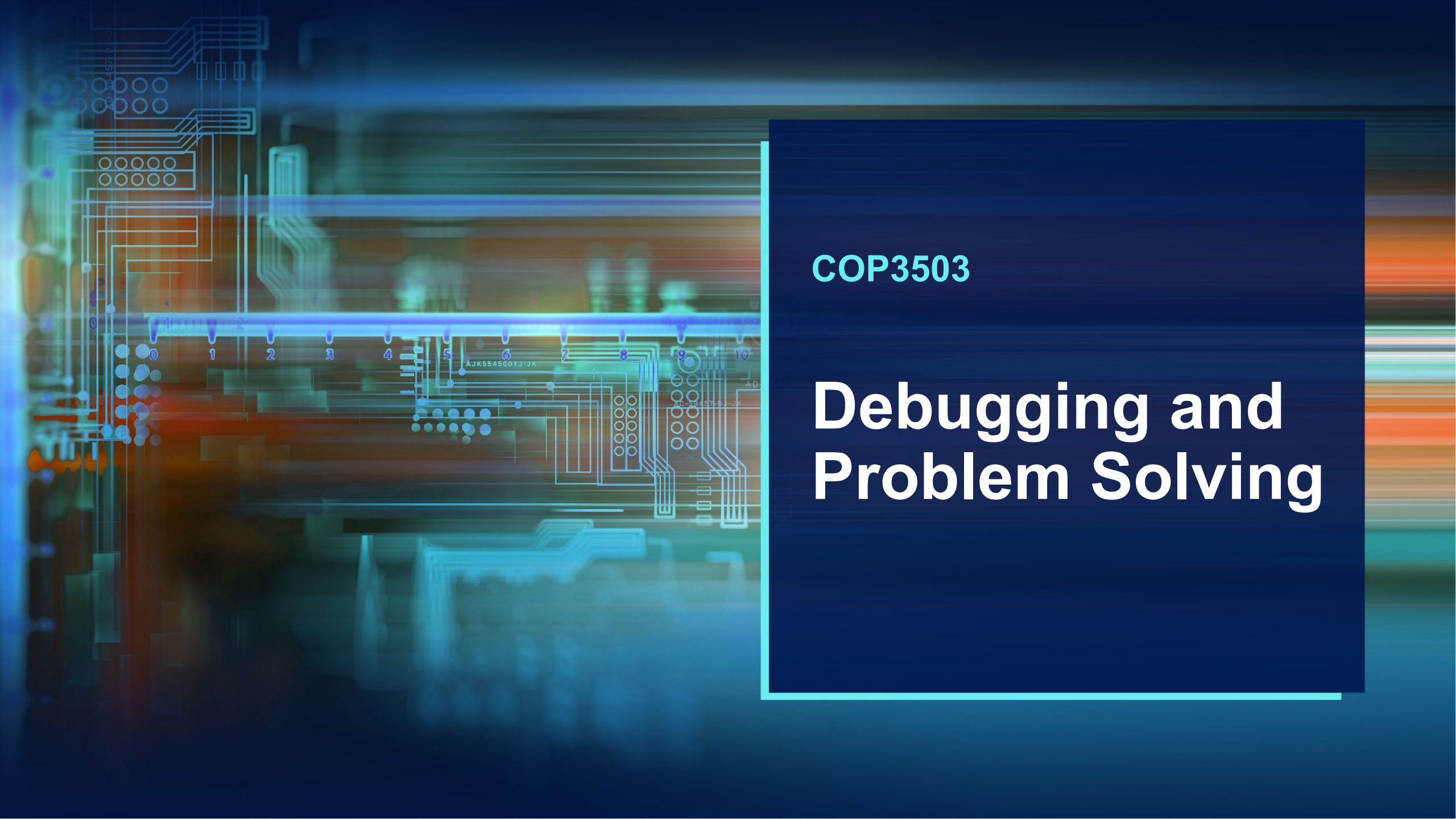


COP3503

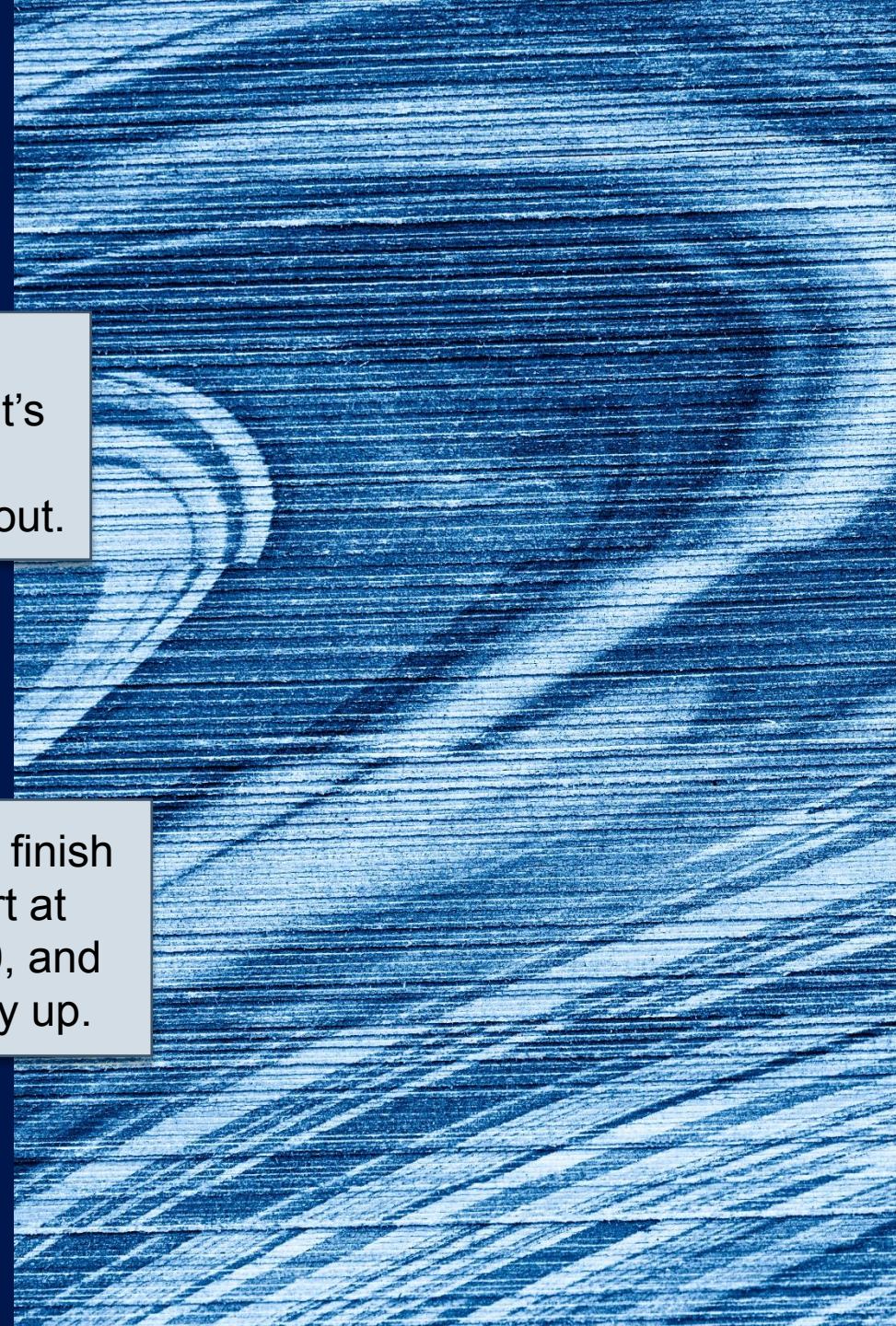
Debugging and Problem Solving

Embrace “I Don’t Know”

- Use this as a starting point.
- I don’t know…
  - but I will try to figure it out.
  - and I will do some research to change that.
  - but I’m working on it.
  - yet.
- Don’t sell yourself short—you know **something** about the problem.
- **The Big Secret™**:
  Everyone has a list of things they don’t know.

