Chapter 24

Pointers

This section is about using a trick where one variable can point to another. We can't use that trick with the variables we have now – we have to use a `class`, which is like a struct, but uses pointers. Pointers also require you to create variables in a different way than declaring them – you `new` them – so this section is also about that.

That's a lot of stuff at once. This is one of the more complicated things in C#, so don't worry if you have to read it a few times.

24.1 First walk-through

Before giving the rules and theory, I'll give a working simple example using classes, pointers and `new`. I'll say exactly what happens in each step, but I'll save the real explanation for later:

A class is defined and used the same as a struct – fields and the dot-rule. This defines a class `Dog`. It's exactly like a struct `Dog` would be, except for the word `class` in front:

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

A special rule for classes says you can't declare actual `Dog`'s. When you think you're declaring one, you're really making a pointer to a `Dog`. You have to create the dog separately. They declares `d1` and `d2`. They're variables, but not dogs. They can each point to a dog. Currently they point nowhere:

```
Dog d1, d2; // not Dogs. pointers to Dogs
```

To check this we can try `d1.name="spike";`. For a struct that would work. But now we get an error. `d1` isn't aimed at a dog yet.

The only way to create a real `Dog` is with the command `new Dog()`. It creates a fresh free-floating `Dog` with no variable name. If we put `d1=` in front, it hooks `d1` up to it:

```
Dog d1, d2;
d1 = new Dog(); // create and hook-up a dog

// d1: o--> | name:
//          | age:
//
// d2 -> (nowhere)
```

In the picture, the box on the right is `not d1`. It's the `Dog` we made with `new`, and it doesn't have a name. `d1` happens to be pointing to it, but that could change.

Even though it has this funny set-up, we can now use `d1` like a struct. These lines understand to follow `d1` to where it points:

```
d1.name="Spike"; d1.age=4;
```

We also read them as if they were structs. This normal-looking line understands it has to follow `d1` and read the name:

```
print( d1.name ); // Spike
```

This next part is where that all matters. `d2=d1;` causes `d2` to point at the same dog as `d1`. This is a totally new thing and something we can only do with pointers:

```
d2 = d1; // make d2 point where d1 points

// d1: o-\
//        -> | name: Spike
//        -> | age: 4
// d2: o-/
```

There's one free-floating dog, and two ways to get to it. We can show this dog-sharing by using one variable to change and the other to read:

```
d1.name = "rover";
print( d2.name ); // rover

d2.age = 9;
print( d1.age ); // 9
```

If you understand how `d1` and `d2` are looking at the same dog, this should make sense. If we co-own a poodle and I shampoo it, "your" poodle was also shampooed.

We can go one step further. At first, our dog seemed like it was `d1`, but I mede a big deal how it wasn't. Then it became a shared dog. We can create a fresh dog for `d1`, leaving `d2` as the sole owner of the original:

```
d1 = new Dog(); // make a 2nd dog, aim d1 at it:

// d1: ---------------------> | name:
//           | name: rover    | age:
//        -> | age: 9
// d2: o-/
```

24.2 Terms and Theory

This section has the rules for each part: pointers, how `new Dog()` really works, and then the exact rules C# uses for pointers and `new` together. It's still a lot at once. After this are more real examples of how we use them.

24.2.1 Pointers

A pointer variable can't store anything on it's own. All it's good for is pointing to a real variable. Suppose you have `int a,b,c,d;` and int-pointer `p1`. The program only has 4 `int`s in it. `p1` isn't an `int` – it's a way to pick out one of them. `p1` can point to `a`, `b`, `c` or `d`. The most common way to think of a pointer is as an arrow.

Pointer isn't a type by itself. It has to be a pointer to some real type – `Dog d1;` is a pointer to a `Dog`, and can only point to `Dog`s. There's no such thing as a generic pointer which can point to a `Dog` or a `Horse` or any type. We just use the word pointer as a shortcut for "pointer to whatever type we were just talking about."

Inside, all pointers are really the same – just arrows. Even if `Horse` was a huge class with 20 fields, a `Horse`-pointer would still be just a simple arrow.

