Chapter 27

Using indexed lists

This chapter is about handling a real list of items. As a warm-up, we've been playing around with strings, treating them as a list of characters. Real lists are better – they can be of any type. But the loops are mostly the same.

There are two basic types of ordered lists in programming. This chapter is about the most common type, where list items are laid side-by-side. This lets us easily loop through them, grow or shrink from the end, but we can't add or subtract from the middle.

The main new thing, and the reason we like these, is each box in the list counts as a full variable. We can assign to them.

27.1 Intro

Every list holds one type, which goes inside a set of angle-braces. This declares a list of integers:

```
List<int> N;
```

It's our first 2-part type. You can't use just `List` by itself, you always need the second part, with a type in those brackets.

Skipping some steps, let's pretend `N` is size 8, and filled with 0's. Looping through is about the same as with strings: check the size using the new dot-`Count`, but each box is still `N[i]`. As normal, indexes go from 0 to size-1, so our `N` has boxes 0 to 7.

This is a basic index loop to print the contents:

```
print(N.Count); // 8 (pretend we finished creating it)
for(int i=0;i<N.Count;i++) print(N[i]); // 0 0 0 0 0 0 0 0
```

The new thing we can do is assign to each box. This changes all the 0's to 6's. It's the same loop, except we get to use `N[i]=`:

```
for(int i=0;i<N.Count;i++) N[i]=6; // 6 6 6 6 6 6 6 6
```

We can't add to a List as easily as a string. We can grow the end by one box at a time. The funny-looking `Add` function does that:

```
N.Add(12); // now N is: 6 6 6 6 6 6 6 6 12
N.Add(3); //  now N is: 6 6 6 6 6 6 6 6 12 3
```

Lists have one drawback. You need a loop to do most things, like printing them, comparing or adding them together.

27.2 Creating Lists

Now for the details we skipped and explanations:

As mentioned, a new rule says `List`'s need to have a type in angle brackets to be finished. As you might guess, leaving it out is an error: `List A;` is no good.

It can be any type. Examples of legal list declarations:

```
List<int> L;
List<string> AnimalNames;
List<float> F1, F2; // multiple rule is fine - 2 lists of floats

List<Vector3> P; // list of points
List<Cow> Barn; // list of the Cow class we made
List<GameObject> Balls; // list of links to balls in the game
```

It's not great that we have to re-use `<` and `>` as another type of parenthesis. But if you've ever written HTML or seen XML or JSON – yep, angle brackets. The computer can always tell them from real compare symbols, and we can too, after some practice.

In a program, angle-brackets go around an extra required type. For example, in `GetComponent<Renderer>`, renderer is the type of thing we're looking for.

`List`'s are a reference type (oh, no!) But it's OK. We have to `new` them, but there won't be any other monkey business. Examples of making usable, size 0, lists:

```
List<int> L = new List<int>(); // all in one line.
AnimalNames=new List<string>(); // we declared it above.
List<float> F1=new List<float(), // The declare several...
            F2=new List<float>(); // ...and init them all trick.
List<Vector3> P=new List<Vector3>();
```

Notice how the same type goes in both places - the declare and the `new`. That was always the rule (like `Dog d = new Dog();`), but we didn't notice as much before since the names weren't as hard to type. It's fine since the drop-down helps you for the second one.

That's also a lot of special symbols packed together. Just remember the `new` has the type followed by the required ending double-parens. So `List<int>` plus `()`.

There are two common ways to think of lists. One is as a set of variables, for example, how much rain fell during each month. It should start as twelve 0's. The other is more like a grocery list - it starts empty, you write things as you think of them, and it's as big as it is.

Starting at size 0 makes sense for that second type. Here's a fakey example where we create a list and add however many random numbers until they add to 100 or more:

```
List<int> N=new List<int>(); // size 0
int sum=0;
while(sum<100) {
  int n=Random.Range(1,10+1); // 1 to 10
  sum+=n;
  N.Add(n);
}
```

Starting from size 0 and using `Add` feels pretty natural for "as many as we need" lists.

For the other kind, "I need twelve 0's for monthly rainfall" we'd rather make it size 12 to begin with. We can't do that, but `Add`'ing twelve 0's, once, at the start works fine:

```
// Make list with 12 zeros:
List<float> MonthRain=new List<float>();
for(int m=0;m<12;m++) MonthRain.Add(0);
```

`Add` grows the list by one box. It's funny in several ways. The first is how it runs like a member variable. This is a common thing we'll see a lot more of later. Hopefully we can see `N.Add(3)` grows `N`, while `Q.Add(3)` grows `Q`.

