Chapter 27

Using indexed lists

This chapter is about making and using a list of items. The chapter on using indexes with strings was a warm-up for this. Looking through a list is about the same as looking through the letters in a string.

The new thing is that each box in a list counts as a real variable. We can assign to them as well as read from them. A list is often a fast way to create a pile of similar variables all at once, giving us the ability to use them in a loop.

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. The boxes in each list can be any type, but they all have to be the same.

Let's pretend `N` is [20 60 90 1]. This loop prints every box:

```
for(int i=0; i<N.Count; i++)
  print(N[i]); // 20 60 90 1
```

It's the same as a loop to print every letter in a string, except lists use `Count` for the length (instead of Length, the way strings do. Why change the name? Beats me).

We can assign to boxes, which is new. In a list, `N[0]` is a working `int` the same way as `d.age`. Boxes in lists are *L-Value*'s. These are legal:

```
// N [20 60 90 1]
N[0] = 333; // [333 60 90 1]
N[3] = -22 // [333 60 90 -22]
```

This loop changes everything in a list to 6:

```
for(int i=0; i<N.Count; i++) // basic "every box" loop
  N[i]=6;
```

We can't add to a List as easily as to a string. We can add one thing at a time, only to the back end, using the `Add` command:

```
// pretend N is [1 2 3]
N.Add(12); // now N is: [1 2 3 12]
N.Add(3); //  now N is: [1 2 3 12 3]

int cc = N.Count; // 5, N knows it has 5 things in it
```

The same as for strings, going off the edge of a list is a run-time *null reference exception* error. For examples `N[-3]` or `N[999]`. If `N` has length 8 then `N[8]` is one past the end, and also an error.

27.2 Creating Lists

Now for the details we skipped and explanations:

As mentioned, `List`'s need to have a type. `List A;` is nothing, and an error. `List<string>` is a list of strings. This should make complete sense. When we look at `N[i]` we need to know exactly what type it will be. We can't have a list where any box could be any type.

More fun list types we can declare:

```
List<string> AnimalNames; // ex: ["cow" "pig" "ant"]
List<float> F1, F2; // 2 lists of floats

List<Vector3> P; // list of points. N[0] is a point
List<Cow> Barn; // list of the Cow class we made
List<GameObject> Balls; // list of links to balls in the game
```

`<` and `>` as parenthesis are called *angle brackets*. We borrowed them from HTML (and XML and JSON). The computer uses them for types.
`GetComponent<Renderer>()` uses them that way – Renderer is the type of thing we're looking for.

`List`'s are a reference type (oh, no!) But it's OK. That only means that we're required to `new` them. There won't be any other monkey business. New lists start with nothing in them. Ex's:

```
List<int> L = new List<int>(); // all on one line is fine
AnimalNames=new List<string>(); // (declared above)
List<float> F1=new List<float(), // same as: int a=0, b=0;
```

```
            F2=new List<float>();
List<Vector3> P=new List<Vector3>(); // list of points
```

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 types didn't have those extra `<>`'s.

We often like how lists start empty. We add what we want, stopping when we have enough. The final list is as big as it is. Here we want a list of random numbers that add to 20 or more:

```
List<int> N=new List<int>(); // size 0
int sum=0;
while(sum<20) {
  int n=Random.Range(1,10+1); // 1 to 10
  sum+=n;
  N.Add(n);
}
```

The final `N` might be [3,8,1,7,4]. The 4 put the total over 20, quitting the loop. We didn't know the final list would have 5 slots.

Other times we want the list to start at an exact size and stay there. For example, we have 6 seats and are arranging people. The list should start out with 6 `""`'s. We can make that with a quick loop:

```
// Make list with 6 empty strings
List<string> SeatingChart=new List<string>();
for(int i=0; i<6; i++)
  SeatingChart.Add("");
```

You may have noticed the funny way `Add` looks – it has a dot like a field. It's a *member* function, which we'll see later. For now, it's enough to know `N.Add(3)` grows `N`, while `Q.Add(3)` grows `Q`.

