Contents

35 Recursion **2**
- 35.1 How calling yourself works . . . 2
- 35.2 Recursive thinking . . . 4
  - 35.2.1 Tree functions . . . 8
  - 35.2.2 Connected / Flood Fill . . . 11
  - 35.2.3 Maze walk . . . 14
- 35.3 Errors . . . 17

36 First class functions **19**
- 36.1 Rules and examples . . . 20
  - 36.1.1 Declaring function pointers . . . 20
  - 36.1.2 Using function pointers . . . 22
- 36.2 Uses . . . 24
  - 36.2.1 As a regular variable . . . 24
  - 36.2.2 Arrays of pointers . . . 25
  - 36.2.3 User-defined keys . . . 26
  - 36.2.4 Passing functions into functions . . . 27
- 36.3 Function pointers as callbacks . . . 31
- 36.4 Just cramming in a function . . . 34
- 36.5 More rules and examples . . . 36
  - 36.5.1 Older syntax, `delegate`, shortcuts . . . 37
- 36.6 Built-in array function-using functions . . . 38

37 Inheritance **40**
- 37.1 Pre-inheritance example . . . 40
- 37.2 Basic Inheritance . . . 42
- 37.3 Intro to Polymorphism . . . 43
- 37.4 Dynamic casting . . . 45
- 37.5 Inheritance and functions . . . 47
- 37.6 dynamic dispatch . . . 51

38 More Inheritance **53**
- 38.1 A story about the terms . . . 53
- 38.2 Misc examples . . . 54

38.3 Cutesy extra rules . . . 56
38.3.1 Privacy . . . 56
38.3.2 abstract base class . . . 56
38.3.3 Inheritance chain, Multiple Inheritance . . . 57
38.3.4 Misc . . . 57
38.4 Interfaces . . . 58
38.5 General inheritance advice . . . 62

39 Linked Lists **65**
39.1 Simple linked list . . . 65
39.1.1 Linked List loops . . . 66
39.1.2 Insert/Remove . . . 69
39.1.3 Removing . . . 72
39.2 Misc . . . 73
39.3 Special cases . . . 76
39.4 Doubly-linked lists . . . 76
39.5 Using templates . . . 80
39.6 Built in Linked List . . . 80
39.7 Array implementation of linked lists . . . 81

40 Big-O notation **83**
40.1 Loop counting logic . . . 84
40.2 Integer array loops . . . 85
40.2.1 Fun with bad loops . . . 89
40.3 Linked Lists . . . 90
40.4 Formal math . . . 91
40.5 More tricks and exceptions . . . 92
40.6 Other data structures . . . 95
40.7 Overview . . . 95
40.8 Numbers . . . 96

Chapter 35

Recursion

This section has only one real rule: a function can call itself. Each time, you get a fresh copy and a fresh set of local variables. That's the whole rule (examples are later.)

That's called recursion. The real trick is being able to make it useful. There are some problems which naturally break into parts and subparts and subsubparts. It works out that a function calling itself is the best way to trace through some of those problems.

The first part of this section is just going over the mechanical rules for recursion – no actually useful examples yet. Then next part will be real, but fakey uses of recursion - doing things recursively that we can already do better without it.

Then the last part will be about real problems where recursion is the best way.

Most people don't "get" recursion at first. That's fine. If you just mess around with it a little then mostly forget it, you'll see a problem years from now and think "hey - that looks like recursion would work" and can relearn it.

35.1 How calling yourself works

A function is allowed to call itself. Just like calling anything else, it waits for the call to end, then continues. The function `crashMe` below is useless, never prints anything and will crash the computer, but it is legal:

```
void crashMe() { crashMe(); print("Arrg"); }
```

When this calls function `crashMe()`, the computer starts running a second fresh copy. The first `crashMe` call is waiting, and the second copy is running. The second copy calls the third copy and waits, and so on.

This makes an infinite number of copies, never getting to line 2 of any of them. A fun fact – each copy takes a little bit of space. So instead of running forever like an infinite loop, it runs out of memory and crashes the entire Unity editor (same as an infinite loop, be sure to save before testing if you changed the Scene.)

Having a chain of `crashMe`'s, all waiting for the next one to finish, isn't the problem. Never stopping is the problem. We can use tricks to make recursion end, just like we use tricks to make loops end.

This next recursive function is still pretty useless, but it will run without crashing:

```
void A(int n) {
  if(n>0) A(n-1);
  print(n);
}
```

It works because of the "make a copy of" rule. Each time we call `A`, we get a new copy of it with a new local `n`.

Saying this in a different way: the idea I gave you before about local variables was oversimplified. I made it seem like you could premake one local `n`, which `A` used whenever it was called. The real trick is we really make a fresh local `n` for each call.

Suppose someone calls `A(5)`. It calls `A(4)`, which calls `A(3)` ...down to `A(0)`, which stops because of the `if`. Here's what the call stack looks like after that:

```
 A       A       A       A       A       A
n:5  |  n:4 |  n:3  |  n:2  |  n:1  |  n:0
```

Five copies of `A` are waiting, with their own local `n`'s. The last copy with `n=0` is running. It prints 0 and pops back to `A(1)`. That local `n` is 1. We print that, pop back and print 2 and so on.

A neat thing, pretend you're the first `A(5)`. All you do is call `A(4)`, wait, then print 5. All that growing and shrinking the call stack is handled by the computer. It seems like this is different because the function you're waiting for is you, but it's not.

The whole thing prints 0 1 2 3 4 5, on different lines. A loop would be easier and faster. This is all just for demo purposes.

For fun, here's the full useless but standard counting recursion example. It prints 5, 4, 3, 2, 1, 0 as it makes the calls, and then 0, 1, 2, 3, 4, 5 on the way back:

```
void B(int n) {
  print(n);
  if(n>0) A(n-1);
  print(n);
}
```

Even if this wasn't calling itself, we can see `B(5);` prints 5 first, than whatever the function call says to, then prints 5 last.

35.2 Recursive thinking

A recursive function starts with a recursive idea. You have something you can solve by doing a little bit of work to make it the same problem, but smaller. Then you have a copy of yourself solve the smaller version, which repeats the process.

Here's an idea to compute powers of 2: to get 2^n, find the next lower one and double it. For example, 2^4 is two times 2^3. Easy, right? How to figure out 2^3? A recursive call to myself. Here's how it looks:

```
int powTwo(int p) {
  if(p<=0) return 1; // 2 to the power of 0 is 1
  else return 2*powTwo(p-1);
}
```

Let's forget about how we can already do this with a loop. Do you see how beautiful that last line is? `return 2*powTwo(p-1)`. It says "call a function to find the next lower power of two, double it, and I'm done."

If `powTwo(3)` computes 8, then `powTwo(4)` will compute 16.

Here's a picture of the largest stack frame if we call `twoPow(4)`:

```
p:4  |  p:3  |  p:2  |  p:1  |  p:0
```

The fun part is when they start returning things. Each one gets the "one lower" answer, doubles it, and returns that. As the functions pop back, the return values look like the second line:

```
   p:4 | p:3 | p:2 | p:1 | p:0
16 <-  8 <-  4 <-  2 <-  1 <-
```

I think of it as a bunch of guys in a row. Each guy asks the one to the right, then waits. Eventually, they hear the answer, double it, and tell that to the guy who asked them.

The formal term for when a recursive function stops (by not calling itself) is the **basis step**. Each recursive call should get closer to the basis step, somehow.

So far, I just subtracted 1 and the basis step was when it got to 0, but you can do other things.

For fun, we could rewrite `powTwo` to hand-check for 0-3. Then `n<=3` would be the basis step. This isn't any better – it runs 3 steps faster, but is longer:

```
int powTwo(int p) {
  if(p<=0) return 1;
  else if(p==1) return 2;
  else if(p==2) return 4;
  else if(p==3) return 8;
  else return 2*powTwo(p-1);
}
```

Finding the smallest item in an array is another one we can solve with recursive thinking, even though a loop would be easier. The plan is: compare the last item to the smallest thing in the rest of the list. How to find that? A recursive call. Stop if the list only has one thing.

Here's the code, with extra print statements:

```
int smallestItem(int[] A, int len) {
  if(len<=0) return -999; // empty list has no answer
  else if(len==1) {
    print("returning A[0]="+A[0]);
    return A[0];
  }

  int li=A[len-1];
  print("len "+len+" calling len "+(len-1)+":");
  int smallRest=smallestItem(A, len-1);
  print("len "+len+" comparing "+li+" and "+smallRest);
  if(li<smallRest) return li;
  else return smallRest;
}
```

If we call `smallestItem(A,4)` with `A=[35, 72, 25, 64]` we'd see this output, showing how it makes the calls, then comes back:

```
len 4 calling len 3:
len 3 calling len 2:
len 2 calling len 1:
returning A[0]=35
len 2 comparing 72 and 35
len 3 comparing 25 and 35
len 4 comparing 64 and 25
```

The most fun part is how the length 4 version calls the length 3 one and waits (first line of output.) The last line is when it's done waiting. We've got

the smallest out of the the rest of the array, and can finally compare it to our 64.

The extra `len` input is a common trick. We want an extra variable to fake-shorten the array, so we just add it as an input. Then we make a front-end to fill it in:

```
int smallestItem(int[] A) { return smallestItem(A, A.Length); }
```

Now the user can call `n=smallestItem(A)`, like normal, which calls the real 2-input version. Front-ends that set up the extra inputs for recursive functions are usually called **drivers**.

This next thing is something new that we don't really need recursion for, but it's not too bad a way to solve it. It's about searching for a number in a sorted list. We can solve that the same way we would for real: look at the middle number, figure out which half our number could be in, look at the middle number in that half .... Eventually we'll find it, or narrow where it could be to nothing.

Thinking of this recursively: the smaller problem is to try to find the number in a smaller list. In the previous problem the list was just 1 smaller. Now it will be half as big. I'll use the same trick where I'll use extra numbers to fake-resize the list. Here's the driver:

```
bool isInList(int[] A, int num) { return bSearch(A, 0, A.Length-1, num); }
```

The second and third inputs are the exact first/last indexes of the part we're searching. The driver just fills in 0 and Length-1, which means we start with the entire list.

`num` is what we want to find. The answer is either yes, it's in the list, or no it's not. Here's the code, with extra prints:

```
bool bSearch(int[] A, int i1, int i2, int num) {
  print("list is "+i1+"-"+i2);
  // note: i1==i2 means the list has 1 thing in it. i1>i2 means it's empty:
  if(i1>i2) {
    print("size 0 list=not found");
    return false;
  }

  int mid=(i1+i2)/2;
  if(num==A[mid]) {
    print("found it at "+mid);
    return true;
    }

  if(num<A[mid]) {
    print("searching H1= "+i1+"-"+mid-1);
```

```
    return bSearch(A, i1, mid-1, num);
  }
  else {
    print("searching H2="+mid+1+"-"+i2);
    return bSearch(A, mid+1, i2, num);
  }
}
```

Suppose `A` has size 101 (which is 0 to 100.) The plan is to check `A[50]`. Then, depending if our number is smaller or larger, we'll check either 0-49 or 51-100.

Look how beautiful the recursive calls are: `bSearch(A, i1, mid-1, num)` checks the first half (from the start, to one before the middle,) and `bSearch(A, mid+1, i2, num);` checks the second half (just past the middle, to the end.)

Here's searching for 60 on the list `[12 16 29 36 39 53 57 62 78 80]`:

```
searching H2= 5-9
searching H1= 5-6
searching H2= 6-6
searching H2= 7-6
size 0 list=not found
```

It cuts the list in half three times to get down to 6 through 6. That's where 60 could have been, but it's not. The last step is trying to cut 6 to 6 in half and realizing we're done and can't find it.

Here's the same list, searching for 36:

```
searching H1= 0-3
searching H2= 2-3
searching H2= 3-3
found it at 3
```

This time the `if` walks us through different halves. When we get down to a size-1 list, that's our number. For fun, here's what happens when we look for 62:

```
searching H2= 5-9
found it at 7
```

We cut it in half once, but then luck out when we try again – the middle is exactly our number. It's not a rule that a recursive function has to go all the way every time.

I skipped a lot of testing steps – this was a lot harder to write than I'm making it look. Anything with indexes is super-easy to be off-by-one, or get wrong index math so it spins forever, skips certain positions, or all sorts of wrongness.

But the really cool thing is this follows the recursion idea. On each call, `i1` and `i2` aren't getting smaller, but they're always getting closer together. The list is always getting smaller.

35.2.1 Tree functions

The computer science term for parents with children, with yet more children, is a tree.

The most common one you see is a folder layout. In a 3D environment, we use an even simpler tree to glue items together. If you haven't seen one, here's how it works:

Suppose we make three long, thin cubes for fingers; then another cube for a hand. We can position the fingers against the hand, but we'd like to formally make them part of it. If we drag them into the hand (the same way we'd drag a file into a folder) they become the hand's children:

```
hand
  finger1
  finger2
  finger3
```

The advantage is we can now move the hand and the fingers move with it.

Just to use the built-in tree commands, here's a non-recursive function that prints the parent and kids:

```
void printKids(Transform tt) {
  print("parent is "+tt.name);
  int childCount=tt.childCount; // this could be 0
  for(int i=0; i<childCount; i++) {
    Transform tt2 = tt.GetChild(i);
    print("child "+i+" is "+t2.name;
  }
}
```

`childCount` is just an integer counting your direct children. `GetChild(childNum)` uses 0-based indexes and returns the `Transform` of that child.

Since those two functions run on Transforms, I made the input be one: we could call this with `printKids(gg.transform);`, or use `public Transform t1;` and just call `printKids(t1);`.

The thing is, real trees are usually a lot messier than this. The same one can be shallow, and deep, and have different numbers of kids. Here's a typical messy one: a body with two different fingered hands and a long forked tail:

```
body
    hand1
        finger1
        finger2
    hand2
        finger1
        finger2
        finger3
    head
    tailA
        tailB
            tailC
                tailSpike1
                tailSpike2
```

The trick to walking through something like that is that trees have a recursive definition. A tree is one gameObject with 0 or more children, which are trees themselves.

In recursion, we're looking for a natural way to break a problem into identical smaller problems. A tree breaks into 1 parent and a bunch of smaller trees. The basis case is when you have no children. Here's a recursive function doing that:

```
void printAllNames(Transform tt) {
  print(tt.name):
  int childCount=tt.childCount;
  for(int i=0; i<childCount; i++) {
    Transform tt2 = tt.GetChild(i);
    printAllNames(tt2); // <-recursive call
  }
}
```

This has one thing we've never done before: it makes *several* recursive calls each time. That's legal, and is really the main use of recursion. This is the first thing we've seen where recursion is the hands-down best solution.

To get an idea of using multiple calls, think about the first call, to `body`. It prints `body`, then calls `printAllNames(hand1)` and waits. When that's done it makes the call for `hand2`, `head` (that's a short wait) and finally `tailA`. Each time we're saying "you're a tree, print yourself."

The same set-up can turn everything red:

```
void colorRecur(Transform tt, Color cc) {
  Renderer rr = tt.GetComponent<Renderer>();
  if(rr!=null) rr.material.color=cc;

  int childCount=tt.childCount;
```

```
  for(int i=0; i<childCount; i++) {
    Transform tt2 = tt.GetChild(i);
    colorRecur(tt2, cc);
  }
}
```

It's the same recursive set-up, but with color-change where printing was. An interesting thing is that it makes no difference what order we color things. We happen to go through the entire first hand before going to the second. Changing that order is tricky, but we rarely need to.

Just to mess around, suppose we want to color children as soon as we see them in the loop; then only jump down to them if they have kids. We can do that. We need a driver to color the original parent, so I renamed the recursive part with an underscore:

```
// driver:
void colorRecur2(Transform tt, Color cc) {
    Renderer rr = tt.GetComponent<Renderer>();
    if(rr!=null) rr.material.color=cc;
    _colorRecur2(tt, cc);
}

// recursive part:
void _colorRecur2(Transform tt, Color cc) {
  int childCount=tt.childCount; // this could be 0
  for(int i=0; i<childCount; i++) {
    Transform tt2 = tt.GetChild(i);

    Renderer rr = tt2.GetComponent<Renderer>();
    if(rr!=null) rr.material.color=cc;

    if(tt2.childCount>0) _colorRecur2(tt2, cc);
  }
}
```

This is a bunch more complicated. I'm writing it out only to show what a puzzle it can be to write recursive functions. There's often a short way that looks totally obvious; but it can take hours of trying head-hurting messy junk to come up with it.

Unity has a super-hacky shortcut for finding children. A `foreach` loop takes a list as input, hitting every item. But C# has a trick for it – if you're not a list, you can fake-up a list that `foreach` will use. Unity did that for Transforms – they're specially written to give `foreach` a list of their children.

It's a total hack, and seriously weird-looking, but `foreach(Transform tt2 in tt)` will make `tt2` hit every one of `tt`'s children.

Here's something using that shortcut to give every finger a random length:

```
void resizeFingers(Transform tt) {
  string w=tt.name;
  if(w.Length>=6 && w.Substring(0,6)=="finger") {
    Vector3 sc=tt.localScale;
    sc.z=Random.Range(0.6f, 1.4f);
    tt.localScale=sc;
  }

  // new super-short "call each child":
  foreach(Transform tt2 in tt) {
    resizeFingers(tt2);
  }
}
```

Here's one final classic tree function, which just counts everything in the entire tree. It's dastardly how something this short actually works:

```
int treeCount(Transform tt) {
  int count=1; // me;
  foreach(Transform tt2 in tt)
    count+=treeCount(tt2);
  return count;
}
```

35.2.2 Connected / Flood Fill

This section is about a 2D grid, like a dungeon map or a maze, where we fill in squares to make walls. We often want to know whether we can walk between all of them – whether we can or can't. For example, maybe we're making a random map and want to be sure it's all connected. Or the game is to make a fence around a cow and we're checking to be sure it can't walk out.

Either way, the basic trick is to pick 1 square and then find every other square we can reach from it. The generic name for that is a flood-fill.

I'll start with the boring set-up for the grid. Each square is either a wall, or open. For now, we'll just color walls blue and open areas white. Here's a simple class to handle one square. `G` will be our permanent link to the real square, which we'll make later:

```
// holds info about one square in the grid
class GridSq {
  public bool isWall;
  public bool isMarked;
  public GameObject G; // pointer to the real square
```

```
  public bool canWalk() { return !isWall && !isMarked; }

  public void setMark() { isMarked=true; colorMe(); }

  public void colorMe() {
    Color cc;
      if(isWall) cc=Color.blue;
      else if(isMarked) cc=Color.green;
      else cc=Color.white;
      G.GetComponent<Renderer>().material.color=cc;
  }
}
```

The variable `isMarked` is a common trick. It doesn't mean anything for real – it's an extra scratch variable to "mark" a space, which we'll need later.

