the player to pick which keyboard key does what (we'll make a menu for that, which I won't show in this example.)

We'll start with just a little struct pairing up the function pointer and current keypress. No array yet:

```
// pretend we have void() functions for drop, pickup and throw

struct KeyActionPair {
  public System.Action theAction; // pointer to drop, pickup or throw
  public char theKey; // key that does it
}
```

Then, the same as the idle action example, we'll make an array pre-filled with these pointers, and also the starting letters:

```
KeyActionPair[] KeyActions; // list of all actions and keys

void Start() {
  KeyActions = new KeyActionPair[3];
```

```
  KeyActions[0].theAction=drop; KeyActions[0].theKey= 'd';
  KeyActions[1].theAction=pickup; KeyActions[1].theKey= 'p';
  KeyActions[2].theAction=throw; KeyActions[2].theKey= 't';
}
```

Now `KeyActions[0].theAction()` runs the `drop` function, and `KeyActions[0].theKey=ch;` changes the key we use to drop things (the user menu we're not writing would do that.)

Unity likes us to read keys individually. So, to check during play, we'll loop through the array: check if that key was pressed; if so, run the function next to it:

```
if(Input.anyKeyDown) { // not needed, but can't hurt
  for(int i=0;i<KeyActions.Length;i++) {
    if(Input.GetKeyDown( KeyActions[i].theKey )) {
      KeyActions[i].theAction(); // run the function going with the key
      break;
    }
  }
}
```

If you know constructors, this would be nicer if the `KeyActionPair` struct had a simple 2-input constructor for setting the the pointer and the key together. I left it out to avoid clutter.

38.2.4 Passing functions into functions

The best way to explain this trick is with an example. Here's a regular function to count how many positive numbers are in an array:

```
int arrayCount(int[] A) {
  int count=0;
  for(int i=0;i<A.Length;i++)
    if(A[i]>0) count++;
  return count;
}
```

It would be great if we could send in a replacement for `>0`. Now that we have function pointers, we can. The first step is to rewrite greater-than-0 as a function:

```
bool isPositive(int x) { return x>0; }
```

Notice how this is a `bool(int)` function. That's what `>0` really is. It takes any integer, and tells us whether it likes it. Rewritten, it's a `System.Func<int, bool>` function.

The last step is to rewrite `arrayCount` to take that as an extra input. There are two changes – the extra input, and using it inside the `if`:

```
int arrayCount(int[] A, System.Func<int,bool> theTest) {
  int count=0;
  for(int i=0;i<A.Length;i++)
    if(theTest(A[i])) count++;
  return count;
}
```

Like any other parameter, `theTest` is going to passed in by the caller. For example:

```
int[] N = {4, 0, -3, 78, -11};
int c = arrayCount(N, isPositive); // 2
```

Notice, on the last line, how `isPositive` is still missing the parens. We're not calling it – we're just passing a pointer. Inside the function, `theTest` points to `isPositive` and `theTest(A[i])` runs it.

After all that work, we can finally count whatever we want, as long as we write a testing function. This counts how many in the array are even:

```
bool isEven(int num) { return num%2==0; }

  // elsewhere in the program:
  int n=arrayCount(N, isEven);
```

Even though `isEven` is a function name, it's passed like an normal variable. It's just copied into `theTest`. Below, in the loop `if(theTest(A[i]))` is now magically running the `isEven` function, counting how many things in the array are even.

Positive and even are 1-line functions. This next example is just to use a longer one. There's nothing new, but I think it helps. The function checks if a number is a square (1, 4, 9, 16 ...,) using a loop (only read it if you like odd loops):

```
bool isSquare(int num) {
  if(num<0) return false;
  int n=0;
  // keep adding 1. Stop when n-squared hits num, or goes past:
  while(n*n<num) n++;
  if(n*n==num) return true;
  else return false;
}
```

There's nothing special about it, and it just checks one number. `isSquare(81)` is true, `isSquare(22)` is false. The key is that it has a `Func<int,bool>` signature, so we can use it in `arrayCount`:

```
int[] A={9,5,25,3,2};
int sqCount = arrayCount(A, isSquare); // 2 (9 and 25)
```

I like this example since it's not special at all. We're calling another function over and over from inside the `arrayCount` loop. But we've always been able to do that. The function we call has another loop in it, making the whole thing a nested loop. But we've also always been able to do that.

The only new thing is being able to plug in any function.

Change each one

A function that changes each item in an array uses pretty much the same idea – we give it a function to use on each item. To make it interesting, I'm going to use an array of `Cat`s.

