Chapter 30

Member functions

When we make a class, we often write some helpful functions using it. For the `Dog` class we had `setDog(d1,"Spike",2)` and `showDog(d1);`. In our minds, those were `Dog` functions.

Member functions is a trick to make them feel more like Dog functions. We can have `d1.set("Spike",2);` and `d1.show();`, and it feels like `d1` is setting itself, or running its personal `show` function.

This is really the other half of how to make a class or struct. After we add fields, we also add functions. Both are used with a dot.

Member functions don't actually do anything new. We're merely moving around where we define some functions and changing the way we call them. This might feel like a lot of rules for not much gain. But it's a nice way to organize things, all of the built-in classes and structs use them, and we'll need them for some Object-Oriented tricks later on.

30.1 Example and motivation

Here's a rewrite of the old setupDog function as a member function:

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

  // this is a member function:
  public void setup(string nm, int howOld) {
    name=nm; age=howOld; // <- name, age? By themselves?
  }
}
```

The basic rule is that we can use `name` and `age` directly. We learned you can never do that – you have to say which Dog it is first. But member functions have that built into the call. `pet2.setup("Spot",5)` remembers `pet2` called

it. Clearly `name` is the name of `pet2`. If we run it again with `d1.setup("",0)`, we're now clearly using `d1`'s fields.

It's a very clever trick. You have to call them using Dog-dot – that's a rule. So a member function is guaranteed to be running for one Dog. It gets to use that Dog's fields in a simple, fast way.

There's nothing special showing how `setup` is a member function. Functions written inside of a class are automatically member functions. As we can see, they use parameters the normal way. They can return things. They have only that one extra rule: always running "for" some Dog.

It helps to see why we're doing all this work. Some of the advantages of having member functions:

- Putting `d1`-dot in front makes it easier to see this is a `Dog` function when we're reading it.
- It also makes it easier to see how `("Spot",2)` is the real input. `d1.setup("Spot",2);` makes it look more like an assignment statement, which it mostly is.
- When we're writing code, it's easier to find the `Dog` functions. We can type `d1`-dot and search the pop-up: the same pop-up that shows the fields also shows the member functions.
- We can use shorter function names. As a regular function, `setupDog` seemed good. But as a member function, `setup` is fine, since we know we're inside `Dog`.
- The "using this dog" `name`/`age` shortcut can make the inside of member functions a little nicer.

These little things add up. Once you know how, instead of writing a helpful function, you write a helpful member function.

Here are the member function rules listed out:

- Member functions are written inside the class. The extra word `public` must be in front, but otherwise they look normal.
- They can only be run by a variable of that type, using variable-dot-function. Like `d1.setup("Spot",2)`. Trying to run them by themself is an error. `setup("Gabby",6);` is no good.
- The variable that runs them is an automatic input. It doesn't have a name inside the function. Instead you use its member variables "naked." In a dog member function, just `name` and `age` means the name and age of the dog that called you.

- Member functions, even in structs, are allowed to change the parts of what called them. For example `frog1.doFrogStuff();` could be changing variables inside of `frog1`.
- The order you write them doesn't matter. Member functions can come before, after or in-between member variables.
- Forgetting `public` in front isn't an error, but you won't be able to use the function (it won't appear in the pop-up after `d1`-dot, and more importantly, it gives a *function not found* error if you type it anyway).

30.2 More simple examples

Dog examples

Here are some more `Dog` member functions, all with no inputs:

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

  public string desc() { return "Name: "+name+" "+age+" years old"; }

  public void reset() { name=""; age=0; }

  public bool isAPuppy() { return age<=2; }
}
```

We know they really all have 1 input – the dog that calls them.

`desc()` is the same simple "help print me" function we've written for structs before. We'd call it like `string w = dog2.desc();`. As usual, it uses `name` and `age` and knows they mean `dog2`'s name and age. Notice how it returns a value the same way as a normal function.

`reset()` is showing the rule of how you're also allowed to change the fields. `pet1.reset();` would blank out the inside of `pet1`.

`isAPuppy()` only reads `age`, which is fine. We've seen the function-in-an-if trick with regular functions. Now we can use `if(d2.isAPuppy())`.