I'll use a string-picture as a cheap way to describe the grid. This one has some interesting differently-shaped areas. The `o`'s are walls:

```
public GameObject gridCube; // drag a flat 1x1 Cube prefab here

GridSq[,] Grid;

string[] GridPic = {
 "      o   ",
 "   oooo   ",
 "          ",
 " o        ",
 "oo    oo o",
 " o    o o ",
 " ooo  o   ",
 " o o  o   "};
```

Then here's the code to make the grid 2d-array, and build it on the screen. I'm assuming `gridCube` is a flatish 1x1 cube:

```
public GameObject gridCube; // drag a flat 1x1 Cube prefab here

GridSq[,] Grid;

void makeGrid() {
  Grid=new GridSq[10,8];
  Vector3 pos; pos.y=0; // used to place the tiles
  for(int x=0; x<10; x++) {
    for(int y=0; y<8; y++) {
      GridSq gs = new GridSq();
      Grid[x,y]=gs;
```

```
      gs.isMarked=false;
      GameObject gg = Instantiate(gridCube);
      gs.G=gg;

      pos.x=-5+1.1f*x; // usual positioning
      pos.z=4-1.1f*y; // putting 0 at top to match how GridPic is written
      gg.transform.position=pos;

      gs.isWall = GridPic[y][x]=='o';
      gs.colorMe();
    }
  }
}
```

Nothing special or new there. As usual, lots of trial and error and off-end errors to get it to line up.

An interesting thing about finding every place we can go, is that it doesn't seem like a recursive problem, and it really isn't. It feels like you'd use maybe a nested loop or something. It just turns out, after hours and days of trying, a recursive trick is the best way to solve it.

The recursive plan is: mark the square you're on, then make recursive calls to the squares all around you. But just quit if you're already marked. As usual, it's crazy that it works, and how short it is for what it does.

`colorConnected` is the recursive function:

```
public int xStart, yStart;
void Start() { colorConnected(xStart, yStart); }

void colorConnected(int x, int y) {
  // just quit if off-edge or not a legal space:
  if(x<0 || x>=10 || y<0 || y>=8) return;
  GridSq gs = Grid[x,y];
  if(!gs.canWalk()) return;

  gs.setMark();

  // all four neighbors:
  colorConnected(x-1, y);
  colorConnected(x+1, y);
  colorConnected(x, y-1);
  colorConnected(x, y+1);
}
```

Noticed I used the trick where you do the work at the top of the code, for just that one spot. Down below, the code can jump inside walls or off the edge

without checking.

This should mark (and turn green) all the squares you can get to from position `(xStart,yStart)`.

35.2.3 Maze walk

The flood-fill trick can be used to find a path through a maze. The basic idea is the same – take one step in each direction and try from there. The difference is we want to stop when we reach the end.

To do that, each call in a direction returns true if it found the exit. If it does, we don't try any other directions.

Here's the code, then some notes. The `mazeWalk` part is just making sure the marks are cleared, calling the real function, then printing the results. `_walkMazeR` is the real recursive function. You should be able to see it looks about like the flood-fill one:

```
int[] xWalk, yWalk;
int walkLen=0; // this is also where the next grid spot goes

void Start() {
  makeGrid(); // from the old grid
  xWalk = new int[8*10]; // enough for every space in the maze
  yWalk = new int[8*10];
  walkMaze();
}

void walkMaze() {
  // clear everything for fresh walk:
  walkLen=0;
  for(int i=0; i<10; i++)
    for(int j=0; j<8; j++) {
      Grid[i,j].isMarked=false;
      Grid[i,j].colorMe();
    }

    // call the function:
    bool found = _walkMazeR(xStart, yStart);

    // Use the list however we like. Just print it, for now:
    if(found) {
      for(int i=0; i<walkLen; i++) {
        int xx=xWalk[i], yy=yWalk[i];
        Grid[xx,yy].setMark(); // cheap way to turn it blue
      }
```

```
    }
    else print("no path");
}

bool _walkMazeR(int x, int y) {
  if(x<0 || x>=10 || y<0 || y>=8) return false;
  GridSq gs=Grid[x,y];
  if(!gs.canWalk()) return false;

  // this space is free, test walk here and try to keep going:
  xWalk[walkLen]=x; yWalk[walkLen]=y; walkLen++;
  gs.isMarked=true;

  if(x==xEnd && y==yEnd) return true;

  if(_walkMazeR(x-1, y)) return true; // found the exit. Done.
  if(_walkMazeR(x+1, y)) return true;
  if(_walkMazeR(x, y+1)) return true;
  if(_walkMazeR(x, y-1)) return true;

  // it was all dead-ends, so erase this step from the list:
  walkLen--;
  //gs.isMarked=false; // no need, and may as well warn others this is a dead-end
  return false;
}
```

Notes:

- I'm using global int-arrays `xWalk` and `yWalk` to save the actual path. Each time we make a call, we add that (x,y) to the path. Each time we give up (all 4 directions were no good,) we take ourself off the list. It's a sneaky, but common trick.

  After the call is made, at the end of non-recursive driver `walkMaze`, a loop just uses them to color the path.

- We still need the marks, to keep us from walking in circles. But they aren't the answer anymore. For example, I leave the marks on dead-ends as an "I already tried this" if we come to it some other way. I sort of cheated using `gs.isMarked=true;` to avoid changing the color.

- As soon as we find the exit, we can quit the whole thing. But there's no good way to say "pop back to before all the recursion." Instead, we have to pop back with `true` one step at a time, like this does.

- This still works like a flood-fill, wandering around however. That works fine for a "tight" maze. But if you have open areas, like a 3x4 wall-less part, this will stupidly wind around in a lot of extra steps.

You can change this to try every possibility and remember the shortest, but that's more work.

About that last point, the easiest way to see how we're really stupidly wandering in a pattern is the watch a maze-walk run slowly.

If you've seen the way Unity times things using `IEnumerator` (I haven't covered it, at all,) here's a version which walks a maze at 5 steps/second, coloring as it goes. It's the same code, but hacked to get delays in it:

```
string[] MazePic = {
  " oo o     ",
  "  o ooo oo",
  " oo o o   ",
  " o  o   o ",
  " o oo ooo ",
  "      o oo",
  " oooo   o ",
  "    o o   "};

public int startX, startY, endX, endY;

// driver. Calls recursive function with starting coords. Not 100% needed:
void mazeSearch() {
  mazeDone=false;
  StartCoroutine(mazeSearch2(startX, startY));
}

// helper function:
bool isInMaze(int x, int y) { return x>=0 && x<10 && y>=0 && y<8; }

// global
bool mazeDone=false;

IEnumerator mazeSearch2(int x, int y) {
MazeSq ms=Maze[x,y];
ms.setMark();
yield return new WaitForSeconds(0.2f);

if(x==endX && y==endY) {
mazeDone=true; yield break;
}

Debug.Log(x+","+y);

// try to walk x-1, x+1, y-1, y+1:
int x2=x-1, y2=y;
```

```
if(isInMaze(x2,y2) && Maze[x2,y2].canWalk()) {
yield return StartCoroutine(mazeSearch2(x2,y2));
if(mazeDone) yield break;
}

x2=x+1; y2=y;
if(isInMaze(x2,y2) && Maze[x2,y2].canWalk()) {
yield return StartCoroutine(mazeSearch2(x2,y2));
if(mazeDone) yield break;
}

x2=x; y2=y-1;
if(isInMaze(x2,y2) && Maze[x2,y2].canWalk()) {
yield return StartCoroutine(mazeSearch2(x2,y2));
if(mazeDone)  yield break;
}

x2=x; y2=y+1;
if(isInMaze(x2,y2) && Maze[x2,y2].canWalk()) {
yield return StartCoroutine(mazeSearch2(x2,y2));
if(mazeDone)  yield break;
}

// no path from here. Walk back
// (guy who called us will try other directions)
ms.isMarked=false; ms.colorMe();
yield return new WaitForSeconds(0.2f);
}
```

35.3 Errors

Obviously, lots of possible errors. My favorites are infinite loops caused by calling the wrong function or using the wrong value.

This tree function accidentally make the recursive call on itself, instead of on a child, so spins forever:

```
void doTreeStuff(Transform T) {
  ...

  for(int i=0; i<T.childCount) {
    Transform t2=T.GetChild(i);
      doTreeStuff(T); // <- opps!! used myself by mistake. Meant to use t2
  }
}
```

In this one, the driver accidentally calls itself, instead of the recursive function:

```
bool findPath(int xStart, int yStart, int xEnd, int yEnd) {
  for(int x=0;x<10;x++) // clear marks, reset color, etc...
     ...

     findPath(xStart, yStart, xEnd, yEnd); // <-- OOPS!! meant to call findPathRecursive
     // this spins forever
}

bool findPathRecursive( ... ) { ... }
```

Chapter 36

First class functions

The one idea in this chapter is letting us use variables for functions.

For example, suppose `f1` was a variable and we assigned it the round-up function: `f1=Mathf.Ceil;`. Then `f1(3.7f)` would be 4. Since it's a variable, we could change it to round-down: `f1=Mathf.Floor;`. Now the same call `f1(3.7f)` would be 3.

We could even go nuts and say `f1=Mathf.Sqrt;`. I don't know what the square-root of 3.7 is, but `f1(3.7f);` would compute it.

Here's how it looks in working code:

```
System.Func<float,float> f1; // declaring f1
f1 = Mathf.Ceil;
print( f1( 3.7f ) ); // 4 - it runs ceiling function

f1 = Mathf.Floor;
print( f1(3.7f) ); // 3 - runs the floor function

f1 = Mathf.Sqrt;
print( f1(3.7f) ); // 1.92
```

From the declaration, you might have figured out that `f1` is limited to `float` input and `float` output functions. But it's not limited to built-in functions. We can point it at one we wrote, and there's even a rule to make up one on the fly (which I'm not showing you, yet):

```
float randAround(float x) { return Random.Range(f/2, f*1.5f); }

f1=randAround;
float x = f1(10); // random from 5 to 15
x = f1(3); // random from 1.5 to 4.5
```

This is going to be a really useful trick. But, as usual, even though you probably guessed them, we should go over the rules first.

36.1 Rules and examples

These things act like pointers – they don't hold a function, they just aim at them. C# has a special name for them (which I'm saving for later.) Most people call them function pointers.

When you write `f1=Mathf.Ceil;`, you're aiming it at that function. That's why there aren't any `()` parens after `Ceil` – you're not calling the function. We've never left them out before, but we've never had this trick before.

Using them works like other pointers. You automatically follow the variable to the function. If you start with `f1=Mathf.Floor;` then `f1(3.7f);` automatically follows `f1` to the function `Floor`, then runs it the normal way.

36.1.1 Declaring function pointers

This is a longish and semi-boring section. Knowing that `System.Func<float, float, float>` can point to Min or Max, but not square root (since it only has one input) is 90% of what you need. Skim this, quit when you get bored, then come back when you need the details.

The technical term for the type of a function is the *signature*, which is just the input and output types. Like regular pointers, there's no such thing as just a generic function pointer. They need to know the signature of functions they can point to.

This example makes `f2` able to aim at functions with two float inputs and a float return. If you look at the sample call, it's obvious why we can't point it to other types of functions:

```
System.Func<float, float, float> f2; // aim at:  float A(float, float)
f2 = Mathf.Max; // legal
f2 = Random.Range; // legal (picks the (float,float) version
f2 = Mathf.Floor; // ERROR, only 1 input (signatures don't match)
float ff = f2(3.4f, 7.9f); // sample call
```

Another example of the same thing, `f3` is for functions with a string input and output:

```
string makeLonger(string w) { w="x"+w+"y"; return w; }
System.Func<string, string> f3;
f3=makeLonger; // legal
f3=Mathf.Max; // not even close to legal
```

The exact way C# makes signatures for these is a little funny, but not too bad. As we saw, you write `System.Func` with angle-brackets. Inside, you list the input types, in order, then the output type *last* (not first, like in real functions – the output goes last.)

For example, `showCat` takes a Cat and returns a string, so has signature flipped as `Func<Cat, string>`:

```
string showCat(Cat c) { return c.name+" is "+c.age+" years old"; }
System.Func<Cat, string> g1 = showCat; // legal. signatures match
```

For a more oddball example, suppose some functions search a `float` array, starting where you tell it, and return one item:

```
float largest(float[] N, int startIndex) { ... }
float nicest(float[] Nums, int startHere) { ... }
```

The signature (inputs then output last) is `Func<float[], int, float>`. We can use it to make a pointer for them:

```
System.Func<float[], int, float> searchFunc;
searchFunc = largest; // legal, signatures match
searchFunc = nicest; // also legal
float num = searchFunc(NList, 3); // sample call
```

This rule works as-is for functions with no inputs. Put the return type by itself. For example, these two die-rolling functions take no input and return an `int`, so have signature `Func<int>`:

```
int twoD6() { return Random.Range(1,7)+Random.Range(1,7); }
int cheatingCoinFlip() { if(Random.value<0.75f) return 0; else return 1; }

System.Func<int> rollerFunc = twoD6; // legal
rollerFunc = cheatingCoinFlip; // also legal
```

For an odder example. `System.Func<Cat[]>` is a no-input function that returns a Cat array.

Functions with no return values are a little different. Instead of `Func` you write `Action` and just list the input types. Two examples of output-less functions and pointers for them:

```
void move(int x, int y) { ... }
System.Action<int, int> f1 = move;

void makeAllCatsThisOld(Cat[] C, int newAge) {
  for(int i=0;i<C.Length;i++) C[i].age=newAge;
}
System.Action<Cat[], int> catChangeFunc = makeAllCatsThisOld;
```

There's one more special case: no inputs and no outputs, like `void doMove();`. For that, you write Action all by itself, with no angle-brackets: `Sysem.Action f1; f1=doMove;`.

Just so you know, there's nothing special about functions with no return values and the Func/Action split. It just worked out that way. You'd normally write `Func<int,void>` for an `int` input and no output, but `void` isn't legal there. Instead of making it be, they decided to create `Action` for no-output signatures. There's nothing special about the word `Action`. It just seemed like a good name.

36.1.2 Using function pointers

Once you have a function pointer, you can do three things with it: point it somewhere, call it, or compare it.

Built-in functions and ones we wrote aren't any different to the computer. One pointer can flip between any functions, as long as the signatures match. The rules for changing where they point are the same as regular pointers. `f1=f2;`, makes `f1` point at the same function as `f2` does. We can even use `f1=null;`

The most interesting thing about using them to call a function is how it looks like a real function call. When you see `func1(5);`, you can't tell right away whether `func1` is a real function or a function pointer. It's similar to how in `c1.age` you don't know if `c1` is a struct or a class. In both cases, C# invisibly follows the pointer for you.

The rest of this is just examples of those rules, and little notes:

This is the standard pointer example, showing how we can aim these wherever we want. `average` is one of our functions, which has the same `float(float, float)` signature as Min and Max:

```
float average(float n1, float n2) { return (n1+n2)/2.0f; }

System<float, float, float> f1, funcPointer2;

void Start() {
  f1 = Mathf.Max;
  funcPointer2 = f1; // <- copy pointer to pointer
  f1 = Mathf.Min;
  float num = f1(5,8); // runs min, 8
  num = funcPointer2(5,8); // 5, f2 didn't follow f1 when we changed it

  f1=average;
```

```
num = f1(5,8); // 6.5
```

One thing to note is the variables aren't "smart" beyond the signature. What I mean is, Min, Max and Average feel like the same kind of math, but we're allowed to aim at any arbitrary function fitting the signature.

For example, we can aim `f1` at Random.Range, or even a useless "always -1" function:

```
float alwaysNeg1(float dummy1, float dummy2) { return -1; }

f1=Mathf.Max; // can go here
f1=Random.Range; // but also here
print( f1(6,9) ); // maybe 7.83?
f1=alwaysNeg1; // useless, but legal
print( f1(6,9) ); // -1
```

The `Random.Range` part is a little interesting, since we know there's a version with two `int`s and another with two `float`s. The trick is, `f1=Random.Range;` has to pick which one we use, using the signature of `f1`. When we come to `f1(6,9)`, we've already chosen the `float,float` version. That's not a big advantage, just kind of neat.

Compare really is the same as with regular pointers. This sets two function pointers and compares them to each other:

```
System.Func<float, float, float> f1, f2;
f1=Mathf.Max; f2=Mathf.Min;
if(f1==f2) print("point to same function");
if(f2==Mathf.Max) print("f2 points to Max");
```

The last line is something new. We can compare function pointers directly to functions. The same as with single-equals, `if(f2==Mathf.Max)` isn't running the function. It's just checking where `f2` is aimed.

As usual, one way of using pointers is to always assign them, so you never need to use `null`. But if you want, you can use `null` to mean not assigned yet, empty or invalid. A typical semi-useful example:

```
System.Func<float, float> fixerFunc=null;
if(mode==0) fixerFunc=Mathf.Floor;
else if(mode==1) fixerFunc=Mathf.Ceil;
if(fixerFunc!=null) result=fixerFunc(result);
```

You might notice this is only fake-useful. We could have just taken the floor or ceiling inside the `if`.

As usual, having `fixerFunc` be `null` isn't an error. Trying to follow a `null` pointer causes the problem. Running `fixerFunc(result)` on a null would give the standard run-time crash with nullReferenceException.

36.2 Uses

This whole section has no new rules – just common ways function-pointers are used.

36.2.1 As a regular variable

Any time you're thinking about making an `int` where 1 stands for function A, 2 for function B, you could make a function pointer instead.

Here's part of a program where our special weapon is either a laser, sonic cannon, plasma blob or nothing. This first version uses an integer to say which one it is (no function pointers, yet):

```
void fireLaser() { ... } // pretend these are written
void fireSonicCannon() { ... }
void firePlasmaBlob() { ... }

int specWeapon = -1; // 0=laser, 1=sonic, 2=plasma, -1=none

void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    if(specWeapon==0) fireLaser();
    else if(specWeapon==1) fireSonicCannon();
    else if(specWeapon==2) firePlasmaBlob();
  }

  void loadNextLevel(int lNum) {
    if(lNum==2) {
      specWeapon=0; // laser
      ...
    }
  }
}
```

The key messy part is having to remember the 0,1,2 weapon table. When we press a space, we need `if`s to decode the table. Then, when we change levels, we need to look at the table to assign the correct number for a laser.

We can simplify this by changing `specWeapon` from a look-up `int` to a function pointer:

```
System.Action specWeaponFunc = null; // no special weapon yet
// remember System.Action is the special syntax for no inputs or output

void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
```

```
    if(specWeaponFunc!=null) specWeaponFunc(); // <- easy call
  }

  void loadNextLevel(int lNum) {
    if(lNum==2) {
      specWeaponFunc=fireLaser; // <= nicer than a 0, and activates pop-up
      ...
    }
  }
}
```

The `if` is gone, and I think `=fireLaser` down below is easier to read than `=0`. Another advantage is when we type `specWeaponFunc=fire`, auto-complete will show us the three possible functions.

36.2.2 Arrays of pointers

If we have several functions, putting them in an array makes it easier to pick one. That's just a standard array trick, but it's worth seeing how we use it.