A simple cat-changing function takes one `Cat` as input and has no output. In other words, they will look like `void(Cat)` or officially `System.Action<Cat>`. Here are two sample functions to change one Cat:

```
void makeOlder(Cat_t c) { c.age++; }

void makeRoyal(Cat_t c) {
  // add "Lord" in front, unless the name already has a Lord in it:
  bool alreadyIsRoyal=false;
  if(c.name.FindFirst("Lord")>=0) alreadyIsRoyal=true;
  if(!alreadyIsRoyal) c.name="Lord "+c.name;
}
```

Our cat-array changing function will take one of these as input, and do it to each thing in the array. This is a little simpler than counting:

```
void changeCats(Cat_t[] C, System.Action<Cat> changer) {
  for(int i=0;i<C.Length;i++) changer(C[i]);
}

  // in the program:
  changeCats( MyCats, makeOlder);
  changeCats( CatList2, makeRoyal);
```

Multiple function inputs

Just to show we can, let's write a function with *two* function inputs. Take the array changer function above, and add a checker. In other words, we'll only make some certain cats royal. Here are two sample cat-checking functions:

```
// two sample cat-testing functions:
bool isHeavy(Cat_t c) { return c.weight>10; }
bool isKitten(Cat_t theCat) { return theCat.age<=2; }
```

These cat-checking functions are like the int-checking ones except cats have more parts we can check.

Now we'll add a cat-checker to the cat-array changing function. There's nothing new here, just a combination of the old checker and changer ideas:

```
// new cat-changer:
void changeSomeCats(Cat_t[] C,
       System.Func<Cat,bool> theTest, System.Action<Cat> theChange) {
  for(int i=0;i<C.Length;i++)
    if(theTest(C[i])==true) theChange(C[i]);
}
```

Notice how the top part is starting to get ugly. We've got a `Func` and an `Action`, and I always get confused how `Func<Cat,bool>` is really `bool Func(Cat)`.

Calling it is nothing special, just put the names of the real functions, in the order it says:

```
// cats 11+ pounds get older:
changeSomeCats(C1, isHeavy, makeOlder);

// young cats become Lords:
changeSomeCats(MyCats, isKitten, makeRoyal);
```

This exact example is a little contrived. But the idea is that function inputs are like any other inputs. Use as many as you need.

Change all children

We can use the idea of a change-function with the visit-all-my-children Unity trick. If you haven't seen it, or have but still don't understand recursion, this will make less sense but is still worth skimming.

As a review, here's the recursive function to touch a gameObject, all of its children and so on. As a stand-in, it just prints the name:

```
void touchAllChildren(Transform t) {
  print( t.name ); // <= test action
  int childCount=t.childCount;
  for(int i=0; i<childCount; i++)
    touchAllChildren(t.GetChild(i)); // <= recursive call
}
```

Printing the name is a stand-in for "do something with `t`". We'll change it to calling some transform-using function we plug in. The two changes are the extra input and the first line:

```
void changeAllChildren(Transform t, System.Action<Transform> action) {
  action(t); // <= do whatever thing we plugged in
  int childCount=t.childCount;
  for(int i=0; i<childCount; i++)
    changeAllChildren(t.GetChild(i), action);
}
```

I named my function pointer `action` since I couldn't think of a better name. Here's a sample function to turn just one thing red, and a call using it to turn all my children red:

```
void turnRed(Transform t) {
  Renderer rr = t.getComponent<Renderer>();
  if(rr!=null) rr.material.color=Color.red;
}

  // dog and all dog descendants turn red:
  changeAllChildren(dog.transform, turnRed);
```

There's nothing really special here. This is the same idea as the Cat-changer. If you have some way to look at a bunch of stuff, even if it's a complicated recursion, you can still use the plug-in function trick. We sometimes say we're *visiting* each item. How we visit them all is one thing. What we do during the visit is the other, separate part.

38.3 Function pointers as callbacks

Unity's physics system is an example of *callbacks*, which are usually done using function pointers.

The key idea is you just write `OnCollisionEnter`, with the inputs it requires. Then it's magically called. You don't need to check for collisions yourself – you just need to know the system is checking for them and will call you when one happens.

The functions you plug into the system are called *Event Handlers* or *Call-Backs* or *Listeners*. The terms are used interchangeably.

The things the system tracks for you are the *Events*. They're nothing special – just what the people writing it thought you might want to be told about.

What usually happens is you have some pre-written event system. You have to figure out what the events are, what the function signatures look like, and what the rules are for hooking up your callback functions.

