Chapter 29

Struct in struct

This section is another with no new rules – just useful tricks. Now that we have structs, and lists, and string indexes, we can combine them to make big data structures.

Usually these are pretty natural - maybe you have several people who can own several dogs each. With practice, that's easy to put into the computer and use.

Big data structures are also good practice writing double and triple loops.

29.1 Lists of structs

Making a list of `Cow` structs looks like this:

```
// review of Cow:
[System.Serializable]
public struct Cow { public string name; public int age; }

public List<Cow> Barn; // list of Cows
```

As we know, in the Inspector you have to type a non-zero number for the size of that list. Popping it open will show lots of `Cow`'s, which you can also pop open.

More interesting is how to search around the Barn in code. The trick is the same as for things like `c1.spotCol.r` for the old cow's with colored spots. Go left-to-right, using the options for that type.

Here's code using `Barn`:

```
int n=Barn.Count; // # of cows, same as normal
n=Barn[0].age; // age of first cow
Barn[1].age=4; // make 2nd cow be 4 years old
Barn[1].name="Lu-Lou Cow";
```

Barn is a list, so we can use Count, or can pick out one item using []. Barn[0] is a Cow, so we can use dot-age or dot-name.

To compare, some errors:

```
Barn.age; // error - have to say which cow
Barn[0].Count; // error - cows don't have a count
```

We can also copy entire cows in and out of the barn. This is really the same boring stuff we did with lists of ints, or normal structs:

```
Cow cc; cc.name="Kow"; cc.age=5;
Barn[2] = cc; // simple var-assign, with a whole cow
Cow c3=Barn[3]; // c3 is a copy of that cow
Barn[4]=Barn[5]; // copy entire cows between list boxes
```

We can create the entire Barn list ourselves, adding each Cow. This makes six 1-year olds named cowy:

```
Barn = new List<Cow>(); // size 0
for(int i=0; i<6; i++) {
  Cow c; c.name="cowy"; c.age=1; // a normal cow
  Barn.Add(c); // <-standard Add
}
```

A list of classes works the same, except they also need to be new'd. This makes a list of six Dog's (Dog is a class):

```
List<Dog> Kennel= new List<Dog>(); // size 0
for(int i=0; i<6; i++) {
  Kennel.Add(new Dog());
}
```

That's a little sneaky. new Dog() creates a Dog and returns a pointer to it. Normally we assign that to a variable, but adding it to the list is just as good, since that's the final place for it.

We could use two steps instead, with a middeman variable:

```
  Dog d=new Dog(); // 2-step version. d is a temp
  Kennel.Add(d);
```

Here's a bugged version. See if you can spot the problem. Hint: how many Dog's are there?:

```
Kennel = new List<Dog>();
Dog d=new Dog(); // dog is a class, so each new creates one
for(int i=0; i<6; i++) {
  Kennel.Add(d); // <-standard Add
}
```

This creates 6 pointers, all aimed at the same Dog.

A neat Unity trick is making a list of `GameObject`'s. This lets us drag many Cubes into the Inspector:

```
public List<GameObject> Blocks; // size in Inspector, drag in Cubes
```

We'd change them the usual way. `Blocks[0].transform.position=pos;` would move the first block. Or this would turn every other block red:

```
Color cc = new Color(1, 0, 0); // red
for(int i=0; i<Blocks.Count; i+=2) {
  Blocks[i].GetComponent<Renderer>().material.color=cc;
}
```

We can also fill that list ourself, by adding instantiated objects. They'll all be in the same spot, but we can move them later:

```
public GameObject ballPrefab; // drag in a Cube prefab
List<GameObject> Blocks;

void Start() {
  Blocks = new List<GameObject>(); // size 0, so far

  for(int i=0; i<10; i++) {
    GameObject gg=Instantiate(ballPrefab);
    Blocks.Add(gg);
  }
}
```

This solves a problem from before. We were able to create dozens of balls, but we had no place to save pointers to them all. A list is the perfect place for that.

The shortcut `Blocks.Add(Instantiate(ballPrefab));` would also work. I prefer the extra step of declaring `gg`. We'll probably want to use it to color or place each fresh ball.