Some game background: when a character hasn't been moving for a while, we usually play an "idle" animation, like stretching. In this example, we have three of them, and want to pick one at random. Pretend each needs a different function to set it up:

```
void beginStretch() { ... } // pretend this starts stretching
void beginHandsOnHips() { ... }
void beginFlex() { ... }
```

We can put them in an array in the usual way:

```
System.Action[] Idles; // array of function pointers

void Start() {
  // set up the array:
  Idles = new System.Action[3];
  Idles[0]=beginStretch;
  Idles[1]=beginHandsOnHips;
  Idles[2]=beginFlex;
}
```

Now we can run `beginStretch` using `Idles[0]()`. That looks funny, but it's just array rules: `Idles[0]` is a function-pointer, so putting `()` after will run the function.

We already know how to pick a random item from a regular array. This picks a random item and runs it:

```
  IdleActions[Random.Range(0, IdleActions.Length)]();
  // runs beginStretch or one of it's friends, at random
```

It might look nicer broken over three lines:

```
  int iaIndex = Random.Range(0, IdleActions.Length);
  System.Action nextAct = IdleActions[iaIndex];
  nextAct(); // run it
```

You're allowed to use the array-creation shortcut. Start could just make the array in one line:
`Idles = {beginStretch, beginHandsOnHips, beginFlex};`.

I used simple void/void functions since it was shorter, but it might be nice to see what it looks like when we need the angle-brackets. Suppose the idle functions took a float input and returned a bool. We'd do it like this:

```
System.Func<float, bool>[] Idles;

void Start() {
  Idles = new System.Func<float, bool>[3];
  ...
  bool b = Idles[0](3); // run the 1st idle function, with input "3"
```

36.2.3 User-defined keys

This is another example of an array of function pointers. The idea is, we have three actions, drop, pickup and throw, and we want the player to pick which keyboard key does what (we'll make a menu for that, which I won't show in this example.)

We'll start with just a little struct pairing up the function pointer and current keypress. No array yet:

```
// pretend we have void() functions for drop, pickup and throw

struct KeyActionPair {
  public System.Action theAction; // pointer to drop, pickup or throw
  public char theKey; // key that does it
}
```

Then, the same as the idle action example, we'll make an array pre-filled with these pointers, and also the starting letters:

```
KeyActionPair[] KeyActions; // list of all actions and keys

void Start() {
  KeyActions = new KeyActionPair[3];
```

```
  KeyActions[0].theAction=drop; KeyActions[0].theKey= 'd';
  KeyActions[1].theAction=pickup; KeyActions[1].theKey= 'p';
  KeyActions[2].theAction=throw; KeyActions[2].theKey= 't';
}
```

Now `KeyActions[0].theAction()` runs the `drop` function, and `KeyActions[0].theKey=ch;` changes the key we use to drop things (the user menu we're not writing would do that.)

Unity likes us to read keys individually. So, to check during play, we'll loop through the array: check if that key was pressed; if so, run the function next to it:

```
if(Input.anyKeyDown) { // not needed, but can't hurt
  for(int i=0;i<KeyActions.Length;i++) {
    if(Input.GetKeyDown( KeyActions[i].theKey )) {
      KeyActions[i].theAction(); // run the function going with the key
      break;
    }
  }
}
```

If you know constructors, this would be nicer if the `KeyActionPair` struct has a simple 2-input constructor for setting the the pointer and the key together. I left it out to avoid clutter.

36.2.4 Passing functions into functions

The best way to explain this trick is with an example. Here's a regular function to count how many positive numbers are in an array:

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

It would be great if we could send in a replacement for `>0`. Now that we have function pointers, we can. The first step is to rewrite greater-than-0 as a function:

```
bool isPositive(int x) { return x>0; }
```

Notice how this is a `bool(int)` function. That's what `>0` really is. It takes any integer, and tells us whether it likes it. Rewritten, it's a `System.Func<int, bool>` function.

The last step is to rewrite `arrayCount` to take that as an extra input. There are two changes – the extra input, and using it inside the `if`:

```
int arrayCount(int[] A, System.Func<int,bool> theTest) {
  int count=0;
  for(int i=0;i<A.Length;i++)
    if(theTest(A[i])) count++;
  return count;
}
```

Like any other parameter, `theTest` is going to passed in by the caller. For example:

```
int[] N = {4, 0, -3, 78, -11};
int c = arrayCount(N, isPositive); // 2
```

Notice, on the last line, how `isPositive` is still missing the parens. We're not calling it – we're just passing a pointer. Inside the function, `theTest` points to `isPositive` and `theTest(A[i])` runs it.

After all that work, we can finally count whatever we want, as long as we write a testing function. This counts how many in the array are even:

```
bool isEven(int num) { return num%2==0; }

  // elsewhere in the program:
  int n=arrayCount(N, isEven);
```

Even though `isEven` is a function name, it's passed like an normal variable. It's just copied into `theTest`. Below, in the loop `if(theTest(A[i]))` is now magically running the `isEven` function, counting how many things in the array are even.

Positive and even are 1-line functions. This next example is just to use a longer one. There's nothing new, but I think it helps. The function checks if a number is a square (1, 4, 9, 16 ...,) using a loop (only read it if you like odd loops):

```
bool isSquare(int num) {
  if(num<0) return false;
  int n=0;
  // keep adding 1. Stop when n-squared hits num, or goes past:
  while(n*n<=num) n++;
  if(n*n==num) return true;
  else return false;
}
```

There's nothing special about it, and it just checks one number. `isSquare(81)` is true, `isSquare(22)` is false. The key is that it has a `Func<int,bool>` signature, so we can use it in `arrayCount`:

```
int[] A={9,5,25,3,2};
int sqCount = arrayCount(A, isSquare); // 2 (9 and 25)
```

I like this example since it's not special at all. We're calling another function over and over from inside the `arrayCount` loop. But we've always been able to do that. The function we call has another loop in it, making the whole thing a nested loop. But we've also always been able to do that.

The only new thing is being able to plug in any function.

Change each one

A function that changes each item in an array uses pretty much the same idea – we give it a function to use on each item. To make it interesting, I'm going to use an array of `Cat`s.

A simple cat-changing function takes one `Cat` as input and has no output. In other words, they will look like `void(Cat)` or officially `System.Action<Cat>`. Here are two sample functions to change one Cat:

```
void makeOlder(Cat_t c) { c.age++; }

void makeRoyal(Cat_t c) {
  // add "Lord" in front, unless the name already has a Lord in it:
  bool alreadyIsRoyal=false;
  if(c.name.FindFirst("Lord")>=0) alreadyIsRoyal=true;
  if(!alreadyIsRoyal) c.name="Lord "+c.name;
}
```

Our cat-array changing function will take one of these as input, and do it to each thing in the array. This is a little simpler than counting:

```
void changeCats(Cat_t[] C, System.Action<Cat> changer) {
  for(int i=0;i<C.Length;i++) changer(C[i]);
}

  // in the program:
  changeCats( MyCats, makeOlder);
  changeCats( CatList2, makeRoyal);
```

Multiple function inputs

Just to show we can, let's write a function with *two* function inputs. Take the array changer function above, and add a checker. In other words, we'll only make some certain cats royal. Here are two sample cat-checking functions:

```
// two sample cat-testing functions:
bool isHeavy(Cat_t c) { return c.weight>10; }
bool isKitten(Cat_t theCat) { return theCat.age<=2; }
```

These cat-checking functions are like the int-checking ones except cats have more parts we can check.

Now we'll add a cat-checker to the cat-array changing function. There's nothing new here, just a combination of the old checker and changer ideas:

```
// new cat-changer:
void changeSomeCats(Cat_t[] C,
       System.Func<Cat,bool> theTest, System.Action<Cat> theChange) {
  for(int i=0;i<C.Length;i++)
    if(theTest(C[i])==true) theChange(C[i]);
}
```

Notice how the top part is starting to get ugly. We've got a `Func` and an `Action`, and I always get confused how `Func<Cat,bool>` is really `bool Func(Cat)`.

Calling it is nothing special, just put the names of the real functions, in the order it says:

```
// cats 11+ pounds get older:
changeSomeCats(C1, isHeavy, makeOlder);

// young cats become Lords:
changeSomeCats(MyCats, isKitten, makeRoyal);
```

This exact example is a little contrived. But the idea is that function inputs are like any other inputs. Use as many as you need.

Change all children

We can use the idea of a change-function with the visit-all-my-children Unity trick. If you haven't seen it, or have but still don't understand recursion, this will make less sense but is still worth skimming.

As a review, here's the recursive function to touch a gameObject, all of its children and so on. As a stand-in, it just prints the name:

```
void touchAllChildren(Transform t) {
  print( t.name ); // <= test action
  int childCount=t.childCount;
  for(int i=0; i<childCount; i++)
    touchAllChildren(t.GetChild(i)); // <= recursive call
}
```

The key is it does something with `t`. We'll change printing the name to calling plugged-in Transform-changing function (the same as a Cat-changing function.) The two changes are the extra input and the first line:

```
void changeAllChildren(Transform t, System.Action<Transform> action) {
  action(t); // <= do whatever thing we plugged in
  int childCount=t.childCount;
  for(int i=0; i<childCount; i++)
    changeAllChildren(t.GetChild(i), action);
}
```

I named my function pointer `action` since I couldn't think of a better name. Here's a sample function to turn just one thing red, and a call using it to turn all my children red:

```
void turnRed(Transform t) {
  Renderer rr = t.getComponent<Renderer>();
  if(rr!=null) rr.material.color=Color.red;
}

  // dog and all dog descendants turn red:
  changeAllChildren(dog.transform, turnRed);
```

There's nothing really special here. This is the same idea as the Cat-changer. If you have some way to look at a bunch of stuff, even if it's a complicated recursion, you can still use the plug-in function trick. We sometimes say we're *visiting* each item. How we visit them all is one thing. What we do during the visit is the other, separate part.

36.3 Function pointers as callbacks

Unity's physics system is an example of *callbacks*, which are usually done using function pointers.

The key idea is you just write `OnCollisionEnter`, with the inputs you're supposed to. Then it's magically called. You don't need to check for collisions yourself – you just need to know the system is checking for them and will call you when it needs to.

The functions you plug into the system are called *Event Handlers* or *Call-Backs* or *Listeners*. The terms are used interchangeably.

The things the system tracks for you are the *Events*. They're nothing special – just what the people writing it thought were important.

What usually happens is you have some pre-written event system. You have to figure out what the events are, what the function signatures look like, and what the rules are for hooking up your callback functions.

Because of that, some examples below aren't complete. I'm trying to show only the important parts, but maybe not doing a great job. Hopefully at least one of these will make some sense:

Drags and Slides

Suppose we have a pre-written script that tracks stray touches and tells you about drag and pinches. The interesting part of that script is this:

```
public System.Action<Vector2> dragCallback=null; // input is x&y movement
public System.Action<float> pinchCallback=null; // input is pinch amount

void fingerUpdate() {
  ...
  // whenever we know there's a slide:
    if(slideCallback!=null)
      dragCallback(dragAmt);
```

The first two lines are the callbacks. They're just global function pointers, which is the point. The first line tells you the dragging callback takes a `Vector2` as input, and you register it by assigning to that variable.

The snippet of code in Update is just showing how the system runs it. It's doing a lot of math I'm not showing, but it just calls `dragCallback` like any other function pointer, which is aimed back at a function you wrote.

Our part of the code might look like this:

```
public fingerTrackScript FT; // link to script

void Start() {
  // aim the callbacks at our functions:
  FT.dragCallback=LRcamSlide;
  FT.pinchCallback=null; // ignore pinches
  ...
}

void LRcamSlide(Vector2 amt) {
  float sideMove=amt.x*20;
  theCam.position+=new Vector3(amt, 0, 0);
}
```

All our code does is write the drag callback and register it (up in Start.) To us, it looks like `LRcamSlide` is magically being called when a finger moves.

I purposely have it only use the `x` part, ignoring the `y`. That sort of thing is common. The pre-written system often provides everything we might possibly need. It's similar to how the `Collision` struct in Unity's `OnCollisionEnter` has a lot of fields you might never use.

Unity's UI callbacks

Unity's canvas/UI system uses callbacks. Buttons have a script on them named `Button`, with a `void()` callback. In other words, you can write a no-input function and tell the button to call it when it's clicked.

This is working code:

```
public UnityEngine.UI.Button btn1; // drag in a Button

// button click function for btn1:
public void testBtnClick() {
  // NOTE: buttons have a child named "Text"
  string w=""+Random.Range(1,1000); // sample random #
  btn1.transform.Find("Text").GetComponent<UnityEngine.UI.Text>().text=w;
}

void Start() {
  // add my testBtnClick function as callback for button1:
  btn1.onClick.AddListener(testBtnClick);
```

The last line looks funny because Buttons don't just have one click callback – they have a list. This is a standard bonus feature that most systems have.

`onClick` is a tiny class holding an array of `void()` pointers. `AddListener` is just a shortcut for adding to the end. Notice how we're still just giving it no-parens `testBtnClick`, which is a function pointer.

If you want to switch to another callback, you have to remove your first one (they give you `RemoveListener` to get rid of just one, or `RemoveAllListeners` which clears the whole array) then add the new one.

It's not easy to figure out that Buttons want a `void()` function. If you look at the tool-tip for `AddListener`, it says the input is a "callback function." Then it says the type is `void UnityAction()`, which means `void()`.

Redirecting OnCollision

This next example is a simple use of a callback. In Unity, collisions will only ever call a script on the object that was hit. If my game has 5 different color balls, doing different things when they get hit, I need a different script for each color. My ball code is spread out over all those scripts and I don't like that.

Instead, I'll write one script, used on every ball, which redirects the collision to a function in my main script:

```
class genericColliderScript : MonoBehaviour {
  // set this to point to the real collision handler:
  public System.Action<Transform, Collision> callBack;

  void OnCollisionEnter(Collision col) {
```

```
    // tell it who you are, and your collision info:
    if(callBack!=null) callBack(transform, col);
  }
}
```

By setting the `callBack` variable, we can tell each block what to do on a collision.

This sample main program has the collision functions for two block colors, and sets the blocks up to use them:

```
void handleRedBlockHit(Transform block, Collision cData) {
  // red blocks just die when hit:
  Destroy(block.gameObject);
}

void handleGreenBlockHit(Transform block, Collision cData) {
  // we get a point when green blocks are hit:
  score++;
}

void Start() {
  // setup a red  block:
  Transform rb=Instantiate(redBlockPF);
  // have it call the red function above, when hit:
  rb.GetComponent<genericColliderScript>().callBack=handleRedBlockHit;
  // for real we'd position it, etc...

  // bunch of green blocks:
  for(int i=0;i<5;i++) {
    Transform gb=Instantiate(greenBlockPF);
    gb.GetComponent<genericColliderScript>().callBack=handleGreenBlockHit;
  }
```

Now all of our block collision code is together, and can easily use our variables if they need to (like `score` for green ones.) It's not a huge improvement, but being able to organize your code is nice.

36.4 Just cramming in a function

For short things, it's a pain to have to write a separate function. There's a shortcut rule to let you write a nameless mini-function directly. This legal code uses the `countSome` function to count numbers less then 10:

```
print( countSome(A, n => n<10));
```

The `=>` symbol was made up just for these mini-functions. A trick to understanding them is you always know the required types. In this example, `countSome` has to take a `bool(int)` function, so it knows `n` stands for the `int` input. A shortcut lets you leave out the `return`: if you just write math, it knows to automatically return it.

There's a longer version doing the same thing:

```
print( countSome(A, (n)=>{return n<10;} ));
```

The syntax can be a little confusing, but if you get the idea you can look up examples easily enough. The official name for these are **lambda expressions**. That's a real computer science term – when functions were invented, back in the 60's, that's what we called them.

You can use this trick to assign; and you can also have multiple inputs. These completely silly examples show both:

```
System.Func<int,int,int> f1;

f1 = (x,y)=>x*y/10; // nameless (very silly) math function
print( f1(5,7) ); // 3 (35/10)
```

Notice how we're leaving out the curly-braces and the `return` again. It assumes `x*y/10` is the answer. The long way is `(x,y)=>{ return x*y/10; }`. No matter what, we leave out the types. We already wrote they're all `int`s when we declared `f1`.

This one has an `if` and two returns, more like a real function. You could write a very long function this way, but it's probably better to pre-write those, the regular way:

```
// nameless max function:
f2 = (x,y)=>{if(x>y) return x; else return y; };
float n = f2(7,9); // 9
```

As you might guess, the main use for this trick is not having to go off and write all those 1-line functions. For my super cat-changer, we can change the name of all 1-year-old cats to Kitty:

```
changeSomeCats(C, c=>c.age==1, c=>{c.name="Kitty";} );
```

Then, a C# note just in case: if you look these up, there's an obsolete syntax for these using the word `delegate`. Ignore that and you'll find this good way further on.

36.5 More rules and examples

A nice built-in place that uses these is sorting. If `A` is an array and you type the partial line `System.Array.Sort(`, you'll see many, many overloads, but one of them takes an array and a compare function.

Before we can use it, we need to know how a standard compare function works. This compares two strings (note how it's named `Compare` in the `string` namespace):

```
print( string.Compare("cat", "dog")); // -1, first lessThan second
print( string.Compare("cat", "cat")); // 0, equal
print( string.Compare("dog", "cat")); // 1, first greaterThan second
```

It seems like you'd return just True/False, but it's better to return less-than, equal-to, or greater-than, using -1, 0 or +1. The way I remember is it returns less-than-zero for less-than, and so on. Doing it this way is pretty common.

Now we can write two compare functions for the `Cat` class. They're just regular functions, but it's nice to have them use the class as a namespace:

```
class Cat {
  public string name; public int age;

  static public compAge(Cat a, Cat b) {
    if(a.age<b.age) return -1;
    if(a.age==b.age) return 0;
    return +1;
  }

  static public int compNameHighLow(Cat c1, Cat c2) {
    return -1*string.Compare(c1.name, c2.name);
  }
}
```

These compare just one Cat to another, which is what the built-in Sort wants. Notice how sneaky I am in the second one. The main thing it does is compare names, using the built-in string compare. Then, it so happens, `-1*` is a way to flip the -1/0/+1 result to go high-to-low.

We can use either cat-compare to sort the entire array:

```
Cat[] C; // pretend this has cats in it

System.Array.Sort(C, Cat.compAge); // sorted by age
System.Array.Sort(C, Cat.compNameHighLow); // by name, backwards
```

Or we can use the nameless function trick. This sorts by the length of the name (using a nested `?:` operator.) Remember that it knows `c1` and `c2` are Cats, since they have to be:

```
Array.Sort(C, (c1,c2)=>{ int a=c1.name.Length, b=c2.name.Length;
                         return a<b?-1:(a==b?0:+1); });
```

36.5.1 Older syntax, `delegate`, shortcuts

C# has some older, alternate syntax based on two things from C++. It wouldn't be important, except if you try to look up C# function pointers, you'll see lots of things done this old way.

The first C++ trick is making an alternate name for a type. This isn't like making a new class, where we had to pick a name. This is making another name for a type just because we feel like it:

```
typedef int wholeNumber;
typedef Cat[] CatList;

wholeNumber n; // n is just an int
CatList C1; // C1 is just an array of cats
```

The other trick is the standard way of declaring function pointers, which is to make them look like the function header (again, this is C++, not C#, so don't worry too much about the details):

```
// declares f1 as a float(float,float) pointer:
float (*f1)(float,float);

f1 = Mathf.Max; // sample assignment
```

The idea is the variable you're declaring goes in the middle, like a function name. Then the stuff around it tells you the signature of what it can point to.

