Chapter 25

Pointers in Unity

So far we've been pushing around Cubes using the "do stuff to myself" shortcut. Now we're ready for the real programming way - reach through a pointer.

We know Unity `new`'s and owns all of those Cubes. But if we can declare a pointer of whatever type a Cube is, and somehow convince Unity to give us a link, we're done. We can use that pointer to move and change it.

If we need to move 3 other pre-made Cubes, we should be able to do the trick three times, using `c1`, `c2` and `c3` as links to the Cubes in question.

It's a semi-new use of pointers. The system creates and owns them. Our `c1` is technically a true pointer – a second alternate arrow to some Cube. But it's not alternate to us – in our script it's the only way to find that Cube, and we're never going to change where it points.

In code broken into parts, which most code is, the "real pointer, just for this part" trick is common.

25.0.1 Pointers to existing Cubes

In Unity the class for things you see is `GameObject`. Cubes and balls, as well as cameras and lights, are declared as GameObjects.

On purpose, GameObject is a class. So a `GameObject` variable is a pointer. If we put it in the Inspector using `public`, we can point it to something by dragging.

This next program uses `g` to reach out and move something. The script needs to be on something to run, but it doesn't matter what. It's not moving itself. The first part:

```
public GameObject g; // drag in a Cube

void Start() {
```

As usual, you select the object with the script on it, and look in the Inspector. You should see `GameObject(none)` after `g`. That's Unity's way of saying it's

currently `null` and reminding us what type is it.

Dragging any Cube into the slot assigns `g` to point there. The slot should change to the Cube's name. Clicking the slot should make the linked Cube pulse and flash yellow.

Here's the whole thing:

```
public GameObject g; // drag in a Cube

void Start() {
  g.transform.position = new Vector3(0,4,0); // just some spot
  g.GetComponent<Renderer>().material.color = new Color(1,1,0); // yellow
}
```

Notice how the first part looks like pointer use: g-dot means to follow `g` to the real gameObject, then to look at the field. When we look at the back half, those are the commands we've been using all along – change position and color.

One funny thing is going on here, because we were previously using change-me shortcuts. The thing we're on is also a GameObject, and we get a free pointer to it named lower-case `gameObject`. Here are the commands, written without change-me shortcuts:

```
gameObject.transform.position = ... // change where I am
g.transform.position = // change where g is

gameObject.GetComponenet<Renderer>().material... // change my color
g.GetComponenet<Renderer>().material... // change g's color
```

Written out like this, it should be easy to see we were always following a pointer. `gameObject` is the free one, to "us". Our new code is just switching it to use a different pointer.

Another fun thing: to show GameObject is just a class, we could write `g=new GameObject();` in Start. We'd see it in Unity's list, at the bottom. The line coloring it would cause an error (there's no Cube - adding it would be several more steps.) It's almost always easier to pre-make Cubes. And we wouldn't be able to find the original dragged-in `g` any more.

But the line `g=new GameObject();` would totally run.

Here's another example using two pointers. You'd drag a different thing onto each (I named then `cube1` and `ball1` for fun, but dragging in 2 Cubes works fine):

```
public GameObject cube1, ball1; // drag in 2 cubes/spheres/etc

void Start() {
```

```
  // reach through each to change colors:
  cube1.GetComponent<Renderer>().material.color = new Color(0, 1, 0); // green
  ball1.GetComponent<Renderer>().material.color = new Color(0, 0, 1); // blue
}
```

Some notes, in no special order:

- Dragging to set these up seems like totally cheating. A "real" program would use code, like `cube1=[fancy pointer lookup]`. We could do that (later). But lots of GUI code editors let you use this trick.
- Even worse, it looks like we're assigning values before the variables are even declared. But it's the same Inspector trick as `public int x;`. When you press Play, everything gets declared, then Unity jumps in and copies all Inspector values and links, then it runs Start.

  There really is a secret "`cube1=[use dragged data to find actual cube]`" in your program.
- You could drag yourself into `cube1`. It will work fine (but only if you have a color.) It has to point to a Cube. If you're a Cube, no problem.
- Totally silly, but as a check, you could drag the same Cube into both slots. It turns blue (green, then blue; which is blue.)
- Running with nothing in a slot gives a standard null reference exception error. It tries to follow the pointer and can't. If only `cube1` was empty, neither would change color – Start's first line crashes and quits.
- After dragging something in, you can set a slot back to null by clicking the far-right double-circle, then (none) from the top of the pop-up.

