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, float and structs.

This is one of those rules we try not to use if there's any better way to make a function. But it's good 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**.

As usual, I'm going to mix rules and simple examples, then later some good examples of when we would and wouldn't use this trick.

34.1 `ref` rules

Let's start with an example that rounds a `float` to the nearest value:

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

Notice how it uses the "turn the input into the answer" shortcut, but then it returns nothing. It even has `void` for the return type, and that's not a mistake. Running it:

```
float q=3.6f;
roundMe(ref q);
print(q); // 4 <--the function changed q!
```

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. It applies to only that one, 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`).
- You also have to write `ref` in front when you call it.
  You shouldn't need to, but in `C#` it's a required extra word.
- Inside the function, the `ref` parameter isn't a new variable box with a copy of the input. It's a direct link to the variable you called it with, like a super-pointer. Changing a `ref` parameter changes the original.
- You have to call it with a variable – you can't use a number or equation. This is because of the "direct link" rule. The `ref` parameter uses the box of the original variable, so there has to be one.

  For example, `roundMe(ref 12.3f)` would try to change 12.3 to 12, which makes no sense. `roundMe(ref x+1.0f);` isn't legal since `x+1` isn't a box. But `roundMe(ref g1.age);` and `roundMe(ref A[2]);` are fine. Both of those count as int boxes.
- The types must be an exact match. So no ints where it wants a float (which works everywhere else, but not here.) That's also because of the "uses the same box" rule.

To sum up, all call-by-reference functions work by changing the input with the `ref` in front. `roundMe` changes input `n` into the answer. In a normal function, that would be a shortcut to not declare a new variable. In a `ref` function it's the entire point.

`roundMe(ref amt);` is like giving the function your `amt` and telling it to fix it. Inside `n=nearest;` is the result-setting line. It's like the return. It reaches back to change `amt`.

Here's one with more inputs. Some review: the `clamp` function takes a number to fix, and a low/high range. If the first number isn't in range, it's clamped to be. `clamp(12,1,10)` is 10, but `clamp(6,1,10)` is just 6. The normal version returns the answer. So `clamp(x,1,10)` might return 1, but `x` is unchanged.

This version changes the input:

```
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` – which has a `ref`. That's how call-by-ref works. 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
```

This next one is a rewrite of `moveTowards`. It pushes the first input towards the second by the amount of the third. The old version returned that number. This version actually changes the first input:

```
void moveMeBy(ref float num, float target, float mvAmt) {
  // get there in one step?
  if(Mathf.Abs(target-num)<=mvAmt) num=target;
  float dir = Mathf.Sign(target-num); // +1 or -1
  num += mvAmt*dir;
}
```

It's the same idea as the last two. Now we can use `moveMeBy(ref v.x, 10, 0.1f);` and it really will change `v.x` to be a little closer to 10.

This last one is a real use – something we couldn't do before. It's the standard classroom example. We can write a function to switch two variables:

```
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) swap(ref low, ref high);
```

This doesn't save typing, but it's still classic function use: we see the helpful name `swap`, we only have to type each variable once, and it's easy to mess up the swap dance. So this way is easier to read and safer.

34.1.1 ref vs. normal notes

I mentioned how we rarely need `ref`. Of those 4 examples, `swap` was the only good one. The other three are worse than using a normal math function. They're good examples, but there's a reason we wrote them first as pure math functions.

At first, `roundMe(ref x);` seems better than the hacky `x=round(x);`. But a real program often wants a temp value:

```
float newx=round(x);
// do stuff with x and newx
x=newx; // now finally update x
```

Likewise we can use the old version in math: `q=round(x+y);` or `if(round(x)>round(y))`. Most of the time we don't want to change the input.

Having two versions, `round` and `roundMe`, tends to be confusing. It's easier to remember to use `x=round(x);` for the rare times you wanted to change `x` "in place", and not have a reference version at all.

When a function can return the one answer, it's usually best to have it do that, instead of changing one input. Call-by-ref is good for the weird stuff - changing two inputs, or when you already have a return value and need to do something else (examples below).

34.2 Output parameters

Sometimes we want a function that returns two or more things. We've already got some ways to do that. But if nothing else works, we can abuse reference parameters to do it.

The rules are the same, but the way we think of them is different.

In the last section, the purpose of `ref` is to give the function a variable to look at and change. For an "output" parameter, we'll use `ref` to give it an empty variable, whose only purpose is to hold the answer.

As a warm-up, this version of `max` turns the third input into the answer:

```
void max(float a, float b, ref float largest) {
  if(a>b) largest=a;
  else largest=b;
}
```

We could use it with `float c=0; largest(4,7, ref c);`, which would change `c` to 7. Notice how it never looks at `largest`. All it does is put the answer there. That's why we call it an output parameter.