C# doesn't have either of those, but it does have a rule combining them. It was the original rule for function pointers. You create a typename using `delegate` and a fake function heading, then use it to declare variables:

```
delegate float chooser(float a, float b);
chooser f1;  f1=Mathf.Max;
```

This defines `chooser` as a type (not a variable.) On the next line, `f1` is declared as a regular function pointer, except using `chooser` as a mandatory shortcut.

In one way it's nicer, since you don't need the special `Func/Action` rule. But in another way it's not, since the extra step is required.

`Sytem.Func<float,float,float>` is really a shortcut for creating a nameless `delegate` on the fly. But it works fine, and I obviously think it's better. But it's good to know the word `delegate` because it's the official name for C# function pointers.

If you look up function pointers, you'll see lots of lots of these old examples. There's even an old version of `x=>x+1` using the word `delegate`. You can safely ignore them and skip to the modern examples.

`System.Func` is only the most common shortcut. They added others. For example, `Comparison<T>` is a shortcut for a standard compare function. It's the same as `System.Func<T,T,int>`. You'll see it inside pre-made compare functions, like `Sort`.

36.6 Built-in array function-using functions

C# has some fun built-in functions using an array and a function pointer. An easy one (which I wrote earlier by hand as `arrayCount`) is counting things we like in an array:

```
// need this at the top:
using System.Linq;

int[] A={4,8,7,1,3,12,5};
int n=A.Count( x=>x>6 ); // 3
```

`Count` is written as an array member function. Putting `using System.Linq` at the top adds it. I used the nameless function shortcut, but it works with anything, for example `A.Count(isPositive)`.

It also works for an array of anything. This counts how many cats have long names:

```
Cat[] C; // pretend this is created
int n=C.Count( c=>c.name.Length>12 );
```

There are two more taking a "do I like this?" function. `Where` returns a shorter list of just what we like. `Select` returns a same-length list of true/false:

```
int[] A={4,8,7,1,3,12,5};
int[] B=A.Where( x=>x>6 ).ToArray(); // B is {8,7,12}

bool[] C=A.Select( x=>x>6 ).ToArray(); // C is {f,T,T,f,f,T,f}
```

Select doesn't seem useful right away, but it's there just in case. The extra `ToArray()` just goes there.

There are two `bool`-returning ones. `All` means "is it true for all of these?" and `Any` means "is it true for any of these":

```
int[] A={4,8,7,1,3,12,5};
// is anything in A more than 10:
bool hasAnyMoreThan10 = A.Any( x=>x>10 ); // true
```

```
// is all of A more than 10:
bool areAllMoreThan10 = A.All( x=>x>10 ); // false

// use a premade function:
if( A.All( isPositive )) print("everything in A is positive");
```

To test whether all of our cats have names, use:
`if(C.Any( c=>c.name=="" ))`

Old version

If you look up Select, Where, Any and All, you'll see some very strange stuff. You can skip to the back for the good version, but it's semi-interesting to see what these other versions are and how it happened.

Most database queries use a mini-language named SQL (Standard Query Language.) It does things like "Select phone numbers from everyone Where country is Canada." Probably more people know it than know C#. Database people know it well, and all the tricks. But some people know and use it through simple dropdowns. It's not a programming language, and the syntax is very different from C#.

Most programming languages have add-ons to use things like SQL. C# just jams a modified version right in the language, naming it all "linq." For fun, type the two searches "C# linq good/bad." People will tell you why it's great, and why it a terrible idea.

This odd SQL-like version is the first thing you'll see when you search for C#'s `Where`, `Select` and so on.

But if you keep reading, they show you the programmer's version (the examples I put above.) Then they explain how the SQL-like version is really for non-programmers, who know SQL. And they explain that it's really just a front-end and you should use the real version if you know computer programming.

For more language archeology, you can look at the various versions of C# where they gradually added and changed things. It won't make you a better programmer, but it's nice knowing that computer languages are designed and changed by humans.

Chapter 37

Inheritance

After playing around with Unity for just a little while, you probably figured out *component* is the word it uses for things you can add to gameObjects. For example, rigidbodies are in the Component menu. You probably also noticed the Inspector doesn't seem to have preassigned slots for each type – it acts like one list with combined meshes, scripts and colliders, all jammed side-by-side.

You may have looked it up and seen component isn't just a word for humans. There's an actual C# class named `Component`, and the `Mesh` class also counts as the `Component` class. What you see in the Inspector really is a single list with different classes in it, which is somehow legal because rigidbodies and even your scripts officially also count as `Component`s.

That trick is accomplished using *inheritance*.

There are three parts to inheritance. Part one is the rules for making a class that grows from another one. Part two is making a pointer that can aim at either class. Part three is about using that pointer to call functions in a nice way.

Part one is mostly useless by itself, but we have to know it for part two. I'll write examples, but don't try to figure out how they'd be useful for real. Part two is how Unity accomplishes the single list of different types of things trick. But part three is where you finally see a real example and can understand why we invented inheritance.

37.1 Pre-inheritance example

The simplest, most basic use of inheritance is when you want to create two classes which have a lot in common. For example, Cats and Dogs will each have name, age and weight.

We might split off common animal data into a "helper" class, maybe named `Animal`:

```
class Animal {
  public string name;
  public int age;
  public float wt;
}
```

Now we can make the `Cat` and `Dog` classes using an `Animal` plus specific variables for that one kind. There are no new rules here yet:

```
class Cat {
  public Animal A;
  int cuteness;
}

class Dog {
  public Animal A;
  public float barkVolume; // in decibles
  public string favoriteChewToy;
}
```

The advantages of this idea are probably obvious – it's basic "don't write the same thing twice." But I'll list them anyway:

- Shorter code (especially if we have lots in common, and create more than just Cat and Dog.)
- Can be easier to read. Once you get used to it, `Animal A;` is the shortest, clearest way of saying a cat has all the basic animal variables, with the exact same spelling as every other Animal.
- Easier to add a common stat. Anything you add to `Animal` automatically goes into `Cat` and `Dog`. It also makes sure all they stay in synch.

Another benefit is we can split off just the `Animal` part. We can have a pointer to it, or send it to a function:

```
Animal a1;
// can point to a cat's or dog's animal:
a1=c1.A; a1=d1.A;

void animalTons(Animal a) { print("Tons="+a.wt/2000; }
animalTons(c1.A); // tons of Cat
animalTons(d1.A); // tons of Dog
```

There are two sort-of drawbacks. We have to remember to `new` the `Animal`, or else get a null reference error. Not a big deal – we'd just put that in the constructor (this has nothing to do with inheritance, but it is a good constructor example):

```
class Cat {
  public Animal A;
  public int cuteness;
  public Cat() { A=new Animal(); }
}
```

The other annoyance is having to use the `A` for some variables, and just having to remember it's an `A`. `c1.cuteness=5;` is easy. `c1.A.name="Henry";` isn't as nice.

37.2 Basic Inheritance

The simplest Inheritance rule makes sharing common variables just a little easier. We start exactly the same as before, by making a class for what they have in common. I'm copying `Animal` here, with no changes:

```
class Animal {
  public string name;
  public int age;
  public float wt;
}
```

Now, the same as before, we want to use this to make `Cat` and `Dog`.

The new Inheritance rule says you put `: Animal` after your name (that's a full colon.) Doing that directly injects everything from Animal into you:

```
class Cat : Animal {
  public int cuteness;
}

class Dog : Animal {
  public float barkVolume;
  public string favoriteChewToy;
}
```

Now Cat and Dog have name, age and wt directly inside them. You don't need to make up an extra name, or use an extra `new`. The `Animal` variables are magically inserted.

To anyone using `Cat`, it looks like a regular class with four variables:

```
Cat c1;
c1.name="Mr. Boots"; // declared in Animal, but used like it was in Cat
c1.age=12; // same
c1.cuteness=4;
```

For some technical terms: we say Cat and Dog inherit from Animal. We'd call Animal the *base class*. It's still just a regular class, but in relation to Cat we'd say it's the base.

We also sometimes say Animal is the *superclass* and Cat is the *sub-class*. That can be confusing, since the subclass has more than the superclass. But everyone uses super and sub that way.

Here's one teaching example using inheritance, with nonsense classes:

```
class Furf { public bool ready; }
class Harhar : Furf { public bool done; }
```

`Furf` is a regular class, that we can use the regular way. `Harhar` also acts like a regular class, with two variables:

```
Furf f1 = new Furf(); f1.ready=true;
Harhar h1 = new Harhar(); h1.ready=false; h1.done=false;
```

This is a pretty simple rule, so far. Not very tricky, but also not much of an improvement over what we could do without it.

37.3 Intro to Polymorphism

The major advantage of using the official inheritance rule is that a `Cat` also counts as an `Animal`. This is a completely new thing, it's the real reason we invented inheritance, but it takes some explaining why it's so good.

Here's a non-useful example just showing the "counts as" rule. We can make a `Cat` or `Dog`, and use an `Animal` to point to it:

```
Cat c1 = new Cat();
Dog d1 = new Dog();

Animal a1 = c1; // <- animal pointing to a cat. this is legal!
a1.name = "Biggles";

a1=d1; // <- same animal points to a dog. also legal
d1.name="Spot";
```

This isn't just technically legal – it really accomplishes what it looks like. An animal pointer can reach into a Cat to change the name, then do the same

thing for a Dog.

Before explaining the rules more, here's a useful example. This ordinary function finds the longest name in an array of `Animal`s:

```
string longestName(Animal[] A) {
  if(A.Length==0) return "";
  string longest=A[0];
  for(int i=1; i<A.Length; i++)
    if(A[i].name.Length>longest.Length) longest=A[i].name;
  return longest;
}
```

The special part is we can call it with arrays of Cats and Dogs, since they count as Animals:

```
Cat[] CList = new Cat[10]; // pretend we fill this with cats
string w = longestName(CList);
// same for an array of Dogs
```

This counts as legal, but also should make sense. The important part of the function is where it uses `A[i].name`, which is the `name` variable of an Animal. Cats and Dogs have that exact thing in them.

Saying they count as animals is like saying you can look at a Cat and squint, to see just the Animal part.

We can also mix `Cat`s and `Dog`s in the same `Animal` array:

```
Animal[] MyPets = new Animal[4];
MyPets[0] = new Cat(); // putting a Cat into a animal array
MyPets[1] = new Dog(); // and a Dog
MyPets[2] = new Cat(); // and so on
```

Before, we could never make an array of mixed types, now we can. To the computer, it's just an array of `Animal`s, so it's happy. We could even send this mixed array into `longestName` and it would work just fine.

Just to be clear, we can't put `Dog`s into an actual `Cat[]` array, or vice-versa. And we can't send an `Animal[]` into a function wanting a `Cat[]`. The only special rule we have is that `Cat`s and `Dog`s count as `Animal`s, because of inheritance. Doing it with an array is just a useful way to use that rule.

The rule for "downcasting" like this is you still have to respect the official type, and can only use the things it has. That sounds more complicated than it is. It means that if you're using an `Animal` pointer, you can change name, age or wt. You can't change cuteness, even if you know it's really pointing to a Cat:

```
Animal a1; // this could be pointing to a Cat or Dog
a1.age=6; // legal
a1.barkVolume=39; // ERROR (even if it really points to a Dog)

MyPets[3].name="Fritz"; // legal
MyPets[3].cuteness=6; // ERROR, Animal doesn't have cuteness
```

That rule is really very natural. One way to see it is that if we have `Animal a1;`, the normal rules say it can only use `Animal` variables. Being able to aim it at a `Cat` or `Dog` doesn't change that.

Another way to think of it is that if an `Animal` can go back and forth pointing to a `Cat` or `Dog`, it can only logically use things they have in common.

37.4 Dynamic casting

Now that an Animal pointer can aim at a Cat or a Dog, we have a new problem. How can we go back? More formally, how can we tell if `a1` is really pointing to a Cat?

Then we have one more problem. Even if we know `a1` points to a Cat, the computer doesn't. `a1.cuteness` is still an error. We'll solve both problems at once.

What we really want to do is create a fresh cat-pointer, write `c1=a1;`, then check if it worked. If it did, we can use `c1` to do Cat stuff. If not, then `a1` wasn't pointing to a Cat. Here's a working example doing that, using our one new rule:

```
void animalFunc(Animal a1) {
  print(a1.name); // warm-up. This part is always fine

  Cat c1 = a1 as Cat; // <- new rule
  if(c1!=null) print(c1.cuteness);
}
```

Using a function is just a way to establish how `a1` could point to just an Animal or a Cat or a Dog. The last two lines are the new part – the standard way to check for a cat. "`as`" is a new keyword.

`c1=a1 as Cat;` says we know `a1` might be a `Cat`, or might not. It says to try `c1=a1;`. If we can't do it, since it isn't a Cat, just make `c1` be `null`. The last line, `if(c1!=null)` is really asking "did it work?"

Technically, it combines three ideas. The first thing is it checks for a Cat. The second is it doesn't just say yes or no. For yes we get the pointer, and for no we get `null`. The last part of the idea is it converts an Animal pointer into a Cat pointer. The place the pointer goes doesn't change, but instead of an

`Animal` pointer to a `Cat`, we get a `Cat` pointer to that same cat.

Here's a working example using that trick to add the cuteness of just the Cats in an array, skipping any Dogs:

```
int totalCuteness(Animal[] A) {
  int cuteSum=0;
  for(int i=0; i<A.Length; i++) {
    Cat c = A[i] as Cat; // check for a Cat. If it is, c points to it
    if(c!=null) cuteSum+=c.cuteness;
  }
  return cuteSum;
}
```

If we want to know the exact sub-type, there's no special way. We just use the "`as`" trick as many times as we need. This separately counts Cats and Dogs:

```
void animalChecker(Animal[] A) {
  int cats=0, dogs=0;
  for(int i=0; i<A.Length; i++) {
    if((A[i] as Cat)!=null) cats++;
    else if((A[i] as Dog)!=null) dogs++;
  }
  print(cats+" "+dogs);
}
```

I got a little sneaky, but it should be obvious what's going on. The real command is `A[i] as Cat`, which returns `null` if it wasn't a Cat (it returns `A[i]` if it was, but we don't care about that here.) Usually we assign it to a variable, but we can cut out the middleman by putting it directly inside the `if`.

Formal rules and examples

This is probably obvious: `as` only works with inheritance chains. In `a1 as Cat`, the first thing has to be a pointer of some base class, and the second thing has to be the name of a subclass. In other words, it has to be a question where the answer really could be yes or no.

For example, `c1 as Dog` is an error. A Cat can never point to a Dog so there's no reason to ask.

`as` won't work with non-inheritance stuff at all. This first part is total junk (for fun, the last part is how to do it, but has nothing do to with inheritance):

```
float g=3.0f;
int n = g as int; // error - only works with inheritance

if((int)f==f)
  print("f ends with point-0"); // sneaky way to check for point-0
```

When you assign to a variable, like in `Dog d1=a1 as Dog;`, you need the same type at both ends. Here's an example to try to confuse you about that:

```
Animal a1 = new Cat(); // animal variable, pointing to real cat
Dog d1 = a1 as Cat; // syntax error -- can't assign cat to a dog
Dog d2 = a1 as Dog; // legal (will be null)
```

Since we know it's a Cat, `a1 as Cat` seems right. But the point of those lines is to check whether it's a Dog. That means we have to try to turn it into a Dog using `a1 as Dog`.

Then here are some notes that might help, but aren't anything you need to know:

- `a1 as Cat` is called dynamic casting since it's casting with inheritance logic. Whenever something looks through an inheritance chain to see what to do, we often call it dynamic.

  The casting part is much simpler than we usually do. Normal casting, like `(int)f` returns a changed value. But for this, you either get what you started with, or `null`. It's technically casting since we get an upgrade from `Animal` to `Cat`.

- `c1=a1;` is an error for the regular reason – we can't assign an `Animal` to a `Cat`. The types aren't an exact match. The special polymorphism rule only goes one way. To review the special rules are:
  `a1=c1;` is special, and always works. And `c1 = a1 as Cat;` is special, always legal, but might be `null`.

- Checking for `null` isn't required. Like everything else, you don't have to check if you know it's safe. `(a1 as Cat).cuteness=5;` is legal. But if it wasn't a cat, `a1 as Cat` would be null. Then dot-cuteness gives the usual null reference exception.

37.5 Inheritance and functions

You also inherit functions from the base class. It works the usual way – they count as being directly inside of you. Here's a quick, boring example. Cat has two functions: the one in it, and the one it got from Animal:

```
class Animal {
  ...
  public string aStats() { return "name: "+name+" "+age+" yrs old"; }
}

class Cat : Animal {
  ...
```

```
  public string cStats() {
    return "cute: "+cuteness;
  }
}
```

We'd use both of these as if they were inside `Cat`, without having to know which used inheritance:

```
Cat c1 = new Cat();
string w1 = c1.aStats() + " " + c1.cStats(); // both look like normal Cat functions
```

Things work the same inside other `Cat` functions. We call "our" functions and the ones we inherit the same way:

```
class Cat : Animal {
  ...
  public string allStats() {
    string s1 = aStats(); // inherited from Animal
    string s2 = cStats(); // defined in Cat
    string result=s1+" "+s2;
    if(wt>10) result+=" heavy";  // use inherited wt variable like normal
    return result;
  }
}
```

In short, inheriting functions from the base class works exactly how you'd expect, with no new rules or surprises.

One thing to note is that inheritance only goes one way. Functions in Cat can use variables in Animal. But functions in Animal are completely normal. Animal by itself never "knows" about things that inherit from it.

There is one new rule for inheritance and functions. You're allowed to re-use a function name in the subclass: you can have a `stats()` in `Animal` and another in `Cat`.

The rules for how that works are: 1) the original is hidden from people outside the class, and 2) inside the class, you can use them both. Use yours normally, and the covered-up one by adding `base.` (base-dot).

Here's a teaching example. Goat and Blob inherit from Animal. There's a `stats` function in Animal, and another in Goat covering it up:

```
class Animal {
  ...
  public string stats() { return "name: "+name+" "+age+" yrs old"; }
}
```

```
class Goat : Animal {
  ...
  // same name. Covers up stats in Animal:
  public stats() { return "baa "+name+" baa"; }
}

class Blob {
  ...
  // not writing a stats() here. We get the one from Animal, like normal
}
```

An outside user sees one `stats` function for each class. Blob inherits the one from Animal. Goat didn't like that one, so used the cover-up rule to hide it with a better one. Outside the class works like this:

```
a1.stats(); // the one from Animal
b1.stats(); // the one from Animal
g1.stats(); // baa - the one from Goat
```

The plan is that some animals will like the basic `stats` function. They can do nothing and get it for free. Others will want their own version, so can overwrite it. The end-user only sees what they need to, and doesn't need to know how it got that way.

This example shows the second rule, how to use both versions from inside the class:

```
class Cat : Animal {
  ...
  public string stats() {
    string w=base.stats(); // run the one from Animal
    return w+", cute: "+cuteness;
  }
}
```

This is pretty common. The one covering it up uses the first one as a helper function. It's typical inheritance thinking. We use `stats` in Animal to do the basic work, and the one in Cat adds on the extra work.