To get a feel these are really pointers, we can create one more GameObject pointer and use it to change both:

```
public GameObject cube1, ball1; // drag in 2 cubes/spheres/etc

void Start() {
  GameObject p=cube1; // <- new line
  p.GetComponent<Renderer>().material.color = new Color(0, 1, 0); // green
  p=ball1;
  p.GetComponent<Renderer>().material.color = new Color(0, 0, 1); // blue
}
```

We saw this same trick with Dogs. `cube1` and `ball1` are acting like real GameObjects, while `p` is a temp pointer that can aim at either. We changed `p`'s color twice, but it was aimed at different things.

A fun example which moves two Cubes at different speeds:

```
public GameObject c1, c2; // drag in two Cubes
int x1=-7, x2=-7;

void Update() {
  x1+=0.1f; if(x1>7) x1=-7; // standard move&wrap
  x2+=0.07f; if(x2>7) x2=-7;

  c1.transform.position = new Vector3(x1, 2, 0); // different y's
  c2.transform.position = new Vector3(x2, -1, 0);
}
```

This is nothing new, except it's neat to have one script run a Cube race. "Unity style" would be to have a script on each cube. This is more like what a standard programmer would write (but either way works as well.)

As promised, we can set-up pointers-to-Cubes completely using code. It's not as good, but it makes a nice example.

`GameObject.Find("ball1")` searches through all the names (the ones in Hierarchy) for ball1, and returns a pointer. It says it returns a `GameObject`, which we know means a pointer to one. We could change the program above to always move "ball1" and "ball2":

```
GameObject c1, c2; // no need for public, since we look them up ourselves
// ...

void Start() {
  c1=GameObject.Find("ball1"); // <- aim c1 at ball1
  c2=GameObject.Find("ball2"); // <- aim c2 at ball2
}
// Update is the same as before
```

To compare: before we would have dragged the thing named "ball1" into `c1`. Now we run `Find`, on "ball1". Either way, `c1` is hooked to it, so runs the same.

Some GameObject.Find notes:

- Notice how it's the `Find` command in the `GameObject` namespace. This is another example of re-using a class as a namespace.
- Wait, so is there a `Dog.Find` command? No. This is a total Unity hack. Every GameObject is always added to the list, even ones we make with `new`. That has to happen so they can be automatically displayed by the system.
- What happens if two things have the same name? You probably get the first one. The thing is, if you were worried, you would have used the drag-in trick. This command is like an alternate.

- Can you look them up by index? In theory they could have given that option, but they didn't. You normally move them around, which changes the numbers but not the name. And, again, you're supposed to be using the drag-in trick anyway.

Finally, let's look at `null` in Unity. If something isn't dragged in, our pointer will be `null`. `Find` returns `null` if it can't find the name. This program hopes you drag something in. If not, it guesses some names, then gives up:

```
public GameObject luckyCube; // drag something in?

void Start() {
  if(luckyCube==null) // they forgot to drag something in
    luckyCube=GameObject.Find("clover");
  if(luckyCube==null) // we couldn't find "clover"
    luckyCube=GameObject.Find("rabbitFoot");

  if(luckyCube!=null)
    luckyCube.GetComponent<Renderer>().material.color = new Color(0,1,0);
  // if it's still null, we're just giving up on it
}
```

`null` is the perfect return value for "can't find it". Functions looking for an index return -1, which seems fine since that's never a legal index. But `null` is even better – its only possible meaning ever is "no such thing".

25.1 Instantiate

Unity has a specialized copy function named `Instantiate`. Here's a check whether you're getting the idea of pointers: after it copies the object, it returns . . . ?

Clearly it returns a pointer to what it just made. Copying something is really a fancy `new` (a clone function, but not important if you hated that example.) When you make something, you have to return a pointer to it so they can use it.

25.1.1 Instantiate as a copy

Here's some simple code using it. Warning, warning, warning: do not use this to copy yourself. The copy will run its copy of the script, copying itself, and so on forever:

```
public GameObject ball1; // drag in some other item

void Start() {
  GameObject bb = Instantiate(ball1);
```

```
  // bb is now aimed at the freshly made copy of ball1

  bb.transform.position = new Vector3(-3, 0, 0);
  bb.GetComponent<Renderer>().material.color = new Color(1,0,0);
}
```

`Instantiate` makes an exact copy of `ball1`, including shape, size and even any scripts on it. `bb` is pointing to it. One thing we never had to worry about with other copies – the new `bb` is in the same spot as `ball1`. So we move it.