A pointer variable can do two things: you can make the arrow point somewhere, or you can follow the arrow and change what's in that box. Since it's a variable, you can make it point somewhere, use it, then make it point somewhere else.

This uses `p` to change `d1`, then to change `d2`:

```
Dog d1 = new Dog(), d2 = new Dog();
// now we have 2 pointers, pointing to 2 Dogs:
```

```
//d1: o--> | name:
//         | age:
//
//d2: o--> | name:
//         | age:

Dog p;
p = d1; // aim p at 1st dog
p.name = "Spike"; // use p to change 1st dog
p = d2; // aim p at 2nd dog
p.name = "Rover"; // use p to change 2nd dog

//d1: o--> | name: Spike
//         | age:
//d2: o--> | name: Rover
//         | age:
```

We used `p.name=` twice. But it was aimed at a different `Dog` each time.

24.2.2 `new` vs. Declare

Variables you create with `new` are handled differently than normal declared variables.

Variables you declare are organized and created ahead of time. The compiler pre-scans functions, looking for all variables. This even includes the special ones inside `if`'s and loops. Then it essentially lays them all out on a sheet of paper as the permanent pre-set arrangement. When you call a function it "creates" all variables at once by laying down that sheet. When the function ends, the sheet is removed. It's all very orderly, neat and fast. As the function runs, it knows ahead of time where every variable will be.

The most complicated it can get is a chain of function calls, which is just a stack of sheets for each function in the chain. That's still not very complicated. We call that system the Stack and say normal, declared variables are Stack variables.

The important part is that while the program runs, it never spends any time deciding where to create or how to find normal variables.

`new` creates things as it runs, like a command. If `new Dog()` is in a loop it doesn't reuse the box like normal – it creates more and more dogs. The things it makes are semi-permanent. They aren't owned by the function – scope rules don't apply to them.

That means `new` can make an unpredictable amount of things in an unpredictable order. Running `new` actually goes out to the other end of memory, called the Heap, and searches for empty space. Since we can destroy Heap variables at any time in any order (see below,) the heap gets lots of gaps of free

space in it. We call it a heap since it's a mess compared to the Stack.

Some fun facts and examples:

- Remember back in the struct chapter `new Vector3(0,4,2)` is a shortcut for assigning all three parts at once. The `new` in this chapter is completely different. It looks the same, making it really confusing.
- Using it like `new Dog();` is the only way. Except you can put any class instead of Dog.
- `new Dog();` creates the dog and returns a pointer to it. That pointer is the only way you can find it. We almost always catch it like `Dog d1=new Dog();`.
- The fields are automatically filled in with 0's and `""`, the same way global variables are. Also like with globals, it's probably better not to count on this.
- They don't have names. In other words, there's no absolute way to look them up. You can only find them with pointers. For example this creates two Dogs, both using `p`:

  ```
  Dog d1, d2;
  Dog p; // temp

  p = new Dog(); // fresh Dog
  d1 = p; // hand the Dog over to real owner, d1

  p = new Dog(); // another fresh Dog (re-use p as temp holder)
  d2 = p; // hand that one over to d2
  ```

  `p=new Dog();` gives us a dog with `p` aimed at it. I thnk it's pretty clear they're two separate things, since we're giving the dog to `d1` and using `p` to make another one.

  `d1` is probably it's forever owner, so we'll probably start thinking of that real dog as `d1`. But there's no rule `d1` couldn't pass it to someone else, later.

- If nothing points to a heap object (something you made with `new`, like any Dog,) we can never find it again. It stays in the program, wasting space. Eventually the system will run a Garbage Collection, which scans for and deletes them.

  For example, this creates 3 dogs, but the first 2 are garbage:

  ```
  Dog p = new Dog(); p.name="Al";
  p = new Dog(); // Al is garbage
  p = new Dog(); // second dog is garbage
  ```

`p` was the only way to find the first dog. When we aim `p` away, Al the Dog still exists somewhere on the Heap, but we have no way to ever find him. Ditto with dog 2.