29.2 Structs with Lists in them

Structs with lists inside them also aren't special, but look interesting. As usual, the list needs to be `new`'d. Here's a `Goat` struct which has a list of everything the goat eats:

```
struct Goat {
  public string name;
  public int age;
  public List<string> eats; // currently null
}
```

Some sample code playing around with `Goat g;`, showing how to use `eats`. This goat will eat tin cans and old shoes:

```
g.name = "Billy";
g.eats.Add("cans"); // error - null reference exception. eats not created yet
g.eats = new List<string>();
g.eats.Add("tin cans");
g.eats.Add("old shoes");
int n=g.eats.Count; // 2. g eats two things
```

Then some errors:

```
g.Add("grapes"); // error. g is not a list
g[0]="raisins"; // same error. g is not a list
int n=g.Count; // ditto
```

We could also create an eat-list first, then assign it all at once. Here `BillyFood` is like a temp, passing the list of foods along to `g2`:

```
List<string> BillyFood = new List<string>();
BillyFood.Add("crates"); BillyFood.Add("barrels");
Goat g2;
g2.eats = BillyFood; // g2 now eats crates and barrels
```

`new`'s can pile up on us. Suppose `Goat` was a class. We'd need to create and and to create it's list:

```
Goat g = new Goat(); // new for class
g.eats=new List<string>(); // still need to new the eats list
```

29.3 Lists of strings tricks

A string can use indexes and has a length. So an list of strings is sort of like a 2-dimensional list. A warm-up review:

```
public List<string> W;  // ["cow","banana","anteater", aardvark"]

int n=W.Count; // 4 words in the list
n=W[0].Length; // 3 letters in cow
```

Now the new-looking part, using two `[]`'s in a row:

```
char ch=W[0][2]; // w (3rd letter of cow)
ch=W[2][0]; // a (1st letter of aardvark)
```

The cool thing is it's not a new rule. `W[0]` is `"cow"`, which is a string. So of course adding `[2]` after grabs the `'w'` character.

Now we can write this cool nested loop to count the total number of `a`'s in every word in the list:

```
int aCount=0;
for(int i=0; i<W.Count; i++) { // each word
  for(int j=0; j<W[i].Length; j++) { // each letter in word
    if( W[i][j]=='a' ) aCount++;
  }
}
```

As usual, it's easy to mix up the `i`'s and `j`'s and get wrong counts and out-of-range crashes. `W[word][letter]` might be better variable names if you don't use `i` and `j` for every nested loop.

A cute trick with arrays of strings is to use them to make a map. This usually isn't the best way, but it's fun and a nice example.

For a 3x4 map, we can make a list with 3 strings, all length 4. We'll make the rule that o is an open space, and X has a block in it:

```
List<string> M=new List<string>();
M.Add("xoox");
M.Add("xxxo");
M.Add("xooo"};
```

We can use a nested loop to make that picture. Cubes go on the X's. This is a copy of the nested grid loop from last chapter, except we check the map before deciding to make a block:

```
for(int i=0; i<M.Count; i++) { // 3 rows
  for(int j=0; j<M[i].Length; j++) { // each letter in word: x or o
    if( M[i][j]=='x' )
      newBlockAt(i,j); // from previous chapter
  }
}
```

It doesn't look as nice as it could since `newBlockAt` leaves a little space between. In this case it would be better to have them touching, like one long solid wall.

To get a real feel for working on a grid, we can count how many walls are next to spot (x,y). We take a pretend step in each of the 4 directions:

```
int nearWalls=0;
if(M[y][x-1]=='x') nearWalls++; // left
if(M[y][x+1]=='x') nearWalls++; // right
if(M[y+1][x]=='x') nearWalls++; // down (b/c of the picture)
if(M[y-1][x]=='x') nearWalls++; // up
```

It crashes if we're on an edge (fixed with `if`'s). Math like this is common for walking around in a grid: `M[y][x-1]` is the space to the left of us, and so on.

29.4 2D lists

A real 2D list is just a list of lists. The cool thing is there are no new rules - we can make them out of what we have now.

For example, `List<List<string>>` is a list of lists of words. The inside part is `List<string>`, a normal list of strings. Then `List< ... >` goes around it. So it's a list, containing lists of strings (if you can figure out the TV show about a serial killer who kills serial killers, you can figure out this).

To get started, suppose we have some lists of words for a Mad Lib:

```
public List<string> Subjects; // ["Bob", "The cat", "My face"];
public List<string> Verbs; // ["runs", "sits", "cries", "inspects", "tolerates"];
public List<string> Adjectives; // ["cowardly", "heavy duty"];
public List<string> DirectObjects;
  // ["clouds", "fish", "tears", "truth", "gold", "needles"];
```

We eventually want to use these to randomly make 4-part sentences, choosing one from each, like "The cat inspects heavy duty truth".