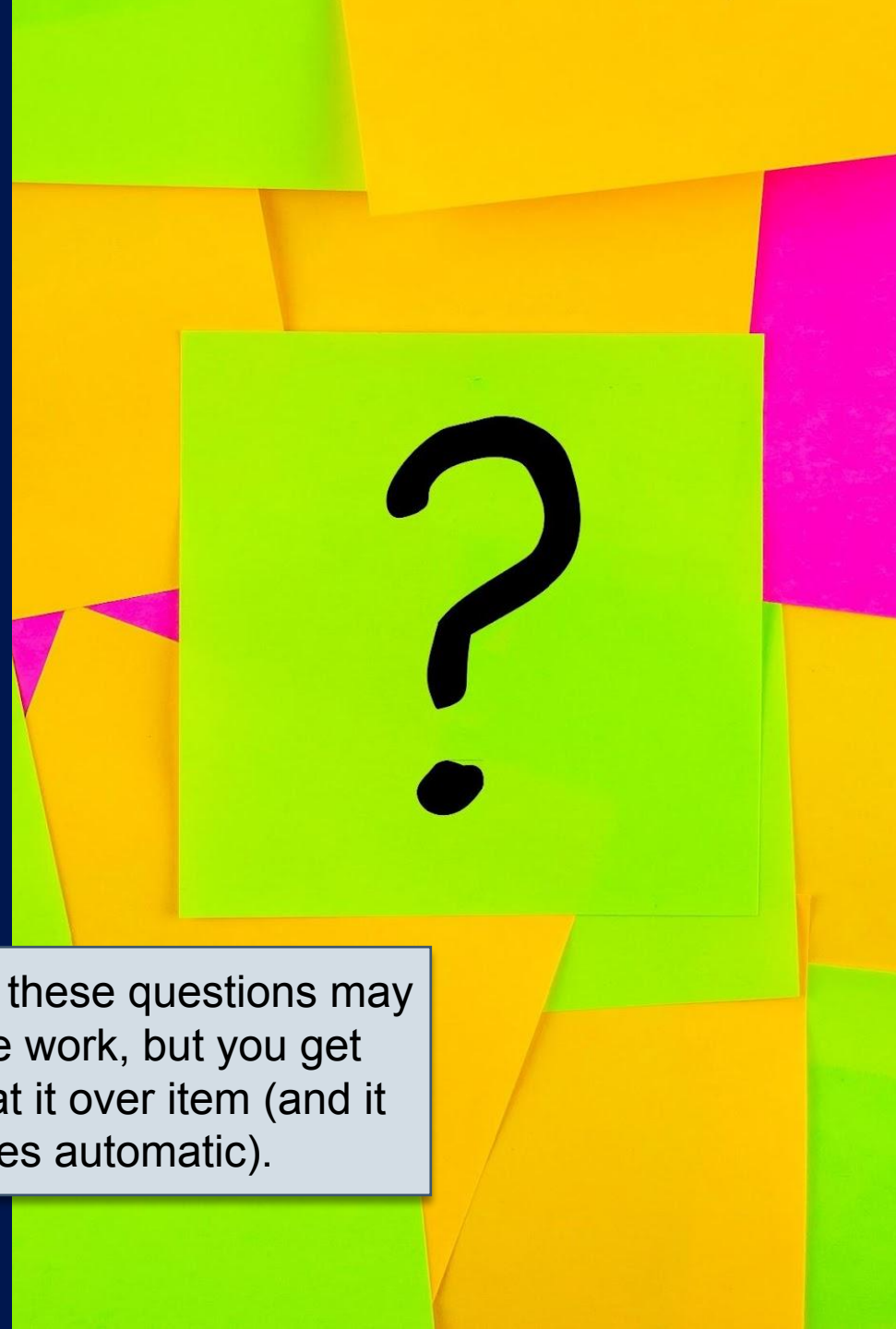
No one expects you to simply know everything. It’s about **working toward a solution**, figuring things out.

We don’t start at the finish line. We don’t start at 100%. We start at 0, and slowly build our way up.



Problem-Solving Involves Questions

- What have you tried so far?
  - What were the results?
  - What worked?
  - What didn't work?
- What do you know about the problem?
  - What can you infer from that information?
- What do you **not** know about the problem?
  - Could these gaps contain possible solutions?



Asking these questions may feel like work, but you get faster at it over item (and it becomes automatic).

What Have You Tried So Far?

- "My program crashes."

"That stinks! Sorry for your troubles!"

- "My program crashes when attempting to do X." (A little better)

"Okay… what have you explored so far?"

- "My program crashes when attempting to do X. I've ruled out (I think) these possibilities, and I think I've narrowed it down to one function, but I don't know why exactly that function is breaking." (Better still!)

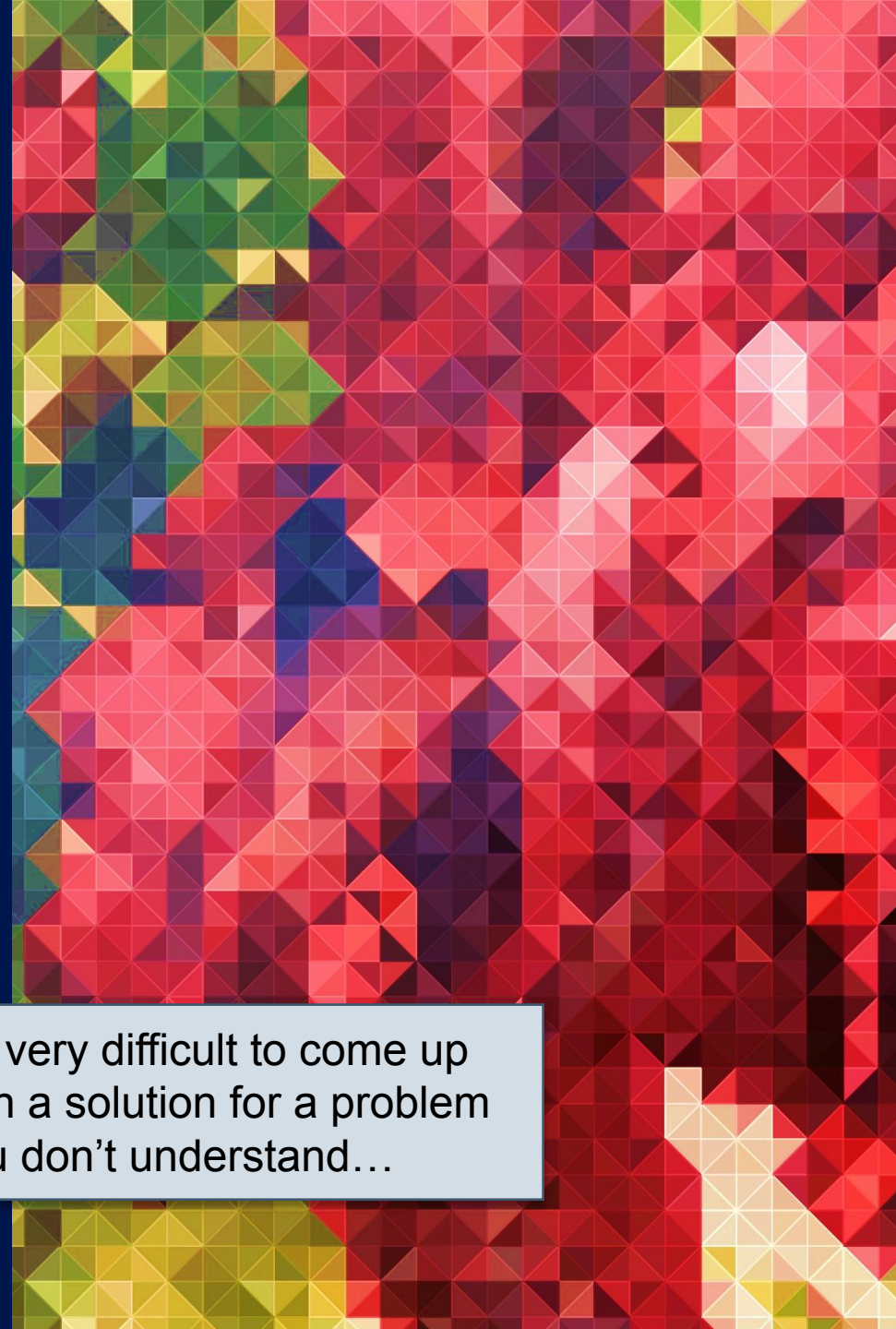
"Oh, well if it's not A or B, have you considered C?"

- "I've tracked the bug down to this particular line. It's breaking because X is a null pointer, or -2 when it should be 0, etc… but I don't know how it got to be that way."

Zeroing in on the problem! You've discovered the **where**, on to the **what**.

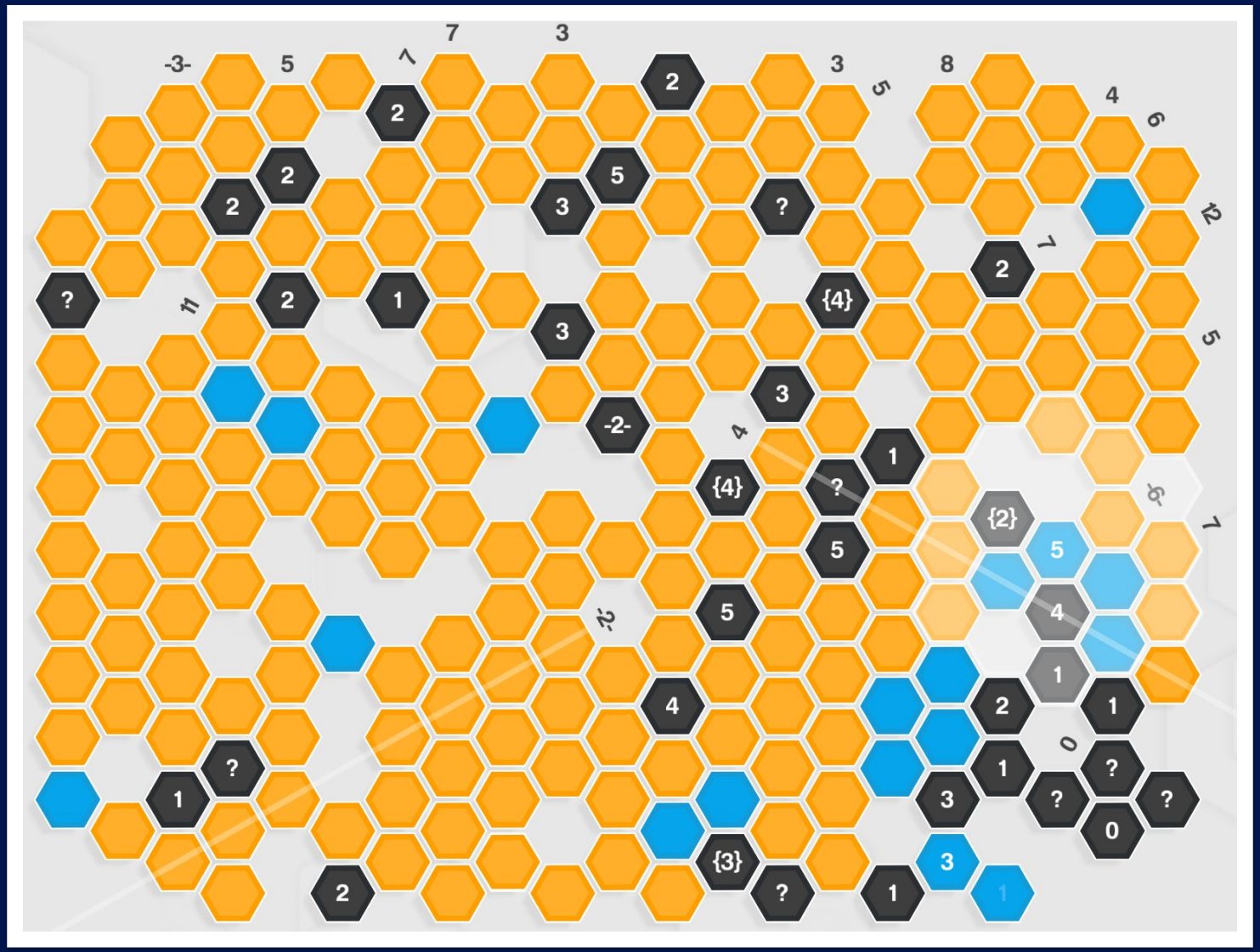
Do You Understand the Problem? (Or Problem Space?)