Member functions with inputs use them in the normal way. This checks whether a dog is within an age range:

```
  public bool inAgeRange(int low, int high) {
    return age>=low && age<=high; // uses age of the calling Dog
  }
```

Now `bool adult = d2.inAgeRange(4,9))` is legal.

You're allowed to have a member function that doesn't use any fields, but you'd never do it. It would be pointlessly confusing. This terrible function picks a random Dog age:

```
class Dog {
  ... // Dog stuff

  public int randomDogAge() { return Random.Range(1,15+1); }
}
```

We be required to call it like `int n=d1.randomDogAge();`. But it doesn't use `d1` for anything, or change it. It runs the exact same no matter what Dog we give it. Why did we require you to run it from a Dog – that's just confusing. It would be better as a normal function.

The most complex member functions take another Dog as input. The standard example is copying an input Dog into ourself. `d1.copyFrom(d2)` makes `d1` a copy of `d2`:

```
  // inside of Dog:
  void copyFrom(Dog dd) {
    name = dd.name; // myName = otherDogsName
    age = dd.age; // myAge = otherDogsAge
  }
}
```

Inside, `dd.name` uses the input Dog, the same as any other function, while `name` is our name.

Here's a simple class for a 2D point with 2 member functions:

```
class Point2 {
  public float x,y;

  public void set(float xx, float yy) { x=xx; y=yy; }

  public float fromCenter() { return Mathf.Sqrt(x*x+y*y); }
}
```

`p1.set(1, 7.2f);` runs the first one. It's allowed to change the inside of `p1`. The input `xx`'s and field `x`'s can be confusing – people try all sorts of naming rules to tell inputs from fields.

`float dist=p1.fromCenter();` runs the second. Nothing special – it reads `p1`'s x and y, and returns the answer (that's the actual formula for distance).

Mover example

Way back in the struct chapter we made something to hold the four variables we need to slowly move and wrap-around. This was it:

```
struct Mover {
  public float val; // the moving value
  public float spd; // amount to change val, each update
  public float min, max; // the limits for val
}
```

That was a nice way to group them. But the code hand-moved each variable, like `m1.val+=m1.spd;`. That would be way better as a function. Now that we know member functions, it would be even better as one of those.

If you remember, `val` goes up or down by `spd`, wrapping around when it hits min or max:

```
struct Mover {
  // the same 4 variables

  public bool doMove() { // returns true if it wrapped around
    bool wrapped=false;
    val+=spd;
    if(val>max) { val=min; wrapped=true; }
    if(val<min) { val=max; wrapped=true; }
    return wrapped;
  }
}
```

`doMove` is nothing special. The math is the same as previously. But it's written next to the variables, inside the function, making it a little easier to find. `mover1.doMove();` runs it. Of course, its purpose is to change `mover1.val`.

We usually write some more helpful member functions. Setting min and max at the same time feels natural, so we should write that:

```
  public void setRange(int low, int high) { // helpful setting function
    if(low>high) low=high; // we may as well check this
    min=low; max=high; // <-setting our min and max
  }
}
```

Here's our final code to fade from red to black, 20 times in a row:

```
Mover redMove;
int laps=0;

void Start() {
```

```
  redMove.setRange(0.2f, 1.0f);
  redMove.val=1.0f; // start at highest #
  redMove.spd=-0.02f; // small since total range is only 0.8
}

void Update() {
  if(laps<20) {
    if(redMove.doMove()) laps++;
    transform.getComponent<Renderer>().material.color=
      new Color(redMove.val,0,0);
}
```

It's supposed to feel as if we're asking the `redMove` variable to run it's own `doMove` function.

BoardSquare

A few chapters ago we made a simple class to hold data for one square in a game board. That particular one allowed the user to toggle between two colors (I called that selecting a space) and it showed which space you were "on" by making it a little smaller.