Another is how it does two things. It grows the list by one box, and puts the new value there. If we just want to make 12 boxes, we might use 0 as a dummy: `N.Add(0)`. But we often want both, like `Flowers.Add("Petunia");`.

`Add` doesn't say where it adds, since everyone knows the back is the obvious place. Any other spot would involve shifting, which would confusingly increase some indexes.

The last funny thing about `Add` is how it uses the type of the list. You still have to give it the correct type. It just happens that each list uses a different version of `Add` for the type it is.

Examples:

```
List<int> N = new List<int>();
N.Add(4);
N.Add("cow"); // error - not an int
N.Add(3.0f); // error - not an int

List<string> W = new List<string>();
W.Add(4); // error - not a string
W.Add("cow"); // this works
W.Add('s'); // error. Arrg! char's aren't strings
W.Add(""+'s'); // but this old trick still works
```

But the usual int to float casting applies for float lists:

```
List<float> F=new List<float>();
F.Add(3); // fine. Adds 3.0f
F.Add(1.5f+""); // error. As usual, becomes "1.5" before trying to add
```

The command to see the length of a list is `N.Count`. Sometimes that seems like a verb, like you're telling it to go out and count, but it means the same thing as length or size. They just liked the way count sounded.

The indexes are always from 0 to Count-1, the same as usual.

It's a little fun to grow lists and check values. For example:

```
List<string> W=new List<string>();
print(W.Count); // 0
W.Add("latin"); // [latin] count is 1
W.Add("greek"); // [latin, greek] count is 2
print( W[0] ); // latin
W.Add("latin"); // [latin, greek, latin] count is 3
for(int i=0;i<3;i++) W.Add("r"); [latin, greek, latin, r, r, r]
```

The last loop really shows how `Add` works. Even though the loop is 0 to 2, the three things it adds just grow `W` from the end.

Each slot in a list counts as a real variable, which means we can do all of our old variable tricks on them. For example with ints:

```
N[0]=3; N[0]++; // now it's 4
N[0]+=5; // add 5 to get 9
N[0]*=2; // double to 18
N[0]--; // down to 17

N[3]=6;
N[0]+=N[3]; // N[0] goes from 17 to 23
```

A list of strings lets us do string tricks to each box (pretend `W` is created):

```
W[2]="cat"; W[2]+="food"; // W[2] is "catfood"
W[2]="red "+W[2]; // red catfood
print(W[2].Length); // string length -- 11 letters

W[2]=""; // red catfood is completely erased, as normal
W[1]="99";
W[2]=N[0]+W[1]+-7; // 2399-7, standard string math, since W[1] is a string
```

The same as for strings, going off the edge is a run-time *null reference exception* error. For examples `N[-3]` or `N[999]`. The same off-by-one problem is in lists: if `N` has size 8 then `N[8]` is just barely off the edge, so crashes.

27.3 More playing with indexes

We can re-use more exercises from the string chapter. These next ones aren't new, except we can add two boxes in a list:

```
// N: 10 30 100 500 9000
int a=1, b=4, c;
c=N[a]+N[b]; // c is 9030. adds boxes #1 and #4
c=a+N[b]; // c is 9002. adds 2 to box #4
c=N[b-a]; // c is 500. looks up box #3
c=N[a-1]+N[a]+N[a+1]; // c is 140.
// add box #a and the two around it
```

Using an index to assign to a list box is new, but not too complicated. More exercises with that:

```
int a=1, b=4;
N[a]=N[b]; // copy box #4 into box #1
N[b-a]=N[0]; // copy box #0 into box #3
N[b-a]=N[a]+N[b]; // copy box #1 + #4 into box #3

N[0]=N[N.Count-1]; // copy last box into the 1st
N[N.Count-1]=N[0]; // copy 1st box into last
```

Looking at how the values change is always helpful. Let's start with a sample list, with 0, 10, 20 . . . 50 in the boxes:

```
List<int> L=new List<int>();
for(int i=0;i<=50;i+=10) L.Add(i);
```

That's fun since we got to use a sequence-style loop. It hits every number, `Add`'s them, and lets the list grow to however many that was.

A common confusion are things like `N[1]=3;` vs. `N[3]=1;`:

```
N[1]=3; N[3]=1;
0  1  2  3  4  5  <- indexes
0  3 20  1 40 50  <- values
```

Or using math in both places (starting with the original list):

```
N[3*3-7]=4-9;
0  1  2  3  4  5  <- indexes
0 10 -5 30 40 50  <- box #2 gets -5
```

The trick is the same as before: calculate which box it is, then do the rest of the line.