Because of that, some examples below aren't complete. I'm trying to show only the important parts, but maybe not doing a great job. Hopefully at least one of these will make some sense:

Drags and Slides

Suppose we have a pre-written script that tracks screen touches, looking for drags and pinches. Each frame it sees a drag or a pinch, it runs our callback function. The interesting part of the code is:

```
// user plugs callbacks into these 2 variables:
public System.Action<Vector2> dragCallback=null; // input is x&y movement
public System.Action<float> pinchCallback=null; // input is pinch amount

// deep inside the code:
void processDrag( ... ) { // pretend this is called for drags
  ...
  if(dragCallback!=null) {
    Vector2 dragAmt = ... touch math here
    dragCallback(dragAmt); // <-call the function we plugged in
  }
```

`dragCallback` is a function pointer. It's purpose is to be a place the user can plug-in their own functions. This code will never change it. The only place it's used is those last two lines.

Notice how we even use the common rule "`null` is legal, and means do nothing."

This user code has `LRcamSlide` to handle drags, which it plugs in with an assignment statement:

```
// pretend this is a preset link to the script, above:
public fingerTrackScript FT;

void Start() {
  // aim the callbacks at our functions:
  FT.dragCallback=LRcamSlide;
  FT.pinchCallback=null; // ignore pinches
  ...
}

void LRcamSlide(Vector2 amt) { // <- or drag callback
  float sideMove=amt.x*20;
  theCam.position+=new Vector3(amt, 0, 0);
}
```

As far as we see, `LRcamSlide` is magically called on drags.

On purpose `LRcamSlide` only uses `x`, to emphasize the pre-written idea. The touch-tracker is written to be used by everyone. It computes and sends all the info anyone might need: `x` and `y`, in a `Vector2`.

Our script is just one thing using it, and we have to follow the rules, which means taking a `Vector2`.

Here's more imaginary user code. Jumping disables camera movement and enables a cheat for pinching:

```
void beginJump() {
   ...
   FT.dragCallback=null; // can't slide while jumping
   FT.pinchCallback=ammoCheat;
   // cheat code: pinch during jump for full ammo
}

void ammoCheat(float p) { // <- need a float to match the pinch signature
  ammo=999;
}
```

This may not be the best way to do it, but it shows how we can plug and unplug at will.

Unity's UI callbacks

Unity's canvas/UI system uses callbacks. Buttons have a script on them named `Button`, with a `void()` callback. In other words, you can write a no-input, no-output function and tell the button to call it when it's clicked.

This is working code:

```
public UnityEngine.UI.Button btn1; // drag in a Button

// button click function for btn1:
public void testBtnClick() {
  string w=""+Random.Range(1,1000); // sample random #
  // NOTE: buttons have a child named "Text"
  btn1.transform.Find("Text").GetComponent<UnityEngine.UI.Text>().text=w;
}

void Start() {
  // add my testBtnClick function as callback for button1:
  btn1.onClick.AddListener(testBtnClick);
```

When the button is clicked, which Unity will check for us, our `testBtnClick` function is magically called.

The last line looks funny because Buttons don't just have one click callback – they have a list. This is a standard bonus feature that most systems have.

`onClick` is a tiny class holding an array of function pointers. `AddListener` is a shortcut for adding to the end. Notice how we're still using no-parens `testBtnClick`. We're sending you the function, not calling it yet.

It's not easy to figure out that Buttons want a `void()` function. If you look at the tool-tip for `AddListener`, it says the input is a "callback function." Then it says the type is `void UnityAction()`, which means `void()`.

Fun fact: this system means you can add several callbacks to a button. It also means changing to a new one requires `RemoveAllListeners` first.

Redirecting OnCollision

This next example is a simple use of a callback. In Unity, collisions will only ever call a script on the object that was hit. If my game has 5 different color balls, doing different things when they get hit, I need a different script for each color. My ball code is spread out over all those scripts and I don't like that.

Instead, I'll write one script, used on every ball, which redirects the collision to a function in my main script:

```
class genericColliderScript : MonoBehaviour {
  // set this to point to the real collision handler:
  public System.Action<Transform, Collision> callBack;

  void OnCollisionEnter(Collision col) {
    // tell it who you are, and your collision info:
    if(callBack!=null) callBack(transform, col);
  }
}
```

By setting the `callBack` variable, we can tell each block what to do on a collision.