We'll put those four lists into a big list:

```
List<List<string>> AllParts=new List<List<string>>(); // a list of lists
AllParts.Add(Subjects);
AllParts.Add(Verbs);
AllParts.Add(Adjectives);
AllParts.Add(DirectObjects);
```

Now `AllParts` has four string lists in it. `AllParts[1]` is the verb list, and `AllParts[1][0]` is the first verb, `"runs"`.

A picture would look like:

```
AllParts
[0] -> Bob, The cat, My face
[1] -> runs, sits, cries, inspects, tolerates
[2] -> cowardly, heavy duty
[3] -> clouds, fish, tears, truth, gold, needles
```

Playing with it to get a feel, we can do things like this:

```
AllParts[2].Add("slow"); // add a new adjective
AllParts[0][1]="The dog"; // replace cat with dog
AllParts.Count; // 4. Four lists of words
AllParts[3].Count; // 6. clouds, fish ... is 6 things
AllParts.Add(new List<string>()); // add 5th sentence part
AllParts[4].Add("?"); AllParts[4].Add("!!!"); // possible sentence enders
```

We can make a random sentence using a loop to pick from each part:

```
string madLib="";
for(int i=0;i<AllParts.Count;i++) {
  if(i!=0) madLib+=" "; // space in front of every word except the first
  int randPos=Random.Range(0, AllParts[i].Count);
  madLib+=AllParts[i][randPos];
}
```

Notice how the last line uses the usual `[][]`. Down the side to the sentence slot we're on now, then across to the random word for that slot.

There's one final super-cool thing a list of lists lets us do. Each of the inside lists counts as a normal list, and we can do normal list things with it. For example, we could write a function to pick a random word from a normal list:

```
string getRandWord(List<string> W) { return W[Random.Range(0, W.Count)]; }
```

This is a totally normal list function, that knows nothing about lists of lists and won't work with them. But we can still use it in our 2D list sentence maker:

```
  for(int i=0;i<AllParts.Count;i++) {
    ...
    madLibs+=getRandWord(AllParts[i]); // <- calling normal list function
  }
```

This works since `AllParts[i]` counts as an ordinary list of strings. `getRandWord` is happy to take it as an input and pick out a random work from that one list.

A 2D integer list of lists feels more griddy, but it's made mostly the same way. This makes a 4x4 grid:

```
List<List<int>> G=new List<List<int>>(); // 2D list
for(int i=0;i<4;i++) G.Add( new List<int>() );
// now we have 4 empty lists. Grow them all by four 0's:
for(int i=0;i<G.Count;i++) // each column
  for(int j=0;j<4;j++) G[i].Add(0); // add four 0's, up this column
```

Probably the most confusing line is `G.Add(new List<int>());`. The inside of it creates a fresh int-list. Normally we assign that to a variable, but we're allowed to skip the middle-man and jam it straight into the list.

We can think of this as a real grid:

```
[3]  0   0   0   0
[2]  0   0   0   0
[1]  0   0   0   0
[0]  0   0   0   0
G-> [0] [1] [2] [3]
```

I put G across the bottom, with the lists going up, so we could use `G[x][y]` in the more normal way. For example, `G[3][0]` is the lower right.

These next two single loops put 7's up the right side, and 8's across the middle (the last 8 overwrites a 7). Notice which slots have the variables:

```
for(int y=0;y<3;y++) G[3][y]=7; // up last column
for(int x=0;x<3;x++) G[x][1]=8; // across 2nd row

// result:
[3]  0   0   0   7
[2]  0   0   0   7
[1]  8   8   8   8
[0]  0   0   0   7
G-> [0] [1] [2] [3]
```

We can even use a normal `List<int>` function on each column, since each column is a list (but not on a row.) This sets everything in an ordinary list to a value:

```
void setVal(List<int> A, int val) {
  for(int i=0;i<A.Count;i++) A[i]=val;
}
```

Now `setVal(G[3],9);` replaces those 7's with 9's. Of course, it only works for columns – each column is one list. There's no way to send a row to `setVal` since each row is 1 box from each of the column lists.

29.5 Larger structures

We can use these rules to make structs with lists of structs, and larger. It's not that complicated, since you make them based on the data you already have.