Then we can watch assigning one box to another:

```
N[2]=N[4]; // copies the 40 into the box with 20:
// 0 10 40 30 40 50
N[3-1]=N[3+1]; // same thing

int k=3;
N[k]=N[k+1]; // pull the 40 into the next lower box
// 0 10 20 40 40 50
k=0; N[k]=N[k+1]; // same thing, but pulls the 10 left into box #0
// 10 10 20 30 40 50
```

`N[k]=N[k+1];` is a "slide-left". If you think of yourself as being on `k`, it pulls the next higher box into you. `N[k-1]=N[k];` is also a slide-left, except it "pushes" your value into the next lower box.

`N[k+1]=N[k];` is like a push slide-right and `N[k]=N[k-1];` would be a pull slide-right – pulls from the next lower box into you.

Swapping two boxes in a list is the same 3-step dance. This switches the first two:

```
temp=N[0]; N[0]=N[1]; N[1]=temp;
```

Then this switches slots `a` and `b`:

```
temp=N[a]; N[a]=N[b]; N[b]=temp;
```

27.4 List loop examples

27.4.1 Initializing loops

If we want a pre-made list with certain values, there are a few tricks. From before, we can make it by `Add`'ing the values.

This makes a list with ten empty strings:

```
List<string> W = new List<string>();
for(int i=0; i<10; i++) W.Add("");
```

With int lists, it's semi-common to start each box with its index. To do that, use `i` in the `Add` command:

```
List<int> NN = new List<int>();
for(int i=0; i<10; i++) NN.Add(i);
```

If we wanted it to be 1,2,3 instead, we'd use `NN.Add(i+1);`. For 10, 20, 30 it would be `NN.Add((i+1)*10);`

When we have a "sequence equation", we can have a loop run through those numbers; or can make a simple n-times loop with the formula inside. These both make a list with 20, 18, 16 . . . 2:

```
List<int> Nums=new List<int>();

// sequence-loop method:
for(int i=20; i>=2; i-=2) Nums.Add(i);

// n-times loop method:
for(int i=0; i<10; i++) Nums.Add(20-i*2);
// check: when i=9, 20-9*2 is 2
```

The second is a pain since we had to pre-compute 20 to 2 by 2's is 10 numbers. But it can be nice having it right there in the code, for when we read it later.

An alternate method, we can grow the list to the correct size using `Add` with a dummy value. Then we use loops and so on to fill the boxes. This size-20 list has goat in the even boxes and cow in odds:

```
List<string> Ani=new List<string>();
for(int i=0;i<20;i++) Ani.Add(""); // grow to size 20
// set evens to goat, odds to cow:
for(int i=0;i<Ani.Count;i+=2) Ani[i]="goat";
for(int i=1;i<Ani.Count;i+=2) Ani[i]="cow";
```

For testing, it can be nice to fill a list with random numbers. The exact range depends on what you're testing. This uses 0 to 100:

```
List<int> R=new List<int>();
for(int i=0; i<20; i++) R.Add(Random.Range(0,101));
```

Character lists aren't very common, since strings are usually better. You might use one for a Hangman game starting with all underscores:

```
List<char> Letters = new List<char>();
for(int i=0; i<12; i++) Letters.Add('_');
```

The drawback is we can't print it, but the advantage is being able to change `Letters[2]='e';`.

A list of `bool`s isn't very exciting, but can be useful. Here we'll make something to check whether a letter was guessed:

```
List<bool> LetterWasUsed=new List<bool>();
for(int i=0; i<26; i++) LetterWasUsed.Add(false); // <-look, a false!
```

If we want a list full of things with no pattern, we can hand-`Add` each one:

```
List<string> ColorWords=new List<string>();
ColorWords.Add("red");
ColorWords.Add("green");
ColorWords.Add("violet");
 ...
```

Or, in Unity we can cheat with `public List<string> ColorsWords;`, without the `new` since Unity does it for us. Then fill it in with the Inspector.

We have to enter a non-zero size. The we can pop it open and enter values for each slot..

A nice use of `Add` is getting lines from a text file, if we know how. Pretend `readLine()` reads one line, or empty-string when it ends:

```
string nextCol=readLine();
while(nextCol.Length>0) {
  ColorWords.Add(nextCol);
  nextCol=readLine(); // at end, so we stop and don't add ""
}
```

Reading from files for real depends on the exact set-up, and often has messy code to start it, but it looks about like this.

