Chapter 24

Pointers

This section is about a trick where one variable can point to another. We can't use it 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 programming, and especially C#, so don't worry if you have to read it a few times.

But pointers are very useful. They're one of the main concepts of an intermediate level coder. I promise that hurting your head reading this chapter will be worth it.

24.1 First walk-through

Before giving the rules and theory, I'll give a working simple example using classes, pointers and `new` (which is completely different from the `new` with structs).

I'll say what happens in each step, but I'll save the real explanation for later, including what they're good for:

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 never declare actual `Dog`'s. When you think you're declaring one, you're really making a pointer to a `Dog`:

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

These are real variables, which can point to any Dog. We have to create the Dog separately. Since we didn't yet, these point to nothing.

We can prove it. Try `d1.name="spike";`. For a struct, that would work. But now we get a *'Use of unassigned variable'* error. It's telling us `d1` isn't pointing to 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 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;
print( d1.name ); // Spike
```

These dots are doing one more step than the old dots – they have to follow the arrow coming out of `d1`. But the end result is the same.

This next part is where arrows matter. `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.

I'll add one more line to the example, another `d1=new Dog();`. It makes a second free-floating Dog with `d1` changed to aim at it:

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

// d1: ---------------------> | name:  (the second Dog)
//           | name: rover    | age:
//        -> | age: 9
// d2: o-/   (original Dog)
```

`d2` is now the sole owner of the original Dog. They no longer share one. `d1.name="Scruffles";` no longer changes `d2`.

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, so we changed two different names.

24.2.2 `new` vs. Declare, Heap vs. Stack

The big thing here is that `new Dog()` is a command. It creates a Dog every time it's run. Normal variables don't work that way.

We need to back up a little into computer theory. There are two general kinds of variables: Stack and Heap. Everything we've used before, including structs, is a Stack variable. The computer is very good at automatically managing those.

For example, variable declarations aren't in the final program. The compiler scans your program, finds all declares, and pre-makes a chart of them. Then it finds everywhere they're used and replaces them with the pre-set spot in the chart. Before the program starts to run, `n=7;` is turned into "memory location 24 = 7".

Functions are mostly the same. The compiler does all of that work, using the locals. The resulting chart is called a stack frame. When you call a function, it grabs enough space to hold the stack frame, in one step. Every local variable is created at once. When the function ends, the whole stack frame is popped off.

If you have a chain of function calls, each adds its stack frame on top. That's why the computer never gets confused by re-using the same variable name for different locals. They're stored in completely different stack frames. Computer chips even have this built-in – they have a spot for the start of the current stack frame. Local variable look-up circuits use it.

As cool as those details are, you don't need to know them. The main point is that normal declared variables are pre-made.

As you might guess, Heap variables are the opposite. You start with none. `new Dog()` is a real command that makes a Dog. Dog's are put into a big area with everything else ever created using `new`. That's why it's called a heap. It's a big pile of whatever, in no special order and without very good record-keeping. The only way to find anything on the heap is through a pointer.

The stack starts at one end of memory. The Heap starts at the other. Here's a picture of a program which has run for a while (Update has called the function doMove. And someone ran `new Dog()` 3 times and `new Horse()` once):

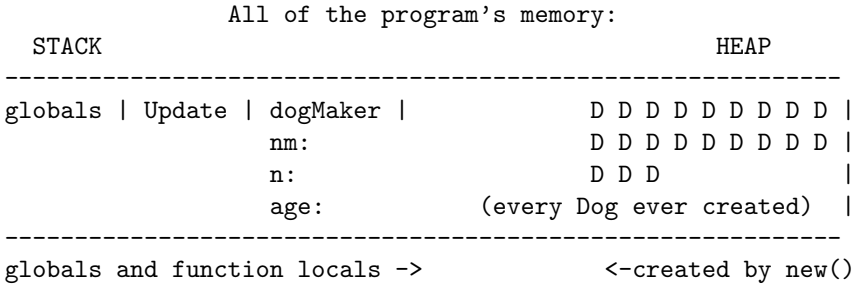
```
                All of the program's memory:
  STACK                                                HEAP
-------------------------------------------------------------
globals | Update | doMove |               Dog Dog Horse Dog |
-------------------------------------------------------------
globals and function locals ->             <-created by new()
```

When `doMove` finishes, its part pops away. The same for Update. But those Dog's and the Horse are just floating on the Heap, living their lives.