Notice how `bb.transform.position=` is the same line we used to move `g` in the previous examples. A pointer is a pointer, no matter how you got it.

The last line changes color, just to show we can, and to make it stand out.

Unity is set up to reset everything when you Stop. In this case, everything you Instantiated goes away.

This next program looks a little cooler, but isn't doing anything new. It makes a new ball when we press space:

```
public GameObject ball1; // drag in some other item

void Update() {
  if(Input.GetKeyDown(KeyCode.Space) {
    GameObject newBall = Instantiate(ball1);

    // move to random spot on screen:
    float x=Random.Range(-7.0f, 7.0f);
    float y=Random.Range(-4.0f, 4.0f);
    newBall.transform.position = new Vector3(x, y, 0);
  }
}
```

One odd thing about this is we don't save the pointers. `newBall` is a local which goes away when the `if` ends. The balls we made are lost to us. Because we're working inside of Unity, that's fine if we never plan to change them.

Instantiate is also overloaded. It makes a nice example. After copying something, you almost always want to move it. To be nice, Unity wrote a "copy and move here" function, which they also named Instantiate. It takes a `Vector3` as the second input.

Here's the program above, rewritten using the 2-input instantiate:

```
public GameObject ball1; // drag in some other item

void Update() {
  if(Input.GetKeyDown(KeyCode.Space) {
    Vector3 newBallPos; // <- pre-make target position
```

```
    newBallPos.x=Random.Range(-7.0f, 7.0f);
    newBallPos.y=Random.Range(-4.0f, 4.0f);
    newBallPos.z=0;

    GameObject newBall = Instantiate(ball1, newBallPos); // <-copy and move
    // or:  Instantiate(ball1, newBallPos);
  }
}
```

We don't need `newBall` to move it anymore – we don't need `newBall` at all. You're allowed to ignore the return value of functions, so we could have called just `Instantiate` with nothing in front.

I'm leaving `newBall` there since we may later want to change, for example, the color.

You might notice how Unity names the new items "Cube (Clone)" or something like that. This is no big deal. It's the same as when you copy a file and get a 2 on the end, or a "Copy of" in front.

Unity decides to change the name, since why not. And it figures you know programming a little, so you'll understand that "Clone" means you made it with something which is basically a Clone command. Of course, like any other copy, it's a completely separate object.

For fun, you could change the name after making it:

```
GameObject newBall = Instantiate(ball1, newBallPos);
newBall.transform.name = "ballA"; // we used name once before
```

25.1.2 Proper Unity use of Instantiate

This entire section has nothing to do with programming. But if you use Instantiate with Unity for real it's probably good to know.

We almost always want to make things out of nothing – like shooting balls, or leaving footprints. That seems completely different, but it's still a simple copy. The trick is to have the original tucked away somewhere no one ever sees it, preferably out-of-the-game.

Unity calls these hidden originals **prefab**'s, short for prefabricated object. To make one, create something as normal, drag it down into the Project section (the same place your scripts live.) You should see a copy appear there, with an icon symbol of a little corner-on cube. Then you can delete the original (you don't have to, but you probably don't want it up there. If it's a ball you can shoot, you don't want the game to start with a stray ball in it.)

You're allowed to drag those prefabs into variables:

```
public GameObject ball; // drag in prefab from Project area

void Start() {
  GameObject newCube=Instantiate(ball);
```

This is the same code as before. The only difference is `ball` is linked to a prefab. The old way, one ball turned into two. This new way a ball appears to pop up out of nowhere. The difference is whether the original was hidden or not.

In real code `Instantiate` should always copy a prefab.

It can be a little confusing how "move me" and "copy me" links are both GameObjects. But we're used to using one type for very different things. This uses both. Hopefully you can tell the difference:

```
public GameObject ball1; // drag in a real ball
public GameObject popupBallPF; // drag in a prefab

void Start() {
  ball1.transform.position = ... // move the real ball
  GameObject newBall = Instantiate(popupBallPF); // copy the hidden prefab
}
```

As a fun note, Instantiating prefabs counts as the Prototype design pattern, if you like design patterns. We could have had a command like createShape, with inputs being the size, color, shape .... Prototype says to pre-make a sample of what you want, probably off-line. So the prefab is the prototype.