27.4.2 Printing

Like structs, there's no built-in command to print an entire list. You usually use an index loop to look at each part. This turns an int list into a string with commas and parens. To get the commas correct, I'm going to do the first one by hand, then have the loop start at 1:

```
string w = "("+N[0]; // add 1st item
for(int i=1; i<N.Count; i++) // loop for everything else
  w+= ", "+N[i];
w+= ")";
print(w); // Ex: (0, 3, 0, 0)
```

A `bool` list is fun to print, since we can shorten true and false to T and F. I'm going to leave out the commas:

```
string w = "(";
for(int i=0; i<LetterWasUsed.Count; i++) {
  string tf;
  if(LetterWasUsed[i]) tf="T";
  else tf="f"; // lowercase f stands out from T better
  w+= tf;
}
w+= ")";
print(w); // Ex: (fffTffTT)
```

It's semi-common to dress up `float`s a little bit when printing any list of them. A list full of 8.333333333's is going to overflow the screen. Unity sometimes rounds floats to tenths when printing them. We can do that for our lists.

I'm going to switch to a loop starting from 0 (see if the comma-logic makes sense):

```
string w; w = "(";
for(int i=0; i<N.Count; i++) {
  if(i!=0) w+=", "; // all but first gets a comma in front
  float nn = ((int)(N[i]*10))/10.0f; // round to 0.1's
  w=w+nn;
}
w+= ")";
print(w); // ex: (5, 4.1, 8.6, -3.2)
```

This would show numbers like 0.003 as just 0. That's usually fine. If your list is full of very small numbers, you wouldn't print it this way.

27.4.3 List Searching

A basic list search is the same as a string search, like counting how many `'e'`s. But, with numbers, we can search for different sorts of things.

This warm-up loop counts how many 7's there are in an `int` list. Nothing special about it:

```
int count=0;
for(int i=0; i<N.Count; i++) {
  if(N[i]==7) count++;
}
```

We can count how many positive numbers there are. This is still pretty much the same, but it *feels* like it should have been harder to do:

```
int count=0;
for(int i=0; i<N.Count; i++) {
  if(N[i]>=1) count++;
}
```

We can do all the math we need inside the loop. This one counts how many are 11, 22, 33 ... 99. If you hate the math, the important part is `N[i]` is the current int, and we can compute lots of stuff for ints:

```
int count=0;
for(int i=0; i<N.Count; i++) {
  int nn = N[i]; // so I don't have to keep typing N[i]
  bool inRange = nn>=11 && nn<=99;
  int d1=n%10; // 1st digit
  int d2=(n/10)%10; // 2nd digit
  if(inRange && d1==d2) count++;
}
```

This counts how many strings in a list are 6 letters or longer:

```
int longWords=0;
for(int i=0; i<W.Count; i++) {
  if(W[i].Length>=6) longWords++;
}
```

Reading `W[i].Length` is the same trick as using structs. `W[i]` is one string, so dot-Length says how many letters. `W[i].Count` is an error, since `W[i]` isn't a list.

Finding the largest number is a clever rewrite of the Price-is-Right trick. Without a list, I did something like `biggest=a; if(b>biggest) biggest=b; if(c>biggest) biggest=c;` and so on with an `if` for all my numbers.

The beauty of a list is we can do the same thing, but using a loop to run every `if`:

```
int biggest = N[0]; // 1st is biggest, so far
for(int i=1; i<N.Count; i++) { // check the rest
  if(N[i]>biggest) biggest = N[i];
}
```

In my mind, the loop says "now give every box past 0 a chance to win." If we "unroll" the loop and write all the lines, it's just `if(N[1]>biggest) biggest=N[1];`, then again for every number after that. Lists are pretty cool.

Here's the same idea, but finding the shortest string in a list. It compares the lengths of the strings. I added test prints for fun:

```
string shortest= W[0];
// print("start: "+shortest); // testing
for(int i=1; i<W.Count; i++) {
  string ww = W[i]; // save to avoid typing W[i] twice
  if(ww.Length<shortest.Length) {
    shortest=ww;
    //print("new shortest: "+shortest); // testing
  }
}
```

If we ran this on `["aardvark", "cow", "horsey", "ox", "bear"]` we'd see it print aardvark, then cow, then ox, and the answer is ox.

27.5 Random item

My random town example used a big `if-else` to pick town names. A list can make picking a random word easier. Put all the words in a list, then pick a random index and use the word there.

This picks one word at random from any list:

```
List<string> Ani = new List<string>();
Ani.Add("frog"); Ani.Add("toad"); // give it some sample animals
Ani.Add("crawdad"); Ani.Add("polywog");

int ai = Random.Range(0, Ani.Count); // 0 to length-1, Perfect!
string aName = Ani[ai];
print("random word is " + aName); // ex: "crawdad"
```

With four words, the indexes are 0 to 3, which is exactly what `Random.Range(0, Ani.Count)` can roll. The weird 1-less rule for a random int helps us here.