One more example, a useless function that makes some Dogs:

```
void dogMaker(string nm) {
  int n = Random.Range(2,6); // dogs to make
  for(int i=0; i<n; i++) {
    int age=i+2;
    new Dog(); // <- each time it runs, makes a Dog
  }
}
```

When we run this, the system instantly creates `nm`, `n` and even `age` for the loop. When it quits they snap away. No Dogs are made yet. When the loop runs, it creates 2 to 5 permanent Dog's. They don't go away when the function ends. Every time it runs, we get 2 to 5 more Dogs. They all live together on the heap:



Heap variables don't have names. They don't care which function made them. They're just free-floating anonymous Dogs. The only way to use one is if you have a pointer to it. So all of those D's are useless space-wasting garbage Dogs. That's why `new Dog();` returns a pointer – it has to. `Dog d1=new Dog();` is the pointer `d1` catching the return value of the `new Dog();` command.

Things on the heap live forever, sort of. Some systems have a command to individually destroy one. Other systems, like Java and `C#` use what's called Garbage Collection. When a Dog get lost – there aren't any more pointers to it – the system eventually auto-deletes it (more on this later).

Fun facts about other `new`'s:

- A reminder: the `new` from structs, like `v=new Vector3(-7,0,0);` is fake. It's not a real `new` command.
- `C#` actually triples-uses the word `new`! There's another one that goes in front of certain functions. It's totally different. I'm so, so sorry.

I mentioned heap variables don't have names, but it seems like `Dog d1=new Dog();` makes a Dog named `d1`. This shows how that's not true. It uses `p` to make two Dogs, handing them off to `d1` and `d2`:

```
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(); // p now aimed at 3rd Dog
```

`d1` is aimed at a Dog originally made for `p`. We could think of that Dog as `d1`, unless we plan to hand it off somewhere else. `d2` is the same way.

24.2.3 Reference types

In general, pointers are like add-ons. You can have an `int` and also a pointer to an `int`. But several languages, including C#, simplify this with a trick. They make it so a type can either never use pointers, or has to use them. That's simpler since you never have a choice.

Our old types are the first way – they can't use pointers. So none of these new rules apply to ints, strings. floats or structs. You can't have a pointer to an int, or use `new int()` to create one out of thin air.

A `class` can use pointers. It has to. You can never declare a normal Dog. This is why `class` and `struct` look like the same thing. They are, in both flavors – never pointer, or always pointer. In practice, when you're creating a struct you have to decide – struct or class? Does this need pointers?

A summary, plus some new rules:

- `Dog d1;` is automatically a pointer since Dog is a class. When a programmer sees `Frog f1;`, they won't know what `f1` is until they check struct or class.
- `new Dog()` is the only way to create an actual Dog. There's no way to declare one.
- For a class, `d1=d2;` doesn't copy the contents. It makes `d1` point to `d2`. As easy way is to remember that `d1` is an arrow. `d1=` tells it where to point.
- `d1==d2` checks whether they point to the same Dog. It's always false if they point to different Dogs. Again, remember they're arrows and it's not all that confusing.

The technical term for pointer-only types is *reference type*. Struct is normal. Class is a Reference Type. I personally would have called them Arrow types, but reference is like "refers to", so not too bad.

24.3 Common pointer/class use

In practice, 90% of class variables are used like structs, with no pointer tricks. The rules were specially tweaked to do this. This code would work the same whether Dog's were classes or structs:

```
Dog d1 = new Dog(); // create d1's permanent Dog
Dog d2 = new Dog(); // create d2's forever Dog
d1.name="Spot"; // exactly the way a struct would do it
d1.name+="ster"; // complicated math is also exactly the same
d1.age = Random.Range(2,10+1); // even this -- looks like a struct
```

We declared `d1` to be the forever "owner" of its Dog. We'll never use any pointer tricks. It's easier to think of `d1` as a normal Dog variable. `d1=d2` will result in pointer weirdness, so we won't do that.

The other way we'll use them is as real pointers. When we declare them, we'll mentally decide they'll never "own" anything. They'll only point to other people's dogs. Here `p` is a pointer-style dog:

```
Dog d1 = new Dog(); // a normal-syle Dog

Dog p; // a sneaky dog pointer. Notice no new
Dog p=d1; // p is temporarily peeking at d1

d1.age=8; // normal setting our age
p.age=8; // sneaky p increasing d1's age
```

Technically that makes 1 Dog with both `d1` and `p` pointing to it. But logically it makes Dog `d1` and a pointer temporarily aimed at it.

