



# Art of Debugging

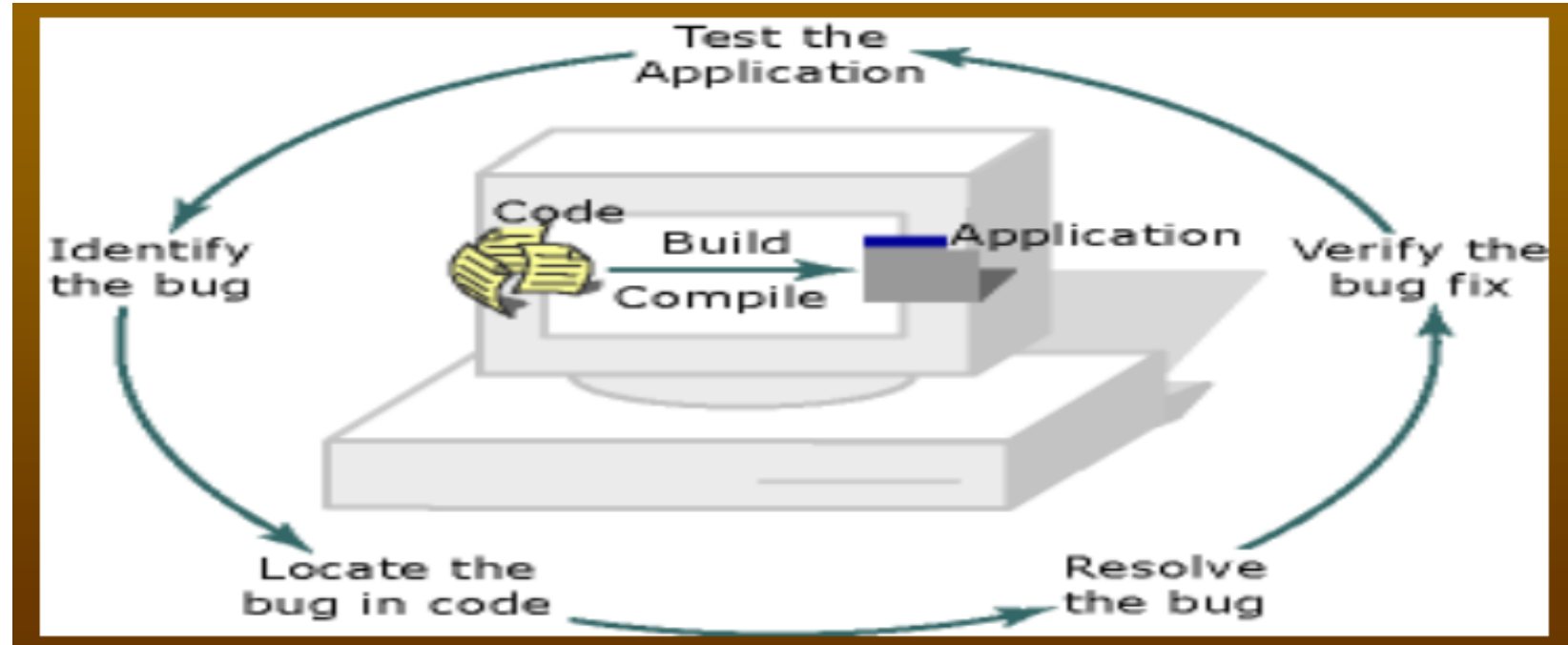
---

YESODA BHARGAVA

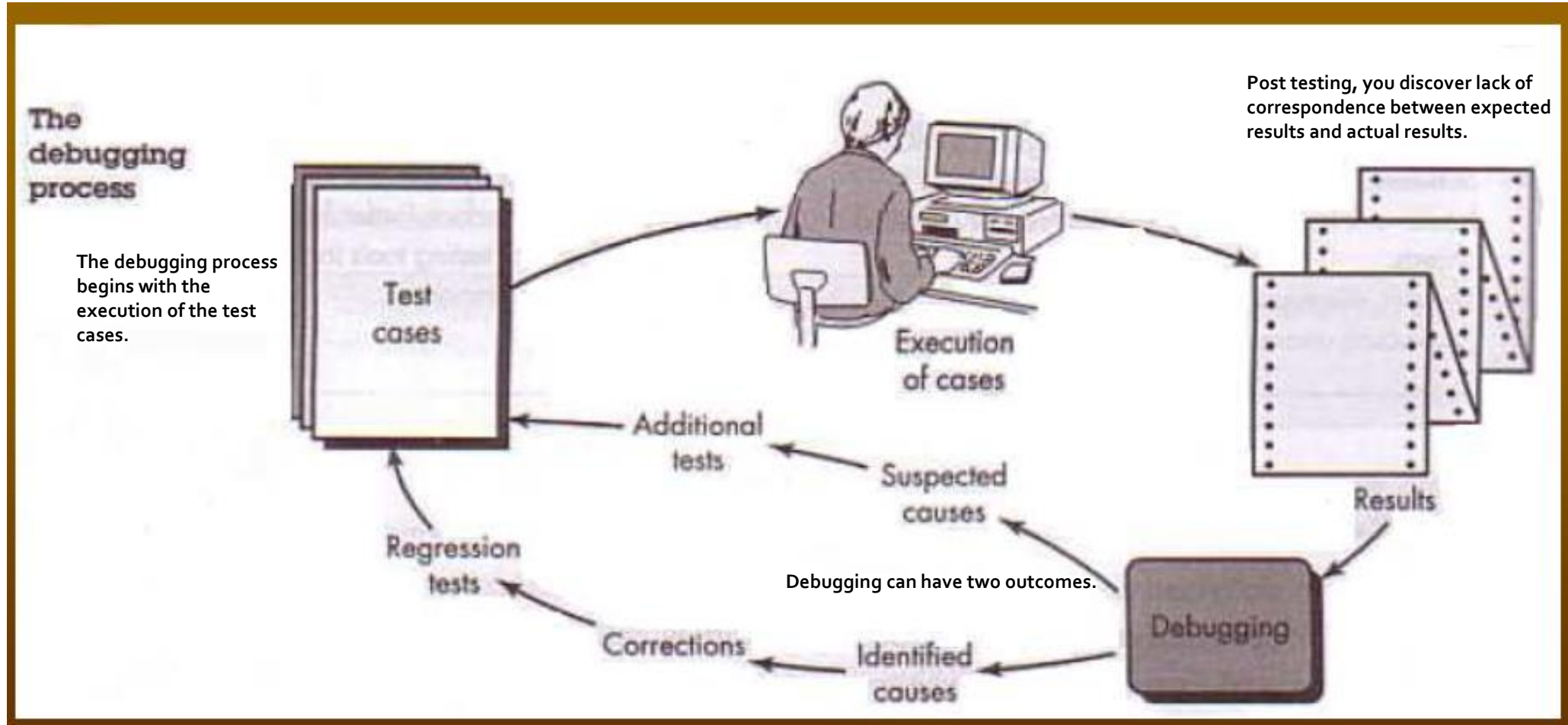
# Introduction

---

- I. Art of Debugging is a process of locating and correcting the causes of known errors.
- II. Locating the error consists of 95% of the activity.



# Debugging Process





What is the  
difference between  
testing and  
debugging?

**Testing** is the process of executing a program or system with the aim of finding errors or bugs.

**Debugging** is the process of correcting these errors or bugs (found during testing).

- I. Debugging is the one aspect of the software production process that programmers enjoy the least.
- II. Your ego may get in the way  
Debugging confirms that programmers are not perfect  
They have committed an error in either the design or the coding of the program.
- III. You may run out of steam  
Of all the software development activities, debugging is the most mentally-taxing activity  
Debugging usually is performed under a tremendous amount of organizational or self-induced pressure to fix the problem as quickly as possible.
- IV. You may lose your way  
Debugging can be mentally taxing because the error you have found could occur in virtually any statement within the program

# What is a bug exactly?

---

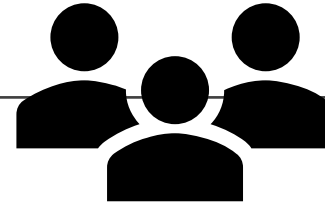
- I. A fault in a program which causes the program to perform in an unintended or unanticipated manner.
- II. A program that contains a large number of bugs, and/or bugs that seriously interfere with its functionality is said to be **buggy**.
- III. Reports detailing bugs in a program are commonly known as **bug reports**, fault report, problem reports, defect reports etc.