For example, making a list of goats (which have lists of what they eat):

```
// repeat of Goat class:
class Goat {
  public string name;
  public List<string> eats;
}

// make goat list:
List<Goat> GG=new List<Goat>();
for(int i=0; i<8; i++) { // make 8 empty goats
  Goat g=new Goat();
  g[i].name = "goat #"+(i+1);
  g[i].eats = new List<string>();
  G[i].eats.Add("goat chow");
  GG.Add(g);
}
```

Notice we have to `new` the goat-list, `new` each goat in the list, and `new` each `eats` list for each goat.

A partial picture:

```
GG 0 -> | name: goat#1
        | eats -->  ["goat chow"]

   1 -> | name: goat#2
        | eats -->  ["goat chow"]

   2 -> | name: goat#2
        | eats -->  ["goat chow"]
```

A few sample lines that do and don't work:

```
int n = GG.Count; // 8 goats
n = GG[0].Count; // ERROR -- goats don't have a length
n = GG[0].eats.Count; // 1 food
string s=GG[0].eats[0]; // "goat chow"
n = GG[0].eats[0].Length; // 9 letters in "goat chow"
GG[2].eats.Add("shoes"); // legal. also eat shoes
GG[2].Add("cake"); // ERROR - goats aren't lists
```

We can run some interesting loops through this. This nested loop counts how many goats like a certain food:

```
int howManyLikeThis(List<Goat> G, string food) {
  int foodCount=0;
  for(int i=0; i<GG.Count; i++) { // check each goat
    // check all foods this goat eats:
    for(int j=0; j<G[i].eats.Count; j++) {
      if(GG[i].eats[j] == food) {
        foodCount++;
        break; // don't double-count this goat; quit eats loop
      }
    }
  }
  return count;
}
```

It checks the obvious way: scan every goat, look through what that goat eats, count it and skip to the next if you get a food match.

Here's a triple loop to count the total Z's in all food items (if five goats like "pizza", one of them twice, this would count as 12 z's):

```
int zCount=0;
for(int i=0; i<GG.Count; i++) { // each goat
  // each food:
  for(int j=0; j<GG[i].eats.Count; j++) {
    // each letter:
    for(int k=0; k<GG[i].eats[j].Length; k++) {
      if(GG[i].eats[j][k]=='z') zCount++;
    }
  }
}
```

`GG[i].eats[j][k]` has four total look-ups, but it's fine since we really need that many: which goat, what it eats, which food, which letter.

29.6 Giant game board example

Previously we used a nested loop to create a 2D grid of Cubes. But we couldn't save pointers to them, which meant we couldn't find and change them later. Now we can, which means we can almost make a game.

Our game will eventually allow you to toggle spaces (selected or un-selected) letting you make crude pixel shapes. First a simple class to hold data for each space:

```
class BoardSquare {
  public GameObject cube; // pointer to Cube at this space
  public bool selected; // selected squares change color
}
```

Then we'll make a simple 2D list, to hold them, for when we make the cubes, later:

```
// 2D grid of boardSquares:
List<List<BoardSquare>> Board; // want 8 wide, 6 high grid

void Start() {
  Board = new List<List<BoardSquare>>(); // main 2D list
  for(int i=0;i<8;i++) {
    Board.Add(new List<BoardSquare>()); // add empty column
    // grow column to 6 high:
    for(int j=0;j<6;j++) Board[i].Add(new BoardSquare());
  }
```

Now `Board.Count` is 8 (going across the bottom.) `Board[0].Count` is 6 (going up.) `Board[7][5].selected=true;` selects the upper-right corner.

Making the blocks and hooking them up is nothing special. We'll use `Board[x][y].cube=gg;` as we create each real Cube, then math to get the proper arrangement:

```
public GameObject cubePrefab; // drag in

  // the rest of Start:
  for(int x=0; x<8; x++) {
    for(int y=0; y<6; y++) {
      Vector3 pos; pos.z=0;
      pos.x=-4.0f+x*1.2f; // x goes across, from left
      pos.y=-3.0f+y*1.2f; // y goes up, from bottom
      Board[i][j].cube = Instantiate(cubePrefab, pos); // create and place shortcut
      Board[i][j].selected=false;
    }
  }
```

That gives us something new. We have 6 by 8 Cubes on the screen, and we now have a way to find and change them all. `Board[x][y].cube` is that Cube.

