Chapter 35

Debugging

I tried to put bits about testing a program and finding errors all through this. But we never wrote any really big programs, and I wanted to sum up, plus mention some common debugging tricks.

35.1 Test as you go

The main trick to debugging code is to write a little, and then test. Put another way, it seems like it would be easier to just not make any mistakes, but that never happens.

The quickest way to get a working program is to write a little, test, write a little more ....

Some of why that's true:

You have to test code that doesn't have any errors, anyway. Because the hard bugs are things you can't even tell where they come from.

Suppose a Cube is in the wrong place. The mistake could be in the equation. Or maybe one of your functions used by the equation is wrong. Or maybe the inputs are wrong. Or maybe the positioner part is using the value wrong, or you forgot to position it at all. Maybe that's all perfect, but someone else is moving it again.

That sort of thing happens a lot. You "know" each part has no mistakes, but you know one does, since the Cube is wrong. So you have go back and test each part. You have to test 6 parts that work perfectly, then one that you thought was perfect, but had a tiny problem.

Once you realize that you'll have to test each part, it's easier to test them just after writing them.

If you write it all, then run it, there will probably be several bugs. Often they overlap, for example both cause the Cube to be a little wrong. You find

one, fix it, but it since doesn't look fixed you put it back the way it was – multiple overlapping bugs are the worst.

Often early testing finds "wrong idea" problems. In a game, you might realize the thing you thought would be fun, isn't. In a web app, you realize the browser feature you're using is buggy.

You especially want to test before copying code. I still make this mistake all the time – I write a few lines to handle a button and test them with just one click. Then copy them with tweaks to 5 more buttons. Sure enough, there's some weird bug when you mouse-over quickly, which I would have seen if I spent 20 more seconds testing the first one. Now I've got to rip out all that tweaked code and start again with the fixed version.

35.2 How/what to test

If there are any commands I'm not sure about, I usually write a small, separate program to test them.

For example, Unity can use layers to make a Raycast skip through certain objects. Maybe that sounds great for what you need.

So make a Scene with two Cubes in a row and Raycast through them. It should hit the first. Play with layers until you can make it go through the first and hit the second. Play with them some more until it can skip both. Try it going the other way.

If your program is going to have 4 layers of different things to skip, test 5 Cubes in a row with all those layers. There will almost certainly be some new rule you didn't know that would have messed you up eventually.

Finally keep the scene for later. Two tests doesn't mean you know it all. Some crazy other bug will seem like a layer problem, maybe, and going back to that scene is a good way to recheck some stuff.

New language features are the same way, but worse. One of the best ways to learn something new is to try to use it in a real program. But keep in mind that's basically the same as a test scene. You can write a useful program, or write a program to learn something. But almost never both.

It's also common you realize the new feature doesn't do what you thought it did. Often you learn a lot, but your trial program is completely unusable.

You don't have to start writing the first part of the program first. You can write functions, then test them with fake function calls in `Start`.

Typical tests are "sanity checks" where you run it with really easy inputs where you know the answer. Another is an edge-case: for a letter grade function, try next to the cutoffs, like 59 and 60. Try 100.

Often you can run it in a loop. A letter grade function can run from 0 to 100 printing `num:grade` on each line. That's quick to scroll through and eyeball.

For some others, 50 random numbers are good.

Sometimes you have to be creative. For a function that checks whether you won tic-tac-toe, you can fake up a board, hand-set almost-win and win boards, and test on those.

Sometimes I write a function to test a function – just to isolate my testing lines.

You can test a main part of the program by faking up functions. If something is suppose to read user input, you can have it temporarily give a random, legal number. If a function is suppose to decide where to walk next, it can temporarily pick 20 feet ahead, or a random spot.

Depending on what you're doing, you'd start in various ways. Often you think of the simplest thing you can get running. Sometimes there's one part you're really not sure about, so think of the fastest way to test that part.

Sometimes you're not quite sure how it should work out, and you just write quick sloppy code, just to get a feel for it. You'll probably delete it later, but it's still worth it.

35.3 Tracking down bugs

No matter how much you test, there's always some bug that pops up out of nowhere. Here's are some tricks to track it down:

- When you find a bug, keep testing to make sure what it really is. Just run normally and try more things.
  If an enemy doesn't die when it should, do they never die, or just sometimes not die? Is it every enemy, or just that one? Is it only when you use a certain weapon? If you wait and shoot it some more, does it die then?