- Can you describe what you're working toward?
  - What you're trying to solve overall?
  - What the current problem is?
- Can you explain the major parts of the current situation?
- How confident are you in your understanding of those parts?
  - If not, then first focus on gaining an understanding



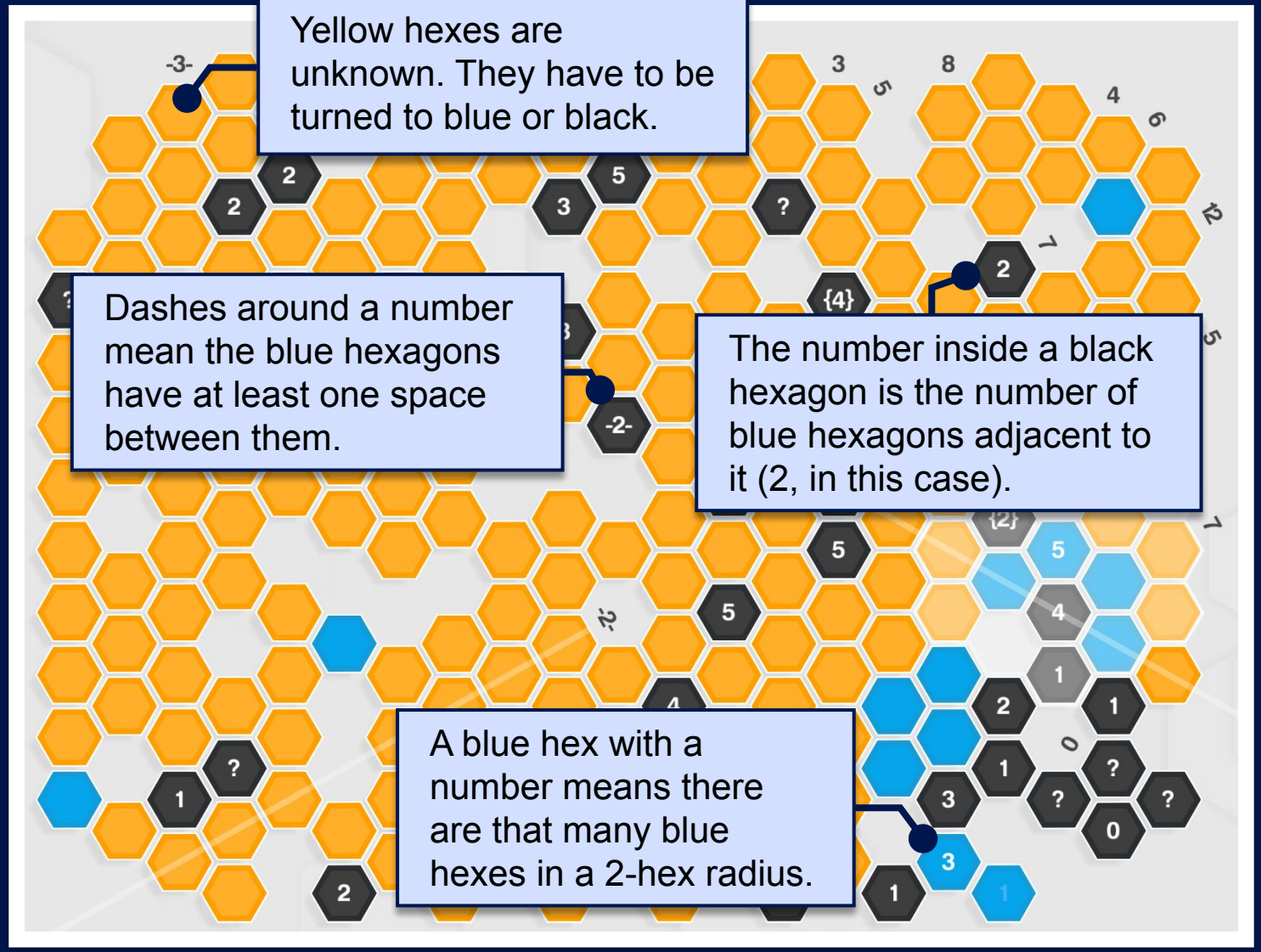
What's Happening in This Picture?

But… what does that **mean**?
What is this **doing**?
Can you **interpret** this?



- This is a level in the puzzle game, Hexcells.
- If asked, what could you say about this image? How would you explain or interpret it?
- You might start with a **literal description**:
  - It is a collection of hexagons.
  - Some are yellow.
  - Some are blue.
  - Some have numbers in them.
  - Numbers are outside some hexagons.
  - There are lines through a few hexagons.

If You Knew About the Game…



- Now you can explain what is happening, instead of providing a literal description.
- You can explain individual elements, and **how they relate to others**.
- This is especially important with code.

Explaining Your Code

- Can you explain your code to someone else?
- Do you know how it works?
  Can you **interpret** what is happening?
- Understanding underlying processes matters
- Even “simple” terminology matters!

Your constructor is okay… but your copy constructor isn’t.

Forget to initialize something?

Change the class and forget to update both constructors?

Before you can solve the problem, you have to understand the fundamental differences between these three lines.

```
Foo x, z;
Foo y = x;
z = y;
```

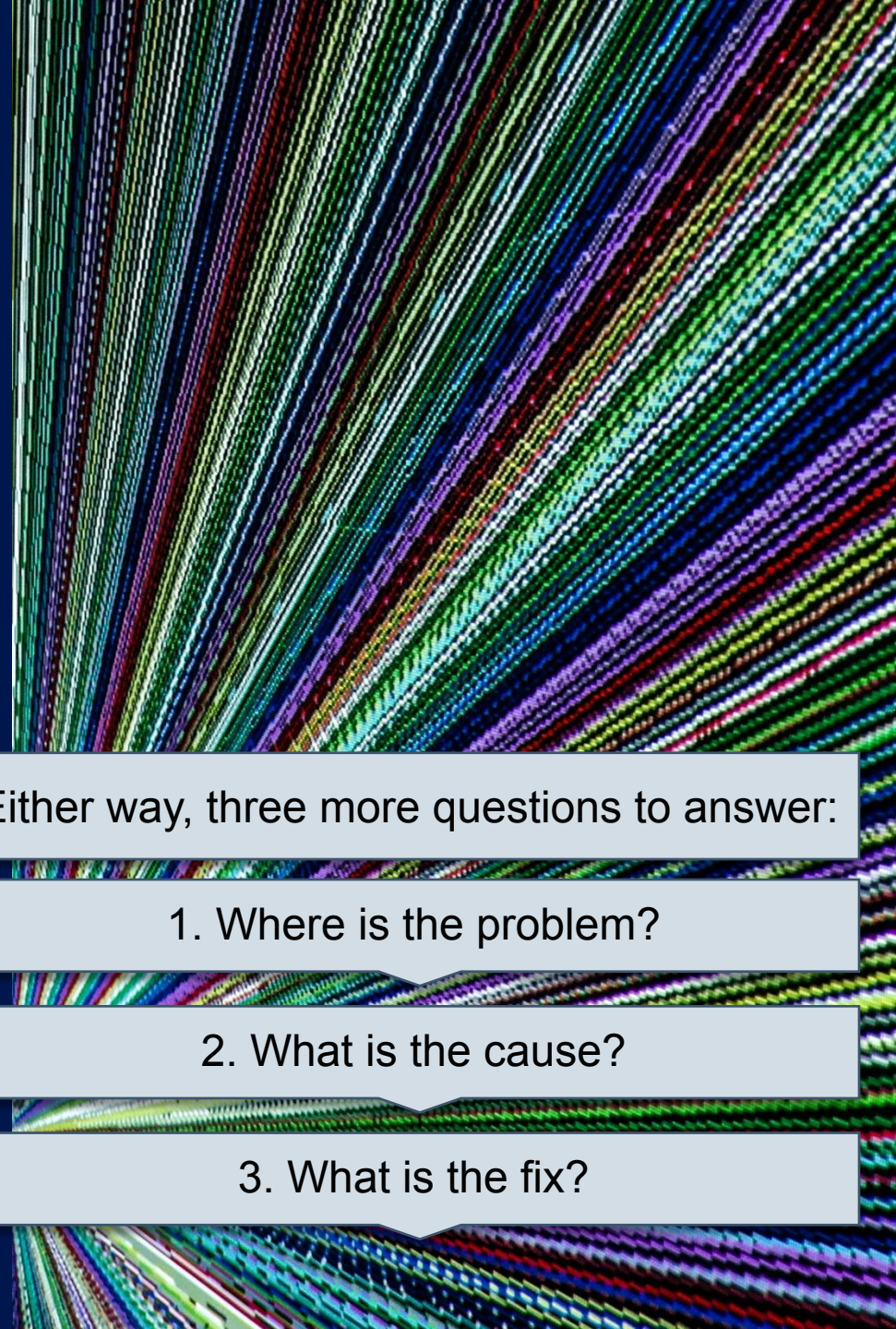
“But not this?”

```
// Invoke default constructor
// Invoke copy constructor
// Invoke copy assignment operator
```

“Why is my code broken when I do this?”