That's not a problem, except it takes time to find and clean them up – the computer has to look through all of your pointers. If you spray garbage in a big loop in a common function, your program may lag and stutter.

A funny thing is C# doesn't have a command to directly destroy a dog. The only way is to "lose" it on purpose.

24.2.3 Reference type

For real, a computer can make a pointer to any type, and any type can either be declared, or created free-floating with `new`. But to simplify things, `C#` strictly divides types into ones that can never use pointers, and ones that have to use pointers.

Normal variables – int, string, fioat, bool, any struct – can never use pointers. You can never create an int pointer aimed at `int a;`. None of these new rules apply to any of our old types. So that's simpler already.

The other half of the simplification is that classes must be pointers and can only be created with `new`. In the `Dog` class example, that was the only way to do it. We can never declare a normal `Dog` variable.

That's simpler since we don't need rules for how to declare a normal Dog.

Together, these rules – can't or must – mean we never have to think about whether to make something be normal, or a pointer. Everything is locked into being one or the other.

The types that use these new rules – only classes so far – are called Reference Types. That's sometimes good to know for error messages. You might also see Java uses reference types – same thing.

As you've seen, the normal variable rules are changed for classes. Here's a summary of what we already know:

- Declaring `Dog d1;` is changed to mean "declare a pointer to a `Dog`."
  We don't have to worry about how to declare a normal `Dog`, since we aren't allowed to.

- `d1=d2;` is changed to mean "change where `d1` points." Or longer "make `d1` point to the same place as `d2`".
  The normal rule where it means to copy from one into the other – that's gone. There isn't even a command to copy a dog.

- Adding a dot after a pointer, like `d1.name` means to follow `d1` to where it points, and then look at the field. They purposely made so following a pointer to a field looks the same as using a normal field in a normal struct.

- `==` is changed to check whether they point to the same thing. It's like asking "is this the same wrench you borrowed from me?" You're not asking if it's the same type of wrench. More on this much later.

24.3 Common pointer/class use

As usual, the rules are good to know, but they don't tell us how to write a program. We should also know the various ways we like to use and think about things.

The most common thing is ignoring these rules. Dog is a class, so we're stuck using pointers to make one, but we can pretend it's a normal variable. For example, in the code below the only way to know `d1` isn't a struct is the extra `new Dog()`. Even the function is completely normal:

```
Dog d1 = new Dog();
d1.name="Rin rin";
d1.name+="ny";
d1.age = Random.Range(2,10+1); // no reason

print( showDog(d1) );

string showDog(Dog d) { return d.name + ": " + d.age + " years old"; }
```

We'll keep `d1` locked onto the dog it made, as the permanent owner. Together they count as one thing. We can think of `d1` as a normal variable. There's no computer rule to do that, this is just describing our plan.

Then we'll think of some dogs as real pointers. Those will never "own" anything. They'll only be used to point to other people's dogs, such as `d1`. Here `p` is a pointer-style dog:

```
Dog d1 = new Dog(); // d1 is a normal dog
d1.age=7; // like a simple struct changing itself

Dog p; // a sneaky dog pointer
Dog p=d1; // p is temporarily peeking at d1

p.age++; // using p to increase d1
d1.age++; // normal increasing my own age
```

Even though the end is really one dog equally shared by two pointers, it's better to think of `p1` as a temporary link to `d1`.

Here's a more realistic example. There are two "real" dogs and one pointer showing the active dog. Pressing A or S switches the pointer:

```
// two real dogs and a pointer:
Dog pet1, pet2; // "real" Dogs
Dog activeDog; // used to select pet1 or pet2

void Start() {
  pet1 = new Dog(), pet2 = new Dog();
  activeDog = pet1; // select pet1 for now
}

void Update() {
  if(Input.GetKeyDown("a")) activeDog = pet1;
  if(Input.GetKeyDown("s")) activeDog = pet2;
  // add 1 to whichever dog is active:
  activeDog.age++;
}
```

Hopefully, to you pet 1 and 2 feel like dogs, but `activeDog` feels like a dog-selector. If you see this picture in your head, you've got the idea of pointers:

```
pet1: o-> | name:      pet2: o-> | name:
          | age:                 | age:
                ?     ?
                 \   /
               activeDog
```

24.4 Functions and pointers

We can use thiese as inputs and return values from functions. The trick is that we can still never have a normal Dog variable. Everything is still a pointer.