Did you know that the first actual #computer “bug” was a dead moth which was stuck in a Harvard Mark II computer in 1947?



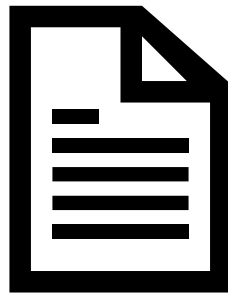


# Causes behind bugs



Mis-understanding of user needs

Unfinished/not detailed Requirements



Lack of documentation



Not enough software testing

Logical errors in design document



# Bug Lifecycle

---

- I. States of a bug:
  - I. New
  - II. Open
  - III. Assign : to developer/team.
  - IV. Test: Test the bug, replicate it.
  - V. Verified : Developer makes necessary code changes and verifies the changes.
  - VI. Deferred : expected to be fixed in next releases. Priority releases.
  - VII. Reopened : if bug continues to exist even after it is fixed by the developer.
  - VIII. Rejected : the bug is not really a bug but a particular behaviour of the system. The developer can reject the bug and explain the tester.
  - IX. Closed : bug is fixed, tested by the tester. The tester closes the issue.

# Common Types of Computer Bugs

## I. Maths bugs

- I. Division by zero
- II. Arithmetic overflow or underflow

## II. Logic bugs

- I. Infinite loops and infinite recursion

## III. Syntax bugs

- I. Use of the wrong operator, such as performing assignment instead of equality.

## IV. Resource bugs

- I. Using an un-initialized variable
- II. Resource leaks, where a finite system resource such as memory or file handles are exhausted by repeated allocation without release

## V. Team working bugs

- I. Comments out of date or incorrect.
- II. Differences between documentation and the actual product.

# Types of Bugs/Defects

- I. Defects detected by the tester are classified into categories by the nature of the defect.
- II. **Showstopper (X)** : The impact of the defect is severe and the system cannot go into the production environment without resolving the defect since an interim solution may not be available.
- III. **Critical (C)** : The impact of the defect is severe, however an interim solution is available. The defect should not hinder the test process in any way.
- IV. **Non Critical (N)** : All defects that are not in the X or C category are deemed to be in the N category. These are also the defects that could potentially be resolved via documentation and user training. These can be GUI defects.

# Showstopper bugs example



Suppose you are doing an online payment.

## Login

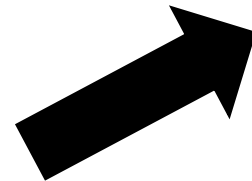
Email

Password

[Forgot your password?](#)

[Log in](#)

You enter correct login details. BUT



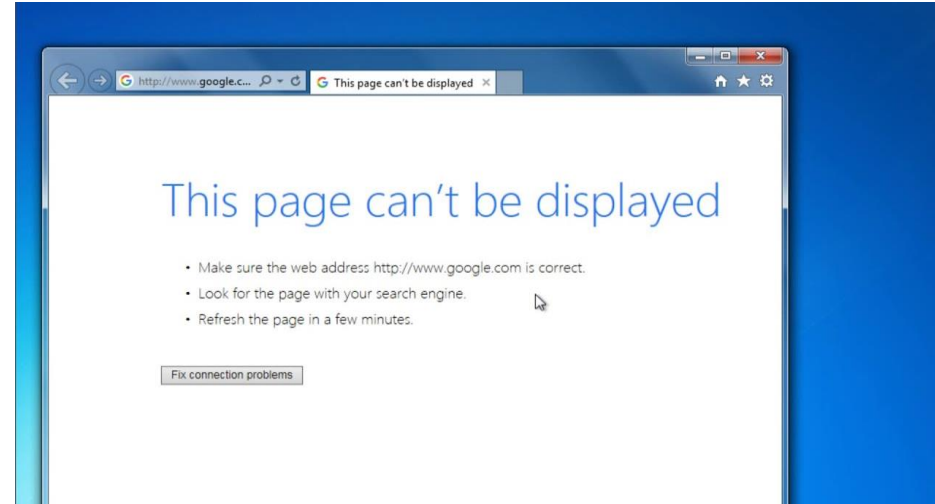
Invalid email or password.