This sample main program has the collision functions for two block colors, and sets the blocks up to use them:

```
void handleRedBlockHit(Transform block, Collision cData) {
  // red blocks just die when hit:
  Destroy(block.gameObject);
}

void handleGreenBlockHit(Transform block, Collision cData) {
  // we get a point when green blocks are hit:
  score++;
}
```

```
void Start() {
  // setup a red  block:
  Transform rb=Instantiate(redBlockPF);
  // have it call the red function above, when hit:
  rb.GetComponent<genericColliderScript>().callBack=handleRedBlockHit;
  // for real we'd position it, etc...

  // bunch of green blocks:
  for(int i=0;i<5;i++) {
    Transform gb=Instantiate(greenBlockPF);
    gb.GetComponent<genericColliderScript>().callBack=handleGreenBlockHit;
  }
```

Now all of our block collision code is together, and can easily use our variables if they need to (like `score` for green ones.) It's not a huge improvement, but being able to organize your code is nice.

38.4 Just cramming in a function

For short things, it's a pain to have to write a separate function. There's a shortcut rule to let you write a nameless mini-function directly. This legal code uses the `countSome` function to count numbers less then 10:

```
print( countSome(A, n => n<10));
```

The `=>` symbol was made up just for these mini-functions. A trick to understanding them is you always know the required types. In this example, `countSome` has to take a `bool(int)` function, so it knows `n` stands for the `int` input. A shortcut lets you leave out the `return`: if you just write math, it knows to automatically return it.

There's a longer version doing the same thing:

```
print( countSome(A, (n)=>{return n<10;} ));
```

The syntax can be a little confusing, but if you get the idea you can look up examples easily enough. The official name for these are **lambda expressions**. That's a real computer science term – when functions were invented, back in the 60's, that's what we called them.

You can use this trick to assign; and you can also have multiple inputs. These completely silly examples show both:

```
System.Func<int,int,int> f1;

f1 = (x,y)=>x*y/10; // nameless (very silly) math function
print( f1(5,7) ); // 3 (35/10)
```

Notice how we're leaving out the curly-braces and the `return` again. It assumes `x*y/10` is the answer. The long way is `(x,y)=>{ return x*y/10; }`. No matter what, we leave out the types. We already wrote they're all `int`s when we declared `f1`.

This one has an `if` and two returns, more like a real function. You could write a very long function this way, but it's probably better to pre-write those, the regular way:

```
// nameless max function:
f2 = (x,y)=>{if(x>y) return x; else return y; };
float n = f2(7,9); // 9
```

As you might guess, the main use for this trick is not having to go off and write all those 1-line functions. For my super cat-changer, we can change the name of all 1-year-old cats to Kitty:

```
changeSomeCats(C, c=>c.age==1, c=>{c.name="Kitty";} );
```

Then, a C# note just in case: if you look these up, there's an obsolete syntax for these using the word `delegate`. Ignore that and you'll find this good way further on.

38.5 Sort plus function input

Most languages have a built-in sort-any-way-you-want, using function pointers. You write a function that says whether two things are in order, send it to the Sort, and it puts the whole list in that order.

Your function always take 2 inputs of the same type. In `C#` -1 means in order and +1 means out-of-order (0 means equal, which is good for non-sorting things.) Here's a working example that sorts Cats by age:

```
int catAgeOrder(Cat c1, Cat c2) {
  if(c1.age<c2.age) return -1; // in order
  if(c1.age>c2.age) return +1; // out of order
  return 0; // equal (won't matter for sorting)
}

public Cat[] C; // pretend this is filled in

  System.Array.Sort(C, catAgeOrder); // sorted by age
```

`System.Array.Sort` is built-in. It has a ton of overloads, but one takes `System.Func<Cat,Cat,int>` as the second input (obviously it keys off of the type of the first input.) It uses the function we gave it to compare, using it to decide when to fix out-of-order. The result will be sorted by age.

This one sorts integers based on absolute value, for example [2, -3, -6, 8, 12, -23] wold count as sorted:

```
public int[] Nums = {8, -3, 2, -23, -6, 12};

  System.Array.Sort(Nums, intAbsOrder); // sorted by "size"

int intAbsOrder(int a, int b) {
  if(a<0) a*=-1; if(b<0) b*=-1; // make both positive if not
  if(a<b) return -1; // in order
  if(a>b) return +1; // out-of-order
  return 0; // equal
}
```

A quick check: `intAbsOrder(5,-6)` should say they're in order, and it does (it sees `5<6` and returns -1.) It should say `(-7,2)` is out of order, and it also does (it sees `7>2` and returns +1).