This function takes two *pointers* to Dog's and return a pointer to a Dog:

```
Dog someDogFunction(Dog a, Dog b) { ... }
```

But we're going to keep our two ways of thinking. Only a few will act like pointers. Usually everything will feel normal – like passing and returning normal Dog's.

24.4.1 Pointer inputs

The showDog function from before is an example where we can pretend Dog is a normal type. It's good for showing the rules. A repeat:

```
string showDog(Dog d) { return d.name+": "+d.age+" years old"; }

string dWord = showDog(pet1);
```

`Dog d` is a pointer. Instead of copying `pet1` into it, the call sets it up to point back to `pet1`. That's normal pointer copying rules. `d` and `pet1` now point to the same dog.

`d.name` isn't reading a copy – it's reading the actual name directly from `pet1`. But we get the same output either way. If we only *read* the input, pointer-using functions are the same as normal ones.

The new thing we can do is change the input for real. This pointer-using function is a short-cut for assigning name and age:

```
void setupDog(Dog d, string nm, int howOld) {
  d.name=nm; d.age=howOld;
}

// sample use:
setupDog( pet1, "Gary", 4);
setupDog( pet2, "Gina", 5);
```

Once we know `d` is a pointer, set up to point to the original dog in the call, the way this works is obvious. It's the same math we did before, except using a function to create the extra pointer:

```
// what that function does:
d=pet1; // <- set during call
d.name=nm; // <- clearly changing pet1
d.age=howOld; // <- ditto
```

The idea of a function like this is saying "here's a pointer you should use to change me."

Notice if `Dog` was a struct the function would do nothing. We'd be changing a copy.

This next one adjusts a `Dog` away from wrong values. It's the same reach-back-and-change-me idea, but plays with the variables more:

```
void adjustDog(Dog checkMe) {
  if(checkMe.age<0) checkMe.age=0;
  else if(checkMe.age>25) checkMe.age=25;
  if(checkMe.name=="") checkMe.name="dog";
}
```

The same as before, calling this with `adjustDog(mascot);` starts by aiming `checkMe` at the mascot, letting us adjust it for real.

This next one is a little different. It copies one `Dog` into another. The second dog isn't being changed – we only read from it, but the first one is:

```
void copy(Dog toDog, Dog fromDog) { // like toDog=fromDog
  toDog.name = fromDog.name;
  toDog.age = fromDog.age;
}
```

There's no built-in way to copy `pet1` into `pet2`, so this is actually useful: `copy(pet2, pet1);`.

To get more of a feel for these, here's some totally abusive uses of these functions:

We should always catch the result of `new Dog()`, like `pet1=new Dog();`. Otherwise we can't use it. But we're not required to. This next one creates a free-floating dog and prints it:

```
string w = showDog( new Dog() ); // ": 0 years old"
```

Inside the function, `d` is the only thing pointing to it. When the function ends, the Dog we created becomes garbage. This is a terrible way to print that, but it's legal.

`copy(pet1, new Dog());` works the same way. `fromDog` is the only pointer to a fresh `["",0]` dog. Those values are copied into `pet1`. After the function returns we get another garbage dog.

A real oddball is `copy( new Dog(), pet2);`. It accomplishes nothing, but is legal. It copies `pet2` into a fresh Dog we can never find again after the function ends. If `pet2` was named Inga, there would be another Inga floating on the heap, with nothing pointing to it.

In all of these, we wasted a little computer memory making an unusable dog. If we ran them in a loop, we'd eat up more and more memory on all those lost dogs, forcing a tiny delay for a garbage collection.

`Dog d1; showDog(d1);` would give an error. `d1` points to nothing, so `d1.name` is an error.