The word `base` in front comes from how Animal is the base class. If you remember, we also call it the super class. Some languages use the word super instead of base, to reach back like this.

Here's one more with the same idea, but using numbers:

```
class Animal {
  ...
  public float cost() { return age*5+wt*2; } // made-up cost formula
}
```

```
class Cat : Animal {
  // cats cost 25% less than most animals:
  public float cost() { return base.cost()*0.75f; }
}
```

I think `base.cost()*0.75f` is easy to read, and says what it means.

This is the same idea as with `stats`. Maybe some types of Animals are happy with the `cost` they inherit. Maybe they all cover it up and use `base.cost()` as a helper function. Maybe all Mice are 50 cents, so they cover up the one in Animal with just `float cost() { return 0.5f; }`.

Inside the class, we're allowed to use a covered-up function from anywhere, not just the function covering it up. In a real program we rarely want to, but we can:

```
class Duck : Animal {
  ...
  public string stats() { return "quack"; }

  public string stats2() {
    // call ours and the one in Animal:
    string w1 = stats() + base.stats();
    return w1;
  }
}
```

I think it's easy to see which is which in `stats()+base.stats()`.

One cool error, if you try to call the function you're covering-up, but forget the `base`-dot, you get an infinite recursive loop:

```
  // oops!! Meant to call base.stats(). Runs forever:
  public string stats() { return stats() + " mieo"; }
```

Constructors and base constructors

Constructors don't cover each other up, but they use the `base` rule in mostly the same way. I'd normally leave this as a detail you can look-up later, but it's a nice example of using `base` to reach back into your base class:

```
class Animal {
  ...
  public Animal() { age=2; } // constructor
}

class Cat : Animal {
```

```
  ...
  // constructor:
  public Cat() : base() { cuteness=5; } // also sets age to 2
}
```

The syntax is a little funny. It's just colon-`base` in that spot. It looks like the inheritance rule, but they're re-using the colon to mean "run the base constructor first." You don't have to use it, but you can.

37.6 dynamic dispatch

Suppose we have an Animal pointer aimed at a Cat and call `a1.cost();`. It's not completely clear if we should call the `cost` function in Animal, or the one hiding it in Cat:

```
Cat c1 = new Cat();
Animal a1 = c1;

a1.cost(); // run the one in Animal, or in Cat?
```

The question is, should we key off of the pointer type, or the real type? Both ways make some sense.

Keying off the pointer type follows the rules. `a1` is an Animal, so can only run Animal functions. It can't even see the `cost` function in Cat.

The problem is, doing it that way gives us the wrong price for the Cat. We purposely covered up the Animal cost function with a better one for Cats. The entire point is to always use it.

Some languages only use the good rule: functions are always based on the real type: if you're a Cat, you always get the Cat `cost` function, even if you call it using an Animal pointer.

The C# family of languages let you chose. You decide when you write the class. If you do nothing special, you get the bad rule, to key off the pointer type. To follows the real type, add two keywords, `virtual` and `override` (details later):

```
class Animal {
  ...
  virtual public string stats() { return "animal"; }
}
class Cat : Animal {
  ...
  override public string stats() { return "cat"; }
}
```

Now, when we run `a1.stats()`, it checks the real thing `a1` points to. It runs the Animal version for real animals, and the Cat version for real cats. If we left out those two words, then `a1.stats()` would always call the Animal `stats` function.

Here's an actual example showing all these rules working together. It takes another array of Animals, which we know will be an array of Cats, or Dogs, or possibly mixed Cats and Dogs; and finds the total cost:

```
float totalCost(Animal[] A) {
  float total=0;
  for(int i=0; i<A.Length; i++) {
    total+=A[i].cost();
  }
  return total;
}
```

It only works if we write `cost` using `virtual` and `override`. Then `A[i].cost()` would compute the correct cost for that Dog or Cat.

If we have several covered-up functions, we have to put `virtual` in front of every one of them in the base class, and `override` in front of every covering-up subclass function. This is a pain. Here's an example where `stats` and `cost` are both virtual:

```
class Animal {
  ...
  virtual public string stats() { return "animal"; }
  virtual public float cost() { return wt*5+age*2; }
}
class Dog : Animal {
  ...
  override public string stats() { return "dog"; }
}
class Cat : Animal {
  ...
  override public string stats() { return "cat"; }
  override public float cost() { return base.cost()*0.75f; } // cats on sale
}
```

Notice how `Dog` didn't write its own `cost` function. As usual, that's fine. If you don't cover a function up, this new rule doesn't even matter. Dogs want to use the basic `cost` in Animal, which is simple and not a problem.

Also notice how the Cat `cost` is still using the one in Animal as a helper. That's still perfectly fine, and works like it did before.

Chapter 38

More Inheritance

The last section was all the basic ideas about inheritance and polymorphism. This section has a few examples, reasoning, and little rules to tweak things.

38.1 A story about the terms

The particular rules and terms C# uses are strange, and probably not worth knowing at first. But if you want to know why things are the way they are, which you don't need to, here's a story:

In the old days we didn't have built-in inheritance. But we had the idea of it, and hand-hacked it into our programs. You made one class, with an `int` for which type it counted as. Then you jammed the class full of every variable for every type of Animal. Then you used `if`s to call member functions for the particular animal you thought you were. Like this:

```
class Animal {
  public int aType; // 0=cat, 1=dog, otherwise generic animal

  // common stats, same as normal:
  public string name;
  public int age;

  // stats for every animal:
  public int cuteness; // only for cats
  public string chewToy; // only dogs

  public float cost() {
    if(aType==0) return catCost();
    else if(aType==1) return dogCost();
    else return animalCost();
  }
```

```
  // add cost function for each animal here:
}
```

Inheritance is just a way to build this trick into the language. Back then, it made total sense to say "you know all of those `if`s you write to call a function based on the real animal type you are? We're going to call that a virtual function and have it be automatic."

That's why call-by-pointer-type is the basic way, because if you know how things really work inside the computer, it is the simplest way. To the computer, that's just a standard function call. Call-by-real type (virtual functions) really are extra work for the computer. It made sense to have the extra `virtual`s and `override`s there to remind us of that.

That's the old detail-oriented way of doing things, giving you every option and making you think about how the computer sees things. C++ does that.

But the point of a computer language is to hide and simplify things you don't need to know. For example, Java (a newer language) gets rid of all those words. Everything is virtual, but you don't have to type anything extra, or even know that word. When you cover up one function with another, it just automatically works the right way.

They could have added a special word to put before functions to mean non-virtual. But they didn't even do that - they figured no one would ever use it. It would just clutter things up trying to explain it.

C# is based on Java and C++. They decided to start with the C++ way of inheritance, because, well, they just did.

38.2 Misc examples

There are several different tricks inheritance can let us do. Unity has some examples using a mix of tricks:

Unity's `Component` is a base class used to cram different things into an array. If you inherit from it, you don't get much except "I count as a Component." This is pretty much what `GetComponent<Rigidbody>()` does to find one:

```
Rigidbody getRBfromArray(Component[] C) {
  for(int i=0;i<C.Length;i++) {
    Rigidbody rb = C[i] as Rigidbody; // dynamic casting test
    if(rb!=null) return rb;
  }
  return null;
}
```

It's also like a label to help you. When you see `class Collider : Component` it's telling you that Colliders, because they're components, go on gameObjects.

Unity uses the `MaskableGraphic` base class as a way to double-up on two real classes that have a lot in common. The two ways to put a 2D image on the screen are `Image` and `RawImage`. They both have a color and material, so we made `MaskableGraphic` for them to inherit from.

Then we can do tricks using what they share:

```
public MaskableGraphic catPicture; // drag in Image or RawImage

  catPicture.color=Color.White; // works if we're an Image or RawImage

// works on Image or RawImage:
void turnRed(MaskableGraphic mg, Color cc) { mg.Color=cc; }
```

The `Monobehaviour` class is for a few things. It inherits from `Component`, so it's using that trick to put your scripts in a gameObject (it goes in the Component list, along with everything else.) It also acts as a label – Unity can loop over the component list using `C[i] as Monobehaviour` to grab all your scripts.

It's a way to give all your scripts a free `transform` and `gameObject` link, which is like giving all of your Animals a name and age.

And then, Start, Update and OnCollision act like virtual functions. They obviously aren't (since they don't have `public override` in front,) but they use another trick that acts that way.

A non-Unity semi-practical example. I want to make a list of Buttons, labels, drop downs and other things which can mostly be clicked. I'll make a base class `menuItem` which has a do-nothing virtual `click` function. Each subclass overrides `click` (or just keeps the base do-nothing click, if you can't click it):

Again, I'm just listing the important parts:

```
class MenuItem {
  public GameObject myGO;
  public virtual void click() {} // override this if your click does something
}

class Label : MenuItem {
  // labels are fine with doing nothing on clicks
}

class Toggle : MenuItem {
  // toggle make clicks flip between true and false:
  public bool value;
  override void click() { value = !value; }
}

class CycleButton : MenuItem {
```

```
  // clicks cycle 0,1,2, 0,1,2:
  int v, max;
  public override void click() { v++; if(v>max) v=0; }
}
```

`MenuItem` as a base class lets us use a `MenuItem[] M;` array for any of these. It also gives a common `myGO` variable. And, of course, `M[i].click()` depends on `click` being a virtual function.

38.3 Cutesy extra rules

Like everything else, inheritance is just a few real ideas (polymorphism and virtual functions) and then a lot of little tweaks and options. There's nothing wrong with these, but they're just a distraction until you know the basics.

38.3.1 Privacy

If you recall, `private` just is a way to help organize a big program. It's a nice way to hide variables from outsiders who shouldn't be using them anyway. So far, the one privacy rule is about what people outside the class can see.

With inheritance, we can make another privacy rule: when you inherit from a class, can you see its variables? We'll grow the privacy rules just a little for that option:

Here are the expended rules:

- `private` means only you. Things that inherit from you can't see your private stuff. They have them, but they can't use them.
- `public` still means everyone can see it – completely outside the class, or other classes that inherit from it.
- A new keyword `protected` means "public if you inherit, private to everyone else." The same as private, it can be used on variables or functions.

Just like the regular `public` rules, these do nothing. The only reason to ever use `protected` is if you think "I sure wish this Animal variable would be in the pop-up for Cat functions, but not for outside." If you never think that, never use `protected` and you'll be fine.

38.3.2 abstract base class

In the Animal, Cat, Dog example, I never use `new Animal()`. I could have, but it wouldn't make any sense to create just a generic Animal. A lot of inheritance is like that, and it's not a problem. But it wouldn't hurt to have an official word for it.

`abstract` in front of the class means you can never use `new` to create one. It does nothing useful except cause errors if you do. It doesn't prevent you from

declaring those variables. `Animal a1=new Cat();` is still legal.

An example:

```
abstract class Animal { ... }

class Cat : Animal { ... } // no change in Cat

Animal a1 = new Cat(); // legal
a1 = new Animal(); // ERROR
```

38.3.3 Inheritance chain, Multiple Inheritance

You're allowed to inherit from a class that inherits from something else. You don't have to list them all – you automatically get everything inherited by the one you inherit. FlyingCats has everything from Cats and Animals:

```
class Cat : Animal { ... }

class FlyingCat : Cat {
  public float airSpeed;
}
```

As usual, everything a FlyingCat inherits looks the same, like `fc1.name`, even though `name` came from way down in Animal.

To use virtual functions, you still put `virtual` in the top one, and `override` in all the ones covering it up. `protected` is unchanged – FlyingCats can see `protected` variables all the way down in Animal.

The other way to inherit from more than one class is to directly inherit from both, which is called multiple inheritance. You list them with commas (the order doesn't matter):

```
class SaleItem { public float price; public int numForSale; }

class Cat : Animal, SaleItem { ... }
class Eagle : Animal { ... } // eagles are not for sale
class Car : Vehicle, SaleItem { ... }
```

This is more flexible than an inheritance chain since we can mix and match. But C# doesn't allow it (trying to is an error.)

38.3.4 Misc

C# has two class keywords you can ignore: `sealed` and `new` (which is a totally different use than `new Cat()`.)

`sealed` adds a restriction that no one can inherit from your class. Like `private`, it doesn't do anything – just makes errors.

`new` is optional. Whenever you could put `override` but don't want to, you can put `new`. It means call-by-pointer-type, which is the one you don't want, and what happens if you just leave it blank.

38.4 Interfaces

Suppose we wrote a base class that had only do-nothing virtual functions, for example:

```
class public Breakable {
  public virtual void use(int times) {}
  public virtual bool isBroken() {return false; }
}
```

The idea is that `use(1)` has a chance to break the item, and `isBroken` is how you can check. But this class gives zero help in how they would work.

We can inherit from it, but we have to write everything from scratch. Here's a sample Hammer and Pen which break in different ways:

```
class Hammer : Breakable {
  // Hammers have a 2% chance to break each use:
  bool broke=false;
  public override use(int times) {
    for(int i=0;i<times;i++) if(Random.value<0.02f) broke=true;
  }
  public override isBroken() {  return broke; }
}

class Pen : Breakable {
  // Pens have 20 uses:
  int usesLeft=20;
  public override use(int times) { usesLeft-=times; }
  public override isBroken() { return usesLeft<=0; }
}
```

That useless base class still lets us use polymorphism tricks. This somewhat silly function will stress-test a Hammer or Pen, since it takes a Breakable:

```
int usesUntilBroken(Breakable b) {
  int timesUsed=0;
  // use until it breaks, or 1000 times (since some things never break):
  while(b.isBroken()==false && timesUsed<1000) {
    b.use(1); timesUsed++;
```

```
  }
  return timesUsed;
}
```

We can improve it just a little by making `Breakable` be `abstract`. There's an extra rule for abstract classes: putting `abstract` in front of a function means you can skip the body, everyone else has to write it, and it counts as virtual:

```
abstract class Breakable {
  public abstract void use(int times);
  public abstract bool isBroken();
}
```

That's really the purest form of the idea. If you inherit from this, you get nothing and are forced to write those functions. In return, you gain the ability for `Breakable b1;` to point to you and use `b1.use(1)` and `b1.isBroken()`.

This concept, just a list of virtual functions you will have, is often called an *interface*.

It's semi-common to inherit from one class for real (you get variables and useful functions,) then tack on an interface. For example, here we use regular inheritance to make Car and Shredder from Machine, and Bird and TurtleElder from Animal, tacking on the Movable interface where needed (this is almost legal C#):

```
// using as an interface:
abstract class Movable {
  public abstract void setTarget(Vector3 pos);
  public abstract void move();
  ...
}

class Car : Machine , Movable { ... }
class Shredder : Machine { ... }
class Bird : Animal, Movable { ... }
class TurtleElder : Animal { ... }
```

Notice how `Movable` can be added to one Machine, but not the other, and then also added to just one Animal out of the two. It's like we're taking a fresh look at all four, not caring if or what base class they used, and just adding Movable to the ones we'll want to move.

Of course, we get no help making them move. We have to hand-write `move` and `setTarget` for Car and Bird to make them really be Movables. But then we can aim `Movable m1;` at either, and call `m1.move();` on Cars and Birds.

Official Interfaces

Traditionally, interfaces are just a way of thinking – if you make a class with only do-nothing virtual functions, we say you're using the interface idea. But C# has an official thing called `interface`. On the one hand, this is just a detail, but I think it also shows the interface idea, so is worth seeing.

Here's Breakable rewritten as an official C# interface. Both versions are the same:

```
// old way:
abstract class Breakable {
  public abstract void use(int times);
  public abstract bool isBroken();
}

// new way:
interface Breakable {
  void use(int times);
  bool isBroken();
}
```

Since you told the computer it was an `interface`, the computer adds the rest (for example, the functions always count as public.)

You "inherit" from these the usual way, and are required to write the functions. You can even declare variables for interfaces, like `Breakable b1;`. That seems funny, but it's the same idea as abstract classes. `Breakable b1;` is for pointing to things that count as Breakables.

The exact C# rules are messy, but you can look up examples pretty easily. For example, you sometimes have to use the interface name as a namespace. Here I'll write `use` that way, but `isBroken` the normal way:

```
class Hammer : Tool , Breakable {
  ...
  int usesLeft=20;
  void Breakable.use() { usesLeft--; }
  public bool isBroken() { return usedLeft<=0; }
 }
```

You really do leave out the `public` the first way, but have to write it the second way; and you can't write `override` but it counts that way. It's a mess. But the error messages are pretty clear, and there's only one way it works. As long as you understand the "I will write these virtual functions, then will count as your type" idea, you'll get it right after a few tries.

Built-in interfaces

We already know how to sort an array using function passing. If you remember, if looks like `Sort(C, youngerCatCompFunc);`, where the second input is a 2-item compare function.

But C# has another way to sort using interfaces. It's not as good, but it makes a nice example.

The first two parts are built-in to C#. There's an interface holding a compare function. It looks funny - the `T` stands for whatever type we are, like a Cat. It only has one input, but it really has two since it's a member function (more about that later):

```
interface IComparable<T> {
  int CompareTo(T t);
}
```

It should return -1/0/+1, which is standard compare function use.

The built-in Sort takes an array of these, and uses the `CompareTo` function to figure out the order. Here's the part we care about:

```
void Sort(IComparable<T>[] A) {
  ...
    // Compare A[i] and A[j]:
    if(A[i].CompareTo(A[j])==1) // out of order
  ...
}
```

The trick is we'll really being giving it an array of Cats or Dogs, or anything that counts as `IComparable` (they don't, yet, but we can make them.) Sort doesn't need to care about exactly what they are, all that matters is they have `CompareTo` written.

Comparing using a member function looks funny - we'd rather use a regular 2-input function, like `Compare(A[i],A[j])`. But interfaces only give us member functions.

We can use those built-ins to sort an array of Cats. Write the `CompareTo` function in Cat, and put `IComparable` in your inherit list. This makes a Cat sortable by age (figuring out the extra Cat in angle-brackets and the other stuff took some trial&error):

```
class Cat : Animal, IComparable<Cat> { // <= added IComparable
  ...
  int System.IComparable<Cat> CompareTo(Cat c2) {
    if(age<c2.age) return -1;
    if(age==c2.age) return 0;
```

```
    return +1;
  }
}
```

Now `System.Array.Sort(Cats);` is legal, and sorts our Cat array. Notice how there isn't an option how to sort. It always uses the one we wrote in the class, which compares by age.

This is a semi-common trick. But if you hate this example, sorting by passing in a compare function is better anyway.

38.5 General inheritance advice

Try to use inheritance for a specific purpose. The main reason for it is when you need a pointer that can go to either, or an array that can hold either.

Suppose you have health pickups, ammo pickups and so on. If you write a simple `Pickup` base class, you can do things like:

```
public Pickup nearestPickup; // health pack or ammo

public Pickup[] AllPickups; // all health and ammo, together
```

Then you might naturally move variables like `timeOut` into the base class. That lets you use `if(nearestPickup.timeout<5)` easily without having to check what kind it is.