Email

Password

[Forgot your password?](#)

[Log in](#)



After payment, you see this message.

These are highest  
priority bugs (P1).

# One more example..

---



For Instance - User is testing a Online Shopping site. After Selecting the required items into the Shopping Cart, he clicks on Order Button.

Expected Result : User Should be Redirected to Order and Billing Page

Actual Result : Application displays a webServer error and displays a http Page not found error page.

A good [article](#) to read related to showstopper bug handling scenario in a company.

**Here's a real life scenario, let's see how many of you can relate to it:**

*It is the middle of the day on a regular Tuesday afternoon. You are the QA Manager of a company developing an Enterprise Application, and last week your team released a minor version that included the fixes from all the patches of the last two months plus four minor features that were requested by product marketing in order to "close some pretty important deals".*

*All of a sudden the phone rings and it is your R&D director "inviting" you to an urgent meeting in his room. You arrive to find there the R&D director, together with his development team managers, the product marketing manager, and the support team leader in charge of your product who is standing next to the whiteboard...*

*As you sit down the support team leader tells all of you about an urgent showstopper that was released in last week's version and that is expected to affect about a third of the companies who install this upgrade.*

**What would you and your team do in this situation?**



# Critical defect

---

- I. Normally when a feature is not usable as it's supposed to be, due to a program defect, or that new code has to be written or sometimes even because some environmental problem has to be handled through the code, a defect may qualify as a critical defect (Priority 2).
- II. This is the defect or issue which should be resolved before the release is made. These defects should be resolved once the Showstopper issues are solved.