The "game" will be to move around and toggle the current square on/off. We'll be able to make a crude picture, which is still pretty cool. I'll break it into using the keys to move, and using space to toggle the color of the cube we're on. Update will call each function:

```
int row=0, col=0; // where we are on the board now
void Update() {
  moveCheck();
  selectCheck();
}
```

`selectCheck` will toggle the color of the current cube. Without movement, we can never leave square (0,0), but that's good enough to test. We can tap space and watch the corner cube change color:

```
void selectCheck() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    BoardSquare bs=Board[col][row]; // <- using row/col to look up current space
    bs.selected = !bs.selected; // flip T/F
    // now set to correct on/off color:
    Color cc=Color.blue;
    if(bs.selected) cc=Color.red;
    bs.cube.GetComponent<Renderer>().material.color = cc;
  }
}
```

The best part of this is the top line with `bs=Board[col][row]`. It picks out the current spot, based on row and col, exactly how we want in a 2D grid. Once we have that, we use `bs.selected` and `bs.cube` to look at the parts.

Without using `bs` as a shortcut, the last line would start with: `Board[col][row].cube.GetComponent`. That's long, but reading it left-to-right should look fine.

Movement is the arrow keys. left/right change the column, up/down change the row. I decided to have off-edge wrap around, which requires 4 ugly `if`'s. The current cube is a tad smaller, to show that we're on it:

```
void moveCheck() {
  int oldRow=row, oldCol=col; // save old row/column

  if(Input.GetKeyDown(KeyCode.RightArrow)) {
    col++; if(col>=8) col=0; } // wrap-around
  if(Input.GetKeyDown(KeyCode.LeftArrow)) {
    col--; if(col<0) col=7; }
  if(Input.GetKeyDown(KeyCode.UpArrow)) {
    row++; if(row>=6) row=0; }
  if(Input.GetKeyDown(KeyCode.DownArrow)) {
    row--; if(row<0) row=5; }

  // if anything changed, show new spot:
  if(oldRow!=row || oldCol!=col) {
    //  Reset old square to normal size:
    Board[oldCol][oldRow].cube.transform.localScale = new Vector3(1,1,1);
    //  Make new square smaller:
    Board[col][row].cube.transform.localScale = new Vector3(0.8f,0.8f,0.8f);
  }
}
```

Doing something special to the current item is always a pain. That's what the second half of the code is doing. We need to check whether we moved, reset to old square's size (which is why we needed to save it in old row and col), and finally change the new square.

Be sure and look at `Board[col][row].cube.transform.localScale`. Long, but pretty.

This type of set-up for a board is common. Each square might need to know the terrain type, buildings in it, who captured it, and so on. A class or struct is good for that. A link to the visible part (`cube`, here) is just one more field.

29.7 List of indexes

This is an old trick, not very common anymore, but it's a fun exercise. Suppose you want to make a list of *some* of the things in a list. If you wanted items 1,4 and 6. You'd make a list `[1,4,6]`, which is an array of indexes. Here it is in code:

```
public List<string> Ani; // ["ant","bear","cow","deer","eel","ferret","goat","hawk"]
public List<int> InZoo; // [1,4,6] // stands for bear,eel,goat

void Start() {
  // print animal names in the zoo:
  for(int i=0;i<InZoo.Count;i++)
    print( Ani[InZoo[i]]); // bear eel goat
}
```

We've never seen anything like `Ani[InZoo[i]]` before. It's a nested look-up, doing the thing above. `InZoo[i]` goes through 1, 4, 6. So `Ani[InZoo[i]]` goes through items 1, 4 and 6: bear, eel, goat.

Here's a longer version of the same trick. Suppose we want to print the animals in a random order. A cheap way is to shuffle a list with 0-7, then read the animals in that order:

```
List<int> RandIndexes=new List<int>();
for(int i=0; i<Ani.Count; i++) RandIndexes.Add(i); // 0-7
shuffle(RandIndexes); // old function. ex: 2,6,0,1,5,7,4,3

for(int i=0;i<RandIndexes;i++)
  print(Ani[RandIndexes[i]]);
```

The last line is the same nested look-up as before.

This is really a version of pointer thinking. We can't have pointers to strings, but we can use the indexes like pointers. It's not common, but it's a nice way to practice.

29.8 Internal list pointers

If we have a list of objects, we can pick out some of them with another list of pointers. For example `AllDogs` is a list of every dog. `SledTeam` won't have any new dogs – it will only point to things in `AllDogs`:

```
public List<Dog> AllDogs; // created and filled in Inspector
List<Dog> SledTeam; // will point to some things in AllDogs

void Start() {
  SledTeam=new List<Dog>();
  for(int i=0;i<4;i++) SledTeam.Add(null); // 4 empty Dogs pointers

  // start with first 4 dogs, but can change:
  for(int i=0;i<4;i++) SledTeam[i]=AllDogs[i];
}
```

You can't tell by the definition. but in our minds `SledTeam` isn't holding Dogs. It will never have any Dogs of its own. It's purpose is to pick 4 dogs out of the main Dog list.