It also takes a different type depending on the list. It's sort of like an overloaded function – int-list Adds takes ints, and so on. It doesn't say it adds to the back, since that's the obvious best place. The front would cause us to change the index of everyone else's box.

`Add` also grows the list by a box and puts the new value there, which is like 2 things at once. Usually we like that – `Flowers.Add("Petunia");`. If all we want to do is add a box, we can use a dummy value like `N.Add(0)`.

An altogether list 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], count is 6
```

27.3 More playing with indexes

As usual, we sometimes need variables and math for the indexes. Assume `W` has several boxes:

```
int i=0;
W[i]="duck"; // box 0
i++;
W[i]="goose"; // box 1
W[i*2]="turkey"; // box 2
```

Setting values of int-lists can look odd, with numbers in and out of the `[]`'s. For examples:

```
// N [0 10 20 30 40]
int i=2;
N[i] // 20
N[i+1] //  30 (the next box)
N[i]+1 // 21 (our box plus 1)
N[i+1]-1; // 29 (next box, minus 1)
```

It can be equally funny-looking assigning int's using some math. Real programs can have lines like these:

```
// N [0 10 20 30 40]
int i=2;
N[i]=i; // [0 10 2 30 40]
N[i+1]=i; // [0 10 20 2 40]
N[i]=i+1; // [0 10 3 30 40]
```

We can use the regular shorcuts. `N[0]+=4;` adds 4 to box 0, and `N[0]++;` adds 1. For string lists, `W[i]+="y";` adds a y to the end of that box.

A new thing we can do is a "slide". `N[k]=N[k+1];` feels like `k` is pulling the next box into it. `N[k-1]=N[k];` is like `k` pushing itself into the next lower box. Both are slide-lefts.

`N[k+1]=N[k];` feels like `k` is pushing itself to the right. `N[k]=N[k-1];` is like `k` pulling from the next lower box. They each slide right. These can all be used in loops to slide the entire list.

Swapping two boxes is fun. Recall a normal swap is `int tmp=a; a=b; b=tmp;`. An array swap is the same:

```
// swap 0 and 1:
temp=N[0]; N[0]=N[1]; N[1]=temp;

// switch positions i1 and i2:
temp=N[i1]; N[i1]=N[i2]; N[i1]=temp;
```

27.4 List loop examples

27.4.1 list-Initializing loops

If we want a pre-made list with certain values, there are a few tricks.

This is a repeat of when we want 10 slots, and will fill them later. We'll `Add` ten empty-strings:

```
List<string> W = new List<string>(); // size 0
for(int i=0; i<10; i++) W.Add(""); // size 10, all ""'s
```

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

```
List<int> NN = new List<int>();
for(int i=0; i<10; i++) NN.Add(i); // [0 1 2 3 4 5 6 7 8 9]
```

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);`

If our list starts with a more interesting sequence, we can have a loop hit the exact numbers, `Add`'ing. This makes a list with 20, 18, 16 ... 2. It will be as long as it needs to be for them to fit:

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

// loops hits exactly 20 18 16 ... 2:
for(int i=20; i>=2; i-=2) Nums.Add(i);
```

Sometimes it's easier to create the list first, then use loops to set the values. Here evens are "goat" and odds are "cow". One loop handles each:

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

Sometimes, for testing, it's nice to fill a list with random numbers. This loop puts 1 to 100 in each slot:

```
List<int> R=new List<int>(); // list of random #'s
for(int i=0; i<20; i++) R.Add(Random.Range(1,101));
```

Character lists aren't very common. Strings are better, usually. But char lists allow us to easily change each character. 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('_'); // twelve underscores
```

Keeping to the hangman theme, we could use a list of 26 bools to remember which letters were guessed. There's nothing special – each slot is `true` or `false`, but it looks odd:

```
List<bool> LetterWasUsed=new List<bool>();
for(int i=0; i<26; i++) LetterWasUsed.Add(false);
```

Often we simply hand-`Add` each item:

```
List<string> ColorWords=new List<string>();
ColorWords.Add("red");
ColorWords.Add("green");
ColorWords.Add("violet"); // [red, green, violet], so far
```

In Unity we can cheat. `public List<string> ColorsWords;` appears in the Inspector. We can pop it open, type a size (it starts at 0), and hand-enter the values. When our program starts, the entire list is magically created.

An older use that you don't see as much is a reading loop. Pretend `readLine()` reads one line from a file, with `""` for when it's past the end. We can write a "go until done" while loop with an `Add` as the body:

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

Most systems don't read this anymore. But it's a fun loop, regardless. `readLine` is driving it, and you can check it works for a 0-line file, and Adds the last line, but not the `""` past-the-end line.

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 `floats` a little bit when printing a list. A list full of 8.333333333's is going to overflow the screen. Unity sometimes rounds them to tenths, for printing. We can do that (I'm using a different trick for the commas):

```
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++;
}
```

`W[i].Length` is 2 parts. `W[i]` is a string. Maybe it's `narwhale"`. So `W[i].Length` is the number of letters in narwhale. To compare, `W[i].Count` would be an error. Strings don't have a Count.

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. One `if` for each number.

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. This is the real reason random int works that way – back in the day everyone knew "random 4" meant 0,1,2, or 3.

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;
  }
}
```

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, closing in. But using [0, 2, 1, 3, 2, 4, 3, 5, 4, 6] makes it stagger-walk to the right. The code:

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

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. `newPos` could be equal to `i`. That won't break anything, and is actually good. A real shuffle has a small chance of keeping each thing in the same place, and so does ours.

One use of a shuffle 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]; // pull next lower into me
N[0] = last;
```

Another quick check: the last slide is at `i=1`, so is `N[1]=N[1-1]`, which is 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. Copy from A into B 1 at a time:
List<int> B=new List<int>();
for(int i=0;i<A.Count;i++) B.Add(A[i]);
```

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]);
```

The `Add` in the second loop always fools me. It's running 0,1,2 ... through `B`. But those numbers are only for `B`. The `Add` puts then on the end of `C`, at whatever box that happens to be.

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 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 usually not a problem since a loop won't try to check it.

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

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. Here's a basic list clone. It's merely the list-copy code from above, in a function:

```
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;
}
```

`List<string> W2 = getListCopy(W1);` runs it.

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 <- indexes
A 3 2 0 4 1 1
B 1 1 1 2 3
  ------------
  4 3 1 6 4 <- pairwise sums
```

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 computed form 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.

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;
}
```

The first line should be: `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 List variables as pointers

We're generally happy pretending that lists are normal variables. We might take advantage of their pointerness to change the contents inside of functions, but that's about it. Even so, we can do real pointer things with lists.

Here `L1` and `L2` count are considered regular lists, while `Lp` acts as a pointer:

```
public List<string> L1=new List<string>(), L2=new List<string();

void Start() {
  List<string> Lp; // pointer
  Lp=L1; // alternate way to get to L1
  Lp.Add("cherry"); // adding to L1
  Lp=L2; // alternate way to get to L2
  Lp.Add("emerald"); // adding to L2
}
```

We can use this trick to choose between 2 lists in the slow-word display:

```
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 W1 or W2:
  if(Random.Range(0,1+1)==0) CurWords=W1;
  else CurWords=W2;
}
```

The rest of the code can use `CurWords` like it was a normal list, even though it's always `W1` or `W2`.

We can write a function returning a pointer to 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;
}
```

The return value isn't a new list – it's a pointer to one of the lists we gave it:

```
List<int> C=getMore7List(N1, N2);
C.Add(99); // adding to end of either N1 or N2
```