Chapter 34

Reference parms

This section is about a new rule where a function can change its inputs for real. We can partly do that now with pointers – `fixName(dog1);` can adjust `dog1` – but this lets us do it with ints, strings, floats, and structs.

We try not to use this rule if there's any better way. But it's good for the times you need it, and it's a basic computer science concept. The normal way, which copies inputs, is named **call-by-value**. This new way is **call-by-reference**.

34.1 `ref` rules

This rounds a `float` to the nearest value, changing it in place:

```
void roundMe(ref float n) { // <- ref is new rule
  int nearest = (int)(n+0.5f); // standard rounding trick
  n=nearest;
}
```

Notice how all it does is change `n` into the correct value. We've seen functions like this before. It works because `n` is a solid link to the variable you gave it:

```
float q=3.6f;
roundMe(ref q); // q is 4
```

Here are the rules for call-by-reference (what you type, mixed in with how it works):

- Write `ref` in the function definition, before the parameter type.
- You also have to write `ref` in front when you call it. That's to remind people that it's a reference parameter.

- The `ref` applies to one thing, but you can use it more than once. Exs: `void A(ref int n, ref float ff)` (both) or `void B(int q, ref string w);` (only `w`).
- Inside the function, the `ref` parameter isn't a new variable box with a copy of the input. It uses the box you gave it. It's like a super-pointer.
- You have to call it with a variable, which must be the correct type. This is a natural consequence of sharing a box. `ref float n` takes a box holding a float.

  For examples, `roundMe(ref 12.3f)` is an error – not a box. `roundMe(ref x+1.0f);` is an error – also not a box. `int n=4; roundMe(ref n)` is an errors – not a float box. But `roundMe(ref c1.wt);` and `roundMe(ref A[2]);` are fine.
- Changing a reference parameter changes the original variable.

In `float q=3.6f; roundMe(ref q);` I like to think that we're giving it `q`. Without the `ref`, we're merely giving it 3.6. With `ref` we're handing over the entire box.

34.2 More `ref` examples

Here's another version of the old Clamp function which changes the input into the correct range, using call-by-ref:

```
void clampMe(ref int num, int low, int high) {
  if(num<low) num=low;
  else if(num>high) num=high;
}
```

Notice how all it does is change `num`. Running it looks like this:

```
int x=12;
clampMe(ref x, 1, 10); // x is lowered to 10, for real

int z=8;
clampMe(ref z, 50, 60); // z is raised to 50
clampMe(ref z, 1, 100); // no change. z still 50
```

Here's moveTowards rewrite. It pushes the input float towards the target, but not past:

```
void moveMeBy(ref float num, float target, float mvAmt) {
  // get there in one step?
  if(Mathf.Abs(target-num)<=mvAmt) { num=target; return; }
  // go towards target, in whichever direction:
```

```
  float dir = Mathf.Sign(target-num); // +1 or -1
  num += mvAmt*dir;
}
```

`moveMeBy(ref n, 10, 0.1f);` pushes `n` 0.1 closer to 10, stopping on it.

Those three are not useful functions. At first, `roundMe(ref x);` seems better than the hacky `x=round(x);`. But a return value let's us do more:

```
float x2=round(x); // using a normal round, which returns the answer
z=round(x+y);
if(round(x)>round(y)) ...
```

A swap is an actually useful call-by-ref function. We've seen the 3-step swap dance. Now we can put it into a function:

```
void swap(ref int a, ref int b) {
  int tmp=a; a=b; b=tmp; // standard 3-part swap dance
}
```

Notice how both have the `ref` in front, since both need to change. A sample use:

```
if(low>high) // switch to put them in order:
  swap(ref low, ref high);
```

Another real one I sometimes use is for computing a min and max. If it needs to, it adjust one of them to contain the new value:

```
void growRange(ref float min, ref float max, float newNum) {
  if(newNum<min) min=newNum;
  if(newNum>max) max=newNum;
}
```

It doesn't seem like much, but it can help find the min and max of a list:

```
int smallest=A[0], largest=A[0];
for(int i=1; i<A.Count; i++)
  // grow to fit A[i] in the range:
  growRange(ref smallest, ref largest, A[i]);
```

If saves me from writing 2 `if`'s, has a descriptive name, and there's no way to write it without using call-by-ref.

34.3 Output parameters

We can use the call-by-ref trick to have a function return 2 things. We'll give the function 2 empty boxes, asking it to put the answers in them.

This function attempts to break a string into 2 words if it has a comma. We can't return 2 things, but we can fake it with 2 "output" parameters:

```
void commaSplit(string w, ref string w1, ref string w2) {
  int commPos = w.IndexOf(',');
  // if no comma, we only have 1 word:
  if(commaPos<0) { w1=w; w2=""; return; }
  w1=w.Substring(0,commaPos); // before comma
  w2=w.Substring(commaPos+1); // after comma
}
```

In our minds, this returns `w1` and `w2`. A sample run:

```
string animal="", noise="";
commaSplit("milk cow,moo-moo", ref animal, ref noise);
// animal is "milk cow", noise is "moo-moo"
```

`commaSplit` feels as if it has 1 input, and 2 slots for the outputs.