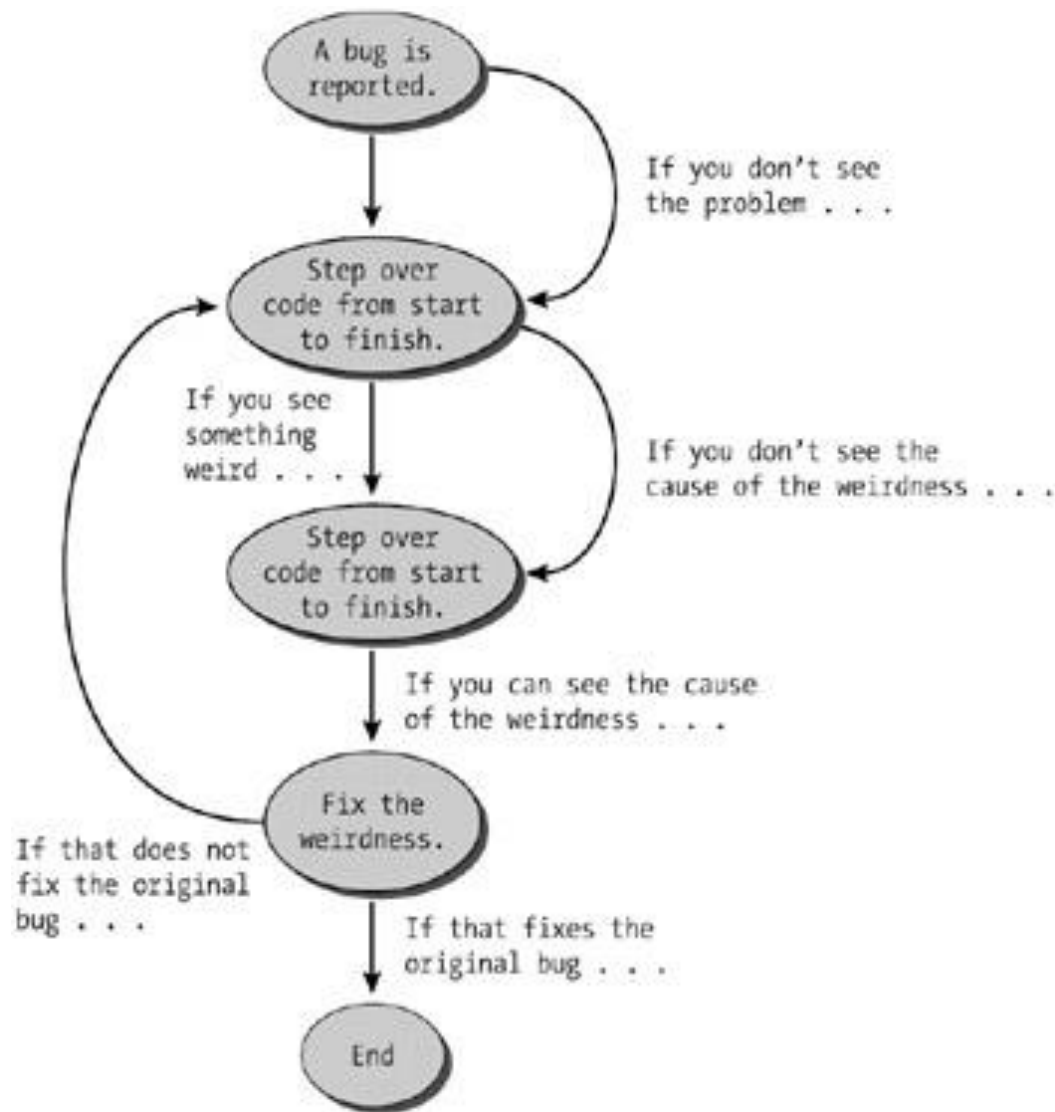
# Introduction

---

- I. Commonly used debugging methods include
  - I. Brute Force
  - II. Induction
  - III. Deduction
  - IV. Backtracking.

