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`. We've seen a different version of `new` with structs. This is a completely different use.

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` hasn't been assigned to point 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 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.

I'll add one more line to the example, another `d1=new Dog();`. It makes a second free-floating Dog with `d1` aimed at it. A funny thing is how `d1` was already aimed at a Dog. That's fine, it moves:

```
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)
```

The neat thing is how the original (Rover, 9) Dog had just `d1`, then was shared, and now only has `d2` pointing at it. This is what I meant about them not having names. The variables pointing to them have names, but they don't.

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 new 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. Classes are a kind we've never seen before: Heap variables.

As you may have guessed, declarations aren't commands – they don't actually create the variable when they're run. Instead the compiler scans your program, pre-making all globals and a chart for every function.

The only variable creation happens at function calls. The system grows the memory just enough to hold the pre-made chart of vars for it. When the function ends, memory shrinks back. If you have a chain of function calls, each adds its chart. If you think of memory as growing "up", it looks like function variable charts are stacked up.

If that picture helps, great. But the main point is that normal declared variables are pretty much all created before the program starts running.

The stack starts at one end of memory. Meanwhile, the Heap starts at the other. It starts empty, grows when you run the `new Dog()` command, and follows none of our old rules. A picture:

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

`new Dog()` is an actual command. It creates a Dog, in the Heap area, when it runs. If it runs twice, it makes another one.

A sample function, doing nothing except showing how these work:

```
void dogMaker(int dNum) {
  int n; // some normal variables to compare
  Cow c; // cows structs are normal
  for(int i=0;i<dNum;i++) {
    int x; // another normal variable, 1 copy at function start
    new Dog(); // <- each time it runs, makes a Dog
  }
}
```

When we run it, the system instantly creates the four declared variables. When it quits they snap away. If we run it five times in a row it pretty much re-uses the same memory for those variables.

But running the function creates no Dogs. The loop creates a bunch, then a bunch more each time we run it. The Dogs it makes just pile up in the Heap area:

```
                All of the program's memory:
  STACK                                                HEAP
-------------------------------------------------------------
globals | Update | dogMaker |              D D D D D D D D D |
                    dNum:                  D D D D D D D D D |
                    n:                     D D D             |
                    c:              (every Dog every created) |
                    x:                                       |
-------------------------------------------------------------
globals and function locals ->              <-created by new()
```

Heap variables don't have names. They don't care which function make 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. Techincally `Dog d1=new Dog();` is 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 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.

When I mentioned heap variables don't have names, it seems like `Dog d1=new Dog();` makes `d1`, Here's another example to show how declared pointer variables have names, but the actual heap Dogs don't. 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
```

The first heap Dog was pointed to by `p`, but now is only pointed to by `d1`. The second heap Dog is currently pointed to by `p` and `d2`.

24.2.3 Reference type

For real, every variable can be on the stack or the heap, and can have a pointer. For example you could have an int-pointer to an int on the heap.

But some languages limit this to make themselves simpler. They say each type needs to always be one or the other: normal variables are limited to never using pointers or living on the heap. The rest *must* be on the heap.

Heap-only variables are called Reference Types. In `C#`, classes are reference types. Structs (and everything else) aren't, because that's the rule they made.

This is a weird, but common rule. Some results:

- If `Dog d1;` automatically makes a Dog pointer, how do I declare a normal Dog? The answer is the language was purposely made so you can't.

- How do I create a pointer to an int? You can't. The language purposely doesn't want you to have integer pointers, so there's no way to make one.
- What about `v=new Vector3()`? That `new` is still fake. The result is the value (0,0,0). If it was a real `new` the result would be a pointer to a sharable heap Vector3. And you can't have pointers to structs, so that's impossible anyway.
- When I create a struct, how do I know if it's a reference type? The system keys off of the kind of thing. In `C#` putting `struct` in front automatically makes it normal. Putting `class` in front automatically makes to reference-type/heap-only.

That last item is what `class` really means – take a normal struct, but make it be a reference type instead. Since classes have to be on the heap, we can redo the rules for reference types.

A summary of what we've seen (and one new thing):

- 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`".
  Fun note: There's no command for the old meaning of `d1=d2;`. If you want to copy everything in `d2` into `d1`, do it by hand.
- `d1.name` knows to follow `d1` to where it points, then look up the name.
- `d1==d2` is true if they point to the same Dog. In other words, it's asking "do these point to the same thing?" It's false if they point to different Dogs, even if the values are all the same. 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.

We usually give every heap Dog a permanent owner. As a shorthand we'll call them by the owner's name. For example we'll say this makes normal Dogs `d1` and `d2`:

```
Dog d1 = new Dog(); // d1 will always point here
Dog d2 = new Dog(); // ditto
d1.name="Spot";
d2.name="Rin rin"; d2.name+="ny";
d1.age = Random.Range(2,10+1); // random age, for fun
```

For real, the third line is changing `name` of the heap Dog which `d1` always points to. But we'll say it's changing `d1`'s name, as a shortcut.

Most of the Dogs we declare will be permanent Dog links.

The other way we'll use them is as real pointers. They'll never "own" anything and will only point to other people's dogs. 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
```

The comments are how we're thinking about it. `d1`'s job is to be a locked-in way to find the heap Dog. `p`'s job is to wander around being an alternate way to find various Dogs.