Solving Common Programming Problems

- Code won't compile? (**syntax error**)
- Code won't execute properly? (**semantic error**)
- Syntax errors are usually easier to deal with.
  - The compiler points out the error, sometimes even suggests how to fix it.
  - Syntax is well-documented.
- Semantic errors are more challenging to deal with.
  - Your code does **something**… just not what you want.
  - You will spend most of your fixing these errors.



Either way, three more questions to answer:

1. Where is the problem?
2. What is the cause?
3. What is the fix?

Syntax Errors (Compiler Errors)

- These are easy(ish)--The compiler says "Hey, your code's busted."
- The **where** is easy! The error references a line, gives you a message about the error
  - Sometimes the message is even helpful!
  - Some languages are better about this than others.
- Fix it, or learn how to
- **Fun Twist**: The issue may actually be elsewhere.
  - Line 514 didn't compile? 514 could be the disease, or just a symptom.
  - The problem with your code could be 3 files over, on line 6.

Semantic Errors

```
int main()
{
    Vehicle vehicles[] = { /* Vehicle Initialization */
    };

    // Showrooms to store the vehicles
    Showroom showroom("Primary Showroom", 3);
    showroom.AddVehicle(&vehicles[0]);
    showroom.AddVehicle(&vehicles[1]);

    Showroom secondary("Fuel-Efficient Showroom", 4);
    secondary.AddVehicle(&vehicles[3]);
    secondary.AddVehicle(&vehicles[4]);
    secondary.AddVehicle(&vehicles[5]);

    // A "parent" object to store the Showrooms
    Dealership dealerShip("COP3503 Vehicle Emporium", 2);
    dealerShip.AddShowroom(&showroom);
    dealerShip.AddShowroom(&secondary);
    dealerShip.ShowInventory();
    return 0;
}
```

- What happens when this code compiles, but doesn't work?
- Where, in **all** of this code, is the problem?

  (Tough to say, right?)
- So, how can we start to tackle this problem? Asking questions can help:
- What are you expecting to happen? What is actually happening?

Simplify the Problem

What if all this code results in no output?

```
int main()
{
     Vehicle vehicles[] = { /* Vehicle Initialization */ };

     // Showrooms to store the vehicles
     Showroom showroom("Primary Showroom", 3);
     showroom.AddVehicle(&vehicles[0]);
     showroom.AddVehicle(&vehicles[1]);

     Showroom secondary("Fuel-Efficient Showroom", 4);
     secondary.AddVehicle(&vehicles[3]);
     secondary.AddVehicle(&vehicles[4]);
     secondary.AddVehicle(&vehicles[5]);

     // A "parent" object to store the Showrooms
     Dealership dealerShip("COP3503 Vehicle Emporium", 2);
     dealerShip.AddShowroom(&showroom);
     dealerShip.AddShowroom(&secondary);
     dealerShip.ShowInventory();
     return 0;
}
```

Simplify! Does this do anything?

```
int main()
{
     Vehicle vehicles[] =
     { /* Vehicle Initialization */ };

     // Showrooms to store the vehicles
     Showroom showroom("Primary Showroom", 3);
     showroom.AddVehicle(&vehicles[0]);
     Showroom.Display();
     return 0;
}
```

If not, simplify! What about this?

```
int main()
{
     Vehicle test("Ford", "Whatever", 0, 0, 0);
     test.Display();
     return 0;
}
```

Slowly Complex… ify… the Problem

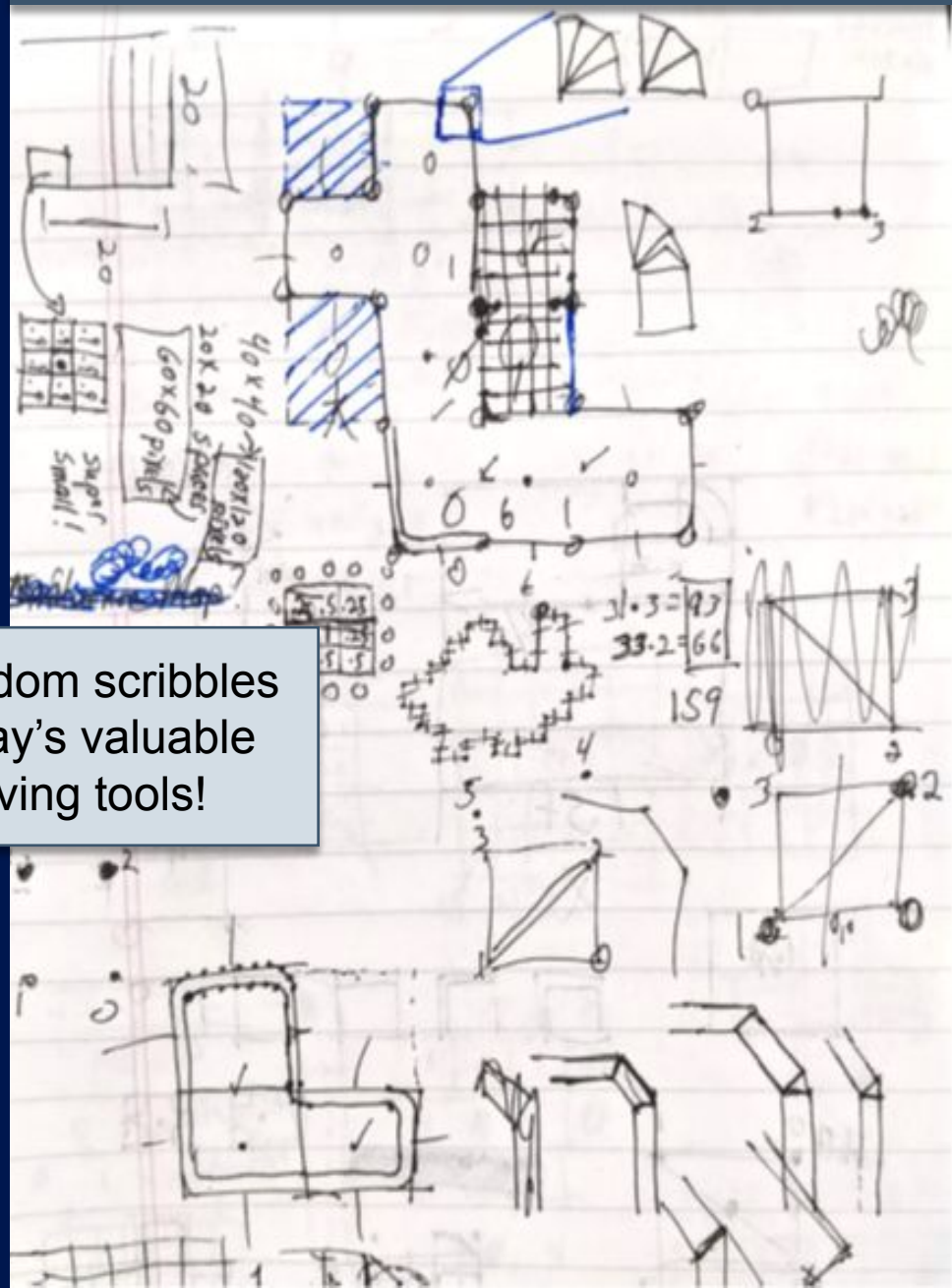
```
do
{
   if (codeWorking)
     AddMore();
   else
     Stop_FixIt();
} while (bugCount > 0);
```

- Gradually reintroduce one function, one class at a time.
- Keep going until you reintroduce the problem.
- That answers, “Where?”
  - The most recently added code is the “problem” code.
  - The issue **might** still be a combination of separate elements.

Trying to Understand the Problem?

- Sketch it out.
- Buy a blank sketchbook as a “scribble pad”.
  - Use a stylus/pen-enabled screen if you want
  - Dry-erase boards are the best $30-40 you’ll ever spend.
- **Something** to get ideas out of your head
  - “Just stare at the screen harder” is not an effective strategy.
- **IMPORTANT: You aren’t making art!**
  - They may be cryptic to anyone looking at them (including yourself, later on).
  - They help **right now**, then get discarded.
- If you don’t do this already, start!