With randomness it's hard to notice code that's off-by-one. For example, `Random.Range(0, Ani.Count+1)` could roll past the end and crash. But you might test it 400 times, never have that happen at random, and think it works. It's also hard to notice if there's 1 animal you can never get.

A hacky trick to pick one word more often is to repeat it in the list. If we add an extra `Ani.Add("toad");` then we have 2 of them. We'll roll a toad twice as often.

27.5.1 A list as a sequence

In the random word example, the list is just a bunch of stuff, and the order doesn't matter. We also like to use lists to make a sequence: use item 0, then a little later use item 1, and so on.

For example, suppose I want the space bar to print words one at a time, from a list.

The trick is to think of it like a slow loop. We'll have an index starting at 0, add 1 each time, and when it hits `Count` we've gone off the edge. Here's how it looks:

```
public List<string> W;
// sample Inspector values: ["eat", "at", "Joes", "food", "gas"];
public int phraseIndex = 0; // index variable for W

//public GameObject theLabel; // optional for on-screen Text

void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    string oneWord = W[phraseIndex];
    print(oneWord);
    phraseIndex++;
    if(phraseIndex>=W.Count) phraseIndex=0; // may as well wrap-around

    //theLabel.GetComponent<UnityEngine.UI.Text>().text=oneWord;
  }
}
```

In a loop, naming the index just `i` is fine. For a global and more spread-out code, I thought `phraseIndex` was a better name.

Here's a completely different-seeming program using the same idea. I want a Cube to pop to preset spots, once a second. The preset locations will be stored in a slow-walked list.

For examples, [-7, 7, -5, 5, -3, 3] would make it dance around the center, and [0, 2, 1, 3, 2, 4, 3, 5, 4, 6] makes it stagger-walk right. The code (again, the slow-walk idea is the same as the last example):

```
public List<float> PosX; // set size and enter x-values in Inspector
public int pi = 0; // index of next position
int delayCounter=0;

void Update() {
  delayCounter--; // the "wait 50 ticks" delay trick
  if(delayCount<0) { // make 1 step:
    delayCount=50;

    float xx = PosX[pi]; // get next position
    pi++;
    if(pi>=PosX.Count) { // check for wrap-around
```

```
      pi=0;
      delayCount+=50; // little extra delay on wrap looks nice
    }

    transform.position = new Vector(xx, 0, 0); // go to next position
  }
}
```

This is just the movement code and the slow-walk-list code crammed together. I used my old countdown delay, just in case you hate the `Time.time` trick I'd switched to using.

The trick can work for anything. Here's a version using a list to control the delay. To keep it simple, this Cube pops left-to-right by 0.5 each tick. The code:

```
public List<int> DelayAmt;
// ex: [10, 10, 100, 100] goes fast, fast, slow, slow
public int di; // index into DelayAmt
int delay; // ticks remaining until next pop

Vector3 pos;

void Start() {
  // start us on left side, with the first delay:
  pos.x=-7; pos.y=0; pos.z=0;
  transform.position = pos;
  // set-up wait for first delay:
  delay=DelayAmt[0];
  di=1; // this will be the next delay
}

void Update() {
  delay--;
  if(delay<0) {
    delay=DelayAmt[di]; // read next delay from the sequence
    di++; if(di>=DelayAmt.Count) di=0;

    // move a little right, with wrap-around. Nothing special:
    pos.x+=0.5f; if(pos.x>7) { pos.x=-7; }
    transform.position = pos;
  }
}
```

It's kind of fun to try different numbers. Changing to `[10, 10, 10, 100]` would have it scramble 3 quick steps, then a pause. `[80, 60, 40, 20, 10, 5]` would have it appear to gather momentum.

This is another one of those programs that took a lot more work that it looks. It took me a few tries to make it start waiting using the first delay, and more work to have it not double-use the first delay.

27.5.2 Variable variables

Sometimes our program needs a pile of nearly identical ints. Using one list to make them all is easier than declaring them individually, and can make the program much simpler through the magic of indexing (seriously, this is a great trick.)

In this example, I want to roll a 6-sided die 50 times and count how many of each number. We need six int's to hold the results, which we can get with a size-6 list. In our minds. `Count[0]` is how many 1's we rolled, up to `Count[5]` is how many 6's. The code:

```
List<int> Count = new List<int>();
for(int i=0;i<6;i++) Count.Add(0); // now we have 6 zeroed-out numbers

// roll 50 dice and count:
for(int i=0; i<50; i++) {
  int roll = Random.Range(1, 7);
  Count[roll-1]++; // <- the very sneaky line
}
```

The magic part is `Count[roll-1]++;`. In our minds, `Count` holds six regular variables. `Count[roll-1]` is using `roll` to look up which one. If `roll` is 1, that gives us `Count[0]`, which holds how many 1's we've gotten. If we roll a 6, the whole line says `Count[5]++;`, which is adding one to the 6's box.