The copy function only works if we created the thing we're copying into. This gives an error:

```
Dog pet1=new Dog(); setupDog(pet1, "Cpt. Spangles",  7); // legal dog
Dog pet2; // un-newed dog
copy(pet2, pet1); // <- error, pet2 doesn't have a dog
```

24.4.2 Pointer outputs

When a function looks like it returns a dog, it's really returning a pointer to a dog. The simplest dog-returning function work like `new` – they create a dog and

return a pointer to it. They feel like they return a regular Dog.

Here's a very basic one:

```
Dog makeDog() { return new Dog(); }
```

The inside, to review, creates a free-floating dog and gives a pointer so we can use it. Then the function returns that pointer back to whomever called it.

`Dog pet1=makeDog();` is a shortcut for `Dog pet1=new Dog();`.

A little different version, using a variable, works the same:

```
Dog makeDog2() { Dog d=new Dog(); return d; }
```

Returning a pointer variable feels different. Remember these things: `d` points to a dog, equals with pointers means "point where I point", and returning is like equals. This returns a pointer to the dog it made.

`Dog pet2=makeDog2();` hooks pet2 up to that fresh dog, same as before.

Finally here's a semi-useful one, that allows us to supply dog info:

```
Dog makeDog(string dogName, int yearsOld) {
  Dog dd = new Dog();
  dd.name=dogName; dd.age=yearsOld;
  return dd;
}
```

`Dog pet1=makeDog("Spike", 6);` is now a nice substitute for a `new` and two assigns. This was the point of the example where `p` made a Dog and passed it along. Here we're using temp `dd` to set up the Dog, then sending that pointer along to the real owner.

A sneaky one that people use for real is clone. It creates and returns a copy of another `Dog`. In other words, it's the same as `makeDog`, except you give it the name and age by giving it a dog to copy:

```
Dog clone(Dog cloneMe) {
  Dog dd = new Dog();
  dd.name = cloneMe.name; dd.age = cloneMe.age;
  return dd;
}
```

Trying to use this shows off more how pointers work. Copy assumes you did the `new`, clone does it for you:

```
Dog pet1 = makeDog("Spike", 6); // dog to copy
Dog d2;
copy(d2, pet1); // ERROR - d2 has no Dog
d2 = clone(pet1); // works, makes a Dog, gives to d2
```

```
copy(d2, pet1); // now this is legal, since d2 has a Dog
d2=clone(pet1); // legal but the old d2 is now garbage
```

The second type of pointer-returning function is very different, and is all about how pointers work. It chooses between existing items and gives you a pointer to one of them.

This function takes 2 dogs and returns a pointer to the oldest:

```
Dog oldest(Dog a, Dog b) {
  if(a.age > b.age) return a;
  return b;
}

// sample use:
Dog dOld = oldest(pet1, pet2);
```

The big difference is we're thinking of `dOld` as a pure pointer. Its purpose is to point to a real dog (`pet1` or `pet2`). It's like `activeDog` from before.

Here's the same idea, randomly choosing between two globals:

```
Dog randomDog() {
  if( Random.Range(0,2)==0 ) return pet1;
  else return pet2;
}

// sample call:
Dog pp = randomDog(); // pp aimed at either pet 1 or 2
```

`pp` should feel like a dog selector, and `pp.age++;` should feel like it's reaching through to one of the real dogs.

And the usual non-pointer-related random notes: each call randomly sets `pp`, but it sticks there until we call it again. Random also isn't afraid to repeat numbers. If `pp` is on `pet1`, calling this has a 50% chance to pick `pet1` again.

These two function types are like the two ways of thinking of pointers: are you returning a regular `Dog`, or are you returning a pointer to someone else's dog?

24.4.3 Turn-input-into-output bug

A neat example of how structs and classes are very different is the "change the input" trick. It doesn't work at all with pointers.

Here's a non-working and working function to make a baby `Dog` from an adult. This first version would work perfectly for a struct, but is a mess for a class:

```
// non-working change input version, which would work for a struct:
Dog babyDog(Dog adultDog) {
  adultDog.name += " Jr.";
  adultDog.age=0;
  return adultDog;
}
```

This incorrectly turns the original adult dog into a puppy. Then returns the same pointer so the "new" puppy is just a shared box with the adult.