Suppose we often want to convert floats like 5.32 into a 5 and a 0.32. We could write a "return-by-reference" function for that. In our minds it has 1 real input, and 2 outputs:

```
void wholeAndFraction(float num, ref int whole, ref float fraction) {
  whole = (int)(num); // ex: (int)5.8 is 5
  fraction = num-whole; // ex: 5.8-5 is 0.8
}
```

We call it by making 2 spots for the results and calling with the as the last 2 inputs:

```
int n1; float f1;
wholeAndFraction(9.34f, out n1, out f1);
// n1 is 9, f1 is 0.34
```

34.3.1 `out` shortcut

As we've seen, there are 2 main ways to use call-by-reference. The first is when we give it a box to possibly adjust. The function will read from it, and might change it or might not. The second is giving blank boxes for the answers. The function has no reason to read from them, and will definitely fill them all with the answers.

C# decided to make an extra rule for that second way: `out` for "output parameter". It's the same as `ref` except with more error messages (you can't read from them, and you have to assign to them).

Here's `commaSplit` rewritten with `out`:

```
void commaSplit(string w, out string w1, out string w2) { // <-out
  int commPos = w.IndexOf(',');
  // if no comma, we only have 1 word:
  if(commaPos<0) { w1=w; w2=""; return; }
  w1=w.Substring(0,commaPos); // before comma
  w2=w.Substring(commaPos+1); // after comma
}
```

It's the exact same, except for the word `out`.

It's another fun example of language design. On the one hand, `out` is a tiny change from `ref`, and you don't need it, and it's another special word to remember. On the other hand, why not build the idea of output parameters into the language?

It turns out that C# is the only language to that chose to add it. In fact, Java decided the whole call-by-reference thing was so rare and hacky that Java doesn't have even `ref`.

34.4 mixing real return and `ref` parms

We often have functions that either give an answer, or they don't work. Often they can return -1 or null, but not always. Sometimes we like to have them return a `bool` saying it they worked, and give back the real answer in a reference parm.

For example, `int.TryParse` is a C# built-in that attempts to convert `w` into an int, or else returns false:

```
string w; // could be "12", or could be "cat"
int num;
bool gotNum = int.TryParse(w, out num);

if(gotNum) print("number is "+num);
else print("failed. w wasn't a number");
```

For `"37"` it will return true, and `num` will be 37. For `"3cat"` it returns false and `num`'s value won't matter.

The more common way to use it looks very strange at first:

```
if(int.TryParse(w, out n)) print("number is "+n);
else print("can't read it");
```

Normally, functions inside of an `if` shouldn't do anything except return true or false. But in this case, it seems fine to have it also fill `n`.

We can rewrite `commaSplit` to use this style. It returns false if there isn't a comma:

```
void commaSplit(string w, ref string w1, ref string w2) {
  int commPos = w.IndexOf(',');
  // if no comma, it didn't work:
  if(commaPos<0) { w1=w; w2=""; return false; }
  w1=w.Substring(0,commaPos);
  w2=w.Substring(commaPos+1);
  return true; // got both words
}
```

The new way to use it:

```
string animal="", noise="";

if(commaSplit(W[0], ref animal, ref noise)) {
  print("The "+animal+" says "+noise);
}
else print("invalid animal/noise pair");
```

34.5 Pointers and References

We can already change classes from inside of a function, so it seems like we wouldn't also need call-by-reference for them. But sometimes we do.

This function to swap two goats (assume Goat is a class) does nothing:

```
void swap(Goat a, Goat b) { Goat tmp=a; a=b; b=temp; } // broken swap
```

Here's a picture after we call `swap(g1, g2);`:

```
// start function call:
  main                    swap
g1 -> [ name: A  age: 4]  <- a
g2 -> [ name: B  age: 7]  <- b
```

If we change `a.age`, we're changing the real age. But making `a` point to `b` is simply moving a local variable. We can't change where `g1` points, only the contents of what it points to.

Adding `ref` in front of `a` and `b` fixes the problem:

```
// working goat swap
void swap(ref Goat a, ref Goat b) { Goat tmp=a; a=b; b=temp; }

swap(ref g1, ref g2);
```

Now changing `a` is also changing `g1`.

This next one is simpler, but somewhat made-up. Suppose we want a function to blank out a goat, which works for `null` goats. This version doesn't work:

```
void blankGoat(Goat g) { // non-working
  if(g==null) g=new Goat(); // <-original goat is still null
  g.name="test goat"; g.age=0; // <- this part is fine
}
```

It's the same problem. We can't make the original goat pointer aim at a different Goat. `ref` fixes it:

```
void blankGoat(ref Goat g) {
  if(g==null) g=new Goat(); // <-changes calling goat to point here
  g.name="test goat"; g.age=0;
}

blankGoat(ref gg1); // works, even if gg1 is null
```

34.6 Raycast example

`Raycast` is probably the most complicated function in Unity – it uses an output parameter with a `bool` return, and has a bunch of overloads and default parameters. It makes a nice example of lots of things, including call-by-reference.