A note: notice how 4 and 7 are fine to go into normal floats. The system converts them to 4.0 and 7.0, the same as always. But the `ref` input `c` had to be a float variable. It would be an error if `c` was an int.

This one is longer (it's an excuse to sneak in some programming and string tricks.) Suppose we have a string with a comma between two numbers, like `"2,7"` and we want a function to convert that into 2 and 7. It will have one string input, and 2 int outputs, using the `ref` trick. Here's the heading:

```
void parseXcommaY(string xCy, ref int x, ref int y) { ... }
```

I used "parse" in the name since that's a technical term for analyzing something and breaking it into parts.

It's not written yet, but a call would look like this:

```
int n1=-1, n2=-1; // outputs
parseXcommaY( "2,7", ref n1, ref n2);
// now n1 is 2, n2 is 7
```

In our minds, `"2,7"` is the only input. Technically `n1` and `n2` are inputs, but we know they count as outputs, and the values in them are ignored.

Writing it is fun. You've seen these string functions before, and the convert, but no big deal if you don't completely remember them. The part that matters is how we set `x` and `y` at the end:

```
void parseXcommaY (string xCy, ref int x, ref int y) {
  int commPos = xCy.IndexOf(',');
  string wx=xCy.Substring(0,commaPos); // before comma
  string wy=xCy.Substring(commaPos+1); // after comma
  // fill in the answer variables:
  x = (float)System.Convert.ToInt32(wx);
  y = (float)System.Convert.ToInt32(wy);
}
```

Again, notice we never read from `x` or `y`. We only use them once, to put the answers into.

This crashes if the input is in the wrong format (no commas, not numbers, extra commas.) For real we'd make it twice as long, checking for problems.

34.2.1 `out` shortcut

We've seen the two ways we use reference parameters: "fix me" parameters or output-only's. Most languages use the same reference feature for both. `C#` decided to split them into two words `ref`, which we've been using, and `out`.

This is a trivial detail not worth knowing – using `ref` for everything is easier. But, it's a good excuse for an example:

This function has one real input, `num`, and gives the int part and the fraction part. For example 4.6 gives answers 4 and 0.6:

```
void wholeAndFraction(float num, out int whole, out float fraction) {
  int n = (int)(num); // ex: (int)5.8 is 5
  float f = num-n;
  // copy into answer variables:
  whole=n;
  fraction=f;
}
```

It uses `out` instead of `ref` but it works the same.

We'd call it the same as before: the real input, and the output variables:

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

So far, it's exactly the same as `ref`. The only difference is the error messages. Since `out` is for output only, they made it a helpful error to try to read from them in the function. But you're allowed to pass them uninitialized – in the sample call, not giving `n1` and `f1` starting values is legal. They even made it a helpful reminder error if the function doesn't assign to every `out` parameter.

Again, don't bother remembering this – just use `ref` for everything. And you won't even use it much. `out` is really an example of how humans pick-and-choose during language design.

34.3 mixing real return and `ref` parms

It seems obvious you shouldn't mix `return` and reference parameters. In other words, if you have two answers, it would be weird to `return` one and use `ref` for the other.

But there is one time when we use both ways: when there are real answers and a "did it work?" answer. Then the style is to return whether it worked, and pass back the actual answers by reference. This is the same idea as the "do stuff and return whether it worked" functions.

Built-in `int.TryParse` works this way. It converts a string into a number, and returns whether it worked (it's a better version of `System.Convert.ToInt32`).

It runs like this:

```
int num;
bool didItWork = int.TryParse(w, out num);

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

The main thing it does is convert `w` into a number, putting it into `num`. As a secondary thing it returns true if it worked. `int.TryParse("3cat",out q);` returns false and `q` is junk.

Most people putting the call directly inside the `if`. This is total abuse and horribly confusing if you don't realize we're mostly running TryParse for the side-effect:

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

One comment about the command: `int.TryParse`? So this gets to use `int` as a namespace? I guess they can do that since they wrote the language.

But it's nice since we can now type int-dot and browse for helpful int functions. We can guess float-dot has some – for example it also has a TryParse.

But how come it's `System.Convert.ToInt32`? Why is `ConvertTo32` in the Convert namespace inside of System? Why isn't it also in `int`? It's because computer languages grow in funny ways. It got in the Convert folder before they decided to use `int` that way, and they didn't want to break old programs by moving it.

My `parseXcommaY` example can be rewritten to return whether it worked. Recall the first version turned `"2,7"` into 2 and 7, but crashed on bad input. This one does all the error checking.

Warning, it's long. But you can probably get the idea:

```
bool parseXcommaY(string xCy, ref int x, ref int y) { // <-new bool return
  int commaPos = xCy.IndexOf(',');
  if(commaPos<0) return false; // no comma
  string wx=xCy.Substring(0,commaPos);
  string wy=xCy.Substring(commaPos+1);
  // NOTE:  x and y are our answers. Changes go all the way back:
  if(!int.TryParse(wx, out x)) return false;
  if(!int.TryParse(wy, out y)) return false;
  return true;
}
```

The last two lines are extra tricky. If TryParse can't decode either number, we should return false. That's not too bad.

But notice how we give TryParse our `x` and `y`. That's super-sneaky. It changes them in us, by reference, but since we're also a reference, they're getting changed way back in the caller. In other words, a 2-function call-by-reference chain works.

Even if you hate that, what matters is we can write:

```
int num1=-1, num2=-1;
if(parseXcommaY(w, ref num1, ref num2)) {
  // do stuff with num1 and num2
}
else print("w has incorrect format");
```

We're still basically calling it to pull out the 2 values, with an added "did it work?" check.

34.4 Pointers and References

We already know functions with pointer inputs can use them to change the original, so it seems like you never need `ref` with pointers. It turns out that

`ref` is better than pointers, and sometimes even pointers need it.

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
```

`swap(g1, g2);` runs it, which does nothing.

The trick is to remember we think of `g1` as a goat, but it's really two things: a nameless heap goat, and our local variable `g1` pointing at it. A function can use a pointer to change the actual goat, but can't change our local `g1` variable without using call-by-ref.

A picture:

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

`a=b;` just makes local `a` point to the second goat. After the dance inside of swap, `a` and `b` cross to aim at each other goat's, but `g1` and `g2` are unchanged.

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, and handle `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
}
```

If the input goat was `null`, this creates a goat, but we have no way to hook the original goat to it – that one will still be null. `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
```

The `ref` allows the function to reach back and change `gg1` when it makes `g` point to the freshly created goat.

This last thing is a fun note, but not so important. Back in the swap, I flipped where the pointers went. In the picture, `g1` and `g2` are crossed after the swap.

Another way to swap goats is to keep the arrows the same, and swap the pointers. This version is more work, but doesn't need call-by-reference:

```
void swap(Goat a, Goat b) { // keep pointers same, swap contents
  string wTmp=a.name; a.name=b.name; b.name=wTmp;
  int tmp=a.age; a.age=b.age; b.age=tmp;
}
```

34.5 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. So it makes a nice example.

This is long and ugly, and not vital to know. Even if you want to use a Raycast for real you can just copy one from the internet. But if you read casually, skipping the parts that hurt your head, there's lots of cool stuff in how raycast is written.

A raycast follows an imaginary ray (a start point, and a direction) through the Scene and checks what it hits. A common use is firing a laser – start at the tip of the laser cannon, go in the direction it's aimed and see what it would hit.

A nice thing about a raycast is it's mostly a pure math function. It uses globals (all of the objects in the Scene,) but it doesn't draw anything, or do anything to what it hits. It just computes things.

With all the combinations of overloads and default parameters, there are 16 versions of it!. Most of them are mostly simplifications of this monster:

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

Right away, we can ignore the last three – they use `=` which means they have default parameters, which means you're supposed to skip them most of the time. That leaves us with this version:

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

The first two inputs are the imaginary ray we're shooting. I like how they're both `Vector3`'s, but they mean different things. The first one is a position

where it starts, but the second is the direction it goes – for example `(1,0,0)` means to go right, not to go through (1,0,0).

The third input isn't as bad as it looks. It's an `out` parameter, which means we just give it an empty variable. The type is a struct named `RaycastHit`. As usual, `hitInfo` means nothing – it's just a variable name they picked.

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
```

But a raycast could also hit nothing. It uses the "output parameter + return if it worked" method. In this case, `Raycast` returns true if it hit something, false if it hits nothing. We usually put them inside of an `if`:

```
if(Physics.Raycast(pos, dir, out RH)) { // fills in RH
  print("hit location is " + RH.point);
  print("hit object is " + RH.transform.name);
}
else { print("missed"); }
```

`RaycastHit` is a specially made struct, whose only purpose is to hold results of a raycast. It's fine to make a struct for just one thing, if it makes the program look nicer.

I'm using two of the fields: `point` – where we hit; and `transform` – what we hit. Don't let `RH.transform` scare you. They just picked `transform` for the field name of what we hit. As usual, it has nothing to do with our personal `transform` variable.

I'll give a useful `Raycast` example later. But first let's look at some of the other 15 versions of `Raycast`:

The rule for Default parameters says we have to fill them from left-to-right. `maxDistance` is first, so we can fill in only that, leaving the next two blank. This checks for obstacles within 20 units – if a block was in front, but 24 units away, it would say `false`:

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

The next item in the default parameters is LayerMask. Suppose we know what a layerMask is, and want to use `LM` for it. The default parm rules say we have to fill in distance first, even if we don't care. I'll use just 9999 for distance:

```
// WRONG, will use LM for maxDistance:
if(Physics.Raycast(pos, dir, out RH, LM)) {

// this works, fill in fake distance:
if(Physics.Raycast(pos, dir, out RH, 9999, LM)) {
```

We still don't know what layerMasks are, but we know enough about function calls to know this is how RayCasts would use them.

There's another overload in that massive list with only 2 inputs. It's for when you don't care what you hit. It still returns true/false, but you skip the Output parameter. It works like a pure `bool` function:

```
if(Physics.Raycast(pos, dir)) print("blocked");
else print("nothing in front of me");
```

That version uses the same default parameters, so you can check if something is 10 meters in front of you:

```
if(Physics.Raycast(pos, dir, 10)) print("blocked within 10");
else print("clear within 10");
```

The list has another set of overloads using a `Ray`. Since we often use `pos` and `dir` together to make an arrow, Unity combined them into a struct named `Ray`. This is all it is:

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

If you happen to already have a `Ray`, you can use it in a raycast version:

```
Ray rr; // set up the ray:
rr.origin = new Vector3(0,0,-10); // same #'s as before, but
rr.direction = new Vector3(0,0,1); // now in a Ray

RaycastHit RH; // will hold output
if(Physics.Raycast(rr, out RH)) { ... }
```

Notice how this is the exact same 2 inputs as the first example. It just happens they're both inside of `rr`. If you use the same position and direction together lots of times, being able to combine them in a Ray is just a nice option.

The last default parameter is an example of the enumerated type trick. Unity has invisible boxes named *triggers* that don't block motion, but can detect when something enters them. They're common for things like health or ammo pickups, or flame zones.

In the past, Unity had one global setting for whether all raycasts hit triggers or skipped through them. It starts as "ignore" – a health pickup doesn't block the path. Newer versions let each raycast say whether it hits or skips triggers.

Here's the cool part: raycast trigger interaction now has 3 options: hit them, skip them, or use the old global setting to decide. The last option is good for upgraded older projects – it's "uh, do it the way I was doing it before."

For those three options they made an enumerated type. This is only used in raycasts:

```
enum QueryTriggerInteraction {UseGlobal, Ignore, Collide};
```

So the last monster parameter:
`QueryTriggerInteraction queryTriggerInteraction=QueryTriggerInteraction.UseGlobal`
is telling you it will use the system setting if you don't say. But you can give it `QueryTriggerInteraction.Collide` if you want to be sure this particular raycast hits triggers.

But the important thing is, you could skip all this, know nothing about triggers, leave out that input, and the raycast would work the way it should. That's the purpose of default parameters.

Here's a semi-useful script you can test raycast with. The player moves along the bottom of the screen, shooting a raycast straight up. Whatever it hits turns red:

```
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;
  }
}
```

```
// easy way to turn stuff red, then back:
// (doesn't save the old color, but good enough for testing)
void setCol(Transform tr, bool isOn) {
  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);
}
```

This would go on a Cube. Moving with a/s should turn things red then back. For fun, put one Cube above another and partway covered. The raycast hits the first thing along the line. Or you could Play then go to Scene mode and drag things through the ray.

34.6 Reference vs. reference?

You may have noticed that we now have two different uses of the word reference: call-by-reference, and also reference types. The way reference got double-used is a funny story:

The original use of the word is call-by-reference as compared to call-by-value. A reference is a locked-on alias for another variable.

Java invented the term Reference Type. It was the first commonly used language to break variable types into "can't use pointers" and "must use pointers."

This re-use of the word seemed fine, since Java doesn't have call-by-reference, so no confusion there. Things like `Dog d1;` are sort of between real pointers and real references. Pointer is considered a scary computer word. So they called `d1` a reference to a Dog, and that seems fine for Java. I personally would have made up a word like Refters or Poinerences. Oh, well.

`C#` was based on Java. In fact, it was originally advertised as a version of Java, for windows. It had to use the term Reference Type for classes. Then they wanted to add call-by-reference back into the language. That's such a standard computer term they had to use the correct name. So `C#` is stuck with two different meaning for the word reference.

The story of why Java doesn't have call-by-ref and `C#` does is another fun

language design issue. As I wrote, you rarely need call-by-reference, and there's always a work-around. And it's one more thing the compiler has to deal with.

Java thinks removing a rarely used feature is better and results in simpler, easier to read code. `C#` thinks more options are always better. People argue about it online.