It's a really clever trick to make a "variable variable." Without a list, if we had 6 individual variables, we'd need a 6-part `if`: `if(roll==1) count1++; else if(roll==2) count2++;` and so on.

Here's the same trick for saving high scores. Pretend we have 10 levels in a game, with the high scores stored in a list. `HighScore[0]` is the high score for the first level, and so on.

The player can jump around between levels somehow. Code like this would look up the high score for the level they're leaving, and update it if needed:

```
List<int> HighScores = new List<int>();
for(int i=0;i<10;i++) HighScores.Add(0); // game has 10 levels
int levelNow; // moves from 0-9 as we change levels

  // check current score as a possible new high score:
  int oldHS = HighScores[levelNow]; // <- like a 10-way if
  if(score>oldHS) {
    HighScores[levelNow]=score; // <- new high score
```

```
    print("new high score"); // testing
  }
```

It's another case where `HighScores[levelNow]` saves us a ten-part `if`.

27.5.3 Moving parts of a list

There are some general tricks involving moving around the boxes in a list. These pop up in a few places, and are good index practice.

A useful one is a shuffle. The final list will have the same things in it, but mixed up. The simplest way to do this is using one loop. Go through each item, and switch it with some random position:

```
for(int i=0; i<N.Count; i++) {
  int newPos = Random.Range(0, N.Count); // random index
  int temp = N[i]; N[i]=N[newPos]; N[newPos]=temp; // swap
}
```

One funny thing – we might switch with ourself. That possibility is actually good. In a real shuffle everything has a small chance to be in it's regular place (randomness is tricky.)

One use of this is randomly rolling 1-6 with no repeats. First create a list of 1 to 6, shuffle it, then slow-walk as you need the numbers:

Partial code:

```
List<int> RandNums = new List<int>();
for(int i=0;i<6;i++) RandNums.Add(i+1); // 1-6
int rnIndex=0; // used to slow-walk through RandNums
// shuffle goes here

int getNextRandNum() {
  if(rnIndex>=6) return -1; // too many
  int answer = RandNums[rnIndex];
  rnIndex++;
  return answer;
}
```

It's reverse thinking. It feels like we should do the random part as we request each new number. But we can pre-load the randomness. In other words, it's like shuffling a deck of cards, then getting random cards by dealing from the top.

Some fun exercises, which you don't use much for real, are rotating a list – moving every variable one space left or right, and wrapping around. Here's a picture of both types of shifts:

```
0  1  2  3  4  5
10 20 30 40 50 60  // starting list

shift left:
20 30 40 50 60 10

shift right:
60 10 20 30 40 50
```

The main problems are the wrap-around and not erasing anything. To rotate left we save `N[0]`, then shift everything from `N[1]` onward to the next lower box, then copy the saved `N[0]` to the right side:

```
int first = N[0];
for(int i=1; i<N.Count; i++)
  N[i-1]=N[i]; // push left
N[N.Count-1] = first; // copy saved first to end
```

As a check, with `i` starting at 1, the first slide is `N[0]=N[1];`, which is correct.

For practice, we could also write that using a pull-left: `N[i]=N[i+1];`. We'd start at 0 and go to 1 less than the end:

```
// different way to rotate left:
int first = N[0];
for(int i=0; i<N.Count-1; i++)
  N[i]=N[i+1]; // pull next higher value into me
N[N.Count-1] = first;
```

Same check: since `i` now starts at 0, the first is `N[0]=N[0+1];` – still correct.

Rotating a list to the right is also good for index practice. We save the last item, then go right to left, sliding items forward one space (for fun, try going left to right instead, it just copies `N[0]` into every box):

```
int last = N[ N.Count-1 ];
for(int i=N.Count-1; i>=1; i--) // right to left, stopping early
  N[i]=N[i-1];
N[0] = last;
```

Another quick check: last slide is at `i=1`, so is `N[1]=N[1-1]`, which is still correct.

27.6 Two list tricks

There are some fun things we can do using two lists.

With lists, copying `List<int> B=A;` doesn't work at all. It just makes `B` point to `A`, with both sharing the same real list. Instead you need to copy the list, a box at a time:

```
// make B a copy of A:
List<int>] B=new List<int>();
for(int i=0;i<A.Count;i++) B.Add(A[i]);
```

It's common to accidentally copy things backwards, so I like to check: the last thing this does is add the last item of `A` to the back end of `B`, so that seems correct.

