Chapter 37

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.

37.1 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, and so on.

Some of why that's true:

Having two bugs at once is horrible, especially if they overlap. You could fix one and not even know. Testing as you go is much more likely to find one error at a time.

You might have a bad plan. Early testing tends to find it before you write lots of stuff that needs to be torn out later when you finally see the problem.

Sometimes part of your programs are basically copies. It's better to find bugs in the original before making 3 copies of the same error.

You'll have to test everything anyway. At some point you'll have an error where you can't even imagine what's causing it. This doesn't seem possible: if something is in the wrong spot, the error is obviously in the equation. But there are times when you've checked the obvious places, and they're fine. Something that can't be causing the error, is causing the error. The only way to find it is to test every part.

Put another way, after you write a simple function that can't possibly be wrong, there's 100% chance of it being a suspect later on. You will definitely at some point think "well, maybe the problem is here – I never tested it".

37.2 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 ignore certain objects. If that sounds great for what you need, make a small test with fresh Cubes. Assume there's a least one thing you don't understand, because there always is, and try to find it.

New language features – anything big – usually take an entire practice project to learn. If you start using a new feature in a real project, it pretty much automatically turns into a practice project. In other words. you can write a useful program, or you can write a program to learn something new. But not both.

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: try numbers just before and after cut-offs. If something checks for out-of-bounds, try things that are barely out, barely in, and use every boundary.

Most bugs are in things that you don't test, not even once, since there's no way it can be wrong.

A test-loop can try lots of numbers, with an `if` to check that the results are at least close. Sometimes you have to be creative – if a function checks a 10x10 board, you'll have to fake one up. That seems like a lot of work, but you'll have to do it anyway when you get the first impossible to track down bug.

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.

37.3 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 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.
- It's very common to be sure the error is in one place, and be completely wrong. At some point, do a quick check of all the things you know can't possibly be causing the problem. If computed value is wrong in a textBox, put just `"cat"` in there. If you don't see `"cat"`, there's something wrong with the box.
- 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.

Some specific tricks:

Add "is this even running?" print statements. If your function to align text isn't working, it might be because it never got called. If your function to make 10 Cubes is mistakenly making 20, it may be working just fine, but was called twice. Run and then check to see it ran everyplace it should, and nowhere it shouldn't:

```
print("A"); // we got to point A
if(n>=0 && active==true) {
  print("B"); // we entered the if statement
```

You can see the *stack trace* from a print statement. 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 – it will show you exactly who's calling it when they shouldn't be.

To really check a function, print all of the values. After running, check to see which ones are wrong:

```
int doStuff(int n) {
  print("starting doStuff, n="+n);
  ...
  print("ending doStuff, ans="+ans);
  return ans;
}
```

Sometimes a function is called too much to read everything it might print. You can narrow it down with `if`'s. This only prints for a Player input:

```
int doStuff(int n, GameObject gg) {
  if(gg.transform.name=="Player") // don't print all the time
    print("doStuff on Player, n="+n);
```

Occasionally you have a problem caused by reading old data. I like to sometimes change-up a print statement to check. If the first line in Start changes to `"In Start version II"` and the output is `"in Start"`, you know something is very wrong.

You may have been reading old output this entire time. Or the system may not have been reading your changes. You may need to find a Recompile-All option, or a way to delete old files to force it to recompile.

Sometimes you can't find a nullReferenceException because of a chain. In the code below, the null could be in `c1` or `favoriteDog` or in `gg`, or there could be no renderer:

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

To track it down, it can be broken into single steps, with prints in-between:

```
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");
// and so on ...
```

That seems like a lot of typing, but it's better than sitting around thinking "`favoriteDog` must be `null`, right? I mean, all dogs have a `gg` and a renderer, don't they?"

Sometimes it helps to temporarilty simplify the program. If the customer data isn't being displayed correctly, it probably wasn't caused when you computed their order. But it could have been. Comment that out. Simplify the program a little bit at a time by commenting out parts. With luck, removing one thing will cause the error to vanish – you found the part causing the bug.

If the problem is with a feature you're not very experienced using, test it in a different mini-program. For example, if this is only the second time you've used

Unity's OnCollisionStay, make a very simple demo of how your real program uses it. There may have been a few things you misunderstood.

Some specific Unity tips:

- You can Pause while running. This allows you to select items and examine values.
- While Paused, you can move items, resize, change colors. You can even add new items. This can be helpful to find the scope of a bug.
- When spawning items in a loop, change the name, such as `"cat"+i`. This might help you realize the bug is always the first one made, or the last, or something else to do with the item #.
- You can make fake Inspector variables to store locals. Suppose you want to watch your velocity:

  ```
  public Vector3 vvv; // a copy of our velocity that we can see

  void Update() {
    vvv = GetComponent<Rigidbody>().velocity;
  ```

Be very careful about hacks. For example, you can't fix the bug when `x<0`. Instead you add an extra `if` statement in Update to manually fix it. Those things usually turn into super hard-to-find bugs later. If you have to – your demo is tonight – put a big comment `HACK FOR BUG #258`

Be sure to remove all of your test lines. I once spent 15 minutes trying to figure out why it always jumps to stage 4. It was an old testing line that jumped to stage 4. Write them down, or make then super-easy to see, or reload your backed-up file.

When you fix a bug, take some time to remember exactly what fixed it. Maybe write down problem/solution. It's very common to see a similar bug and all you can remember is "I tried this, looked here for a long time, but the problem was . . . arrg! I forgot!".

Don't make casual fixes. When you're looking through a function for a bug, it's irresistible to do a little clean up and quickly fix things that are "obviously" wrong. That seems crazy – who would do that? But when you see `i<L.Count-2` in a loop, it's so, so hard not to think "minus 2? How did that get there? I can save so much time if I fix it now".