Chapter 35

Arrays

Arrays are crude versions of `List`'s. They work the same, with one huge drawback: they have a permanent size which can't change. That's pretty bad. You want Lists for almost everything.

We learn about arrays because they're the real way computers store sequences. They're built into many languages. Lists are actually made using arrays. Many built-in functions use arrays because they're simple, often good enough, and you can turn them into Lists if you need.

35.1 Basic array rules

Array index loops use `N[i]` and `N.Length`. This sets everything in an array to 0:

```
for(int i=0; i<Nums.Length; i++)
  Nums[i]=0;
```

Besides `Length` instead of `Count`, this is identical to how a List does it.

Like `List`'s, arrays are a 2-part type. `[]` means some sort of array, with the type in front. `int[]` means "array of ints". Others:

```
int[] A; // array of ints
float[] B, C; // 2 arrays of floats
bool[] LightIsOn; // array of bools
Cow[] Barn1; // array of Cows
```

Don't confuse the square brackets with indexing. This is a special use for declaring. The inside is always empty. `int[]` is the only way to do it.

`C#` arrays are reference types. They start as `null` and need to be created with `new`. You're required to give them the permanent size, in square brackets:

```
int[] A=new int[5]; // A is a length 5 int-array

float[] B=new float[5], C=new float[20];
// B is a length 5 float array, C is length 20

string[] W;
W = new string[12]; // length 12 string array
```

This is another completely new rule and a different use for `[5]`. Again, don't confuse it with an index.

All of the rules together, this makes a size 10 array full of the word cat:

```
string[] Animals=new string[10]; // size 10 empty array
for(int i=0; i<Animals.Length; i++)
  Animals[i]="cat";
```

The rule where arrays can never change size goes together with the rule where you have to give a starting size in the `new`. Without being able to add items, the only way an array can ever get a size is during creation.

It has to do with how memory works. Memory is like a line of boxes and things tend to be created from one end. After you `new` a size 10 array, the next things you create will come immediately after. There's no room to grow most arrays. Even if there were, the memory allocator isn't really set up to grow an existing chunk.

35.2 Arrays and functions

Functions take arrays as inputs and outputs in the usual way. As before, `int[]` means it takes an array of int's. The input array can be any size. For example, this counts how many of a certain number are in an array:

```
int countNumsIn(int[] A, int countMe) {
  int count=0;
  for(int i=0; i<A.Length; i++) if(A[i]==countMe) count++;
  return count;
}
```

It looks the same as the List version, except for `int[]` instead of `List<int>` and `Length` instead of `Count`.

Because they're pointers, you can use a function to change the contents of an array. This makes every number be even. The array we give it will be changed:

```
void forceEvenValues(int[] N) {
  for(int i=0; i<N.Length; i++)
    if(N[i]%2!=0) N[i]++; // make it even by adding 1
}
```

Returning an array uses the same syntax: `string[]` in front means it returns an array of strings. This creates and returns an array of the size you want, filled with the word you give it:

```
string[] makeSameWordArray(int len, string wd) {
  string[] Result=new string[len]; // make array, final size
  for(int i=0; i<Result.Length; i++)
    Result[i]=wd;
  return Result;
}
```

We could use it like `string[] A=makeSameWordArray(5,"cat");`.

35.2.1 Array work-arounds

An advantage of `L.Add(3)` with Lists is not needing to know the final size in advance. Since arrays need to know, making an array often has 2 steps. First we count how long it needs to be, then we create it and fill it up.

This returns an array with all of the words longer than 3 letters:

```
string[] longWords(string[] A) {
  // Step 1. Find # of words in answer:
  int count=0;
  for(int i=0;i<A.Length;i++)
    if(A[i].Length>3) count++; // length of string

  // Step 2. Make the answer:
  string[] R=new string[count]; // using the count we computed
  int ri=0; // index into R
  for(int i=0;i<A.Length;i++) {
    if(A[i].Length>3) {
      R[ri]=A[i];
      ri++;
    }
  }
}
```

Notice how we needed an extra index for the answer in the second loop. Suppose it turns out the final list should have items 2, 5, 6 and 9. We need `ri` so we can copy those into 0, 1, 2, and 3 of the result. It was a lot easier being able to use `Result.Add(A[i]);`.

Adding 2 arrays end-to-end isn't as bad. We know the final size is the sum of the sizes, but we need math to add the second to the end:

```
string[] combineEtoE(string[] A, string[] B) {
  // we need to pre-make the result array to the final size:
```

```
  string[] R=new string[A.Length+B.Length];
  for(int i=0; i<A.Length;i++) C[i]=A[i]; // copy A
  for(int i=0; i<B.Length; i++) C[i+A.Length]=B[i]; // slide-copy B
  return C;
}
```

If we had a List, the second loop would be simply `C.Add(B[i]);`.

Arrays can change size, sort of. We can create new empty array, 1 box larger; copy everything over; then aim at the new array. This adds `"zebra"` to the end of string array `W`:

```
string[] W2=new string[W.Length+1]; // 1 larger than W
// copy everything over:
for(int i=0; i<W.Length; i++) W2[i]=W[i];
W2[W2.Length-1]="zebra"; // last, new, spot is "zebra"
W=W2; // aim at new array
```

The old array is abandoned and garbage-collected. That works, but if we add a lot it's slow and makes a ton of garbage.

35.3 Max size / current size trick

We often fake an array that can grow by making it the maximum size and marking how much we really use with an extra int:

```
string[] W = new string[20]; // 20 is maximum size
int wsz=0; // current size is 0

W[wsz]="dog"; wsz++; // adding dog
W[wsz]="horse"; wsz++; // adding horse
```

The end result: `wsz` is 2. `W` counts as size 2. And `W[0]` and `W[1]` are our animals.