Combining two lists end to end uses the same idea. Start with a fresh one, copying `A` then `B` into it:

```
List<int> A, B; // pretend these are both created and filled in

List<int> C = newList<int>();
for(int i=0; i<A.Count;i++) C.Add(A[i]);
for(int i=0; i<B.Count; i++) C.Add(B[i]);
```

27.7 Size 0 and 1 lists

Usually a size-0 list is temporary until we add our items. But sometimes the final list is size 0. That's fine. If `D` is a list of the dogs you hate, and you like all dogs, it's size-0. A loop over a size-0 list is won't cause an error – it quits right away.

In a size 0 list, `D[0]` is an error. You can't look up any boxes, because there aren't any. That's not a problem since you have to read `D.Count` and you'll see it's zero.

Having a final list be size 1 also seems like a waste. But it's often what we want. You only hate 1 dog, but you could have hated a lot more.

Even though `D[0]` is the only box, you still have to write it out. `D="Spike";` is an error. The computer won't figure out that `D[0]` is the only place it could go.

27.8 Functions with lists

List input and outputs to functions don't have any new rules, but they're nice to see anyway.

27.8.1 Lists as inputs

A list can be an input to a function. You just put the type, like `List<int>` or `List<string>`.

This checks whether a string list contains a certain word:

```
bool hasWord(List<string> W, string findMe) {
  for(int i=0; i<W.Count; i++)
    if(W[i] == findMe) return true;
  return false;
}
```

You pass a list the usual way, with just the name:

```
List<string> Cows; // ex: ["bessy", "moo-moo", "Lou"];

if( hasWord(Cows, "Lou") ) ...
```

Double equals, `==`, won't properly compare lists, since they're pointers. As usual, `if(A==B)` checks whether both point to the same list, which is not useful. To check whether two different lists are the same, we need to hand-walk through both, comparing pairs at each index:

```
bool equals(List<int> A, List<int> B) {
  if(A.Count != B.Count) return false; // not same size is not-equal
  for(int i=0; i<A.Count; i++) {
    if(A[i]!=B[i]) return false;
  }
  return true;
}
```

Same as before, you run it with just the names: `if(equals(N1,N2))`.

A fun one is checking whether a list has all numbers in increasing order, like [3, 8, 25, 26, 30]. It works like the double-letters example for strings. Every number compares to the one before it, and needs to be larger:

```
bool isIncreasing(List<int> N) {
  for(int i=1; i<N.Count; i++) // start at 2nd item
    if(N[i-1]>=N[i]) return false; // greater than the one before me?
  return true;
}
```

The early return true/false logic can be tricky. If just one pair isn't going up, the overall answer is false. We may as well quit and say that. But we can't say the entire list is increasing until we're checked them all.

Since lists are pointers, changing a list in a function is changing the real list. In other words, you can write functions whose purpose is to change a list. This changes negative numbers in a list to zero:

```
void fixNegatives(List<int> N) {
  for(int i=0; i<N.Count; i++)
    if(N[i]<0) N[i]=0;
}
```

We'd call it like `fixNegatives(N1);`.

27.8.2 Returning lists

We can return lists from functions. Usually the function creates the list and returns a pointer to it.

This function makes a list full of 0's of whatever size you want. Notice how the return type is `List<int>`:

```
List<int> zeroList(int sz) {
  List<int> A = new List<int>();
  for(int i=0; i<sz; i++) A.Add(0);
  return A;
}
```

List inputs and outputs together are common. If you remember from classes, `A=B;` won't make a copy – it's just moving pointers. But a function can make the copy:

```
List<string> getListCopy(List<string> A) {
  List<string> B=new List<string>();
  for(int i=0;i<A.Count;i++) B.Add(A);
  return B;
}
```

We can put the combine end-to-end code in a function. I think it's clear enough this takes two lists and returns another:

```
List<string> combineEtoE(List<string> A, List<string> B) {
  List<string> C = new List<string>(); // same code as above:
  for(int i=0;i<A.Count;i++) C.Add(A[i]);
  for(int i=0;i<B.Count;i++) C.Add(B[i]);
  return C;
}
```

Another exercise which is fun, but not all that common, is pair-wise adding two lists (pair-wise means using matching positions, like `A[0]+B[0]`.) It looks like this:

```
  0 1 2 3 4 5
A 3 2 0 4 1 1
B 1 1 1 2 3
  ------------
  4 3 1 6 4
```

If they aren't the same length the options are to stop when the shorter runs out, or to pretend the rest are 0's. I did it the first way, since it's simpler. Here's a function returning a pair-wise list add:

```
List<int> pairwiseAdd(List<int> A, List<int> B) {
  // length of shortest:
  int len=A.Count;
  if(B.Count<len) len=B.Count;

  List<int> C = new List<int>();
  for(int i=0;i<len;i++) C.Add(A[i]+B[i]); // <- math in the Add
  return C;
}
```

Returning a list based on the input list is always fun. This returns only the even numbers. It's pretty much the same as removing all `z`'s from a string:

```
List<int> evensOnly(List<int> N) {
  List<int> Result=new List<int>();
  for(int i=0;i<N.Count;i++)
    if(N[i]%2==0) Result.Add(N[i]);
  return Result;
}
```

Notice this could return a length-0 list, if everything in the input is odd, and that's fine.