Here's a semi-realistic example. There are two "real" dogs and one pointer showing the active one. Pressing A or S switches the pointer. The age of the active dog increases:

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

void Start() {
  pet1 = new Dog(), pet2 = new Dog(); // make the real Dogs
  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++;
}
```

If you see this picture in your head, you've got the idea of pointers:

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

`actveDog` selects `pet1` or `pet2`, by aiming at it. `activeDog.age++;` is really increasing pet 1 or 2's age.

24.4 Functions and pointers

Classes can be function inputs and return values. They're still always pointers, but we'll use our two ways of thinking: most of the time we'll pretend they're normal variables; but occasionally we'll use them as real pointers.

24.4.1 Pointer inputs

We can call a function with class inputs and pretend they're structs. This normal-looking function converts a `Dog` into a string:

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

string dWord = showDog(pet1); // Rover: 2 years old
```

There's nothing special about it. But a neat thing: `d` is a pointer. It's aimed back at the actual Dog that called us.

Because of pointers, functions can change their inputs. This fills a Dog with the name and age we give:

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

`setupDog(pet1, "Gary", 4);` actually changes `pet1`. It works because `d` points back to what called us. Changing `d` is really changing `pet1`:

```
pet1: o-> | name:
          | age:
              ^
Function      |
           d: o
          nm: Gary
         age: 4
```

Even though they're in different areas, `d` can aim at `pet1` and change it remotely, through the magic of pointers.

This next one is about the same. It adjusts a `Dog` away from wrong values:

```
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";
}
```

`checkMe(pet1);` changes `pet1` for real, if it had a bad age or name.

A common class function copies one into another. The first Dog gets the contents of the second:

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

`copyDog(pet1, pet2);` turns `pet1` into a copy of `pet2` without making them share a Dog. This are useful, since there's no built-in way to copy the contents of one Dog into another.

24.4.2 Pointer outputs

When a function looks like it returns a Dog, it's really returning a pointer to a dog. But most of them seem like normal functions. Only a few return a "real" pointer.

Normal `Dog` outputs

The simplest dog-returning functions work like `new` – they create a dog and return a pointer to it.

Here's a very basic one which is merely a shortcut for `new`:

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

We could run it like `Dog d1 = makeDog();`. It works since `new Dog()` creates a general purpose free-floating heap Dog. It's not tied to the function like local variables are. When the function ends, that Dog is still there and `d1` is pointing to it.

This version does that same thing using a middle-man variable:

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

When the function ends, local variable `d` is destroyed. But once again, the actual Dog is still there and we get it. This version shows that even more:

```
Dog makeDog3() {
  Dog d = new Dog();
  d.name="dog"; d.age=2;
  return d;
}
```

It's easier to see how `d` is basically a temporary. It's used to set up the Dog's base stats, before it's passed along to us. `Dog pet1=makeDog3();` gets a 2-year old dog named dog.

Finally, here a useful version, which allows us to give a name and age:

```
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 assignment statements.

Let's backup a little and compare `setupDog` and `makeDog`:

```
Dog d1=makeDog("Rufus", 6); // creates Dog and fills it in

Dog dog2 = new Dog();
setupDog(dog2, "Roofus", 5); // this works on created Dogs only

Dog dd;
setupDog(dd, "A", 5); // ERROR - dd isn't aimed at a Dog
```

MakeDog creates a Dog, setupDog assumes you already have one. It's about that extra step classes need to create the real object.

For more fun, suppose we call makeDog twice in a row. It's not really a problem:

```
Dog d2;
d2=makeDog("B",6);
d2=makeDog("BB",8); // abandon 1st Dog for this one
```

That makes 2 Dogs and throws away the first one. No one has a pointer to it, so it can never be found. A picture:

```
 d1: o     | name: B  (original Dog, lost)
     |     | age: 6
     |
     \ ->  | name: BB  (Dog from 2nd makeDog)
           | age: 8
```

It's the same thing that happens if you write `d1=new Dog();` twice in a row. No harm, but a wasted Dog.

Another common Dog-making function is a clone. It creates a copy of another `Dog`:

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

It's pretty much the same thing as the function taking a name and age. `Dog d2 = clone(d1);` gets a copy.

To sum up:

```
Dog d1 = makeDog("X", 6); // start with a created d1
Dog d2; // 2 ways to copy it into d2

// both of these make a fresh Dog, with same value as d1:
d2 = clone(d1); // clone creates a Dog
d2 = new Dog(); copyTo(d2, d1); // copyTo assumes you have a Dog

// This makes them share a Dog. Not the same as a copy or clone:
d2=d1;
```

Pointer `Dog` outputs

More rare functions return a true pointer. They give you back something aimed at an existing Dog.

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;
}
```

The advantage is that we can use the output like a pointer, like `activeDog` in an earlier example. We can use it to change the selected Dog:

```
Dog dOld = oldest(pet1, pet2); // dOld is a pointer to pet1 or 2
dOld.name += " the elder"; // changing pet1 or 2
```

This next one cheats a little by assuming we have globals named `pet1` and `pet2`. It randomly chooses one:

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

`Dog pp = randomDog(); pp.age++;` would randomly age pet1 or 2. The logic is the same. In out minds `pp` is merely a pointer, aimed at some existing Dog, used to remotely play with it.