We can make those member functions:

```
class BoardSquare {
  public GameObject gg; // same as before
  public bool selected; // select to flip the color

  public toggleSelect() { // flips in on/off
    selected=!selected;
    Color cc=Color.blue;
    if(selected) cc=Color.red;
    gg.GetComponent<Renderer>().material.color = cc;
  }

  public highlight(bool isOn) { // show the current square
    float sz=1.0f; if(isOn) sz=0.8f; // highlight makes us a little smaller
    Vector3 sz = new Vector3(sz, sz, sz);
    gg.transform.localScale = sz;
  }
}
```

We can now change squares like this:

```
Board[oldCol][oldRow].highlight(false); // undo old
Board[col][row].highlight(true); // select the new one
```

It looks somewhat nice – a space in the board is turning its highlight on or off. The pop-up trick also works: `Board[col][row].` (a dot) causes the member functions, including highlight, to pop-up.

When someone presses space we'd use `Board[x][y].toggleSelect();`. That handles the coloring.

In both of these examples, 95% of the work was "hey, that should be a function". The last 5% was seeing that it may as well be a member function.

30.3 Calling your own member functions

Member functions can call other member functions. They use the same short-cut rule as variables – no dot, simply write the name. Here Dog's have a `d1.superDesc()` which uses their `desc()` function:

```
class Dog {
  public string name;  public int age;
  public string desc() { return "Name: "+name+", "+age+" years old"; }

  // new superDesc:
  public string superDesc() {
    string dw = desc(); // <- call "my" desc() function
    if(name=="dog" && age==0) dw="Basic Puppy";
    return = "<<<< "+ dw +" >>>>>";
  }
}
```

We call `desc()` and it means the `desc()` of the current Dog. In other words, `d1.superDesc()` knows it's also running `desc()` for `d1`.

Here's a completely fake function that uses two Dogs – the calling Dog and an extra input Dog. It shows how the rules work even for funny stuff:

```
class Dog {
  // ...
  public string doubleSuperDog(Dog other) {
    string w1 = "dog#1 "+superDesc(); // runs my superDesc
    string w2 = " dog#2 "+other.superDesc(); // runs its superDesc
    return w1 + w2;
  }
}
```

The first call to `superDesc()` knows to use ours, which knows to use our `desc()`. The second call, `other.superDesc()`, runs on the other dog, which runs `desc()` on the other dog. The computer remembers which dog you're on, even when you flip back-and-forth.

30.4 Built-in member functions

Most of the pre-made structs and classes have member functions. Now that we know how they work, we can enter variable-dot and look for them.

30.4.1 Unity member functions

`Vector3` has a `Set` member function – we can write `Vector3 pos; pos.Set(-7,2,0);`. It's merely a shortcut for hand-setting them. It looks like this:

```
struct Vector3 {
  public float x, y, z;

  public void Set(float new_x, float new_y, float new_z) {
    x=new_x; y=new_y; z=new_z;
  }
}
```

There's nothing special about `new_x`, that's just the name they picked.

Oddly, `Color` doesn't have a `Set` function or anything like it. They didn't write one.

I used `transform.Translate(1,0,0)` several chapters ago without explaining the dot. Now we know it's a member function. `transform` is a variable (you get it for free in Unity if your script is on a gameObject). When you type `transform`-dot, you can see field `position` and member function `Translate`.

If you have `GameObject g;`, typing `g`-dot shows a big list, including one named `SetActive(bool);`. That tells us we can use `g.SetActive(false);` to hide it (and again with true to un-hide).

Put another way, we knew there had to be a way to temporarily de-activate a Cube. There is, and it's a member function.

30.4.2 String member functions

I lied when I wrote that `string` was a basic variable type. You probably figured out that `string` is way more complicated than something like `int`, `float` or `bool`. `string` is really a built-in C# class with a bunch of special rules to make it act like a basic type.

But since it's really a class, it has member functions. As usual, we can type `w`-dot to find them. Lots of them are really fun:

- `w.Contains("a")` is true when `w` has an `'a'` in it. We know this is just an index loop. We've written it already, but it's still a nice shortcut.

- `w.StartsWith("abc");` and `w.EndsWith("abc")` are also simple loops, which we're already written.

- `string w2 = w.SubStr(2,4);` gives a sub-string – start from index 2 and take 4 letters (so positions 2, 3, 4 and 5). `w.Substr(2,2)` on `"catfish"` gives you `"tf"`.

- `string w2 = w.Insert(3,"ABC");` adds `"ABC"` just before index 3 (`"chicken"` would become `"chiABCcken"`) As usual, it returns a changed copy. You'd need to use `w=w.Insert(3,"ABC");` to change yourself.