Since lists are pointers, there's one rarer type of list-returning function where it doesn't make anything – it just returns a pointer to an existing list. This returns whichever list has more 7's:

```
List<int> getMore7List(List<int> A, List<int> B) {
  int s0=sevenCount(A); // pretend this function exists
  int s1=sevenCount(B);
  if(s0>s1) return A;
  return B;
}
```

Again, notice how it's not creating anything. This is true pointer-use. In `C=getMore7List(N1, N2);`, `C` would be pointing to `N1` or `N2` and `C.Add(11);` would grow that other list for real.

27.9 errors

Lists give you the usual pointer errors. Using one that hasn't been `new`'d will give run-time error *null reference exception*:

```
List<int> A; // as a global, this is set to null

A[0]=4; // null reference exception
print( A.Count ); // null reference exception
```

We'll get all the usual list-index-out-of-bounds errors if we go past the end. And the last index is still `N.Count-1`.

A common off-end is assuming two lists are the same length, forgetting to compare sizes. This throws a null-ref error if `B` is shorter than `A`:

```
for(int i=0; i<A.Count; i++) A[i]=B[i];
// out-of-range error if B is shorter than A
```

Another semi-common one is not checking for size 0 (or sometimes even 1.) This function returns the smallest item in the list. It crashes if the input is size 0:

```
int indexOfSmallest(int[] N) {
  int ans=N[0]; // <-- oops. forgot to check this exists
  for(int i=1;i<N.Count;i++) if(N[i]<ans ans=N[i];
  return i;
}
```

We should probably add `if(N.Count==0) return -1;`.

Another reminder: since these are pointers we get the same pointer non-error errors as with classes: `A=B;` and `if(A==b)` work in the funny pointer way.

`List` lives in the namespace `System.Collections.Generic`. Pre-made Unity files have that line on the top. If you write your own from scratch, `List` by itself will give "not found" errors.

You can write out `System.Collections.Generic.List<float> F;`. Or, easier, copy that `using` line to the top of your file.

27.10 Assign shortcut

For testing, it's a pain to make sample lists with lots of `Add`'s. There's a way to make them all at once, which is very ugly, but still shorter:

```
List<int> A = new List<int>(new int[]{5,8,2,12});
List<string> Animals = new List<string>()(new string[]{"cow","hen","pig"});
```

It makes a normal list, so we could still use `Animals.Add("worm");` later, if we somehow needed to.

If you were wondering, there are some languages where you can just write `N=(3,5,8);` and have a list. But those languages have their own places you have to work extra hard to do something that seems like it should be easy.

27.11 Lists variables as pointers

This section is just a copy of the "Dogs as pointers" examples, except using Lists. There's nothing actually new, except we've never seen pointer tricks plus Lists.

We can make a pure-pointer List. In this example `L1` and `L2` count as actual lists, while `Lp` acts as a pointer:

```
public List<string> L1 = new List<string>(), L2=new List<string();
// pretend these are set up
void Start() {
  List<string> Lp;
  Lp=L1; Lp.Add("cherry");
  Lp=L2; Lp.Add("emerald");
}
```

We only have two Lists, and `Lp` is being used to add one thing to the end of each.

We can use this trick to select one List. Suppose we have the slow-walk-word-printer. This uses `curWords` to select between `W1` and `W2`:

```
public List<string> W1; // ["eat", "at", "Joes", "get", "gas"]
public List<string> W2; // ['loose", "lips", "sink", "ships"]
List<string> CurWords = null; // a pointer to W1 or W2

void Start() {
  // randomly aim at either sentence:
  if(Random.Range(0,1+1)==0) curWords=W1;
  else curWords=W2;
}

void Update() {
  ...
    print(CurWords[wi]);
    wi++;
    if(wi>=CurWords.Count) {
      // switch to the other sentence:
      if(CurWords==W1) CurWords=W2;
      else CurWords=W2;
    }
    ...
}
```

The logic is the same as every other true pointer: we can treat `CurWords` like an actual list, but we're always aiming it at a real list owned by someone else.