Turn-input-into-output problem

A neat example of how structs and classes are very different is the "change the input" trick. If Dog was a struct, this would give us a baby Dog, named after the mother:

```
// this would work if Dog was a struct:
Dog babyDog(Dog adultDog) {
  adultDog.name += " Jr.";
  adultDog.age=0;
  return adultDog;
}
```

`adultDog` is a copy. It's fine to change it and return it as the answer.

But with Dog as a class this is a mess. First it changes the original Dog, then it returns a pointer to the same Dog instead of making a fresh one. `Dog baby1 = babyDog(pet1);` causes both Dogs to be aimed at the same thing.

To make a fresh baby Dog, we need a `new`. This works when Dog is a class:

```
Dog babyDog(Dog adultDog) {
  Dog baby = new Dog(); // required to create a different Dog
  baby.name = adultDog.name+" Jr.";
  baby.age=0;
  return baby;
}
```

This version is basically a modified clone function. It gives us a copy of `adultDog`, but younger.

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. But it's harmless. `null` is simply a permanent place any pointer can change itself to, that means nowhere. If it helps, `null` is really just 0.

An example of `null` not causing havok:

```
Dog p = pet1;
p=null; // p moves off of pet1. pet1 is fine
p=pet2; p.age++; // p can be used again. The null didn't break it
```

A common use for `null` is showing that you're done using a pointer:

```
p=oldest(pet1, pet2);
p.age-=2; // oldest dog gets younger
p=null; // p pointing nowhere signals that we're done with it
```

If the function continues, `p=null;` let's us know we're done changing the oldest Dog, and gives an error if we accidentally use `p` again without aiming it somewhere else.

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 get a run-time error: *NullReferenceException: Object reference not set to an instance of an object.* That seems clear enough.

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

```
if(d1!=null) d1.name="X"; // this is safe
else print("can't set name -- d1 isn't pointing to anything");
```

Checks for `!=null` are very common. Often they're error checks at times when the variable should never be `null`. 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; }
  // if nowhere to copy from, blank us out:
  else { dTo.name=""; dTo.age=0; }
}
```

The correct thing to do on an error can be tricky. If we have `dogCopy(d1, dOoops)` and `dOoops` is accidentally `null`, what should happen? If it really should never be `null`, maybe we should print an error message and crash. That gives us a chance to fix it. Maybe I should have set the name to NULL COPY

and age to -999.

Sometimes pointing nowhere isn't a mistake. Often we want a "nothing selected" option. `null` is the best way to say that. Here's the dog-age-adder from way back where the D-key selects "no dog":

```
void Update() {
  if(Input.GetKeyDown("a")) activeDog = pet1; // pet1 and pet2 are globals
  if(Input.GetKeyDown("s")) activeDog = pet2;
  if(Input.GetKeyDown("d")) activeDog = null; // <- no dog selected

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

Logically, `activeDog` can can be 3 things, each as good as the other: pet1, pet2, or nothing.

The common term for a pointer set to null is a **null pointer**. 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).

The same problem with a local Dog gives a little different error:

```
void Start() {
  Dog d1; // forgot the new. We think d1 is a struct
  d1.name="Canine"+Random.Range(1,10);
  d1.age=0;
```

We get a normal error this time *Use of unassigned local variable 'd1'*. That's because `d1` is still a pointer, and local variables aren't automatically initialized. It's not even `null`. The computer can tell it's definitely an error to follow an arrow which has never been set.

24.7 Pointer compare (==)

Oh, no!! More rules! These are some examples of the pointer `==` rule. If you remember, `d1==d2` checks whether they are sharing the same Dog.

Here's an example why we like this rule. The A key switches between pet 1 and 2:

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

This works even if the pets have the exact same values. `if(activePet==pet1)` checks whether we're pointing exactly at `pet1`.

Then here's a wrong way to use pointer `==`. The test is always false since they are two different Dogs:

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

We'd need to write it out: `if(pet1.name==pet2.name && pet1.age==pet2.age)`. Don't get confused about `==` with non-pointers. `if(pet1.name==pet2.name)` is a normal string compare.

24.8 Garbage Collection

This is a fun section that's not super important. The short version: remember how in the top section I showed how you could waste memory by creating lots of Dogs with `new`'s you didn't need? That's not a problem. The computer will eventually clean that up. Don't worry about using too many.