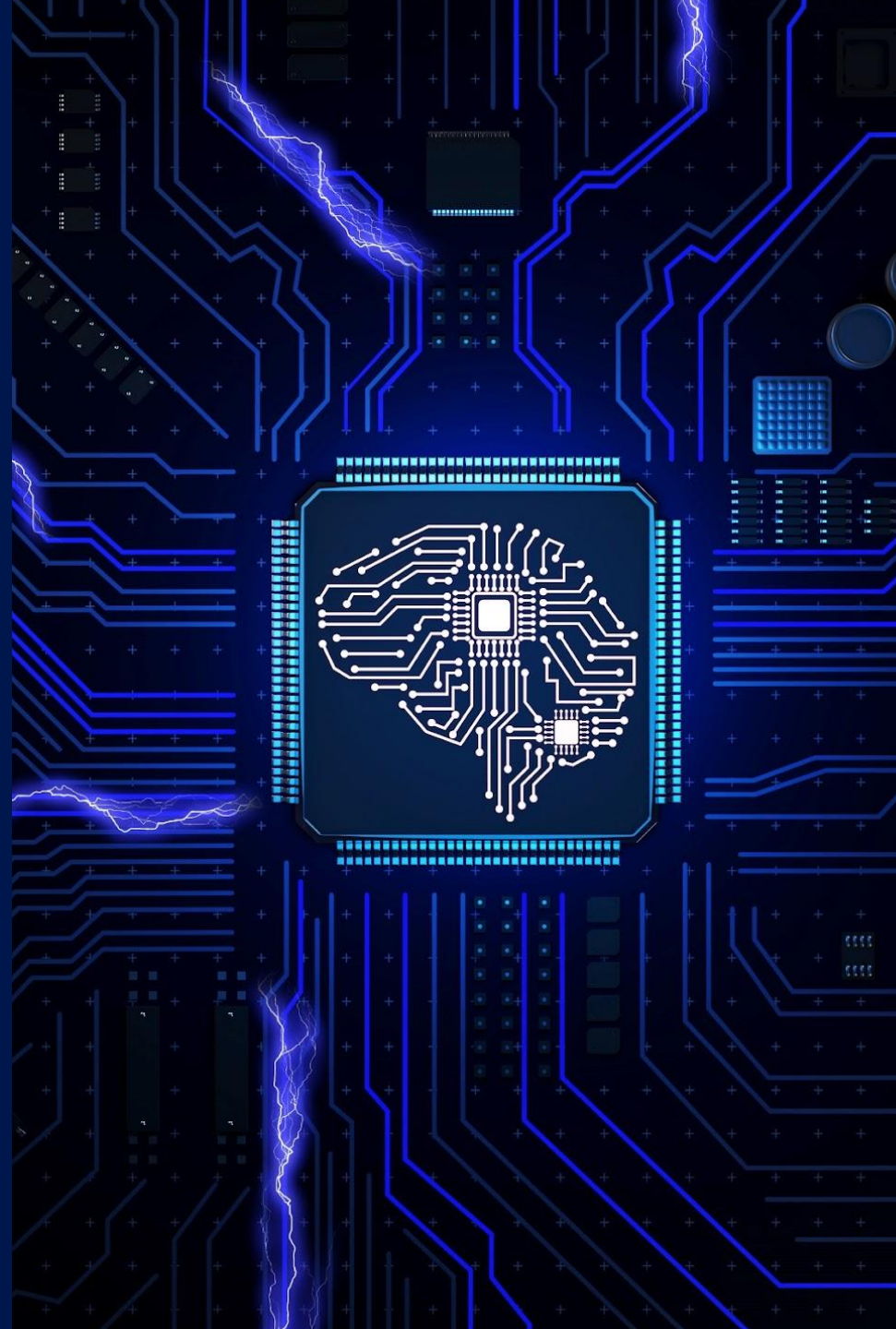
A page of one of my scribble pads



Today’s random scribbles are yesterday’s valuable problem-solving tools!

Think About Your Code

- You will spend more time **reading** code than writing it.
- Think about what you (or someone else) wrote.
- No, really. Think. What does it do? How does it work?
- If you think you're thinking about it, think again.
  You aren't.
- This takes time. Real time spent on a problem.



Reading vs. Writing

- How many times have you spent time tracking down a bug, only to discover…
- The error was some “off by 1” mistake.
- You forgot something simple like a negative sign.
- You wrote `(x = 10)` instead of `(x == 10)` in an `if` statement.
- Or some other trivial thing?
- Time spent finding the bug: hours
- Time spent **fixing** the bug: seconds

I’ve done all of these, many (many, many…) times.

Large or small, a bug is a bug. How many characters it takes to fix it is irrelevant!

Rubber Duck Debugging

- Uh, say what now?
- Having problems with your code?
  Explain it, line by line, to a rubber duck.
- Forces you to think about your code
  - Can’t explain it if you don’t know what it does
- Also forces you to speak it, and hear it
  - This uses different parts of your brain.
- Doesn’t have to be a rubber duck
  - Action figure, potted plant, **annoyed coworkers**…

Space Out Your Code

It May Be Easier to Read

This is a lot of code mashed into a small space

```
void Deallocate(void *ptr){
   auto loc = std::find(allocs.begin(),
               allocs.end(), ptr);
   auto index = loc - allocs.begin();
   if (index < allocs.size()){
      allocs[index] = nullptr;
      dealloCount++;
      std::free(ptr);
   }
}
```

1. Space your code out
2. Make sense of it
3. Clean it up when you're done

Same code, just spaced out a bit!

```
void Deallocate(void *ptr)
{
   auto loc = std::find(allocs.begin(),
               allocs.end(), ptr);

   unsigned int index = loc - allocs.begin();

   if (index < allocs.size())
   {
      allocs[index] = nullptr;

      dealloCount++;

      std::free(ptr);
   }
}
```

What does this line do?

What about this one?

What about this one?

Sometimes, Our Own Eyes Seem to Betray Us…

```
vector<Superhero> heroes;
vector<Supervillain> villains;
int hInd; // Hero Index
int vInd; // Villain Index

/* Elsewhere... */
string heroName = heroes[hInd].GetName();
string villainName = villains[hInd].GetName();
```

Code that looks very similar to other code… Your eyes glance over this with very little effort.

Your Brain Ignores Your Eyes…

What you wrote:

```
string villainName = villains[hInd].GetName();
```

What you “see” when you look for errors:

```
string notBroken = working[
```

Of course, that index is right. That couldn’t **possibly** be the issue so why would I even check for it? What kind of amateur would I be if used the wrong index?

```
].NotBroken();
```

It’s not about what our code **looks** like… it’s about what the code **actually does**.

Another Example

This is working code, bringing three separate arrays of data together into a single object.

```
int pIndex = polyIter.vertexIndex(i);
int nIndex = polyIter.normalIndex(i);
int uIndex;  polyIter.getUVIndex(i, index);

vert.X  = points[pIndex].x;
vert.Y  = points[pIndex].y;
vert.Z  = points[pIndex].z;
vert.NX = normals[nIndex].x;
vert.NY = normals[nIndex].y;
vert.NZ = normals[nIndex].z;
vert.U  = U[uIndex];
vert.V  = V[uIndex];
```

This is **almost** working code.

```
int pIndex = polyIter.vertexIndex(i);
int nIndex = polyIter.normalIndex(i);
int uIndex;  polyIter.getUVIndex(i, index);

t.X  = points[pIndex].x;
t.X  = points[pIndex].y;
t.X  = points[pIndex].z;
vert.NX = normals[pIndex].x;
vert.NY = normals[pIndex].y;
vert.NZ = normals[pIndex].z;
vert.U  = U[uIndex];
vert.V  = U[uIndex];
```

Whoops , copy/paste error

Whoops , copy/paste error

Whoops , copy/paste error

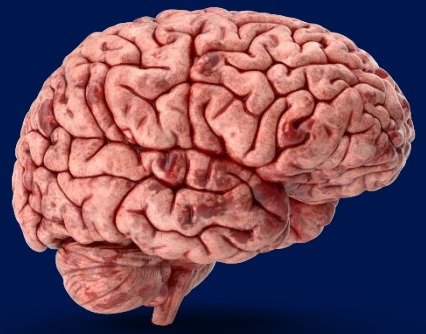
Your Eyes vs. Your Brain

What your eyes see:

```
vert.X  = points[pIndex].x;
vert.X  = points[pIndex].y;
vert.X  = points[pIndex].z;
vert.NX = normals[pIndex].x;
vert.NY = normals[pIndex].y;
vert.NZ = normals[pIndex].z;
vert.U  = U[uIndex];
vert.V  = U[uIndex];
```

What the brain says the code looks like:

```
working = somestuff[perfect].x;
correct = somestuff[booyah!].y;
goodjob = somestuff[winner!].z;
notabug = somestuff[winner!].x;
HaveAnA = somestuff[chicken].y;
itworks = somestuff[dinner!].z;
bugfree = somestuff[working];
waytobe = somestuff[correct];
```