What it actually does is pretend to shine a laser pointer in the 3D world. It tells us what it hit, if anything. We use it to shoot a game laser, or check where a mouse is aimed, or check whether we have room to move in a certain direction.

With all the combinations of overloads and default parameters, there are 16 versions! Four are simplifications of this monster:

```
bool Raycast(Vector3 origin, Vector3 direction, out RaycastHit hitInfo,
   float maxDistance = Mathf.Infinity, int layerMask = DefaultRaycastLayers,
QueryTriggerInteraction queryTriggerInteraction = QueryTriggerInteraction.UseGlobal)
```

The last 3 use default parameters. That means we can ignore them, leave them out, and the system fills them with the standard values. That leaves us with this:

```
bool Raycast(Vector3 origin, Vector3 direction, out RaycastHit hitInfo)
```

The first two inputs are the imaginary ray we're shooting: where it starts in 3D, and the arrow to follow coming out of the start.

The third is the answer. `out` means it's an output parameter, which means we merely need to give it a blank `RaycastHit` struct. That's a specially made

struct, whose only purpose is to hold results of a raycast. As usual, `hitInfo` is an unimportant variable name.

So here's a legal use of raycast. It shoots from the camera, straight forward:

```
Vector3 pos = new Vector3(0,0,-10); // where camera is
Vector3 dir = new Vector3(0,0,1); // forward along Z

RaycastHit RH; // will hold output
Physics.Raycast(pos, dir, out RH)); // now RH has info
print("We hit " + RH.transform.name);
```

But a raycast could also hit nothing. That's why it returns a `bool`. False means it missed. We use it like this:

```
if(Physics.Raycast(pos, dir, out RH)) {
  print("We hit " + RH.transform.name);
}
else { print("We missed everything"); }
```

Let's play around just a little more. The 3 parameters we skipped are distance, layerMask, and triggerInteraction. We have to fill them in from left-to-right, which means we can use distance.

This shoots a ray for 20 units (if something is 21 units away, it won't reach and returns false):

```
// shoot a ray only 20 units forward:
if(Physics.Raycast(pos, dir, out RH, 20)) {
  print("hit "+RH.transform.name);
}
```

Another version uses a `Ray` as the first input. Here's the struct `Ray`. It's incredibly simple, for a struct:

```
struct Ray {
  public Vector3 origin;
  public Vector3 direction;
}
```

All it does is group together a starting position and a direction arrow. Here's a raycast using a Ray:

```
Ray rr;
rr.origin = new Vector3(0,0,-10); // using the same positions
rr.direction = new Vector3(0,0,1); // and arrow as earlier

RaycastHit RH;
if(Physics.Raycast(rr, out RH)) { ... }
```

Finally, here's a semi-useful script to test raycasts. The player moves along the bottom of the screen, shooting a raycast straight up. Whatever it hits turns red, then white when we move away:

```
Vector3 upDir = new Vector3(0,1,0); // This won't change
Transform oldHit = null; // the thing we were hitting last frame

void Update() {
  movePlayer();

  RaycastHit hitInfo; // results go in here
  Transform newHit; // what's above us this frame (can be nothing)
  if(Physics.Raycast(transform.position, upDir, out hitInfo)) {
    newHit = hitInfo.transform;
  }
  else {
    newHit=null;
  }
  // do something if we hit something different:
  if(newHit != oldHit) {
    if(oldHit!=null) setColor(oldHit, false); // reset old to white
    if(newHit!=null) setColor(newHit, true); // color new thing red
    oldHit=newHit;
  }
}

void setCol(Transform tr, bool isOn) { // on=red, off=white
  Color cc = Color.white;
  if(isOn) cc=Color.red;
  tr.GetComponent<Renderer>().material.color = cc;
}

float xPos=0;
void movePlayer() {
  if(Input.GetKey("a")) xPos-=0.1f;
  else if(Input.GetKey("s")) xPos+=0.1f;
  xPos = Mathf.Clamp(xPos,-7,7);
  transform.position=new Vector3(xPos,-3.5f,0);
}
```

34.7 Reference vs. reference

You may have noticed that we now have two different uses of the word reference: call-by-reference, and reference types. That's an unfortunate accident, especially since they have similar meanings, but different.

Reference types are actual variables, with their own boxes. Which are pointers. Call-by-reference variables aren't their own variables. They're alternate names for the original box.

It's common to say `Dog d1;` is a reference. Inside a function where `w1` was passed-by-reference we'd also call `w1` a reference. Yikes!

Then take this function:

```
void A(Dog d1, ref Dog d2, ref int x) { ... }
```

`d1` is a reference variable passed by value. `d2` is a reference passed by reference. `x` is a value-type passed by reference. Yikes, yikes!