A secondary reason for inheritance is you just notice two classes have a lot in common. This was my first Cat/Dog example. Don't think too hard about this type, or pretty soon you'll be making Animal, Mammel, Feline classes that don't help you much.

If several classes have a lot in common, it makes sense to put all that into a common class. Then inherit it, or just declare `public Animal A;` in your class. If you aren't using polymorphism tricks, it won't really matter.

For very simple stuff, a type field is fine. In the Cat/Dog example, suppose we want to distinguish a Cat from a Dog, but we don't need cuteness or barkVolume. In that case we wouldn't need sub-classes. Just the Animal class with `public int animalType;` would be fine (or make an `enum`.)

You can have a base class and be a MonoBehaviour. In the Health pickup example, you may have started with `class HealthPickup : MonoBehaviour`. If you want to add `Pickup` as a base class, do this:

```
class Pickup : MonoBehaviour {
  public float timeout;
  ...
```

```
}

class HealthPickup : Pickup {
  ...
}
```

That's just a standard inheritance chain. HealthPickup inherits MonoBehaviour from Pickup, so it can be dragged into a GameObject, and can use `transform` and `Update()`.

Don't assume a subclass has to be "bigger" than the base class. Sometimes you have a class that does more than what you need, and it's easy to inherit from it and add restrictions, sort of.

In this example, I've got a button that flips through options as you click it, like 0-4 or 0-15. I can re-use it for a simple 0/1 toggle:

```
class MultiButton {
  public int val, maxVal; // value goes from 0 to maxVal
  public override void click() { val++; if(val>maxVal) val=0;
}

class Toggle : MultiButton {
  public MultiButton() { val=0; maxVal=1; }
}
```

`Toggle` is really just a shortcut for making a MultiButton and setting maxVal to 1. But there's nothing wrong with good shortcuts.

Lots of `a1 as Cats` mean you might want virtual functions. The job of a virtual function is to automatically figure out which class you are and do the right thing. So if you can, let it make your job easier.

An example. This hand-checks the real type and decides what to do:

```
HealthPickup h = i1 as HealthPickup;
if(h!=null) gainHealth(h.healAmount);
AmmoPickup a = i1 as AmmoPickup;
if(a!=null) addAmmo(a.ammoType, a.bullets);
 ...
```

You could rewrite that as a virtual `applyPickup` function:

```
abstract class Pickup {
  public virtual void applyPickup(Player p) {}
  ...
}

class HealthPickup : Pickup {
```

```
public int healAmount;
public override void applyPickup(player p) { p.gainHealth(healAmount); }
...
```

Then you can call `i1.applyPickup(thePlayer)` and have it automatically check which type it was and do the right thing.

In theory, besides being shorter code, it's also nicer to have all the healthPack stuff in the HealthPack script where you can find it.

Be a little careful of inheritance fads. Inheritance used to be really hot, as in you got a promotion if you used it enough, even if you didn't need to. You can find inheritance stuff on-line that seems to just be about turning one class into three. Tthere's a good chance the goal really is just to have more classes.

Chapter 39

Linked Lists

A linked list is an alternative to an array. It's a list of items where each is "linked" to the next with a pointer. Sometimes it's a better way to store things than an array. It's a basic data structure that all programmers learn.

But mostly it's a good exercise. Playing with linked lists is a traditional way to really get good with pointers. I've also got some fun functions, a new type of loop, and using templates (generics.)

39.1 Simple linked list

An array makes a single chunk of memory with each item next to the other. We can find `A[5]` because it's exactly 5 ints past `A[0]`. We can just jump to it using math.

A linked list doesn't bother putting them in order. The items are scattered around, with each one pointing to the next. We call the item plus the pointer a **node** (but you know it's just a small class.) Linked lists aren't built-in the way arrays are. They're just an idea that we have to make ourselves.

A linked list named `L1` holding 2, 7, 4, 9 could look like this. The lines stand for pointers:

```
        9-->null
         \
L1--> 2   4
       \ /
        7
```

I moved them around to emphasize how we need to use the pointers to find which item is next.

Each node has two things – the value and the pointer – so we have to make a class:

```
class IntNode {
  public int val;
  public IntNode next; // pointer to next one, or null for last
}
```

Our picture written out as IntNodes would look like this. Each number is a created IntNode, and our link to it is `IntNode L1;`, which is aimed at the 2:

```
     --------       --------       -------       -------
L1->|val next|  /->|val next|  /->|val next|  /->|val next|  /->null
    | 2    o-|-/   | 7    o-|-/   | 4    o-|-/   | 9    o-|-/
     --------       --------       --------       --------
```

If you look at the second field, `public IntNode next;`, you might think "wait – we have an IntNode inside of another IntNode?" No – the picture is the right way to think of it.

If you remember from the pointer chapter, a quirk of C# (and Java) is there are two ideas of pointer variables. The "normal" way is we own it, it's inside of us, and we have to create it with `new`. The other is a real pointer: we don't create it, we just use it to point to what someone else made.

`next` is a real pointer. Each IntNode "knows about" the one after it, using `next`.

39.1.1 Linked List loops

I'm going to save how we make these for later, since using them is a better way to understand them.

A standard linked list loop starts a pointer on the first item (called the *head*) and follows it one node at a time until we go off the edge with `null`.

This loop prints the list. For example `printList(L1);`:

```
void printList(IntNode head) {
  IntNode p = head; // loop variable. Aim at 1st item
  while(p!=null) { // not past the end
    print(p.val);
    p=p.next; // go to next node
  }
}
```

We can rewrite it as a `for` loop:

```
for(IntNode p=head; p!=null; p=p.next)
  print(p.val);
```

This isn't even abusive – it's a perfect use of a `for`. In the parens we can clearly see `p` is the loop variable, then where it starts, when it stops and how it

moves. Down below we can focus on only what we do as `p` hits each node.

Here's a quick review of pointers and how they work in that loop. Nothing new:

- Remember that pointers don't "reach through" other pointers. `p=head;` makes `p` point to the first item. When we change `p` later, it won't affect `head`.
- Likewise, `p=p.next;` simply makes `p` move down one link in the chain. In this picture, `p` is the node with 7 and `p.next` is the node with 4 (it's really the arrow from 7 to 4, but what matters is it's pointing to the 4):

  ```
  L1--> 2--> 7--> 4--> 9-->null
             A
             |
             p
  ```

  After `p=p.next;`, `p` is aimed at the node with 4. It just slides down one node.
- `p` will eventually go past the last node and be `null`. That's fine. The loop checks for `null`, which is what you're supposed to do.

  If you think about it, regular array loops also go past the end, so this is the same.

We can use a pointer loop to count how many items are in the list. It's not much more complicated than printing them:

```
int getLength(IntNode head) {
  int len=0;
  for(IntNode p=head; p!=null; p=p.next) len++;
  return len;
}
```

As a quick check, suppose `L1` is `null`. When we call `myLen=getLength(L1);` then `p` starts as null, the loop won't run, and it returns 0, which is correct.

Another simple pointer loop is finding the largest item. As usual, I'll start out with the 1st as the largest, then have the loop start at the second:

```
int largest(IntNode head) {
  if(head==null) return -9999; // list is empty
  int largest=head.val; // 1st item is largest, for now
  for(IntNode p=head.next; p!=null; p=p.next)
    if(p.val>largest) largest=p.val;
  return largest;
}
```

The most interesting thing is how much it's like the array version of largest (once you get used to the different loop.)

Checking whether something is in the list is another simple pointer loop. For fun I'll use the semi-slimy shortcut where we reuse the input as the loop variable (that's why the 1st part of the `for` is blank):

```
bool isIn(IntNode head, int findMe) {
  for(;head!=null; head=head.next)
    if(head.val==findMe) return true;
  return false;
}
```

If this was an array, we'd rather find the position of the item, or -1 if it's not there. For a linked list we'd rather find the node with that item (or `null` if it's not there):

```
IntNode findNodeWithVal(IntNode head, int findMe) {
  for(IntNode p=head; p!=null; p=p.next)
    if(p.val==findMe) return p;
  return null;
}
```

If you don't trust the return-from-middle trick we can rewrite (this is just playing with loop conditions, which is always fun):

```
IntNode findNodeWithVal(IntNode head, int findMe) {
  for(IntNode p=head; p!=null && p.val!=findMel; p=p.next) {}
  return p; // null == not found
}
```

The other way to search is for a certain position. In arrays this is so easy (`A[4]`) that we don't even think about it. In a linked list we have to count that far from the start.

It's a fun loop, and a little tricky – we're still moving `p` but we're also counting. We don't want to make `p` be null, since it means the position was past the end, but we have to check just in case:

```
IntNode getNodeAtIndex(IntNode head, int wantIndex) {
  int i=0;
  IntNode p=head;
  while(i<wantIndex && p!=null) { p=p.next; i++; }
  return p;
  // NOTE: if index was too big, this returns null, which is the right answer
}
```

We'd use it like `IntNode pp = getNodeAtIndex(L1,4);`. To get the value, we'd use `pp.val`.

This next one is very sneaky – it uses the returned node. I want to count how many 7's are in a list. The plan is to use `findNodeWithVal` to find the first 7, count it, go to the next node, use `findNodeWithVal` starting from there to find the next 7 ... until we run out of list:

```
// count how many 7's there are in L1:
int count=0;
IntNode p=L1; // start looking at start of the list
while(p!=null) {
  p=findNodeWithVal(p, 7); // moves p to next 7 node, or null
  if(p!=null) { count++; p=p.next; }
}
```

We've seen the same idea in an array loop, except we keep searching and increasing `i`.

Fun fact: if we forgot `p=p.next;` after finding a 7, this would be an infinite loop, finding the same 7 over and over.

Another fun fact about all linked list loops: suppose the last thing in the list didn't point to `null`. Instead it pointed back to some previous node (the first one, somewhere in the middle, or even to itself.) Then loops over it would run forever, thinking the list was infinite.

39.1.2 Insert/Remove

The advantage of a linked list is we can easily insert an item into any position. We just splice it in by changing a few pointers. For example, here's a picture adding 55 to the start of our old list:

```
L1  2--> 7--> 4--> 9-->null
  \ |
   55
```

If it bothers you that memory is really numbered and is more like a line, here's a more memory-accurate picture of how splicing in 55 might look:

```
   --------------------
  /                    \
L1  2--> 7--> 4--> 9<>  55
     \                 /
      -----------------
```

To do this in code we need to create a new node, point it to the old first item, then make the head point to it:

```
void insertToFront(ref IntNode head, int newVal) {
  IntNode nn = new IntNode(); nn.val=newVal;
  nn.next=head; // point at old 1st item
  head=nn; // make this first item
}
```

We'd run this with `insertToFront(ref L1, 55);` (it needs to be passed by reference, since we're changing where `L1` points.)

Having to make a new node should make sense. If we have 4 items, we have 4 nodes. We don't need or want extra nodes hanging around. So if we want a new item, someone has to make the new node for it.

The order of the last two lines is important. If we started with `head=nn;` then we'd lose the list. Another way to write it would be:

```
// alternate hook-up code:
  IntNode oldFirst = head; // saved 1st item
  head=nn; // point to new 1st item...
  nn.next=oldFirst; // ...which points to old 1st item
}
```

One last thing to check – does this work inserting into an empty list? `L1` starts as `null`. This makes the new new point to `null`, so it works.

We can now finally create that 2, 7, 4, 9 list we've been using. Since we're adding to the front, we have to add them in reverse order:

```
IntNode L1=null;
insertToFront(9); // L1-->9
insertToFront(4); // L1-->4--9
insertToFront(7); // L1-->7-->4-->9
insertToFront(2); // L1-->2-->7-->4-->9
```

Inserting in any other position requires us to have the node before it. For example, inserting 15 after the 7 involves changing the `next` pointer from 7:

```
L1--> 2--> 7  4--> 9-->null
            \ |
             15
```

The code is pretty much the same:

```
void insertAfterNode(IntNode p, int newVal) {
  IntNode nn=new IntNode(); nn.val=newVal;
  nn.next=p.next; // new node points to the one after p
  p.next=nn; // p points to us
}
```

It's easy to get confused by the two `p.next`'s. Remember the trick with arrows is if it's on the left side, we only care about the the box where it starts, which we're changing. If it's on the right side, we only care about what it points to and not where it's coming from.

We could insert after the 7 like this:

```
// add a 15 after the 1st 7:
IntNode p = findNodeWithVal(L1, 7);
if(p!=null) insertAfterNode(p, 15);
```

Just for fun, here's a hacky way to add 15 to the end of the list, using our previous functions. Find the length, subtract 1, find that node, and insert after it:

```
int len=getLength(L1);
if(len==0) insertToFront(ref L1, 15);
else {
  IntNode last=getNodeAtIndex(len-1);
  insertAfterNode(last, 15);
}
```

Also just for fun, we could write a function that adds to the end a little nicer:

```
void insertAtEnd(ref IntNode head, int newVal) {
  IntNode nn=new IntNode(); nn.val=newVal;
  nn.next=null; // since it's the last item

  if(head==null) { head=nn; return; }
  // loop stops where we're _about_ to go off the edge:
  IntNode p=head;
  while(p.next!=null) p=p.next;
  p.next=nn;
}
```

Using `while(p.next!=null)` is a common trick to look ahead, stopping just before we go off the end. Another common way is keeping the normal loop that has `p` fall of the end, but adding an extra trailing variable:

```
// find last item using a trailer:
IntNode p=head, pPrev=null;
while(p!=null) {
  pPrev=p; // old value of p, just before moving it
  p=p.next;
}
// now p is off edge and pPrev is last node
```

This might look a little better, or not.

39.1.3 Removing

Removing an item is like inserting it. We just reroute the pointer around the item to remove. Like inserting, the first item is a special case, since we have to change `L1`.

Here's what it looks like to remove the first item of of 2, 7, 4, 9:

```
L1    2--> 7--> 4--> 9-->null
  \       /
   -------
```

It looks odd that 2 still points to 7. We could set it to `null`, but it won't matter. The key things are the list now starts with 7, then 4 then 9; and there's no way to get to 2 anymore. Eventually the garbage collector will remove it.

The code looks like this:

```
void removeFirst(ref IntNode head) {
  if(head==null) return; // there isn't a 1st item
  head=head.next;
}
```

As a check: if there's only 1 item, this would make `head` point to `null`, which is correct.

Removing something that's not the first item is just odd. We need the node *in front* of the one to remove. That's not very good (we'll design a better linked list to fix it, later.) Here's how it looks:

```
void removeAfter(IntNode beforeNode) {
  if(beforeNode.next==null) return; // nothing after us to remove
  beforeNode.next = beforeNode.next.next;
}
```

`beforeNode.next.next` means to follow the arrows twice. `beforeNode.next` is the one after us, so `(beforeNode.next).next` is two steps past (I added the parens, but it works the same either way.)

This last one is long and confusing, but it shows off pointers pretty well. We want to remove all negative numbers from a list. The loop will check whether the *next* number is negative. As usual, the first is a special case:

```
void removeNegatives(ref IntNode head) {
  if(head==null) return; // no items in list
  // removing 1st is special case, since we have to change head:
  // also, there might be several negatives at the start, so we need a loop!!:
  while(head.val<0) {
    head=head.next;
```

```
    if(head==null) return; // whole list was negative numbers
  }
  // check non-first items, looking 1 ahead:
  IntNode p=head;
  while(p.next!=null) { // while not last item
    if(p.next.val<0) p.next=p.next.next; // cut out next item
    else p=p.next; // move normally to next item
  }
}
```

This mess has all the linked list badness, and the usual deleting complications (the head is a special case; need to look-ahead one node so we can remove it; don't move forward when you delete something.)

Whenever I have to write something like this, it's always got a few little errors, including infinite loops and exciting null reference exceptions, and needs solid testing to shake them out.

39.2 Misc

For testing, it would be nice to make a linked list with just 0, 1, 2 ... in it. We'll use the trick where we insert them backwards into the front (the same way we made the 2,7,4,9 sample list):

```
IntList makeSeqList(int len) { // list from 0 to len-1
  IntList head=null;
  for(int i=len-1;i>=0;i--) // backwards, since adding to front
    insertToFront(ref head, i);
  return head;
}
```

For testing, it might also be nice to convert an array into a linked list. We can use the same trick – go through it backwards, adding to the front:

```
IntNode arrayToLinkList1() {
  IntNode head=null;
  for(int i=A.Length-1; i>=0; i--)
    insertToFront(ref head, A[i]);
  return head;
}
```

We'd use it like `IntNode L1=arrayToLinkList(new int[]{2,7,4,9})` (that's the C# way to write a constant array. Or we could just use `A`.)

Just for fun, what if we didn't solve this by thinking "adding to the front is easy" and working backwards from there? There are some other ways we could brute-force our way into this.

We might start with a regular array loop, adding to the end:

```
IntNode arrayToLinkList2() {
  IntNode head=null;
  for(int i=0; i<A.Length; i++)
    insertAtEnd(ref head, A[i]); // gah -- this is a loop
  return head;
}
```

We could rewrite that using an extra `tail` pointer to keep track of the end (we always call the last item in the list the tail.) That would get rid of the extra loops, but make it much uglier and harder to debug.

Another array-like way is we might imagine copying one array into another. First we'll make an "empty" linked list of the correct size. We'll use add-to-front, so can't put numbers in it yet. Then we'll loop through both together:

```
IntNode arrayToLinkList3(int[] A) {
  IntList head=null;
  // make enough "empty" nodes:
  for(int i=0;i<A.Length;i++)
    insertToFront(ref head, -1); // dummy value
  // now copy values:
  IntNode p=head; int i=0; // march these side-by-side
  while(i<A.Length) {
    p.val=A[i];
    i++; p=p.next; // 1 step ahead in both
  }
}
```

Reversing a linked list is another one that has a nice way if we think about linked lists, and some clumsy ways if we try to copy the array way of thinking.

Reversing an array looks like this:

```
// swap items in 1st and second half:
for(int i=0;i<A.Length/2;i++) // swap A[i] and A[A.Length-1-i]
```

We can brute force loop this to work with a linked list, but each function call is a loop, so this runs slow:

```
void reverse1(IntNode head) {
  int len=getLength(head);
  // swap 1st half with back half:
  for(int i=0;i<len/2;i++) {
    IntNode n1=getNodeAtIndex(head,i);
    IntNode n2=getNodeAtIndex(head,len-1-i);
    // standard swap:
    int temp=n1.val; n1.val=n2.val; n2.val=temp;
  }
}
```

The nicer version rearranges the nodes, something that requires a linked-list way of thinking. We go through the list, remove each node, and add it to the front of a new list (which will be reversed):

```
void reverse(ref IntNode head) { // head will change
  IntNode R=null; // temp holder for reverse list
  IntNode p=head; // loop variable
  while(p!=null) {
    IntNode savedNext=p.next;
    p.next=R; R=p; // insert p into the front of R
    p=savedNext;
  }
  head=R;
}
```

Notice how I couldn't use `insertToFront`. That takes a number and creates a node for it. In our case, we already have the node. We only want to change some pointers. The loop also had to save `p.next` at the top, before changing it.