The slightly longer version: those extra Dog's are called *garbage*. If enough get made, memory runs out and the program crashes. Every so often, C# automatically hand-checks and deletes them. That's called *garbage collection*.

And now the long, boring version: the computer is excellent at managing regular variables, using the stack frame idea. Heap variables – Dogs made with `new` – are more versatile, but the system can't manage them. When you lose your last pointer to one, the system doesn't know.

The only way to find useless Dogs is to look at every declared Dog in the entire program, see where they point, mark those Dogs as in use; then go back and remove every unused Dog. That's garbage collection. It happens automatically. There are some tricks, but it's not fast.

In theory, a sloppily written program, that creates and loses lots of extra Dogs, will have a small hiccup every few minutes. It's pausing to run garbage collection to free up memory. In practice you don't notice.

It's common to make garbage without trying. This program has a 1% chance to reset the Dog's name and age, except it uses `new`, so turns the old Dog into garbage:

```
public Dog d1;

void Update() {
  ...
  // 1% chance to reset the Dog's name and age:
  if(Random.Range(0,100)==0)
    d1 = new Dog("Spot",0); // creates new Dog. Old one is garbage
    // non-garbage way:
    // { d1.name="Spot"; d1.age=0; }
}
```

It's not many extra Dogs, so it's fine. I think the "slow" version with `new` is easier to read

If you know you're done using a Dog, it seems as if there would be a command to delete it, just to speed things up, but there isn't. `d1=null;` is the closest we can come. The next garbage collection step will clean up your old, lost Dog.

The Reference Type system was invented to make automatic garbage collection work. That's partly the reason we can never have pointers to `int`'s or declare a normal Dog. Eliminating those options made garbage collection possible.

There are two other systems, not in C#, to handle garbage. C++ has no garbage collection. Instead it has a `delete(d1)` command. If you forget to use it, the garbage is there forever. C++ programs are very careful that every `new` has a `delete`. In return, C++ pointers can be from any type and can go anywhere.

Other languages use Reference Counting. Every Dog has an extra counter for how many things point to it, automatically updated. When the count hits 0,

the system destroys it. The same as C#, you don't need to do anything. It runs a little slower, but makes it up by not needing to pause for garbage collection.

24.9 Why struct `new`'s?

C# purposely makes it so classes and structs use `new` to mean different things. That's odd. Let's compare them again, then try to see why they made that rule.

Vector3 is a struct. It's a normal variable. `new` is only used for the optional constructor shortcut to fill in several variables at once. But for a class, `new` is required to make it exist:

```
Vector3 v; // v now exists and is ready to use
v = new Vector3(); // not needed. Shortcut to set to (000)

Dog d1; // there are no Dogs, just a pointer to nowhere
d1 = new Dog(); // required. creates the Dog
```

We don't normally like to double-use like this. For example we invented `==` for comparing since we didn't want to use `=` to mean assign and compare.

For `new`, the idea is that not everyone knows class vs. struct very well. Here are the "real" commands (this is how every other language does it):

```
Vector3 v = Vector3(4,0,1); // simple assign
Dog d1 = new Dog(); // create a Dog on the heap
```

That's nice – `new` means to actually make something. No `new` means you're not making something new.

But C# was partly written for beginners to just jump in. They don't know what `new` is, and why you'd use it one place but not the other. For them they added the rule that structs get a fake `new`. Everything looks the same.

24.10 Old-style pointers

C# has 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.

To repeat, you aren't allow to use these. They only run in "unsafe mode", and no C# programs are ever written in unsafe mode (this includes Unity programs).

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

```
int n1, n2; // normal ints
int* p1, p2; // pointers to ints (int* means "pointer to int")
```

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
```

This is cool because the types always match. `p2=p1;` is two int-pointers. `&n1` calculates a pointer to `n1`, so the types also match. It's confusing at first to remember those symbols, but once you do, it's easier to read.

We have to use a special symbol to follow a pointer's arrow:

```
p2 = &n1; // p1 aims at n1
p2 = 5; // error. Trying to make p2 point to "5"
(*p2) = 5; // follow arrow and change n1
```

But once you figure out the rule, `(*p2)=5;` spells out how `p2` is a pointer remotely changing something.

Those written-out rules allow us to use `=` both ways, pointer assign and copy:

```
*p1 = *p2; // copy the data between the boxes they point to
p1 = p2; // make p1 point where p2 does
```

You don't need struct vs. class with real pointers. A struct can be either:

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

And again, this is completely not important to know. But you might get a feel for reference types. These are the original rules.