- `string w3 = w.Remove(2,4);` gets rid of 4 characters starting at 2 (it returns the result.)

  A funny thing about this, and the ones before it, are that we know member functions can change themselves, but these don't. We prefer returning the changed version and not changing the original.

- `w.ToLower()` returns an all-lower case version of `w`. It's used in a clever trick for case-insensitive compares. `if(w.ToLower()=="done")` is true for `"done"`, or `"DONE"` or `"Done"` ....

Those are all simple loops, mostly things we're written before. But it's still nice to have them "in" the computer as easy-to-find member functions. Also, maybe, they used some tricks to run them a little faster than our versions.

30.4.3 List member functions

Back in the `List` chapter, we used `L.Add(6);` to grow `L` by one box. That's obviously a member function.

There are a few more:

- `L.Clear();` resets the list to size 0.

- `L.RemoveAt(0);` gets rid of the first item (it slides everything else down, using a loop, then subtracts 1 from the list size).

  It works for any index. `L.RemoveAt(L.Count-1);` gets rid of the last item, which doesn't need a slide-loop at all.

- `L.IndexOf("cat");` searches for that word and returns the index, or -1 if it's not there. We've written this before. `L.Contains("cat");` is the same but worse – it only returns true or false.

- `L.GetRange(2,4));` is like substring. It returns a list of 4 items from `L` starting at index 2.

There are more than that, but that should give the idea.

30.5 `this`

Inside of a member function, you can use `this` to mean "me". For example you could write `this.name` instead of just `name`.

Times when you need it are rare and strange. Suppose there's a real global function taking a dog as input, which our member function needs to run on itself:

```
class Dog {
  public void dogFunc() {
    // oh no! I need to run dogUsingRealGlobalFunc with me as input:
    int n=dogUsingRealGlobalFunc(this);
    ...
  }
}

int dogUsingRealGlobalFunc(Dog d) { ... }
```

There's no good reason to have a global function like that – it would be a member function. But on big projects funny stuff can happen. Sometimes you're stuck using not-quite-right things.

You'll sometimes see a style where every member variable has `this` in front:

```
class thisUsingFunctions {
  public int a, b;
  // using extra this's for fun:
  string desc() { return this.a+" "+this.b; }

  void setup(int a, int b) { this.a=a; this.b=b; }
}
```

In the second function, using `this` let us re-use the names in the inputs. `this.a=a;` copies the input `a` into the member variable `a`.

30.6 Constructors

We've already seen functions like `makeDog("Rex",8)` which create and return a Dog. But languages like to make those official. We've seen it: `p1=new Vector3(1,0,9);` or `cc=new Color(0,0,1);`. Those are officially called *constructor*'s.

An example of a Dog constructor. This allows us to write `Dog d1 = new Dog("K-9",12);`:

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

```
  public Dog(string nm, int howOld) { name=nm; age=howOld; }
}
```

The body of the function is normal, even boring. But the part in front looks strange. Special rules for constructors:

- The name of the function is the name of the class.
- Don't write the return type. Leave it blank. The return type is automatically an item of that class.
- Don't create the object or return it. The system does both of those things for you. Your only job is to set the fields.
- You can't run them using variable-dot. They only run using a `new`. In fact, every time you use `new`, it needs to find a matching constructor.
- Overloading is allowed. You can have multiple constructors with different inputs.

Here are several for a Cow struct. The let us call `new Cow()` or `new Cow("Lu-lu",7,1475)` or `new Cow("Alice")` and do various odd things:

```
struct Cow {
  public string name;
  public int age;
  public float wt;

  // with no inputs, create a standard cow:
  public Cow() { name="Bessy"; age=2; wt=1500; }

  // or provide all 3 inputs:
  public Cow(string nm, int years, float pounds) {
    name=nm; age=years; wt=pounds;
  }

  // this one is just silly, but legal:
  public Cow(string nm) {
    name = "Madam " + nm;
    age = Random.Range(1,10+1);
    if(nm.Length>5) wt=1000;
    else wt=1500;
  }
}
```

All three of them simply set the 3 Cow member variables. The first one replaces the old `new Cow()`. Our basic cow is now Bessy, 2, and 1500. The

second is the standard one that copies the inputs straight into the fields. The third shows that we can do anything a normal function would do, as long as we set all 3 fields.