We can also use the nameless function trick. This sorts from longest name to shortest. I'll skip returning 0, for equals, to save space, and use the `?:` if-shortcut:

```
public Cat[] C; // fill with cats
System.Array.Sort(C, (a,b) => a.name.Length>=b.name.Length?-1:+1 );
```

The nameless function says "if the first name is at least as long as the second name, they're in order."

A sneaky use of a nameless function is to flip another one. The -1/0/+1 answer times -1 reverses it perfectly, sorting backwards:

```
public Cat[] C; // fill with cats

// sorts cats by age, high to low:
System.Array.Sort(C, (a,b)=> -1*catAgeOrder(a,b) );
```

This next part is just a little interesting: we like to build these compare functions into namespaces. For example, `C#` can't compare strings alphabetically using `<`. Instead it has a -1/0/+1-style function named `Compare` using string as a namespace:

```
print( string.Compare("cat", "dog")); // -1
print( string.Compare("cat", "cat")); // 0
print( string.Compare("dog", "cat")); // 1

System.Array.Sort(W, string.Compare); // sort strings by alpha
```

When we have a class that we think we may want to sort, we could write some sort-how functions inside it:

```
class Cat {
  public string name; public int age;

  // add possible sort functions:
  static public int compAge(Cat a, Cat b) { ... } // -1/0/+1
  static public int compNameLen(Cat c1, Cat c2) { ... }

}
```

Then would could use `Sort(C, Cat.compName);` to sort by name.

38.6 Built-in array function-using functions

C# has some fun built-in functions using an array and a function pointer. An easy one (which I wrote earlier by hand as `arrayCount`) is counting things we like in an array:

```
// need this at the top:
using System.Linq;

int[] A={4,8,7,1,3,12,5};
// give it a function that likes things more than 6:
int n=A.Count( x=> x>6 ); // 3
```

`Count` is written as an array member function. Putting `using System.Linq` at the top adds it. I used the nameless function shortcut, but it works with anything, for example `A.Count(isPositive)`.

It also works for an array of anything. This counts how many cats have long names:

```
Cat[] C; // pretend this is created
int n=C.Count( c=> c.name.Length>12 );
```

There are two more like this. `Where` returns a shorter list of just what we like. `Select` returns a same-length list of true/false:

```
int[] A={4,8,7,1,3,12,5};
int[] B=A.Where( x=>x>6 ).ToArray(); // B is {8,7,12}

bool[] C=A.Select( x=>x>6 ).ToArray(); // C is {f,T,T,f,f,T,f}
```

Select doesn't seem useful right away, but it's there just in case. The extra `ToArray()` just goes there.

There are two `bool`-returning ones. `All` means "is it true for all of these?" and `Any` means "is it true for any of these":

```
int[] A={4,8,7,1,3,12,5};
// is anything in A more than 10:
bool hasAnyMoreThan10 = A.Any( x=> x>10 ); // true

// is all of A more than 10:
bool areAllMoreThan10 = A.All( x=> x>10 ); // false

// Or use a premade function:
if( A.All( isPositive )) print("everything in A is positive"); // true
```

A semi useful example `if(C.Any( c=> c.name=="" ))` checks whether any of our cats don't have names yet.

If you hate these, the important thing is merely knowing they exist. The computer thinks function pointers are so cool that it adds built-ins using them.

38.7 Older `delegate` syntax

C# has another way to create function pointers. It wouldn't be worth mentioning except it uses a common computer trick – a Typedef. It's also a fun trip into computer language archaeology.

A typedef is a way to make a shortcut definition. A typical one might let you use `intGrid` as a short version of `List<List<int>>` (remember that a List of Lists is how you make a grid.)

`C#` doesn't have that in general, but it has a version that only works for function pointers. This make `chooser` be a shortcut for `system.Func<float, float, float>`:

```
delegate float chooser(float a, float b);

chooser f1; // declare f1 as function pointer
f1=Mathf.Max;  f1(7,3); // this is legal. answer is 3
```

`delegate` is a keyword. It lets the computer know the rest of the line is creating a shortcut. The trick is it means "`chooser` stands for functions that look like this."

That's the way most languages declare function pointers – by making something that looks like the function signature it takes. `C#`'s `System.Func` stuff is more of an oddball.

If you look at the official type for `System.Array.Sort`, it says it takes `Comparison<float>` as the second input (it will have whatever type your array is.) That's another use of the shortcut.

As we know, it takes a `System.Func<float,float,int>` function – compare two floats and return a -1/0/+1 int. They just thought naming the shortcut Comparison was a nice hint.