If we want a fresh box for the puppy, we need to use `new`:

```
Dog babyDog(Dog adultDog) {
  Dog baby = new Dog(); // <- new line
  baby.name = adultDog.name+" Jr.";
  baby.age=0;
  return baby;
}

// sample use:
Dog pet1Kid = babyDog(pet1);
```

It's really just a modified clone function.

24.5 null

Pointers are allowed to point nowhere. The official value is `null` (all lower-case.)

We often use it to initialize a "real" pointer, as a hint:

```
Dog d1=new Dog(), d2=new Dog(); // real dogs
Dog activeDog=null; // pointer, currently aimed nowhere
```

As we all know from movies, a null-ray is the most devastating sort of ray. It's common to think `null` in a program is like that – that it destroys what we're pointing to. It's simply changing where we point. If it helps, `null` is really just 0.

An example of it not causing havok:

```
Dog p = pet1;
p=null; // pet1 is fine
p=pet2; p.age++; // p is also fine. The null didn't break it
```

A common use is showing you're done using a pointer:

```
p=pet2;
p.age= ...
p.name=...
p=null; // since we're done, move it away from pet2
```

That last `p=null;` line isn't needed, but I think it looks nice.

The most special thing about `null` is, obviously, you can't use it. Trying to is an error:

```
Dog d1=null;
d1.age++; // error

d1=pet1; d1.name="X"; // fine
d1=null; d1.name="X"; // error
```

We can check for `null` using `==` and `!=`:

```
if(d1!=null) d1.name="X"; // this is safe
```

Checks for `!=null` are very common. Sometimes they're error checks. Here's a copy function that won't crash if you give it null dogs:

```
void dogCopy(Dog dTo, Dog dFrom) {
  if(dTo==null) return;
  // nowhere to copy to. May as well quit

  if(dFrom!=null) { dTo.name=dFrom.name; dTo.age = dFrom.age; } // real copy
  // if nowhere to copy from, blank us out:
  else { dTo.name=""; dTo.age=0; }
}
```

A horrible but legal use would be `copy(pet1, null);`, which would blank out `pet1` (only because of the special `if(dFrom!=null)` check.)

Sometimes pointing nowhere isn't a mistake. Sometimes we want that option and checking for `null` is really natural. Here's the dog-age-adder where D selects no dog:

```
void Update() {
  if(Input.GetKeyDown("a")) activeDog = pet1;
  if(Input.GetKeyDown("s")) activeDog = pet2;
  if(Input.GetKeyDown("d")) activeDog = null; // <- new line

  if(activeDog!=null) activeDog.age++;
}
```

The official term for a pointer set to null is **null pointer**. It doesn't mean anything more than that, even though it sounds scary. For example `if(p!=null)` is checking for a null pointer.

24.6 Errors

Almost all the errors we get will be crashes from trying to follow a pointer set to `null`. The error message is: *nullReferenceException. Object not set to an instance of an object.*

A common way to get that error is forgetting to `new` it. If you remember, all global variables are auto-inited to 0 or `""`. Pointers are auto-inited to `null`:

```
Dog dg; // global starts at null

void Start() {
  dg.age = 5; // null reference exception
  int n = dg.age; // same error (except it crashed on the line above)
}
```

This is a run-time error, which means the program will run normally, then crash when it comes to the line. As usual, all of Unity won't crash. Getting these is no worse than getting any other error (except an infinite loop.)

To be nice, Unity keeps running after it should crash. It aborts the current function, but keeps running Updates. This Unity program will print A, then C over and over, with red errors sprayed in-between:

```
Dog d;

void Start() {
  print("A");
  print(d.age); // crash - quits Start
  print("B"); // never gets here
}

void Update() {
  print("C");
  print(d.age); // crash - quits this Update
  print("D"); // never gets here
}
```

It crashes out of that particular Update, but it runs the next, which also crashes. A real program would crash and quit after the first one.