  Usually more testing will give clues about where the problem really is, or isn't.

- Don't assume it's what you just did. Maybe your game crashed after you added balloons. But it really always had a bug when the score goes past 50, and balloons just happened to be the first thing to do that.

- Use prints to narrow things down. Somewhere in the program, some number isn't what it should be. Enough prints can find it and track down who's doing it.

- Sometimes, after trying to figure out some old, horribly awful section you wrote, that keeps having problems, you decide to just rewrite it. Only do it if you think this section is so bad that it's going to suck up more and more time. Lots of people go nuts on rewrites. If a section looks terrible, but works, it's fine.

More details about that stuff:

In Unity, you can copy a local variable into a temporary `public` global, so you can watch it in the Inspector. This lets us watch local `vel`, using debugging variable `vvv`:

```
public Vector3 vvv; // temp public to see velocity
void someFunc() {
  Vector3 vel = ... // normal local in function
  vvv = vel; // test line to copy it into temp Inspector var
  ...
}
```

You can just cheat, make all weapons do 9999 damage, make the space bar kill the nearest enemy. Or make it redo the random balloon placement.

You can see the **stack trace** from a print. It will list all functions that were called to get to this one. Suppose blocks are removed for no reason. Add a print in the `removeBlock` function. When you read them, you might see the function chain: Update, CheckOverlap, RemoveBlock. So now you know the `checkOverlap` function is removing blocks when it shouldn't.

You can add a lot of prints. At first, just "I got here" prints can help:

```
print("A");
if(key) {
  print("B");
   ...
}
```

Later on, expand them to print all your data. Then you can look for wrong numbers:

```
int doStuff(int n) {
  print("starting doStuff, n="+n);
  ...

  print("ending doStuff, ans="+ans);
  return ans;
}
```

Sometimes a function is called hundreds of times, which is far too much to print. You can be clever. This might be called for everything, but we have a problem with just the player, so add a check to only print then:

```
int doStuff(int n, GameObject gg) {
  if(gg.transform.name=="Player")
    print("doStuff on Player, n="+n);
}
```

Sometimes you can't find a nullReferenceException because you have a long pointer chain. In this, the null could be `c1` or `favoriteDog` or `gg` . . . :

```
float r = c1.favoriteDog.gg.GetComponent<Renderer>().material.color.r;
```

It can be broken into single steps to see which one is null:

```
if(c1==null) print("no c1");
Dog d=c1.favoriteDog;
if(d==null) print("no dog!");
else print("got dog "+d.name);
GameObject gg = d.gg;
if(gg==null) print("no gameObject");
else print("got gameObject "+gg.transform.name);
Renderer rr = gg.GetComponent<Renderer>();
if(rr==null) print("no renderer");
else print("got rr");
```

This will tell us exactly which part was null (it will still crash, but we'll know where.)

The real version of adding prints is using a debugger with break-points. But depending on what sort of program it is, you can't always do that. And prints work pretty well.

Try commenting out chunks until the error goes away. Maybe you think the part that checks for a legal phone number is breaking something. Comment out `checkLegalPN();`. Often, the error still happens, so that wasn't it.

Suppose you added cats and dogs and start getting funny errors. Try getting rid of them – commenting out the part that makes them. Sometimes you keep getting the same error, so it wasn't the cats and dogs after all.

It's very common to be sure the error is in one place, and be completely wrong. Often a light bulb dance can save lots of time: if a lamp doesn't go on, try it plugged in somewhere else, check if the clock is working, check if the bulb works in another lamp.

Check out every part of your program the same way to be sure you're fixing the part where the problem really is.

With some tough errors you can go the other way – start a fresh mini-program. Gradually add simplified parts of your old one, and test. Eventually you'll find the one thing that worked wrong. Or maybe the finished simplified program works correctly. That still good since now you know the basic plan is fine.

Finally, this is more of a tip, at some point you'll be trying to fix something and you'll notice code that makes no sense. Maybe you see `cats+1` and you're positive it should just be `cats`. You can't figure out why that `+1` is there.

You'll be tempted to fix it, but don't. What are the odds someone added a +1 by mistake without completely breaking the function? If a function has things in it you don't understand, that's pretty much a giant warning not to mess with it.

Also, if you wrote that part, make a note to yourself about needing to make better comments.