Chapter 25

Pointers in Unity

25.1 Using 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. We merely need to make a pointer variable of the right type, and convince Unity to link it. It's like we have only `Dog p;`. Then Unity creates `Dog pet1 = new Dog();` and hooks up our `p` to `pet1`. Our only dog variable, `p` is our personal link.

This chapter has one more semi-common shortcut trick known sometimes as caching. It's not important, but you'll see it a lot and it also makes a nice example.

25.1.1 Pointers to existing Cubes

In Unity the main 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. We still need to drag the script into something in order to make it run, but it doesn't matter what, since we're not using the "change me" trick. 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 `Cube2` (whatever the name is). 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.

Here's another using two pointers. You'd drag a different thing onto each (I named then `cube1` and `ball1` for fun – they can be anything):

```
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 actually can do that, later. But this is no worse than the number Inspector trick.
- You could drag yourself into one of your slots and it will work fine. For example, the script could be on a Cube: you can drag yourself into `cube1` and something else into `ball1`.
- Totally silly, but as a check, you could drag yourself into both slots. It will turn yourself blue (as usual, it turns you green, then blue; then you see the result, which is blue.)
- Running with nothing in the 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.
- Just to mess around, we could write `g=new GameObject();` as the first thing in Start. That would make an empty gameObject. We'd even see

in the list, at the bottom. Coloring it would cause an error, since there's nothing to color. Since we moved `g` from it, we'd never be able to find the original Cube we linked.

There's no reason to ever do that, but it proves `GameObject` is just a class.

- Wait, why did `transform.position=` grow to become `g.transform.position=`? That's not like the old rules where it's `pet1.age` or `p.age`.

  That's what I meant about how we've been using a shortcut before. Officially you write `this.transform.position=` to move yourself, where `this` is a special pointer to "me." That's messy, complicated, and no one does it. But you can see how we're using proper pointer logic, changing `this` to `cube1`.

To get a feel these are really pointers, we can rewrite using this old trick to color them:

```
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
}
```

This looks like both kinds of pointers. `p` is clearly a true pointer, who's only job is to aim at some other real Dog. `cube1` and `ball1` feel like regular variables. They're both our permanent links to those things.

Here's one more example which moves two Cubes at different speeds.

```
public GameObject c1, c2;
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 is kind of neat to have one script able to run a Cube race. "Unity style" would be to have a script on each cube, moving only itself. This is more like what a standard programmer would write (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 fill them in
// ...

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 the Find command, which does the same thing.

A fun, non-pointer note about `GameObject.Find`. If you recall, the dot in the middle means the function is named `Find` and is in the `GameObject` namespace. It's the same trick as `Vector3.Distance`. GameObject is a class, but we can also re-use it as a namespace for gaemObject-related functions.

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;

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
}
```

25.2 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.

25.2.1 Instantiate as a copy

Here's some simple code using it. Warning, warning, warning: do not use this to copy yourself. The copy will copy 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);
}
```

It's a completely boring Clone function. But it's pretty cool how Unity knows to copy the size and color; and the way it adds the copy to the scene; and how `ball1` was just there, waiting to be copied.

When you stop, all the copies go away.

This 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. To our part of the program, the balls we made are lost, since we don't have a pointer to them.

The way we can still see them isn't a contradiction of the rules - it's just the way running inside Unity changes them a little. The system grabs and saves

things we make with Instantiate, since it was programmed that way.

Instantiate also makes a nice example of an overloaded function. Most people copy something, then move it on the next line. 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 other instantiate:

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

void Update() {
  if(Input.GetKeyDown(KeyCode.Space) {
    Vector3 newBallPos; // <- need to pre-make position, here
    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
  }
}
```

A funny thing with this one is we don't use `newBall` for anything. If we wanted to change the color, we'd need it, but in this case we don't.

If you remember, you're allowed to ignore a return value: `max(4,8)` is legal, but pointless. But `Instantiate(ball1, newBallPos);` by itself makes sense. We get a copy, and don't need the pointer.

Notes about names of copied stuff:

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 of the new thing, just because. And it figures you know programming a little, so you'll understand "Clone." But it doesn't mean anything. 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";
```

25.2.2 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 even 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. In real code `Instantiate` should probably always work like that and copy a prefab.

Both things are gameObjects, so it can be a little confusing. In this code both things are GameObjects, but different kinds. 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
}
```

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 short-cut, 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.

25.2.3 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;

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;
  }
}
```

This is mostly the same as the `activeDog` example. `ball1` and `ball2` act like real balls, but `currentBall` is merely a ball selector.

25.2.4 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( ... );
  }
}
```

A little longer version, suppose we have a link to the floor. If we change the color often we can 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 =  ...
}
```

25.3 New'ing Inspector variables

Unity has a special rule that anything in the Inspector gets `new`'d, for free. 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
}
```

The idea is, we can fill in `dg`'s name and age in the Inspector. But the only possible way to copy those into `dg` before the program starts is to have a real Dog, which requires a `new`.

Besides that, everything else is the same. As usual, you can move or change it all later.