There's another use of prefabs which has no relation to programming at all, but we may as well see it while we're here.

Suppose we need to position a bunch of mailboxes (tall blue cubes.) We'd make one, then drag it to make a prefab. Now you can easily make copies by dragging that prefab mailbox in, where-ever you want it.

Another cute feature, suppose you need them to all be a little taller. Growing the prefab magically grows every real mailbox created from it.

This can be confusing. The linked-copy trick is only a manual editing shortcut, and only works when Play is turned off. Things you make with instantiate are completely separate things, with no funny change-one-change-all tricks.

25.1.3 Variable GameObject pointers

This is a longer example of using a `GameObject` variables as a real pointer. This script moves an object across from left to right, using pointer `currentBall`. On each wrap-around, it randomly picks ball1 or ball2 (it might take a few trips for it to switch to ball2).

It looks nicest if you can tell them apart, like a cube and a ball, or different colors:

```
public GameObject ball1, ball2; // point these at two different-looking objects
GameObject currentBall; // pointer

Vector3 pos; // for currentBall, whichever ball it is

void Start() {
  // start on ball1:
  currentBall = ball1;

  pos.y=0; pos.z=0; pos.x=-7; // middle-left
}

void Update() {
  pos.x += 0.1f;
  currentBall.transform.position = pos;

  if(pos.x>7) { // wrap:
    pos.x=-7;
    pos.y=Random.Range(-3.0f, 3.0f); // pick new y
    // randomly pick ball1 or 2:
    if( Random.Range(0.0f, 1.0f)<0.5f ) // coin flip
      currentBall=ball1;
    else
      currentBall=ball2;
  }
}
```

Notice how only `currentBall` is being moved, which is always really ball1 or 2. It's like the `activeDog` example from last chapter.

25.1.4 Pointers for "caching"

This is a shortcut that won't make your program any better, but you'll see it occasionally and it makes a nice example. It's got some just-because stuff brought in which I won't be using later.

You may have noticed `GetComponent<Renderer>().material.color` is a 3-part lookup because of the 2 dots. It finds the renderer, then than material, then the color. Color is a struct, but the other two are classes, which means we can save a pointer to them.

The type of `material` is `Material`. It just is. So we can declare a variable of that type for the shortcut:

```
Material myMat; // will be a short-cut to our Material
```

```
void Start() {
  // notice this is the first part of the real color command:
  myMat = GetComponent<Renderer>().material;
}

void Update() {
    // ...
    myMat.color = new Color( ... );
  }
}
```

That last line jumps straight to our material, then finds the color, the same as before. The only thing it accomplishes is to shorten the command. And, more importantly, showing we can save a pointer to anything that's a class.

A little longer version, suppose we have a link to the floor. If we change the color often we could create a shortcut for color changing:

```
public GameObject floor; // drag the floor into this
Material floorMat; // saved shortcut for changing floor color

void Start() {
  // hook up floor color shortcut:
  floorMat = floor.GetComponent<Renderer>().material;
}

void Update() {
    // ...
    // move and color floor:
    floor.transform.position = ...
    floorMat.color =  ... // <- using our shortcut pointer
}
```

Again, not really a useful trick at this point – just a cute example if it makes sense to you.

25.2 New'ing Inspector variables

This is one more thing that seems funny using Unity. Our classes need to be created with `new`, but anything in the Inspector comes with slots for us to fill in, like it already exists.

Clearly, Unity is giving Inspector pointers a free `new`. For example:

```
// recall we need this line to make it display:
[System.Serializable]
```

```
public class Dog {
  public string name;
  public int age;
}

public Dog dg; // now dg is in the Inspector

void Start() {
  dg.age=7; // legal, it's been new'd
}
```

In the Inspector for `dg`, we may have entered `"Spot"` and 4. When this runs, the system `new`'s `dg` for us and jams those values into it. We could use `new` again, but there's no need and we'd lose the starting values.

So the rule about needing a `new` isn't being broken – it just looks that way.

But wait, `public GameObject g;` can be `null` – the system isn't `new`'ing it for us.

That's just more hidden programming. The system keys off of `[System.Serializable]` and also what type of built-in it is. It's programmed to have your public GameObjects act like pointers, and not owners. In other words, it assumes you never want to type in all the values to make a Cube when you can just click to make one easier.

But what if you want a public Dog which links to some other pre-made Dog? You can do it, but you won't get any extra help from Unity (we need the "find another script" trick, which we'll see later.)