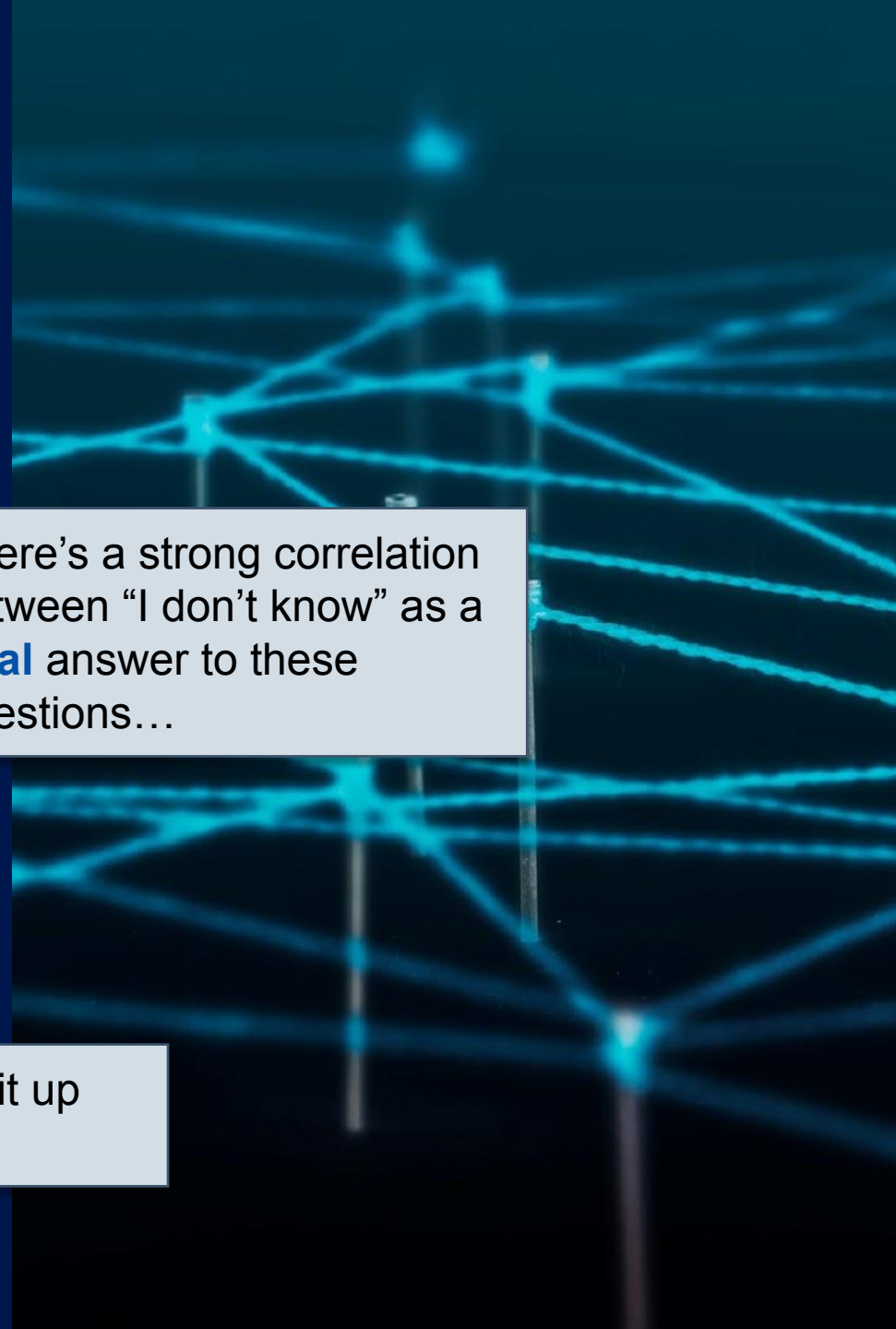
"Listen, eyes… this is brain you're talking to. I'm **the brain**. I process so much data so quickly it would melt your brain. If you had one. Which you don't, because you're the eyes. I don't even listen to you half the time—I already know what you're going to say."

The Worst Enemy When Debugging?

- **Lack of information**
- What do you **want** your code to do?
- What is happening on or before this line of code?
- What is the value of this variable or parameter?
  - What value are you expecting?
- Is it being passed by value or by reference? Which do you **want**?
- Lack of information leads to… "What do I type to make it work?"

There's a strong correlation between "I don't know" as a **final** answer to these questions…

…and then following it up with this question.



Print Debugging

- Sprinkle “print” statements throughout your code.
  - `cout` in C++
  - `Console.WriteLine` in C#
  - `print()` in others
- Time-consuming
- Have to clean it all up later
- Tried and true – been around forever, still helpful

```
void Deallocate(void *ptr)
{
    cout << "Starting deallocation function\n";
    auto loc = std::find(allocs.begin(),
                allocs.end(), ptr);

    unsigned int index = loc - allocs.begin();
    cout << "Value of index: " << index;
    if (index < allocs.size())
    {
        allocs[index] = nullptr;
        dealloCount++;
        std::free(ptr);
        cout << "Deallocation happened!";
    }
}
```

Using a Debugger

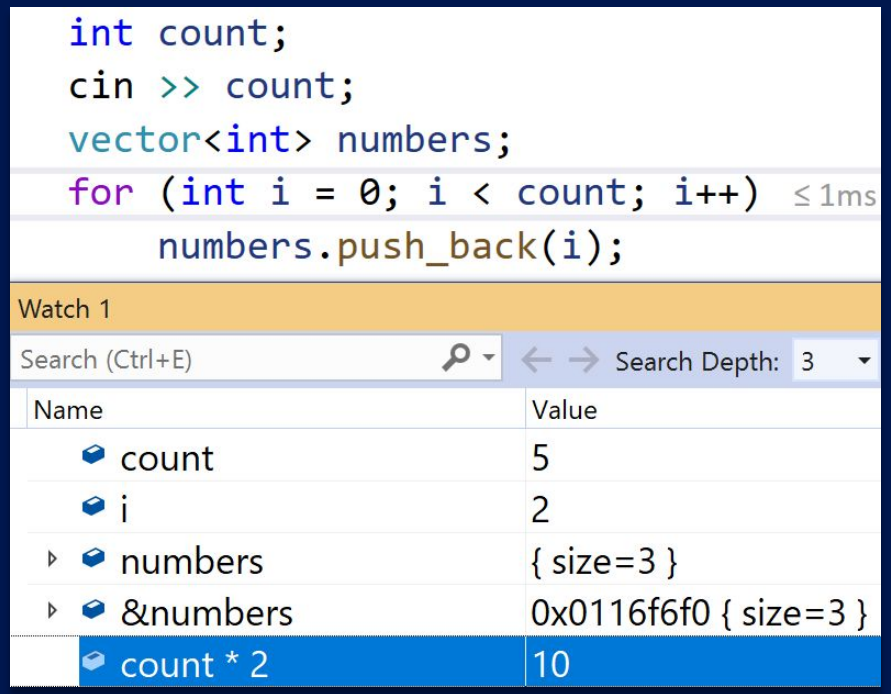
- Intimidating at first, but very powerful
- Much of the power is in a few simple operations:

Set Breakpoints	Step Over	Step Into	Step Out
Stop your program here, pause for a moment	Execute this line of code	Jump into this function for closer inspection	Exit a function and return to calling function

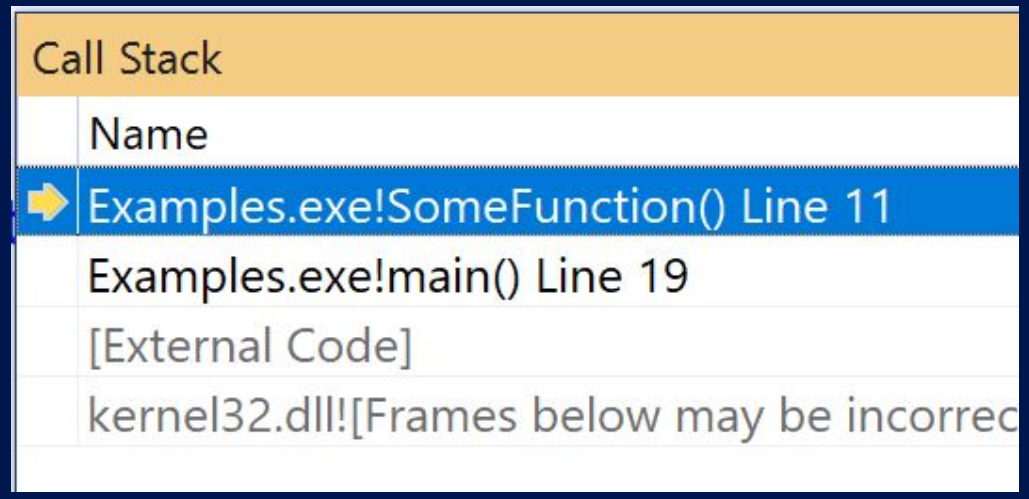
- Every debugger should have these, and learning to use them is vital.

Debugger Information Windows

- Debuggers have other windows to provide information.
- **Watch**: Show information about variables
- **Call Stack**: See what functions are awaiting execution



These two windows are useful in almost every debugging situation.



Learn. To. Use. A. Debugger.

- No matter your IDE of choice
  (Visual Studio, CLion, Eclipse, etc…)
- At the very least:
  - How do you start (and stop) debugging?
  - How do you set/disable a break point?
  - How do you step through your code one line at a time?
  - How do you view variables (and their values) in the current scope?
  - How can you view the call stack?
- Most IDES have a “Debug” menu that contains some or all of these—start exploring!
- Do it. Do it today. The sooner you start, the better.