If you recall, some functions change us, and some return changed copies of us (for example, `w.ToLower()`.) If we wanted a reversed *copy* we could do this:

```
IntNode makeReversedCopy(IntNode head) {
  IntNode A=null;
  for(IntNode p=head; p!=null; p=p.next)
    insertToFront(ref A, p.val);
  return A;
}
```

Another way to reverse a list starts with a linked-list-y idea, but isn't quite as nice when we write it. What if we flip every arrow in the picture? Make every node point to the one in before it, instead of after it:

```
void reverse3(ref IntNode head) {
  IntNode p=head; // loop variable
  IntNode prev=null; // trails behind p
  while(p!=null) {
    IntNode savedNext=p.next; // save next node
    p.next=prev; // switch p to point to the node behind it
    // now move the loop ahead a node:
    prev=p;
    p=savedNext;
  }
  head=prev; // when p is off end, prev is last node
}
```

This ends up being the same as the insert-to-front way, except `R` is replaced by `prev`, and we're thinking of it differently.

We can be more extreme with tearing apart a linked list. For example, we can split the list into even/odd:

```
IntNode odds=null, evens=null;
IntNode p=L1;
for(p!=null) {
  IntNode savedNext=p.next;
  if(p.val%2==0) { p.next=evens; evens=p; }
  else  { p.next=odds; odds=p; }
  p=savedNext;
}
// frontAdd on evens and odds reversed them, fix it:
reverse2(ref evens); reverse2(ref odds);
```

39.3 Special cases

Linked-list loops are also fun because, as we've seen, they have a lot of special cases: 1st item, last item or only item. I skipped mentioning this for most examples, but for real we have to think about each one.

For examples, adding after a node: it's not possible for our new item to be the first one or only one, but it could be the last. It so happens the regular code works for adding the last item, but we still had to check.

Our look-ahead loop using `while(p.next!null)` will crash on an empty list (since `p` starts as null.) We had to handle that as a special case.

But the trick is to ignore special cases at first. Assume you're in the middle of a long list and write general purpose code to solve it. Then go back and think about each special case. Some will work anyway, some won't and need us to add extra `if`'s.

39.4 Doubly-linked lists

For real, we like to have each node point to the next one and the previous one, like this:

```
class IntNode {
  public int val;
  public IntNode prev; // node in front of us. null==1st node
  public IntNode next;
}
```

Here's the new picture. The top row is `next`, the bottom is `prev`:

```
head-->| |-->| |-->| |-->| |-->| |-->null
       |2|   |7|   |4|   |9|   |3|
null<--| |<--| |<--| |<--| |<--| |<--tail
```

Advantages are we can look through the list backwards, we can find both nodes around us for deleting, we can insert to either side of us, and if we really need to we can make a loop that can walk back and forth.

Obviously we still have to go through one item at a time.

The drawback is we have to change twice as many pointers when we move things around.

The other improvement we usually make is saving the head, tail and length in a class. Then we write all the functions as member functions. The new linked list class:

```
class IntList {
  public IntNode head=null, tail=null;
  public int len=0;
  // insert, remove ... as member functions:
  // remember they get to use head, tail and len for free
}
```

Now instead if having `L1` be a simple `IntNode` pointer, we need `IntList L1 = new IntList();`. We have `L1.head=null;`, pretty much the same as before.

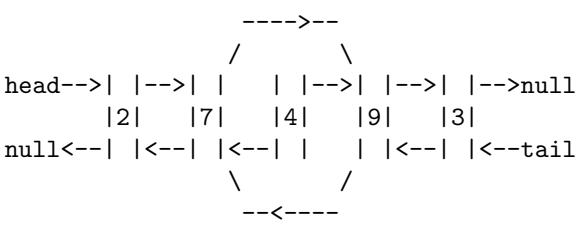
One fun thing we can do with this is search from either end. To find an index, we'll start from the back if it would be quicker:

```
// this is a member function:
public IntNode nodeAtIndex(int index) {
  if(index<0 || index>=len) return null; // off edge
  if(index<len/2) { // 1st half - search from front:
    IntNode p=head;
    for(int i=0;i<index;i++) p=p.next;
    return p;
  }
  else { // search from back:
    IntNode p=tail;
    for(int i=len-1;i>index;i--) p=p.prev;
    return p;
  }
}
```

We could do this before, but this way averages twice as fast. Plus it's fun.

Our removing ability from before was terrible, since we needed to know the previous node. Now if we have a pointer to a node, we can remove it.

Here's a picture how the two arrows change to remove the 4. Notice how the arrows from 4 still don't change, since they don't matter anyway:



If `nn` was pointing to the 4, these are the lines to change the arrow that way:

```
nn.prev.next=nn.next; // the node behind us points to node past us
nn.next.prev=nn.prev; // node past us points to node behind us
```

`nn.prev.next=` can be confusing. It's changing the `next` arrow of `nn.prev`. That's the top arrow in the picture.

The real remove member function needs to worry about removing the first and last, and needs to adjust the `len` variable:

```
public void removeNode(IntNode p) {
  // Adjust node in front of us. Special case if first item:
  if(p!=head) p.prev.next=p.next;
  else head=p.next;
  // Adjust node after us. Special case if last item:
  if(p!=tail) p.next.prev=p.prev;
  else tail=p.prev;
  len--;
}
```

A fun thing to do is trace this out for all the cases: middle, first, last, only node. It happens to work for them all (if it didn't, we'd fix with more `if`'s.)

Adding to the front is the same as before, except we have to update tail and len:

```
public void frontAdd(int newVal) {
  IntNode nn=new IntNode(); nn.val=newVal;
  nn.next=head; nn.prev=null; head=nn;
  // fix next node, or tail:
  if(len>0) nn.next.prev=nn;
  else tail=nn;
  len++;
}
```

Since we have the tail, we can quickly add to the back. It's the same as the front, except backwards, but it's still nice to see:

```
public void backAdd(int newVal) {
  // if list is empty, reuse frontAdd:
  if(len==0) { frontAdd(newVal); return; }
  IntNode nn=new IntNode(); nn.val=newVal;
  nn.prev=tail; nn.next=null; tail=nn;
  nn.prev.next=nn; // previous last node points to us
  len++;
}
```

Let me say again that these get confusing. I always have to draw a picture, recheck the code to be sure what `nn.prev.next` is, and hand-move the arrows.

Before, we could insert something after a node. Now we can choose to insert before or after. Here's insertAfter. We need to change 4 arrows total (the 2 from us and the 2 to us.) Inserting the last is still a special case:

```
public void insertAfter(IntNode p, int newVal) {
  if(p==tail) { backAdd(newVal); return; }
  IntNode nn=new IntNode(); nn.val=newVal;
  nn.next=p.next; nn.prev=p; // our arrows
  p.next=nn;
  nn.next.prev=nn;
  len++;
}
```

We can be really sneaky writing `insertBefore`, by going back a node and reusing insertAfter:

```
public void insertBefore(IntNode p, int newVal) {
  if(p==head) { frontAdd(newVal); return; }
  insertAfter(p.prev, newVal);
}
```

The other things we could write are frontRemove and backRemove. Searching for a particular item is the same as before. Reversing is still front-adding into a new list. Array-to-list is still backwards through the array and using front adds.

There are a few very, very fancy version of linked lists with even more pointers, but you rarely use them and the math is hard. Just a next and prev pointer is a standard grown-up linked list.

39.5 Using templates

This section is just about using templates, and not really about linked lists at all. Except almost every built-in linked list uses templates.

We could make a different linked list node for strings, floats, Cats ..., but that would be a pain. Switching what they hold is why template classes were invented (C# renamed them *generics*.)

Our node type will look like:

```
class Node<T> {
  public T val;
  public Node<T> next, prev;
}
```

Remember the first `Node<T>` defines us as requiring an extra input. It says the real classes will be `Node<int>` or `Node<string>` or even `Node<Cat[]>` (each item in the list is an array of Cats.)

Then the `T's` inside automatically get filled in. If we create a `Node<string> nn = new Node<string>();`, then the next and prev fields are automatically also `Node<string>`'s.

The base list class would be:

```
class LinkList<T> {
  public Node<T> head=null, tail=null;
  int len=0;

  public addFirst(T newVal) {
    Node<T> nn = new Node<T>(); nn.val=newVal;
    nn.next=head; head=nn;
    ...
  }
}
```

We'd use this with `LinkList<string> L1 = new LinkList<string>();`

The inside `T`'s are also filled in with `string`'s. For example, calling `L1.addFirst("cow");` knows to create a new `Node<string>` for our cow.

39.6 Built in Linked List

Most languages have built-in doubly linked lists that work about the same. C#'s is nothing special, but it's interesting to see how they change a few things.

It's obviously a template class. The node type is named `LinkedListNode<T>`. You'll only use this for loops or as the result of a search. The whole type is

`LinkedList<T>`. The member variables are private, but are obviously `head`, `tail` and a length.

The functions to manipulate them are the same ones we've been using, named a little differently:

```
LinkedList<string> L1 = new LinkedList<string>();
L1.AddFirst("bb"); L1.AddFirst("aa");
L1.AddLast("ccc"); // aa, bb, ccc

LinkedListNode<string> nn = L1.Find("bb");
L1.AddBefore(nn,"a2"); // aa, a2, bb, ccc
L1.Remove(nn); // aa, a2, ccc
```

You can't see the head or tail directly, but you can use `L1.First` and `L1.Last` to get them (why not call these head and tail? C# is partly designed for entry-level programmers who aren't used to those terms.)

Since everything is private, you can't directly change the pointers, but, for example, you can still tear a list apart using built-in commands. This pulls all the odd nodes out of `L2`, moving them into another one:

```
LinkedList<int> Odds = new LinkedList<int>();
LinkedListNode<int> p = L2.First; // loop pointer
while(p!=null) {
  LinkedListNode<int> savedNext = p.Next;
  if(p.Value%2==1) {
    L2.Remove(p);
    Odds.AddFirst(p); // overloaded to let you add a node
  }
  p=savedNext;
}
```

39.7 Array implementation of linked lists

This section isn't very useful, but it's more practice for thinking about how linked list work and more playing with array indexes. It is a real thing, but it's only used in very specific circumstances.

The trick is, we pre-make an array with all of nodes we'll ever use. Each `next` pointer will be the index of the next item. The `head` pointer is the index of the first item.

This negates a big feature of linked lists – we won't be able to grow it all we want. But we can still do the other nice stuff: insert and remove from anywhere.

The nodes will look like this:

```
struct Node {
  public string val; // using strings to be less confusing
  public int next; // index in array of next item. 0 to length-1
}
```

Then we'd create an array making all the nodes we think we'll ever need: `Node[] AllNodes = new Node[10];`. In our minds, this is ten unused nodes. For now, suppose `next=-999;` marks it as unused.

We'll start `int L1=-1;`. It holds the 0-9 index of the first node in the list, with -1 for null.

The spots would be probably be used in order, but if we kept adding and removing we might get something like this. The list is boxes 3, 5, 2, 0, 7. Each box has the number of the next one:

```
L1: 3

   0     1     2     3     4     5     6     7     8
 ----- ----- ----- ----- ----- ----- ----- ----- -----
| ddd |     | ccc | aaa |     | bbb |     | eee |     |
|  7  |     |  0  |  5  |     |  2  |     | -1  |     |
 ----- ----- ----- ----- ----- ----- ----- ----- -----
```

A loop to print them out looks a lot like a standard pointer loop. `p` will jump through 3, 5, 2, 0, 7 then -1:

```
for(int p=L1; p!=-1; p=AllNodes[p].next)
  print(AllNodes[p].val);
```

Inserting to the front is similar. Pretend we have a function to get an unused node:

```
int nn = getUnusedIndex(); // 0-9 of a free node
AllNodes[nn].next=L1; L1=nn;
```

Deleting the node after `p` should also look similar:

```
int p2 = AllNodes[p].next;
int p3 = AllNodes[AllNodes[p].next].next; // ugg
AllNode[p].next=p3; // skip past
AllNodes[p2].next=-999; // mark as unused
```

These examples are singly linked, but it works doubly-linked as well.

Chapter 40

Big-O notation

In a previous section about making a faster program I wrote that none of the obvious tricks worked very well. We can mess around rearranging `if`s and equations and, if we turn our program into an incomprehensible bug-ridden mess and are very lucky, maybe we get a x2 speed-up.

Profiling is just more of that same thing. All it does is help you use those little tricks, like telling you which functions get used the most. Or, after you rearrange some `if`'s it will tell you the function is *slower* so you need to keep trying. In a game, it will often tell you the program is plenty fast, but the graphics card is being swamped. You just need to use pictures with a smaller pixel-count.

Profilers are great, especially for code written by a lot of people which you suspect is sloppily assembled. If you're going to do that stuff, go ahead and use one. But it's still checking the couch for spare change compare to the real speed-up trick.

The one thing that gives a huge speed-up is avoiding extra loops. If you have a loop that runs 1,000 times, and you realize you can do the same thing without one, that's about a 1,000 times speed-up.

Avoiding unnecessary loops is the major trick for worrying about speed.

Nothing in this chapter will let you do things you couldn't do before, or make your program look nicer or anything else useful. It just does one thing, which is make it faster. But if you're lucky, it will make it *a lot* faster.

The first section explains that idea more. The second part has examples of using it to classify common array loops; then examples of removing loops by using a better plan. Then a little about how the loops are different for a linked-list.

The section after that introduces the formal way we write big-O notation. It's useful to know because it's the way computer manuals list speeds. Then

there's more examples, data structures and the last section is a summary of how we use this thinking in general program writing.

40.1 Loop counting logic

The math and motivation are long and boring, so I'm going to skip ahead to the end results and what we do about it. I'll go over the explanation in a section at the end, and hopefully the examples will help. Just keep in mind there will be a few places where I'm just saying something is true without really showing why.

To help speed up programs, we classify every function by the worst nested loop it has. We don't worry about how many times the loops run, or how many lines are in them. We list them as only: not a loop, single loop, nested loop, triple loop and so on.

Just to make sure, here's a sample triple loop, using 3 arrays:

```
for(int i=0;i<A.Length;i++)
  for(int j=0;j<B.Length;j++) {
    count1++;
    for(int k=0;k<C.Length;k++)
      count2++;
    }
```

If all the arrays are about 1,000 long, `count1` will be a million and `count2` will be a billion. Each nesting is 1,000 times more steps.

That's why we simplify so much. In general, any single loop is much worse than any non-loop, any nested loop is much worse than any single loop, and so on.

After we label a function with the nested level, we try to reduce it. We don't waste time trying to make the loop run fewer times, or do less work each time. All we do is try to think of a plan that uses less nesting. That's where we get the big x100 speed-ups.

If we have a single loop, we try to think of a way to not use a loop. If we have a nested loop, we try to think of a way to do the same thing with only a single loop. And so on.

Again, I'm not really explaining the math or the thinking, but here are some comments why this plan sort of makes sense:

- This system says that non-loops are the simplest thing and all count the same, as one step. That's pretty much true. Compared to a big loop, the most complicated math and `if`s is insignificant. And we can't get a good x100 speed up for them, since they don't have any loops to remove.
- If you speed up a single loop by running it fewer times, or having less lines inside, this system says it's still a single loop, so those speed-ups

didn't matter. That's also pretty much true. Those are just x2 speed ups. Removing the loop would get you x100.

- A nested loop might run quickly on something small, and not be a problem at all. But sizes tend to grow and it will be. For example, a nested loop on 12 cats might be only 144 steps. But as the pet store grows we'll have 100 cats and the loop will take 10,000 steps.
- After we've eliminated extra loops, the classifications give a good estimate of speed. If a function we want is labelled as a nested loop, yikes! If we plan to put a loop around it we may want to think of a better way.
- A function with two loops in a row doesn't seem to fit in our system, but it counts as a single loop. It's just twice as slow as one loop, which doesn't matter. We have to eliminate both to make it a non-loop (getting rid of just one is like a useless x2 speed-up.)
- Sometimes we have fixed, tiny loops, and they obviously don't count. For example, a 9 step nested loop to set up a Tic-Tac-Tow board.

  It's usually pretty obvious. We have either very small loops that will never grow, or loops running 100+ times which will eventually get larger.

40.2 Integer array loops

The standard loops (or non-loops) on int arrays make good examples of how to apply the classifications.

Finding an item at a certain position in a list is a one step non-loop. We knew that, but I wanted to point out we don't always have to loop to do things with an array.

The first item is one step, `A[0]`, but finding the middle and last are three: `A[A.Length/2]` and `A[A.Length-1]`. We don't even care. It falls under the "basically 1 step" non-loop rule.

Counting how many times something occurs in an array is a single loop:

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

A funny thing is it takes longer depending how many there are (`count++;` takes a step.) We can ignore whether the inside runs 1 or 2 steps and just count it as a loop.

Checking whether something is in an array is even more variable. It might be the first item, finding it halfway is average, and if it's not there we always run the full loop:

```
bool contains(int[] A, int findMe) {
  bool found=false;
  for(int i=0;i<A.Length; i++) if(A[i]==findMe) { found=true; break; }
  return found;
}
```

This is really about the best, average and worst time a loop takes. Usually those are about the same – within half or double of each other. In this case the minimum of 1 or 2 steps is obviously very rare. The average is obviously a loop.

A way to look at it is the `break;` is like a x2 speed up trick. It's better than nothing, but it's still a loop.

Some loops average only 1 or 2 steps. They almost never run the full time. We'd probably count them as non-loops (the math can be tricky,) but those are rare and you'll know them when you see them.

Finding the smallest item is also variable speed But it still easily counts as a standard big-O loop:

```
int indexSmall=0;
for(int i=1;i<A.Length;i++)
  if(A[i]<A[indexSmall]) indexSmall=i;
```

I like how we starting the loop at 1 not so much for a tiny speed-up, but more since it seems weird to compare `A[0]` with itself.

Checking if a loop has duplicates is our first nested loop. The standard way is for the inner loop to only check items *after* us, and to quit early if we find a match:

```
bool hasDuplicates(int[] A) {
  for(int i=0;i<A.Length-1;i++)
    // compare to everything after it:
    for(int j=i+1;j<A.Length;j++) if(A[i]==A[j]) return true;
  return false;
}
```

This is another example where we ignore small tricks. This runs only 1/2 a million steps if there are no duplicates, and half that (on average) if there is one. But that's still about the same speed as a regular nested loop, and much, much slower than a single loop would be.

Finding the most common item is another nested loop, which shows off how our categories might help. This is a modified "largest item" loop, except it compares how many copies:

```
int mostCommonItem(int[] A) {
  int mostNum=-999;
  int mostCount=-1; // anyone can beat this
  for(int i=0;i<A.Length;i++) {
    int cur=A[i];
    int count=howManyOf(A, cur);
    if(count>mostCount) { mostNum=cur; mostCount=count; }
  }
}
```

We already labelled `howManyOf` as a loop. So this whole thing is a nested loop. We probably would have seen this on our own. The big-O idea of thinking only of loops just makes it a little easier.

Inserting and removing items from a list is funny, since they change the size. A common trick is having a too-big array and an extra `size` variable.

With that, adding and removing from the end is 1 step: insert is `A[size]=newVal; size++;` and remove is `size--;`. Technically insert is 3 steps, but the important thing is they're both not loops.

