# Chapter 35

# Arrays

Array are crude versions of `List`'s. Things like `A[i]=3;` and loops work the same, but they have a huge drawback: you have to pick a permanent size and can never grow them.

That's pretty bad, and a List is what you want for almost everything. But some of the built-in functions use arrays, so it's nice to get a quick overview. And since they're such a pain they're an excuse to write some fun pointer-using functions.

A funny thing about arrays is they're built-in. There's special syntax bits made specially to make arrays work. By contrast, Lists are made using standard class rules.

## 35.1 Array rules

Like `List`'s, arrays are a 2-part type. To declare one, put empty square brackets, with the naked type in front:

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

This is a completely different use of square brackets. In a definition they mean "array of" and are always empty. `string[]` means an array of strings.

`C#` arrays are a reference type. You have to use `new` and you have to also give it the permanent size. We re-use square brackets *again* for this:

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

This is a totally special use of `new`. There aren't any regular parens after it, because we made up a new rule for arrays. These are array-new brackets where the size goes inside.

Altogether there are 3 different `[]` rules: the usual `A[i]` index; the ones as part of the type; and the ones giving the final size after a `new`.

Arrays have one built-in function: `A.Length` tells you how long they are. Why do string and arrays use Length, but Lists use Count? Beats me.

To show all these rules, this fills an array with 10 random numbers:

```
int[] Nums=new int[10];
for(int i=0;i<Nums.Length;i++)
  Nums[i]=Random.Range(1,101);
```

With a List we'd build this using `Add`. But since arrays are always created with the final size, and have no Add, creation loops always snap to the size, then place items using an index loop.

This counts how many cows are in a string array of any size. It's identical to the way a List would do it, except for Length/Count:

```
int count=0;
for(int i=0;i<Ani.Length;i++)
  if(Ani[i]=="cow") count++;
```

## 35.2 shortcuts

Arrays have two special syntaxes to insta-create them. When you declare an array, you can list the items in curly-braces with commas. The system counts them and makes the array that size:

```
string[] Ani = {"cow", "hen", "pig"};
int[] N = {1, 7, 12, -5, 7};
```

As usual, there's nothing special about starting values. `Ani` is size 3 forever, since that's how arrays work. but `Ani[0]="cat";` is fine.

The second insta-create shortcut is for everywhere else. You add `new type[]` in front of the curly braces:

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

Arrays can't change size, but you can throw away the old one and make another. That's why the two lines above are legal.

In case you were wondering, this is all completely hacktarific. Arrays are like dinosaurs from the early days of computing and haven't changed much.

## 35.3 Arrays and functions

Functions take arrays as inputs and outputs using *type*`[]`. For example, this counts how many of a certain number are in an array. "`int[] A`" means it takes an array input:

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

The input array can be any size. `A` is just a pointer going back to whatever real array was in the call. Those can't change size, but `A` can point to any of them.

Returning an array uses the same syntax. This creates an array of the size you want, filled with that word:

```
string[] makeSameWordList(int len, string wd) {
  string[] Result=new string[len]; // <-standard array creation
  for(int i=0;i<Result.Length;i++)
    Result[i]=wd;
  return Result;
}
```

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

### 35.3.1 Array work-arounds

Nor being able to grow arrays results in some strange work-arounds, but also some fun loops. You often need to count once for the size, make the array, then go again to place the items.

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

```
string[] longWords(string[] A) {
  // 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
```

```
  // make answer:
  string[] R=new string[count];
  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 the dance the second loop does. Since we can't use `Add` to grow `R`, we need separate index `ri`. It remembers the next open space and only goes up when we add an item.

Adding 2 arrays end-to-end is more fun with indexing. We make the final array, copy the first. Then slide-copy the second:

```
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];
  for(int i=0; i<B.Length; i++) C[i+A.Length]=B[i];
  return C;
}
```

Getting the math right for the second one is tricky – it's easy to be off-by-one. Again, since we can't just grow it with an `Add`.

We can fake an `Add` by making a new array 1 size larger, copying everything, and putting the new item in the last spot:

```
string[] addToEnd(string A, string addMe) {
  string[] R=new string[A.Length+1]; // 1 larger than input
  // copy old to larger new:
  for(int i=0;i<A.Length;i++) R[i]=A[i];
  R[R.Length-1]=addMe; // the new spot
  return R;
}
```

We'd use it like `Ani=addToEnd(Ani, "cow");`. It's a huge waste, since it turns the old array into garbage each time, but it works.

## 35.4 Max size / real size tricks

A common trick to fake growing an array is making a too-large one, with an extra variable for how much we're really using. This works nicely when a class

can hide it.

This makes a dog where `Eats` can hold 5 things, while `foodCount` is how many are really in it:

```
class Dog {
  public string name;
  string[] Eats; // null, for now
  int foodCount=0;

  public void setupDog() { Eats=new string[5]; foodCount=0; }
```

We can add a new food (up to 5) by adding it to the next empty slot:

```
  public void addFood(string f) {
    if(foodCount>=Eats.Length) return; // no space left
    Eats[foodCount]=f;
    foodCount++;
  }
}
```

Using `foodCount` is like the dance with `ri` from a few examples up. It's the next free index. But it's also the number of used slots in `Eats`.

When we search the foods, we use `foodCount` as the size:

```
  public bool canEatThis(string f) {
    for(int i=0;i<foodCount;i++) // <-loop up to foodCount
      if(Eats[i]==f) return true;
    return false;
  }
```

We can improve `addFood` to grow it by 5 when we run out of space. This is the same logic as growing by 1: make a new larger array, then copy the old one over:

```
  public void addFood(string f) {
    if(foodCount>=Eats.Length) {
      string[] EE=new string[Eats.Length+5]; // room for 5 more
      for(int i=0;i<Eats.Length; i++) EE[i]=Eats[i]; copy old values
      Eats=EE; // old Eats is garbage, new Eats is +5
    }
    Eats[foodCount]=f;
    foodCount++;
  }
}
```

Yikes!

## 35.5 List / array conversion

Lists are made using arrays. They actually work like `Eats` plus `foodCount` above. When `Add` runs out of space the hidden array doubles in size. You can think of a List as an array with every sliding/copying loop you'd need, prewritten.

There are functions to convert between lists and arrays.

Recall `List<int> L=new List<int>(new int[]{1,4,6,8,11});` is the way to insta-create a List. Now we can see the inside is really the instant array trick. The whole thing is just making a List from an array. This is also allowed:

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

We can convert back using `int[] A=L.toArray();`. It makes a copy, as an array.

What can happen is you'll see weirdness like `L=new List(f(L.ToArray()));`. That means you wanted to call function `f` that takes and returns an array. You have a List, but you couldn't find a List version of that function. So you convert to an array inside the call, and use `new List()` to convert the array answer back to a List.

## 35.6 OverlapSphere Unity example

Unity's `OverlapSphere` is a pretty typical function for finding things in a game area. An obvious use is finding everything in the radius of an explosion. It returns an array of everything touching a 3D ball from where you say.

This finds everything within 2 spaces of (0,5,0):

```
Vector3 center=new Vector3(0,5,0);
Collider[] HitStuff=Physics.OverlapSphere(center, 2);
```

`HitStuff` is the array. It's an array of Colliders, which are optional parts of gameObjects (anything that "takes up space" has a collider. If you've ever walked around a 5-sided barrel and noticed it counted as round, that's because it had a round collider.)

Typical code to look at what we hit:

```
for(int i=0;i<HitStuff.Length;i++) { // <- normal array loop
  print( HitStuff[i].transform.name + " got hit");
}
```

There's another version using the too-big array trick. You create a re-usable array, large enough to hold everything and give it to the function. It's filled, only up to the limit, and also returns how many slots were used:

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

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

for(int i=0; i<hits; i++) { // <-use returned count
  // do something with HitStuff[i]
}
```

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

OverlapNonAlloc is typical nit-picky stuff. Arrays are a little faster than Lists, and reusing your own array is a tiny bit faster than making a new one every time. And having a partly-filled array – well, that's a pain, but it's not any slower.

The only time to ever worry about this is if you're making an action game for a cell-phone and don't mind spending a few extra days debugging weird array problems.

## 35.7 2D arrays

If you remember, 2D lists are just lists of list – 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];
```

Like with lists, `A[0]` is the first row – a normal length four array. `A[0].Length` is four, And `A[0][0]` is one int.

We call these ragged arrays since each row could be a different length. Lists of lists are also ragged.

### 35.7.2 Real 2D [,] arrays

Various languages also have a special syntax to make perfectly rectangular 2D arrays. This is obscure, but fun to see things we tried.

A "real" 2D 3 by 4 array would look 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 this is one thing with 2 lengths. `M.Length` tells you the *total* number of items – 12 – which is usually not very helpful.

To use it for real, `M.Rank` tells you how many dimensions it has, and `M.GetLength(i)` tells you the size of each dimension. For this, `M.Rank` is 2, `M.GetLength(0)` is 3 and `M.GetLength(1)` is 4.

This uses them in a double-loop to change everything to 7's:

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

In practice, you'd probably have the width and height saved in variables and use those instead of `GetLength`.

Or, for really real, you'd make a class `Grid` and use a member function like `G.at(3,2)` to find things. Then the inside would be one of these 2D array types or just Lists.