Words of wisdom:

1. The fastest way to debug is to write perfect code!
2. The second fastest way is to use the debugger.

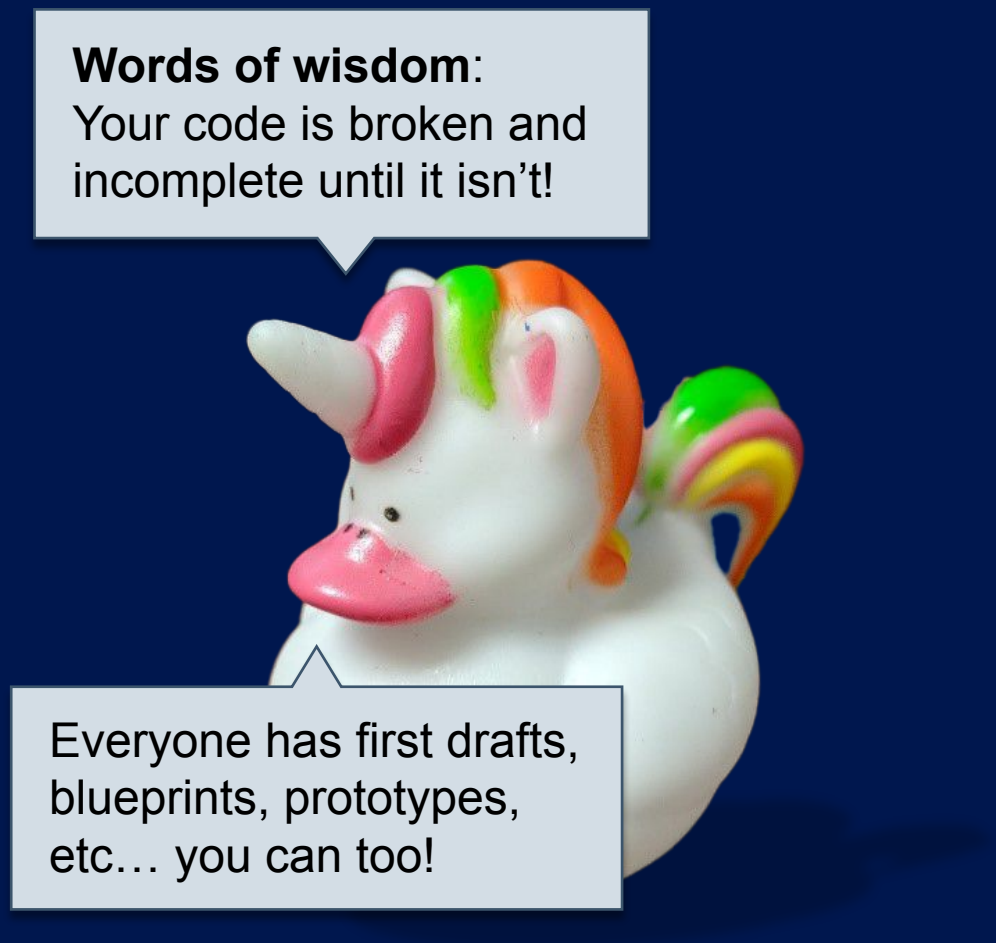
Debugging Is Just Asking More Questions

```
int main(void)
{
    SomeClass x;

    // 1. How can you set a breakpoint here?
    // 2. How can you step into Foo()?
    // 3. What is the value of “someValue” after Foo()?
    int someValue = Foo();

    // 4. How can you step over (or into) this function?
    x.Bar();

    return 0;
}
```

The process of answering such questions in this example is the same as it would be in more complex scenarios.

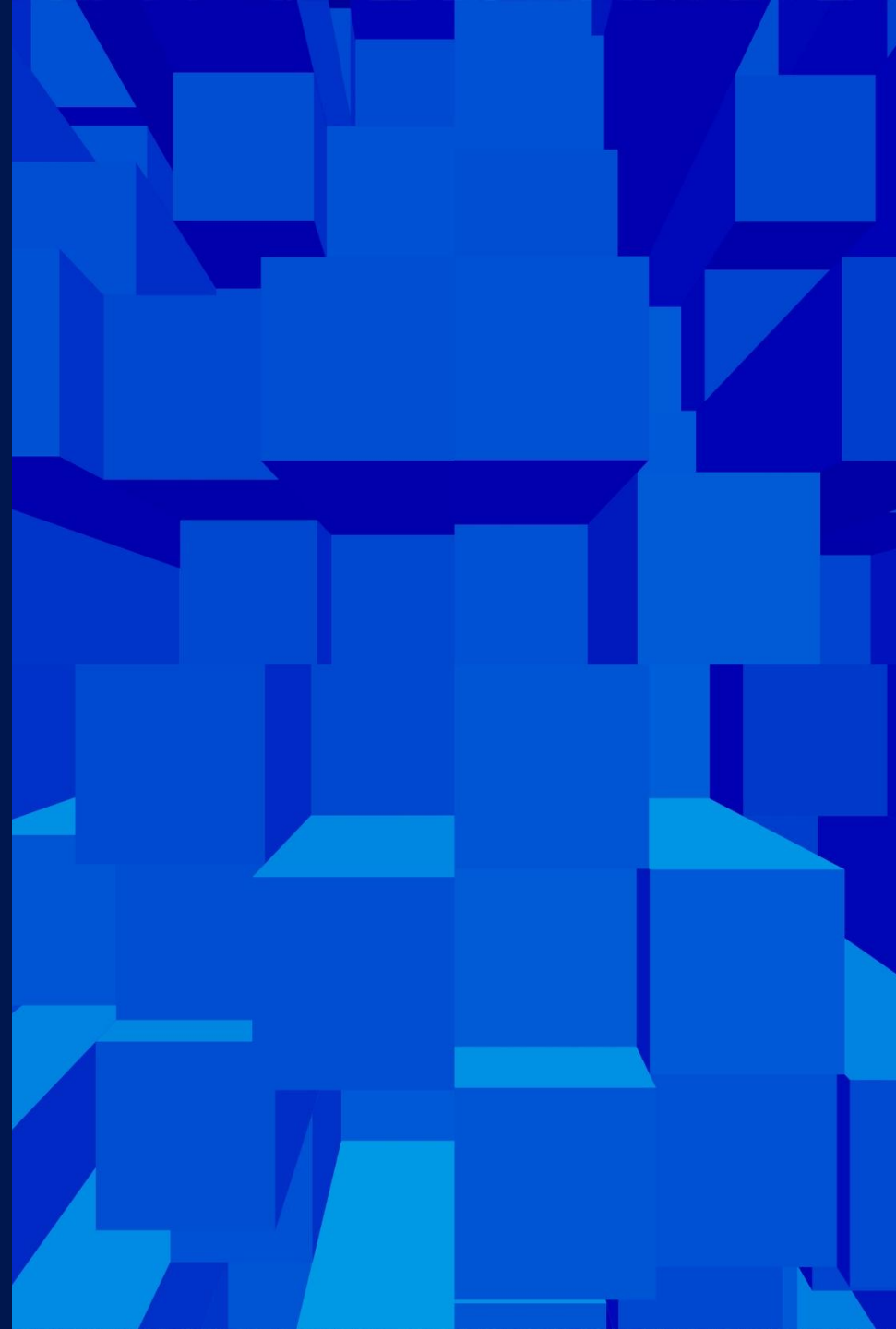
Another Thing…

- It’s okay to write code that isn’t perfect!
- Don’t be afraid to write code that’s:
  - Busted
  - Incomplete
  - Obviously inefficient
  - Ugly
  - On the way to writing better code