This would print the names of all dogs on our team:

```
  for(int i=0;i<SledTeam.Count;i++) {
    if(SledTeam[i]==null) print("(missing)");
    else print(SledTeam[i].name);
```

Here, checking `SledTeam[i]==null;` makes perfect sense. `null` is a perfectly good value – we haven't picked a Dog for that position yet. We're also imagining `SledDog[i].name`as reaching over into `AllDogs`:

```
AllDogs:  0  1  2  3  4  5  6  7  8  (9 real dogs)
             ^  ^     ^  ^
              \/     /  /
              /\    /  /
SledTeam:    o  o  o  o  (4 arrows)
```

There's another version of this trick where a class has pointers to itself inside of it. For example, all of our Rabbits can have another Rabbit best friend:

```
[System.Serializable]
public class Rabbit {
  public string name;
  public int age;

  [System.NonSerialized]
  public Rabbit bestFriend;
  // not part of me -- an arrow to another rabbit
}

public List<Rabbit> AllRabbits; // set up in Inspector
```

Clearly, we can't have an actual rabbit in a rabbit, since that would have another rabbit inside of it, going on forever. But it's fine to have `bestFriend` be an arrow. We'll never use `new` on it. Its purpose is to point out some other pre-made rabbit (or be `null` if we have no best friend).

A neat thing is that Unity can't tell. It assumes gameObject variables pointer to gameObjects it created. But it assumes a rabbit is a rabbit and tries to make one for `bestFriend`, triggering infinite rabbits. A special safely check kicks in after it makes 7 levels of nested rabbits and gives an error.

`[System.NonSerialized]` is there to explain to Unity that `bestFriend` is like a gameObject pointer – it's only an arrow. It says not to create it. All of this can get complicated quickly, but deep down it's the Real vs. Arrow issue that all reference types have.

This code makes every pair of rabbits be mutual best friends:

```
for(int i=1;i<AllRabbits.Count;i+=2) { // 1,3,5,7 ...
  Rabbit r1=AllRabbits[i-1], r2=AllRabbits[i];
  r1.bestFriend=r2; // Ex: Bunny[0] and [1] mutually like each other
  r2.bestFriend=r1;
}
```

Or we could randomly assign best friends, with some anti-social rabbits whose best friends are `null` (1 in 6 chance):

```
for(int i=0;i<AllRabbits.Count;i++) {
  Rabbit fr=null; // no best friend, so far
  if(Random.Range(1,6+1)!=1) { // 1 in 6 for no friend
    int friendNum=Random.Range(0,AllRabbits.Count);
    if(friendNum!=i) fr=AllRabbits[friendNum];
    // can't be friends with yourself
  }
  AllRabbits[i].bestFriend=fr;
  // note: assigning null is legal, and fine
}
```

Basically, `bestFriend` could aim at any other rabbit, or `null`, and can double-up. That's not as fair as pairs of best-friend rabbits, or maybe it is, if you're a rabbit.

Now that we've set best friend arrows, we can use them. This would count how many rabbits have a best friend named Thumper:

```
int count=0;
for(int i=0;i<AllRabbits.Count;i++) {
  Rabbit bf=AllRabbits[i].bestFriend;
  if(bf!=null) { // we may not have a best friend
    if(bf.name=="Thumper") count++;
  }
}
```

Finding the most popular rabbit needs a nested loop. We'll check every rabbit, counting how many other rabbits like it, remembering the highest total:

```
int bestIndex=-1; // most popular rabbit, so far (Price is Right strategy)
int bestVotes=-999; // votes for that rabbit
for(int i=0; i<AllRabbits.Count; i++) {
  Rabbit currentBunny=AllRabbits[i]; // shortcut to this bunny
  int votes=0; // how many other bunnies like this one:
  for(int j=0; j<AllRabbits.Count; j++) {
    if(AllRabbits[j].bestFriend==currentBunny) votes++;
  }
  if(votes>bestVotes) { bestIndex=i; bestVotes=votes; }
}
if(bestIndex<0) print("no rabbits like any other rabbits");
else print("Most popular is "+AllRabbits[bestIndex]);
```

It's a nested loop because each rabbit knows who it likes, but not who likes it. We need another loop for that.