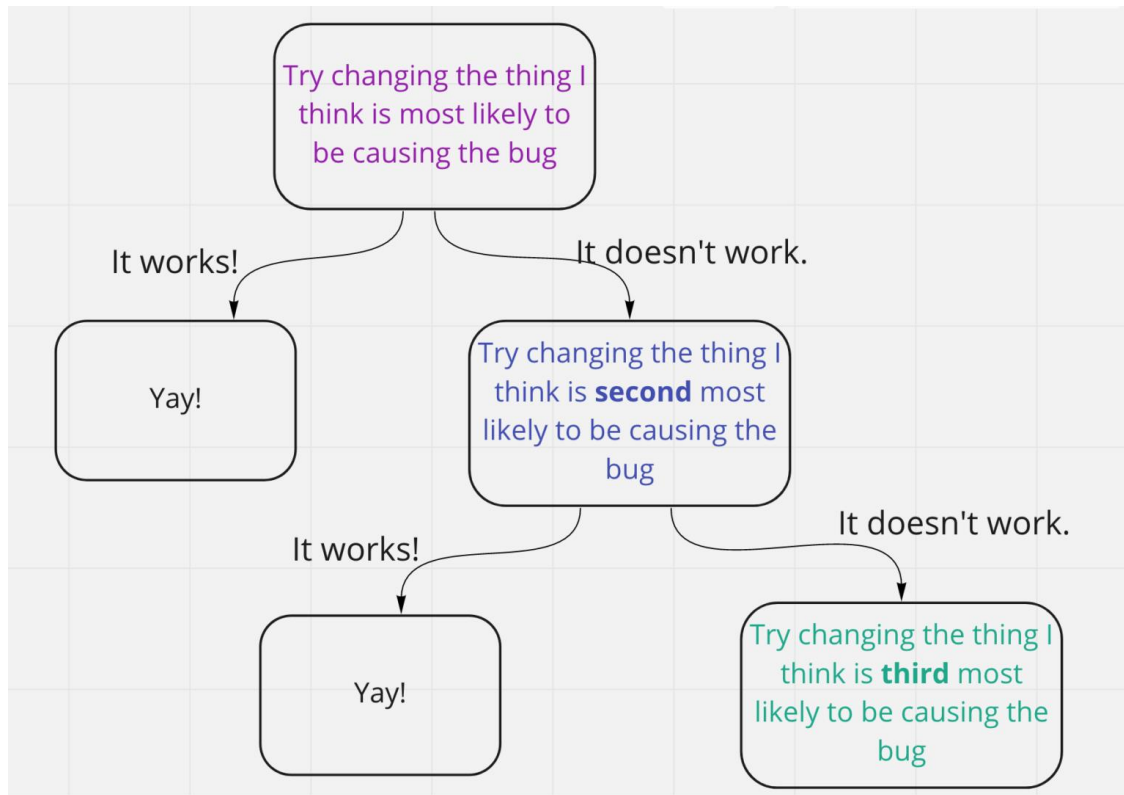
# Debugging by brute force

- I. The brute-force methods are characterized by either debugging with a memory dump; scattering print statements throughout the program, or debugging with automated debugging tools.
- II. Least efficient for isolating the cause of the software error.
- III. Often the least successful. This involves the developer manually searching through stack-traces, memory-dumps, log files, and so on, for traces of the error.
- IV. Extra output statements, in addition to break points, are often added to the code in order to examine what the software is doing at every step.
- V. The biggest problem with the brute-force methods is that they ignore the most powerful debugging tool in existence, a well trained and disciplined human brain.
- VI. In terms of the speed and accuracy of finding the error, people who use their brains rather than a set of "aids" seem to exhibit superior performance.
- VII. Hence, the use of brute-force methods is recommended only when all other methods fail or as a supplement to (not a substitute for) the thought processes described later.



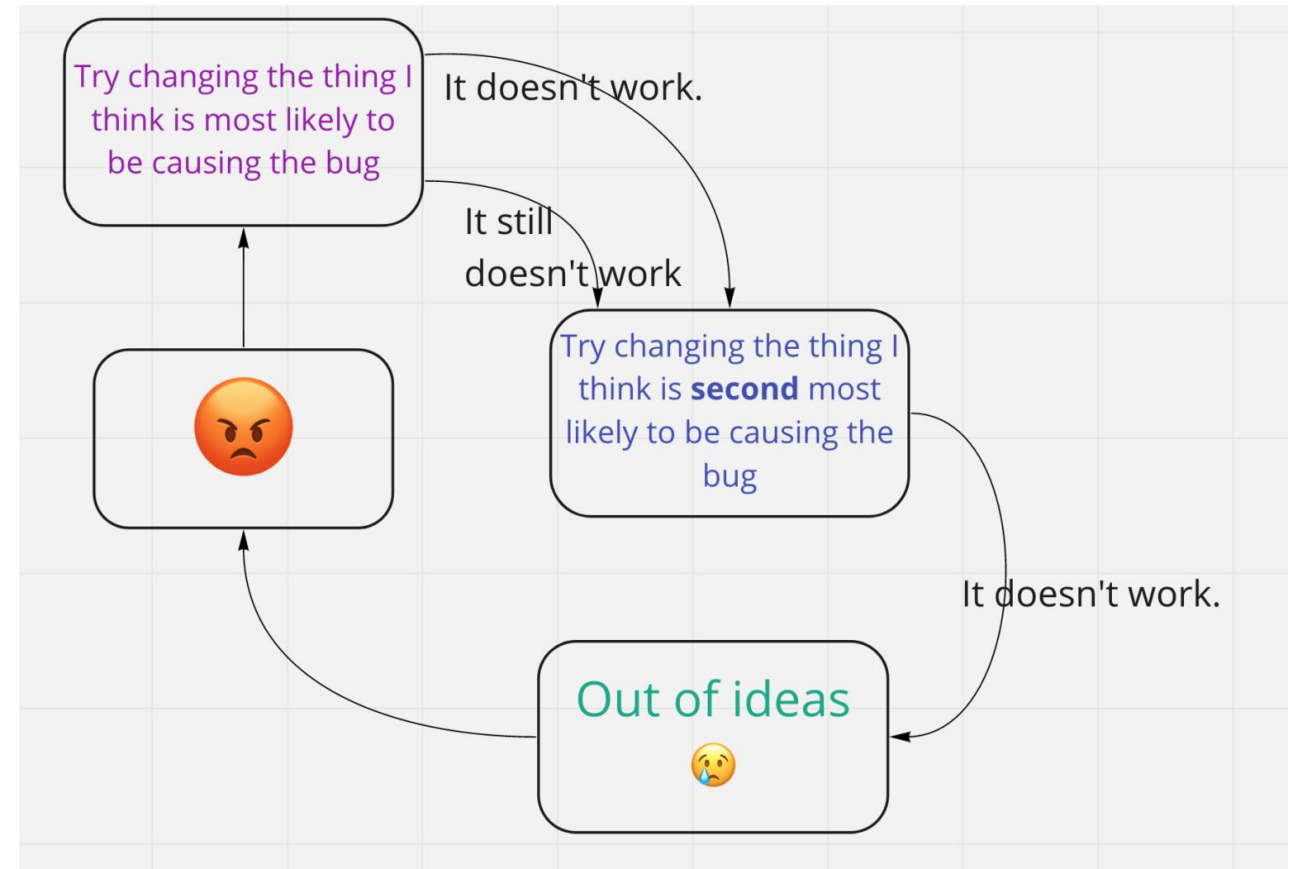
Brute force approach to debugging.