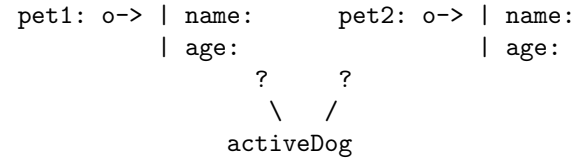
Here's a more realistic example. There are two "real" dogs and one pointer showing the active one. Pressing A or S switches the pointer:

```
// 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(); // 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++;
}
```

Hopefully 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:



If we make the two pets public (and Dog public and add [System.Serializable]) we'll see one of their age's always zoom up. But we're only ever changing `activeDog.age`.

24.4 Functions and pointers

Pointers types can be inputs and return values from functions. They're still always pointers, but we'll use our two ways of thinking. Maybe will work and act like normal variables. Other times we'll be using them as real pointers.

24.4.1 Pointer inputs

This completely turns a `Dog` into a string. It looks and works exactly as if Dog's was a struct:

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

string dWord = showDog(pet1);
```

The way it works is different, but the end result is the same:

When we call it, `pet1` is copied into `d`. Since they're pointers, `d` is pointing back to `pet1`. That's what pointer copy means. Then, when we read `d.name`, we're reading directly from `pet1`'s box.

With a struct, or any normal variable, we have a copy. With a class we have a pointer back to the real variable.

I didn't say it, but in the heading `Dog d` is declaring a pointer, since that's the rule. We can never declare a Dog, not even as a parameter.

Any function where we only *read* the input works the same with pointers as it would have before.

But since the function has a pointer to the real Dog, it can change the input. We couldn't do that before, since all we had were copies. This shortcut function puts values into a Dog:

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

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

All you need to know is that `d` is pointing back to someone else's Dog. This is the same as the shared Dog examples from before. like this:

```
d=pet1; // <- first function call does this
d.name="Gary"; d.age=4; // changing pet1
d=pets2; // <- second function call
d.name="Gina"; d.age=5; // changing pet2
```

Once we know `d` is a pointer, and the only real Dogs are pet1 and 2(*), this is easy.
(*) Remember we're using the shorthand where `pet1` really means "the heap Dog pet1 is permanently aimed at."

This next one 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";
}
```

`checlkMe` is pointing to the original Dog. `setupDog` used that to change it. This function reads and changes it. But it's the same shared-dog idea both times.

This next one is a little different. It copies one `Dog` into another:

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

We're taking advantage of the pointer to change the first Dog, but not the second one.

Copy functions like 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. The simplest dog-returning functions 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. And remember, Dogs on the heap don't belong to any function – they live as long as anyone has a pointer to them.

We can now use `Dog pet1=makeDog();` as a shortcut for `Dog pet1=new Dog();`. You have to point at a Dog made with `new`, but if you can get someone else to make it for you, that's a deal.

This version does that same thing, but look different since it uses a middle-man variable:

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

Remember these things: `d` points to a dog; equals with pointers means "point where I point"; and returning is like equals. All together, this returns a pointer to the Dog it made. Imagine `d` is an arrow – this returns the arrow.

`Dog pet2=makeDog2();` sets up `pet2` with a fresh Dog.

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.

But wait, where does the old `setupDog` fit? That function filled in a *pre-existing* Dog. This new one makes the Dog first:

```
Dog d1;
setupDog(d1, "A", 5); // ERROR - d1 aimed nowhere. No new's
d1=makeDog("B", 6); // legal

setupDog(d1, "AA",7); // now this is legal. d1 has a Dog
d1=makeDog("BB", 8); // also legal, but wasteful
```

The line throws away a Dog, which is no big deal, but makes a nice example. 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 version of this, that people use for real, is clone. It creates a Dog and fills it in, but using the values of another `Dog`. It "clones" that Dog:

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

A typical use is faking a normal =:

```
Dog d1, d2;
d1=makeDog("X", 6);
  d2=d1; // NOPE - gives shared Dog
d2=clone(d1); // YES. Second Dog, copy of first
```

If you understand this next thing, you're understanding pointers: once `d2` exists, we could copy scooby into it two ways: `copy(d2, scooby);`, or `d2=clone(scooby):`. The second way leave behind a garbage Dog.

Moving on, the other pointer-returning function gives you a pointer to an existing item. It doesn't create any Dogs. Instead it acts like a true pointer.