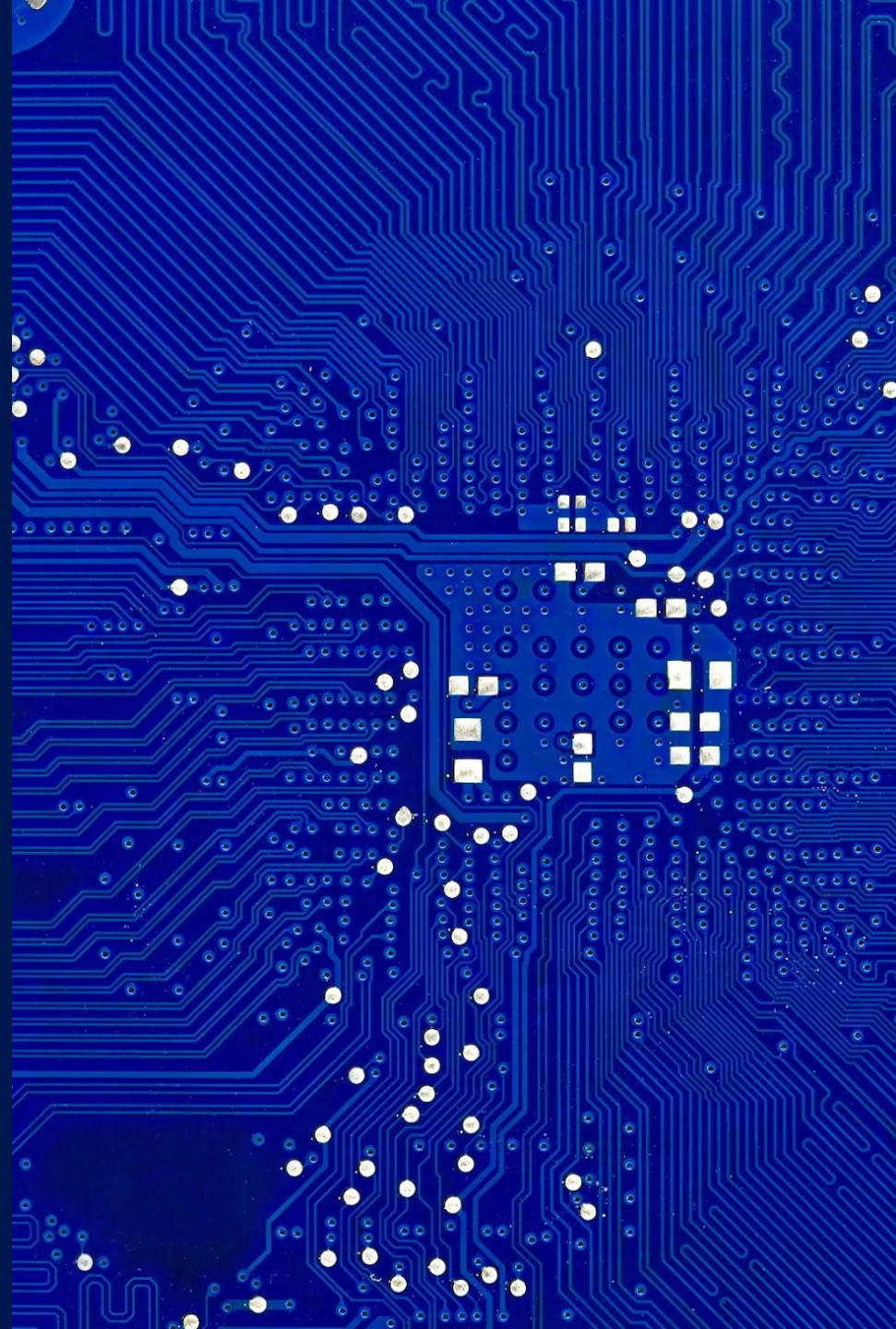
In Other Words

- Solve the problem **first**—figure out how things work
- Later, you can solve the problem:
  - Elegantly
  - Efficiently
  - Ingeniously
  - Etc…
- Don’t get stuck repeating the first step over and over until it’s perfect!



You Will Screw It Up Sometimes

- And that’s okay.
- As long as you fix that mess you made, it’s all good!
- Everyone who has ever programmed has screwed something up.
- It’s not about being flawless.
- It’s about identifying problems, the source of those problems, and then devising solutions.
- **Then**…implementing and verifying those solutions
- Then…do it all over again, for the next problem!



Go Easy on Yourself

- Programming is hard.
- You're learning a new language.
  - One that, if you mispronounce something, doesn't say "I'm sorry, could you please clarify that?"
  - Instead, the response you get is:

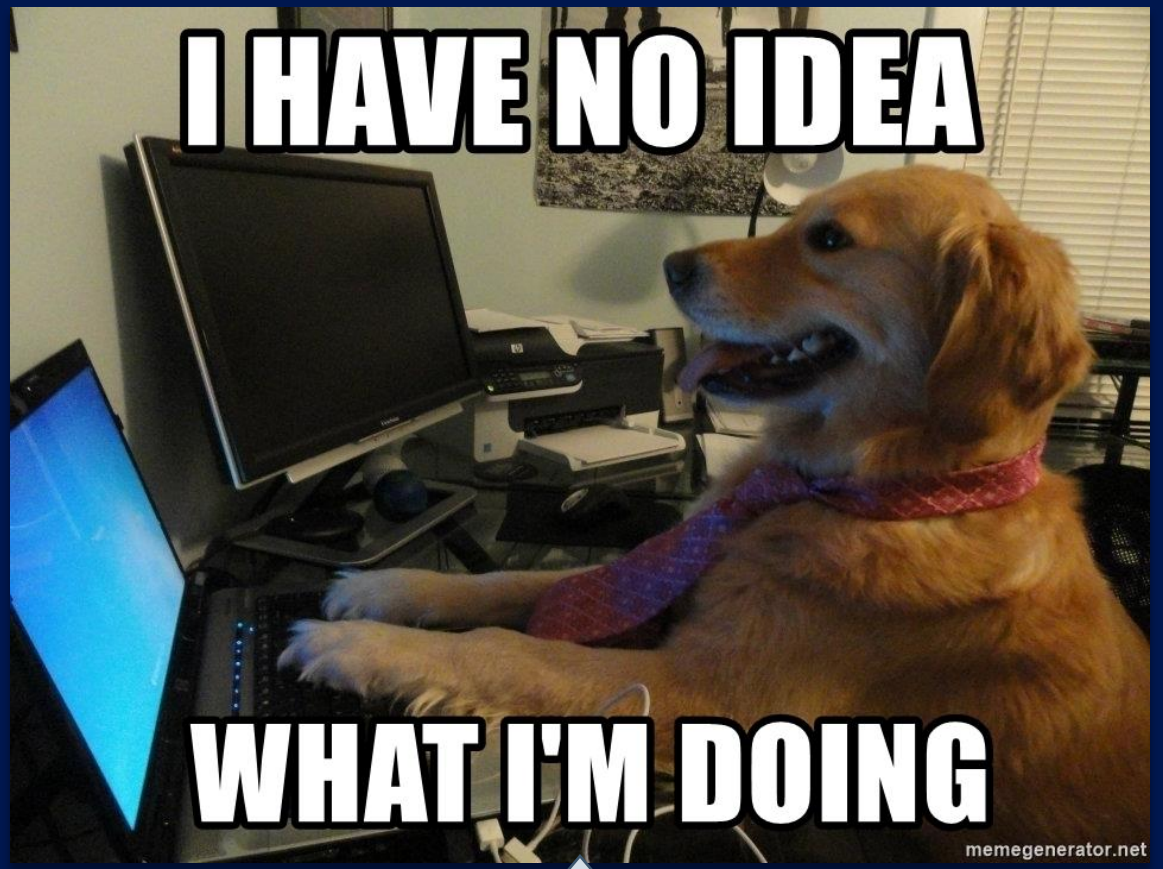
The point of all this:
It's hard to learn this stuff. Give yourself a pass if you don't get it right away.

```
SEG_FAULT mem address 0x001A394C
unresolved external symbol "void __cdecl Foo(void)" (?Foo@@YAXXZ)
0xBAADF00D
0xFEEEFEEE
0xCDCDCDCD
0xDEADDEAD
```

Not making these up.

Give Yourself Time

- No one learns programming in a day.
- Some concepts take a long time to “click”.
- 10 years from now? No one will care:
  - If it took you a month to really understand what a pointer was.
  - If it took you 6 days to complete an assignment while your classmate got it done in 4.
- As children we take a long time to learn how to walk, talk, tie our shoes, etc…
- Learning to program? Much harder than all that!



Some days are like this.
And that’s okay!

One Last Thing

"There is nothing noble in being superior to your fellow man; true nobility is being superior to your former self."

- Ernest Hemingway

That's the magic right there.

Are you better today than you were yesterday?

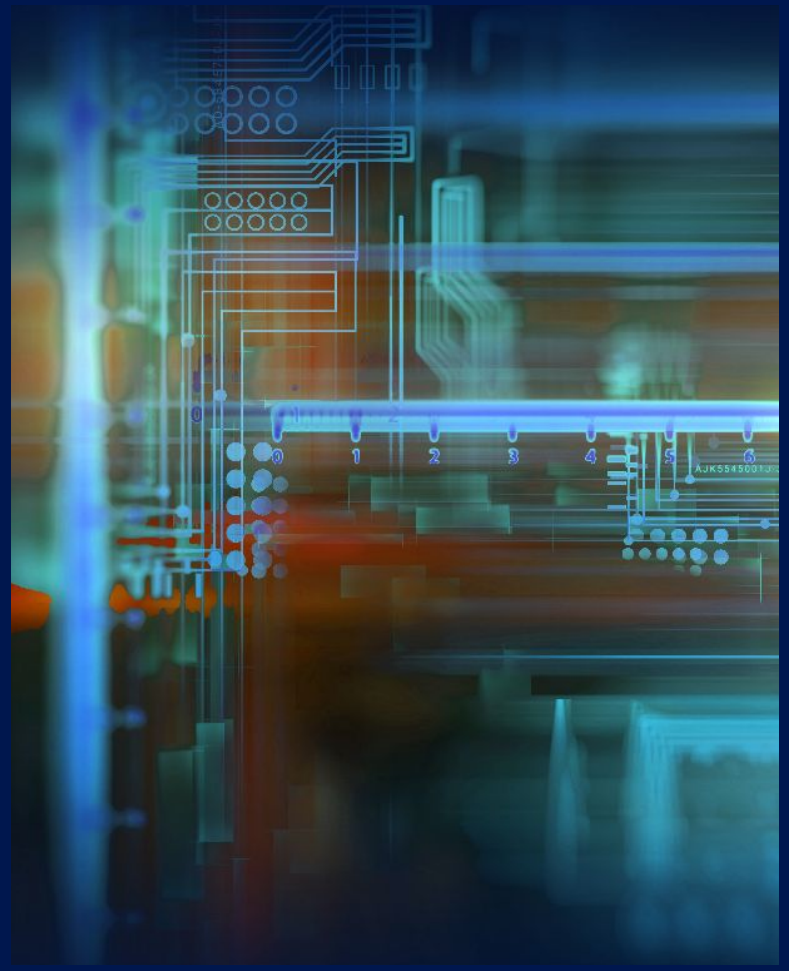
There's always someone who knows a little more, is a little faster, has a little more experience… don't worry about them.

Focus on improving yourself, and you'll be just fine.

Recap

- Problem solving is about **finding** answers, not just **having** them.
- To solve problems effectively, we need **information**.
- To get information, we have to **ask questions** (and answer them).
- **Reading code** is just as important as writing code—reading code gives you information!
- There are many debugging techniques to get us more information
  - Using a debugger
  - Print debugging
  - Rubber duck debugging (quack! quack!)
- They all have a place, and learning to use them is critical

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.