Removing from the middle is a loop. Everything past it slides back to fill the hole. That's probably about 1/2 the array on average (does our program like to delete nearer the start or the end?) It counts as a loop:

```
// remove from middle:
// slide forward everything past removed item, to fill the hole:
for(int i=remIndex+1;i<size;i++) A[i-1]=A[i];
size--;
```

Sorted Arrays

Suppose an array is sorted. Sometimes it's not too hard to keep it sorted as we go. Or maybe it won't take too long to sort it when we need to. If it's sorted, there are a few things we can do faster:

Finding the smallest and largest now take only 1 step (first item, last item.) This seems like cheating, since sorting obviously takes longer than a loop. But if it was already sorted, 1 step is pretty cool.

Checking for duplicates in a sorted list is only a single loop: check whether each item is the same as the one next to it. This loop starts at 1 and compares to the item before it:

```
bool hasDuplicatesSorted(int[] A) {
  for(int i=1;i<A.Length;i++) if(A[i]==A[i-1]) return true;
  return false;
}
```

We removed a loop, sort of. Duplicates on a size 1,000 list went from 250,000 steps to about 750.

If you remember, finding an item in a sorted list can be done with a binary search, which is an odd cut-in-half loop. You find which half of the array your item should be in (which you can do since it's sorted,) then find which half of that half it should be in, and so on.

The important thing is it's much faster than a regular loop, but it's still sort of a loop. For 1,000 items it takes 10 steps. It takes 20 steps for a million length list. We'll make a new category for this, and call it a "cut-in-half" loop.

The end result is searching for an item went from about 500 steps to 10 steps. We almost removed a loop – 50 times faster is pretty impressive.

Just for fun, here binary search, which you can skip. I wrote a binary search in the recursion section, but we can write it as a loop. This picks a middle, figures out the half we should look in, and repeats until we're down to 1 or 0 items in our half:

```
bool isInSorted(int[] A, int findMe) {
  int start=0, end=A.Length-1;
  while(start<end) { // at least 2 things in it
    int mid=(start+end)/2;
    if(findMe<=A[mid]) end=mid;
    else start=mid+1;
  }
  if(start>end) return false; // 0 items
  return A[start]==findMe; // 1 item. Is it our number?
}
```

Finding the most common item is down from a nested loop to a single loop. Identical items are in a row, so we can count as part of the only loop (there's code for this later, but the idea should be clear enough.)

Sorting

Sometimes the array just happens to be sorted. Often it's almost sorted and a few easy changes can make it completely sorted. Even if it's not, it's often faster to sort it, then check for whatever you wanted.

The easy sorts are nested loops: bubble sort, selection sort and insertion sort. But sorting is pretty important and over the decades we've figured out some faster, but complicated ways of doing it. The built-in sort, `Array.Sort(A);`, uses one of those.

Fast sorts are a nested loop where one is just a cut-in-half loop. That means sorts are just a little bit worse than a single loop. For a length 1,000 array, the

cut-in-half loop runs 10 times, for only 10,000 steps total (much better than hundreds of thousands for a nested sort.)

If you want to look them up, the fast sorts are named merge sort, heap sort and quick sort. They all work differently, but all use a regular loop nested with a cut-in-half loop.

This means for anything that takes a real nested loop – duplicates or finding the most common – we can usually do it a few hundred times faster by sorting first, than using a single loop.

Suppose we want to check whether two arrays are the same. If would be a nested loop. But we can go faster by sorting both, then using a single loop to compare items side-by-side.

40.2.1 Fun with bad loops

C# has a class `List` which is really just a wrapper around an array. For example, `B.RemoveAt(50);` gets rid of `B[50]` by running the slide-down loop. It's made for when you want an easy-to-use array and don't care about speed.

Here's a quick demo using it:

```
List<int> B = new List<int>{4,12,8,29}; // C# allows this shortcut
for(int i=50;i<100;i+=5;) B.Add(i); // adds 50, 55, ... to the end. B will grow
int n=B[3]; // normal indexing. n is 65
int i1=B.IndexOf(8); // search position of 8 (this is a loop)
B.RemoveAt(7); // slide-down loop
B.Remove(29); // search loop + slide down. Counts as 1 loop
B.Insert(7, 785); // insert 7 at position 785. Slide-back loop
```

If you don't think about which of these are loops, you can write some slow programs. Suppose we want to get rid of all negative numbers. We can use a back-to-front loop, deleting as we see them (going backwards makes it easier not to skip over a number when we delete):

```
for(int i=B.Count-1; i>=0; i--)
  if(B[i]<0) B.RemoveAt(i);
```

This looks pretty short, but `RemoveAt` is a loop, so this is a nested loop. Yikes.

It turns out we can write a multiple remove using a single loop. There's a built-in `RemoveAll` that works that way, but it requires you know how to send a function: `B.RemoveAll(n=>n<0)` removes negative numbers from `B`.

Just for fun, this is a single loop that creates a new array with all the negative numbers removed. The last loop is a standard "copy only some things," using a floating index for the new array:

```
// count how long the new array should be:
int len=0;
for(int i=0;i<A.Length;i++) if(A[i]>=0) len++;
int[] Temp = new int[len];
// copy positive#'s into new array:
int i2=0; // next open spot in Temp
for(int i=0;i<A.Length;i++)
  if(A[i]>=0) { Temp[i2]=A[i]; i2++; }
  // else leave this out by not copying it
A=Temp;
```

Another accidental bad loop: suppose we're trying to make a list going from 99 down to 1. A clever plan is to count forwards, from 1 to 99, but add each item to the *front* of the list:

```
List<int> B = new List<int>(); // size 0
for(int i=1; i<=99; i++) B.Insert(0,i); // add to front (position 0)
```

But we know inserting to the front is a loop, so this is an unnecessary nested loop.

40.3 Linked Lists

If you haven't seen linked lists, the important part is that each item is connected by pointers to the ones before and after. This means that if we have an item, we can remove it by just changing a few pointers. We can also easily insert to the front, back, or the middle.

But this also means we can't jump to `A[50]` in one step. We need a loop from the front following the next-item pointer 49 times. That also means we can't do a binary search – we can't just jump to the middle.

The example where we make 99 to 1 by adding 1 to 99 to the front works fine with a linked list. This is a single loop:

```
LinkedList<int> N = new LinkedList<int>(); // empty list
for(int i=1;i<=99;i++) {
  N.AddFirst(i); // only 1 step
}
```

The main reason we use linked lists is for times when you'll loop through the whole list, and delete items where-ever you find them.

For example, assume we have a linked list `K` and occasionally add new killer robots to it. This processes everything in the list, removing dead ones. It's a little long. The main idea is that "remove current item" is one step:

```
// loop variable is a pointer. Start it at the first one:
```

```
LinkedListNode<KillerRobot> ptr = K.First;
  while(ptr!=null) {
    // in case we have to delete this one, get pointer to the next robot now:
    LinkedListNode<KillerRobot> ptrNext=ptr.Next;
    KillerRobot kr = ptr.Value; // current robot
    // process kr somehow
    if(kr.health<0) {
      K.Remove(ptr); // takes only a 1 step, instead of a loop for an array
    }
    ptr=ptrNext; // will be null if last item
  }
```

By switching from an array to a linked list, we turned a nested loop into a single loop. Of course, we can only use that trick if this is the only thing we do. If some other function searches or wants to jump around we'll have to think more.

40.4 Formal math

Instead of talking about loops, we really measure speeds using formulas. They're shorter and more accurate. For example, a nested loop over 1,000 items takes 1,000 x 1,000 steps, which is the size squared. We just write that as $O(n^2)$.

The rules are:

- A capital O and parens goes around the whole thing. This is why it's called big-O notation.
- If there's no loop just write $O(1)$, which stands for "basically one step."
- Pick a variable to stand for the size of the input, usually n. This is often the size of the array.
- Write an equation using n with no constants and only the highest term. This is pretty much what we've been doing before.

The numbers of times you can cut something in half is the *log* (really the log base-2, but they're all the same except for multiplication.) The times we already know work out as:

$O(1)$ - not a loop
$O(log(n))$ - cut in half loop
$O(n)$ - single loop
$O(n * log(n))$ - good sorting loop (cut-in-half around a normal loop)
$O(n^2)$ - nested loop
$O(n^3)$ - triple nested loop

If you read technical stuff, they'll tell you inserting to an array is $O(n)$, which means it takes about n steps where n is the size of the array.

Formally we look at these as asymptotically. That means can can graph lots of things as n increases, step way back and we'll see these categories. If you graph $3n^2$ and $n^2 + n$ they'll run very close to just n^2 (when n gets large enough.) That's the formal math part of why we simplify to no constants and the largest term.

There is a formal math definition of big-O, which someone had to figure out and prove it made sense, but you can safely skip it.

There are an infinite about of big-O categories, for example $O(\sqrt{n})$ or $O(n^{1.9})$. But the ones I listed are the ones that come up in real programs. For example, you could write a loop that takes the square root of the size and spins that many times, and it would be $O(\sqrt{n})$. But there's almost no real problems you'd solve that way.

Cut-in-half loops are the only common funny ones, which are $O(log(n))$. If you're wondering, a cut-in-thirds loop is also $O(log(n))$. It's about 30% fewer loops, but more work inside, so it evens out.

Sometimes you use two variables. For example, a loop that compares every item in one array to every item in another would technically take $O(m \cdot n)$, where those are the two array sizes. But they're usually pretty close so calling it $O(n^2)$ is fine.

40.5 More tricks and exceptions

Obviously, not all loops are problems, and not all problems are loops. We're really looking for things that more and more time as the input grows. A big array loop is just the most common way that happens.

Some notes:

Small, fixed-sized loops don't count. For example, suppose you're on a grid and want to check the 5x5 area near you. A nice way to do it is with this nested loop:

```
// get sides of 5x5 area around me (me -2 and +2) not off edge:
int x1 = Mathf.Max(myX-2,0), x2=Mathf.Min(myX+2,width-1);
int y1 = Mathf.Max(myY-2,0), y2=Mathf.Min(myY+2,height-1);
for(int x=x1;x<=x2;x++)
  for(int y=y1;y<=y2;y++) {
    if(x==myX && y=myY) continue; // skip space I'm on
    // check Grid[x][y] ...
  }
```

This runs at most 25 times. If the grid grows to 5,000 by 5,000 – this still runs 25 times. Since we're really trying to estimate the time it takes, we call this $O(1)$.

In Unity3D, `GetComponent` and `transform.Find` are also often tiny fixed-sized loops that should count as $O(1)$. GetComponent loops through all of your components until it finds the one you wanted. But I've never had a gameObject with more than six components. Realistically, it's 6 steps and counts as $O(1)$.

Likewise I often use `transform.Find` when I know I have two children and will never have any more (for example, a label with children: Text and Background.) It loops through them, but it's 2 steps, tops, so we'd call it $O(1)$.

`GameObject.Find` is more like a real $O(n)$ loop, since you probably have hundreds of game objects, and will have more in the future.

This is an example of linked list vs array. They decided GameObjects are in a linked list to make removing them faster. That makes finding one slower, but it's easy to program around that loop by just saving links to the ones you need.

For more fun, some people like to use empty gameObjects as folders, so `transform.Find` could count as $O(n)$. For example, we put a few hundred pickups inside of an empty named `pickupHolder`. Now it would be reasonable to say `transform.Find("healthPack5")` is $O(n)$.

To see why that matters, suppose we do something with every health pickup, using a loop to get health pack 0, 1 and so on:

```
for(int i=0;i<maxPacks;i++) {
  Transform tt = pickupHolder.Find("healthPack"+i); // <-big loop
  // do something with tt
}
```

If we're thinking in big-O terms, we might spot this as a nested loop, running about 100 times slower than it should. We could get a massive speedup rewriting as a single loop that goes through all children, skipping the non-health packs.

Count the work, not the loops. Sometimes a nested loop just goes through the array once, so is really $O(n)$. For example this finds the longest run of numbers in a sorted array (if you don't want to read it, the important thing it's a natural way to solve this, and looks like a nested loop):

```
int maxRunLen=-1, maxRunVal=-999;
for(int i=0;i<A.Length;) { // will increase i inside the loop
  int runLen=1; int val=A[i]; // set-up to count this number
  i++;
  // count the duplicates, check for a new winner:
  while(i<A.Length && A[i]==val) { i++; runLen++; }
```

```
  if(runLen>maxRunLen) { maxRunLen=runLen; maxRunVal=val; }
}
```

If you trace it, this goes through the array just one time. Both loops push `i` ahead. If the array has 1,000 items, this takes 1,000 steps. So it's really $O(n)$.

Likewise, it's possible to write a single loop that runs like nested loops (use `if`s to move `i` and `j`.) And we're already seen the cut-in-half loop.

Non-array loops count. These aren't as common, but sometimes you just loop over numbers. If you check every angle from 0 to 180, going by 0.1, that's about 2,000 steps. A nested loop might check a different angle on 0-90 by 0.1's. It seems fair to call that $O(n^2)$, and to try to think of a less loop-using way.

Sometimes you have to pick `n`. If a loop counts up to a number, `n` is the number. But if a loop checks each digit of a number, `n` is just the number of digits, which isn't that large (but a double or triple-nested loop can still get big, fast.)

If you have a 50x50 grid you might say n is 50, or you might think of n as 2,500 (which is the number of actual spaces, but is also 50 squared.) If the grid is more rectangular, like 10x1000, n = the number of squares probably makes more sense.

If seems funny that we can just pick it, but we're only using it for comparison. It a "pick one and stick with it" situation.

Recursion counts, and is tricky. A loop is the most common way to spin 1,000 times. But if you run through an array using recursive calls, time is time. For example, this recursive loop-faking function is $O(n)$:

```
bool allPositive(int[] A, int startIndex=0) {
  if(startIndex>=A.Length) return true; // made it to the end w/o quiting
  if(A[startIndex]<=0) return false; // found a negative
  return allPositive(A, startIndex+1); // keep looking
}
```

Recursive functions that call themselves twice or more can be tricky. Some blast out of control with $O(2^n)$ – exponential time (which is as bad as it sounds – worse than the most nested loop.) But some work out to just $O(n)$, like flood-fill only sees each square 4 times.

The recursive tree-search function (the one that visits all children, grandchildren) looks scary, but it visits each child once, so obviously takes just $O(n)$ time (where n is how many total children.) And we often use it on simple objects where we know there will be a fixed, small number of kids, so it's more like $O(1)$.

40.6 Other data structures

There are a few more list-like structures that are only useful because they have different big-O's for various things. This is just a very rough summary:

- B-tree. A clever way to have a root item with 2 children, which each have 2 children, and so on, making a big pyramid. Search, insert and delete are all $O(log(n))$. The drawback is you can't put things in a certain order – it's automatically sorted.
- K-tree, red/black tree: similar to B-trees. B-trees have 2 children, K-trees can have more. A red/black tree is a type of B-tree.
- Heap: a tree where finding the smallest item is $O(1)$ and insert/remove is $O(log(n))$. The items can't be in any order, and search takes $O(n)$. It's when for when you always want to handle the first thing in line. For example, storing Unity's waiting coroutines.

  A priority queue is often made using a heap (it's for taking items in order of highest priority, which may change.)

- Hash table. Search, insert and delete are $O(1)$! (on average.) Everything else is bad. There's no order, not even sorted. Finding the smallest is $O(n)$. Finding the next largest item in the list is $O(n)$. It takes up more space than a tree and requires extra set-up.

  C# calls theirs a Dictionary.

They're a little more complicated than that. Knowing the details and how and when to use them is why Com Sci grads get the big money.

40.7 Overview

Altogether, including ideas from the previous section on efficiency:

- In many places you don't care about how fast the code runs: things that rarely happen, that already have an on-purpose time-delay, or "how does this look?" code which you know will be either deleted or improved later.
- Many things run fast enough. A 2D tap-tap-tap puzzle game will run fast enough with easy-to-read code where you don't worry about speed and maybe do some "slow" things.

Assuming we're in the section where we think speed might matter:

- As you plan, think about arrays vs. linked lists, based on the big-O's of whatever you want to do. Generally, if you need to jump around a lot, you have to use an array. If you need to insert/delete lots from anywhere

except the end, you need a linked-list (you can fake changing the end of an array by using a bigger array than you need, with an extra `itemsUsed` variable.)

Maybe think about another fancy data structure (but not until you've tried them in a test project first, since nothing new ever works the way you thought it did.)

- As you write, think a little about the big-Os of things. If you have 50 monsters, each running an array loop which calls an $O(n)$ function, that's like $O(n^3)$. It might be worth trying to get that down.
- For arrays and lists, often sorting them, or keeping them sorted as you add new items is a big speed-up (instead of adding new items to the end, add them in the correct position. That takes longer, but you might search 1,000 times for every one time you add.)
- After you've gotten all the big-O speed-ups and want to look for small ones, use big-O estimates for where to start. For example, speed up the insides of nested loops before singles.

40.8 Numbers

This is the section with the numbers, which you can skip.

Measuring the time instructions take is fuzzy. Maybe adding takes 1 step, but adding floats takes longer than ints. But sometimes the computer does two things at once. And different CPUs can take different times.

`A[i]++;` is maybe 4 steps? Look up `i`, jump to that spot in `A`, add 1, then save it. But they might not all take the same time, so about 4 steps.

Things like square root and trig functions "take a long time," but not really. They might take 10 or 20 steps each. A big, fat trig-using math expression might take 80 steps. If you can get rid of some trig you don't need, you might get it down to 62.

This is what I mean by non-loops not mattering. 80 to 62 isn't a big speed-up compared to x100 faster, and 80 is a small number compared to a few thousand.

If we want to accurately measure the smallest array loop, it might take 7 steps. Just moving the loop is `i++` (2 steps?) and `i<A.Length` (3 steps?) and then `A[i]=0;` is 2 or 3 steps? The hypothetical simple 100 length loop is 700 steps total, making the 80-step math seem even smaller.

If we add a second line inside the loop, that doesn't really double the time. It's more like 7 steps increasing to 9. Put another way, cutting 2 lines inside a loop to 1 is really only a 20% speed-up.

If we put our 80-step math equation in a 1,000 step loop, of course it matters. We're taking 80,000 steps total. But reducing the middle to 62 steps is still less than a x2 speed-up. Removing the loop is still the best thing.

A worse big-O isn't always larger, but will always *eventually* be much larger. Suppose we have a single loop with 50 steps in it, and a sneaky nested loop with only 1 step in it. The times look like this:

```
size  single  nested
20  |   1,000       400
50  |   2,500     2,500
100 |   5,000    10,000
200 |  10,000    40,000
500 |  25,000   250,000
```

We might have a size 100 single loop and a size 20 nested which is currently much faster. But if they get larger, the nested loop quickly sucks up all the time.

And if you only have small arrays, you're probably not having speed issues anyway.

The other thing we have to worry about is working on the "fat" parts of our program, but big-O usually takes care of that:

Suppose our program has 10 equal-sized chunks. A 100x reduction of one means we still take 90.1% of the original time. But what really happens is the worse big-O stuff is also the longest, by a lot. For real, if one of those 10 parts has a nested loop, it's probably 90% of the time of the entire program.

In practice, the worst big-O functions are also the ones hogging up almost all of the time.