Constructors are member functions in a strange way. The first thing they do is create a Dog or Cow. Then they run as member functions on that. When we use `name=nm;`, we're assigning to the thing that we're about to return.

Here's one more set of constructors for the old `FullName` class:

```
struct FullName {
  public string first, last;

  public FullName() { first=last=""; } // the do-nothing one

  public FullName(string fName, string lName) {
    if(lName=="") lName="(no last name)";
    first=fName; last=lName;
  }
```

We can use `f1=new FullName();` as usual, or `f1=new FullName("Cher","");`, who gets (no last name) for a last name.

Constructors are allowed to call other functions. It's common to have a shared `setup` function:

```
struct Cow {
  ...
  public Cow(string nm, int years, float pounds) { setup(nm, years, pounds); }
  public Cow(int years, float pounds) { setup("generic cow", years, pounds); }
  public Cow() { setup("",0,0); }

  public void setup(string w, int ag, float lbs) { name=w; age=ag; wt=lbs; }
}
```

Every constructor figures out what it wants to values to be, and calls `setup` as a quick way to assign them.

Horribly confusing `C#` constructor rules

`C#` has two very strange rules for constructors. One is about class/struct and we've seen it:

For a class, `new Dog("Spot",4)` allocates a real Dog, on the heap. Then it gets filled in by your code and a pointer is returned (automatically, without you writing anything). But for a struct, `new Cow("Bessy",4,1500)` creates a temporary Cow. It can be copied into a Cow variable.

It's the same funny rule as before.

The other strange rule – when you write a class, you get a free constructor. When you write Cow, you automatically get `new Cow()` that makes empty strings and zero's.

But if you write a single constructor of your own, the free one goes away. If you wanted it, you have to re-write it yourself. That's why `FullName` had it. There's no special reason for this rule – other languages do it differently.

30.7 Scope

This is probably obvious: the names of member functions are only for inside the class. It's the same rule as for member variables. It's fine to use the same member function names in different classes, or anywhere else outside the class.

This has three `doStuff`'s, which are fine since they're in different scopes:

```
struct A {
  public void doStuff() { print("A stuff"); }
}

struct B {
  public void doStuff(); { print("B stuff"); }
}

void doStuff() { print("script stuff");}
```

There's never going to be any confusion between `a1.doStuff()`, `b1.doStuff();` and just regular `doStuff();`.

30.8 Style: member vs. non-member

Member functions are just regular functions that we thought looked nicer using the variable-dot notation. They look best when there's one special item. `d1.copyFrom(d2)` is fine since `d1` is changing itself, using `d2`.

But compares look nicer as normal functions, since neither is special. Below shows checking for 2 equal dogs written both ways:

```
class Dog {
  public bool isEqual(Dog d) { return name==d.name && age==d.age; }
}

// as a non-member:
bool isEqual(Dog d1, Dog d2) { return d1.name==d2.name && d1.age==d2.age; }
```

The first one is `if(d1.isEqual(d2))`. The second is `if(isEqual(dog1, dog2))`.

`Vector3.Distance(v1, v2);` is another one like this. It could have been `v1.Distance(v2);`. But since neither point is more special than the other, the first way seems nicer.

30.9 Special struct in script rule

This is the last tricky rule. Consider a Cow defined inside one of our scripts:

```
public class cowUser : Monobehaviour {
  public int minCowWt;

  struct Cow {
    public string name;
    public float wt;

    public void set(string nm, string cowWt) {
      // this is not allowed to use minCowWt
    }
  }
```

There are 2 ways to look at `Cow`. The most obvious way is that every `Cow` must be created by a `cowUser`. A `Cow` would then automatically know about who made it and be able to use those variables.

That's called an *inner class*. Some languages use it, like Java, but not C#.

The other way to look at it is that `Cow` just happens to be written inside. But it's really a completely independent struct. Any other script could declare one. A `Cow` can't depend on belonging to a particular `cowUser`.

That's the way C# does it.