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);
dOld.name += " the elder"; // changing pet1 or pet2
```

In our minds, `dOld` is one of the pure-pointer Dogs. It's job is to aim at pet 1 or 2 and be used to change whichever one. It's like `activeDog` from before.

Here's the same idea, randomly choosing between two globals (pretend pet 1 and 2 are 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.

This next thing is a extra tricky: `randomDog().age++;` is legal!! It adds 1 the age of either pet 1 or 2. It only works because the function returns a true pointer, to something than already exists.

But for real, it's almost always better to store it in `pp` and use 2 steps.

To sum up: when a function returns a pointer, it can works 2 ways. Many times it creates and returns a Dog. It's like a `new` with bonus features. It doesn't involve much pointer-thinking.

The other way is returning a real pointer – here's a link to *someone else's* Dog.

24.4.3 Turn-input-into-output problem

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 the D-key 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++;
}
```

It always adds to pet 1 or 2, or temporarily stops when we press D.

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.)

24.7 Pointer compare (==)

Oh, no!! More rules. But this one is pretty easy. It's some examples of the last pointer rule for `==`. If you remember, it checks whether the *pointers* are equal,

which means whether they point to the same spot.

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.

24.8 Garbage Collection

This is fun section that's not super important. The only part to know: if you lose a Dog, it's not a problem, but try not to lose too many.

As I wrote, local variables in functions are easily created and destroyed as the program runs. Calling a function a thousand times pretty much re-uses the same space.

But using `new` to create something on the heap permanently uses that space. The computer doesn't know which ones you're using and which ones are lost since you don't have a pointer to them.

Suppose we accidentally use `makeDog` to change both parts of a Dog. It works, but there's a hidden `new` in it. This creates an extra Dog and loses the old one:

```
Dog d; d = new ...
  ...
if(...) // possibly redo all Dog values
  d = makeDog("Spike", 3); // Ooops. creates another dog
  // setDog(d, "Spike",3); // nicer way
```

The problem is if we run it a dozen times each Update (maybe in a loop.) To the computer, we're requesting about a thousand extra Dogs a second. If we don't do something, we'll run out of memory.

There are 2 solutions to this. #1 is to add a `delete` command and make it your problem. Don't get a fresh Dog by mistake, and if you don't need one anymore, use `delete` to free up the memory. If you forget, that memory is permanently used up, called a memory leak.

The advantage is it runs pretty quickly. The drawback is that memory leaks eventually cause an out-of-memory crash. That's why older versions of Windows needed to be restarted once a week.

Solution # 2 is to have the system track down lost Dogs. That sounds pretty nice, but if it was easy we'd have done it.

Every so often, when the program thinks we might have too much on the heap, it stops running your program and runs a **garbage collection**. It examines all of your local variables and marks everything they point to. Everything that wasn't marked gets deleted.

If you have a lot of variables, and Dogs can point to other Dogs (we'll see that much later,) this can take a while, which gives the program a little hick-up.

I've never had a problem, but constantly making and throwing away Dogs could result in a noticable stutter.

Another way to lose a Dog is making a local for a function:

```
void dogLoser(Dog inputDog) {
  int n;
  Dog d = new Dog(); // scratch local Dog
  Cow c; // cows are structs
   ...
}
```

All four normal variables are destroyed when this quits. `inputDog` is fine since the caller has a pointer to it. But the heap Dog we created becomes garbage. I think the system checks specially for those things. But sometimes

you make some garbage and it's fine.

A really strange thing about this system is you can't delete a Dog yourself. `pet1=null;` will lose the pet1 Dog. Sometime in the future, it will get garbage collected. If you want to save time by telling the computer to free the memory now, you can't.

24.9 Struct `new`'s

This section is more just for fun, in case you've wondering why they decided to mix fake `new`'s in with the real ones.

Pointers and the Heap have a reputation for being difficult to learn. They're not actually difficult, like multi-variable calculus, but they stick out as tougher than functions and nested if's. And crashes due to memory leaks are especially hard to track down.

Inventing the more limited Reference Types system makes garbage collection possible. It also makes it so you can sort-of use pointers and the heap without knowing much about them.

That's the key. They don't want you to see `c=Cat();` and `d=new Dog();` and ask why one has a `new` and the other doesn't. They'd have to explain stack vs. heap and pointers.

So everything has a extra `new`. Even basic types have them:

```
int n = new int(); // same as n=0
string w = new string(); // same as w=""
bool b = new bool(); // same as b=false
Cow c = new Cow(); // shortcut for name="", age=0
Dog d= new Dog(); // <- only this one is required
```

Even though the last works completely differently than the others, they all look the same. "Always use `new`" is an easy rule to remember.

Likewise, we memorize that functions can change their inputs, but only for classes (which is fine since most inputs you want to change are classes anyway.) Avoiding `=` and `==` with classes is another "just because rule." And you learn some class variables are "references" to other ones, since they just are.

Pointers and stack vs. heap and structs are advanced topics in that system (structs are advanced because you have to explain stack vs. heap to say how they're different from classes.) You have to unlearn the old class rules, but maybe that's fine since you're smarter now.

In theory it's a way to quickly train up to a good enough level. But either way, the extra `new`'s for structs are there so you don't wonder why only classes have them.

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.

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.