24.7 Fake `new`'s

This is a repeat from way up top. I wanted to give more examples.

In programming in general, `new` means to create something on the heap. `C#` is a rare language that reuses new as a way to make an ordinary value.

`new` with any basic type is just a fancy way to make 0:

```
int n = new int(); // same as n=0
string w = new string(); // same as w=""
bool b = new bool(); // same as b=false
```

The key is these are all regular variables and the `new`'s aren't doing anything.

From the struct chapter, we also know you use an extra `new` to create a constant. These are all extra `new`'s that do nothing (remember `Vector3` and `Color` are simple structs):

```
struct Cow { public string name; public float wt; } // <-struct, not class

Cow c1 = new Cow(); // merely sets name "", and wt 0

Vector3 v = new Vector3(0,7,0); // simple assignment shortcut
Color c1 = new Color(1,0,1); // ditto
```

Nothing is being created. In any other language we'd write `c1=Cow();` and `v=Vector3(9,7,0);`, without that confusing `new`.

The key in `C#` is not to be fooled into thinking the `new` means pointers. When you see something like `Frog f=new Frog();`, you have to look up whether frog is a struct or a class.

And remember, if it's a struct, `Frog f2=f;` makes a second identical frog. If it's a class, you get a pointer to the first one.

24.8 Pointer compare (==)

Oh, no!! More rules. But this one is pretty easy. It's some examples of the "is this the one you borrowed from me?" compare example from way up top.

The only data in a pointer is where it points. When you compare two pointers with `==`, you're asking if they point to the same place. `==` won't compare the contents.

Here's an example of a good use of this rule. I want the A key to switch between pet 1 and 2:

```
void Update() {
  if(Input.GetKeyDown("a")) { // switch between dogs:
    if(activeDog==pet1) activeDog=pet2;
    else activeDog=pet1;
  }
  activeDog.age++;
}
```

The `if` is asking whether we're pointing to `pet1`. If we're pointing to `pet2`, even if has the same name and age as `pet1`, the answer is false. We're pointing

to a different dog is all that matters.

Then here's a wrong way to use it. This tries to compare different dogs. `==` doesn't do that:

```
if(pet1==pet2) {
  print("please re-enter dogs using different data");
}
```

This will be false every time. It's asking whether pet 1 and 2 point to the same place.

Don't get confused about `==` with non-pointers. `if(pet1.name==pet2.name)` is a normal compare, since those are both strings. As usual, the final type is what matters.

If you're wondering how you can do a "are these 2 dogs identical" compare – you have to write it out yourself, comparing name and age.

24.9 Old-style pointers

C# does have real pointers. You aren't allowed to use them, but it might be nice to see them, for comparison. They require you to write out every step, so you get to see what the shortcuts are doing.

This example declares `int`s and pointers to `int`s:

```
int n1, n2; // normal ints
int* p1, p2; // pointers to ints
```

Remembering `int*` means "pointer to an int" can be a pain. That's why `Dog d1;` just knows to make a pointer to a Dog. But being able to write both types of things can be nice.

Likewise, we can point to a real int, but need another symbol:

```
p1 = n1; // error - can't copy an int into a pointer to an int
p1 = &n1; // p1 points to the real box n1
p2 = p1; // so does p2. Copy pointer to pointer
```

We have to use a special symbol to follow the arrow:

```
p2 = p1; // change where p2 points (same as before)
(*p2) = 5; // follow arrow and change n1
```

Those tricky rules mean we can copy a pointer, or copy values from where they point:

```
*p1 = *p2; // copy from where p1 points into where p2 points
p1 = p2; // just change p1 to point where p2 points
```

With a struct, you can tell if you have a pointer, or not:

```
Cat c1; // actual cat
Cat* cp = &c1; // pointer to c1
c1.name="Tabby"; // change directly
(*cp).age=5; // change c1 using a pointer
```

To do anything, you have to remember the extra symbols, which means you can get a lot more errors. But you also have more options.

And again, this is completely not important to know. But you might get a feel for the simplified reference type rules if you see the full rules they're simplifying.