We usually put them in a class:

```
class GrowArray {
  public string[] W;
  public int sz=0; // current "size" (the amount being used)

  public void reset(int maxSize) { W=new string[maxSize]; sz=0; }

  public void Add(string ww) {
    if(sz>=W.Length) { print("no room"); return; }
    W[sz]=ww; sz++;
  }
}
```

It's the same trick, but using a class covers it up and makes it easier to use.

When we search the list, we have to remember to use `i<sz` instead of `i<W.Length`.

This is actually how the List class works. It has a hidden array. But instead of quitting when the array is full, it creates a new array twice as big and copies.

35.4 creation shortcuts

Because arrays are built-in, they have an easy shortcut for making them pre-filled. `{2,8,6}` makes a size 3 array with those numbers. It can only be used when the array is declared:

```
string[] Ani = {"cow", "hen", "pig"}; // size 3 array of strings
int[] N = {1, 7, 12, -5, 7}; // size 5 array of ints
```

The system counts the items, creates an array that size, and fills it with those values. As usual, there's nothing special about starting values. You can change them later.

To use the shortcut anywhere else, you need extra in front. Add `new int[]`:

```
Ani = new string[]{"worm", "slug", "snail","bug"};
N = new int[]{3,4,7,4,9.12};
```

As with most array syntax, it's not part of a pattern or larger rule. It works that way because that's the rule we made.

You can even use it inside a function call, often for testing:

```
testMe(new string[]{"ant","bug"});
testMe(new string[]{"x","y","z"});
```

35.5 List / array conversion

Lists have a constructor taking an array as input:

```
int[] A={4,8,12,7}; // sample array
List<int> L=new List(A); // new List, same items as A
```

We often use it with the array-creating shortcut. It's long, but shorter than hand-adding everything:

```
L=new List<int>(new int[]{1,4,6,8,11});
```

Lists have a function to return an array of what they are:

```
int[] N = L.toArray();
```

It makes a copy, so we can change the contents of `N` without changing `L`.

A common use for this is a function that takes an array as input, when we have it in a list:

```
List<float> LF; // we use this list
oldArrayUsingFunction(LF.toArray());
```

35.6 OverlapSphere Unity example

Unity's `OverlapSphere` function returns an array. You call it with an imaginary ball, in the 3D world. It tells you every Cube, Ball and so on partly inside of it. This finds everything within 2 spaces of (0,5,0):

```
Vector3 center=new Vector3(0,5,0);
Collider[] Hits; // an array of colliders, currently null
Hits=Physics.OverlapSphere(center, 2); // returns an array

for(int i=0; i<Hits.Length; i++) {
  string w=Hits[i].transform.name;
  print("hit " + w);
}
```

It returns an array of Colliders (they go on gameObjects). It's an array instead of a List since all you're going to do is look through it. An array is fine for that.

Every time you run it, a new array is created and thrown away when you're done. That's not a big deal. But OverlapSphere has a version which re-uses your array. You make one as big as you think you need and hand it over. OverlapSphere fills it and returns how many there were:

```
Collider[] H=new Collider[10]; // 10 should be plenty

int hits=Physics.OverlapSphereNonAlloc(center, 2, H);

for(int i=0; i<hits; i++) { // <- loop up to hits, not H.Length
```

If you overlapped 3 things, they would be in 0-2 of `H`, the return value would be 3, and the rest of `H` would be old junk values.

35.7 2D arrays

If you remember, 2D lists are just lists of lists – no extra rules needed. Arrays have that method, sort of, plus they have a special perfect-rectangle version.

These are mostly good as examples of the crazy rules people jam into computer languages.

35.7.1 Ragged [][] arrays

An array of arrays is mostly like a list of lists, except for one funny rule. Write two pairs of []'s in the definition:

```
int[][] A; // A is an array of arrays
```

This would make A be 3 by 4:

```
A=new int[3][]; // <-backwards, but this is the rule
for(int i=0; i<A.Length; i++) // A.Length is 3
  A[i]=new int[4]; // each slot is another size-4 array
```

Like with lists, A[0] is the first row. It's a single length four array: A[0].Length is four. A[0][0] is one int in the grid.

We call these ragged arrays since each row could be a different length. We made them all length 4, but they didn't have to be.

35.7.2 Real 2D [,] arrays

C# also includes an older special syntax to make perfectly rectangular 2D arrays. A "real" 2D 3 by 4 array would be created like this:

```
int[,] M; // M is a 2D array
M = new int[3,4]; // creates it 3x4, all at once
M[0,0]=7; // use both inside one set of []'s
```

It comes with a special version of the insta-array shortcut:

```
int[,] K = {{1,2,3},{4,5,6}}; // K is 2x3

int[,] J = {{4,7,9}, // using white space to
            {1,1,1}, // make it look like a grid
            {2,6,3}}; // this make a 3x3
```

The rule for finding the lengths is fun, since it has 2 lengths. M.Length tells you the *total* number of items – 12 – which isn't very helpful.

M.GetLength(i) gets the size of each dimension, starting at 0. M.Rank tells you how many dimensions it has (1 is a normal array, 2 is 2D, and so on).

This uses them in a double-loop to change everything to 7's in any sized 2D array:

```
for(int i=0;i<M.GetLength(0); i++)
  for(int j=0; j<M.GetLength(1); j++)
    M[i,j]=7;
```