- You, the debugger steps across the code from start to finish until you notice something odd.
- "Wait, that variable looks wrong—how did that happen?" Then you start over and step through the code again, looking for the source of that oddity.
- When you reach the end and don't see the source of the problem, you start over and step through the code once more.
- Repeat over and over and over again until finally, "Oh, I see now—the problem is such-and-such. Now how do I fix that?"
- Rather than encouraging you to think about the problem, it is largely a hit-or-miss method.



← How we think...!

But sometimes...



# Debugging by Induction

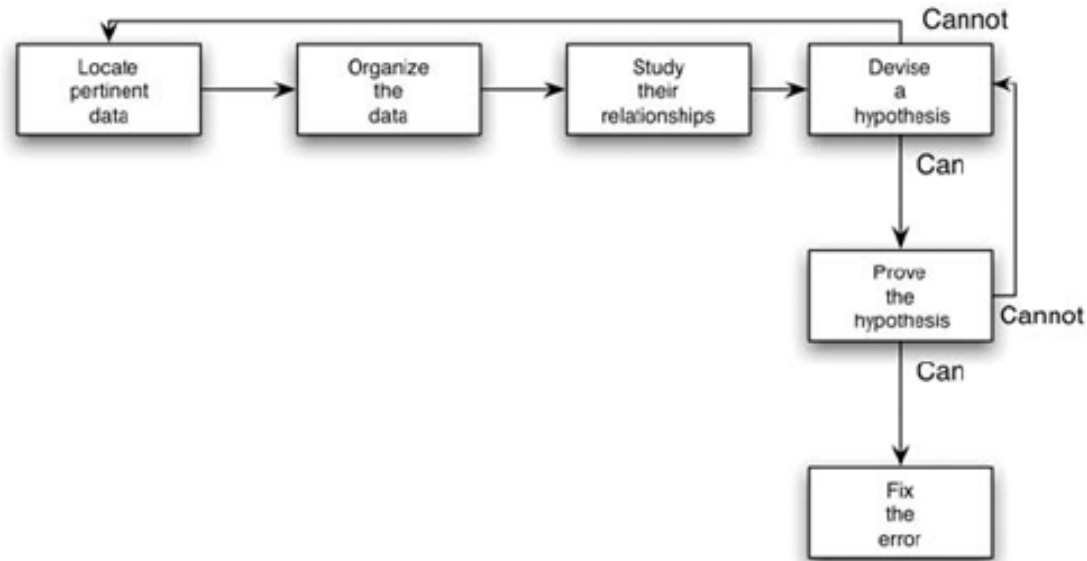
---

I. Involves the following steps:

- I. **Collect the available information:** Enumerate known facts about the observed failure and known facts concerning successful test cases. What are the observed symptoms? When did the error occur? How does the failure case differ from the successful cases?
- II. **Look for patterns:** Examine the collected information for conditions that differentiate the failure case from successful cases.
- III. **Form one or more hypothesis:** Derive one or more hypothesis from the observed relationships. If no hypotheses are apparent, re-examine the available information and collect additional information. If several hypothesis emerge, rank them in order of most likely to least likely.
- IV. **Prove or disprove each hypothesis.** Do not proceed further until this step is complete.
- V. **Implement appropriate corrections.**
- VI. **Verify the correction :** Rerun the failure case to be sure that the fix corrects the observed symptom. If the fix is not successful, go to step I.



**Induction approach** : Processing from the particulars to general.



Locate the data:

- A major mistake debuggers make is failing to take account of all available data or symptoms about the problem.
- The first step is the enumeration of all you know about what the program did correctly and what it did incorrectly—the symptoms that led you to believe there was an error.
- Additional valuable clues are provided by similar, but different, test cases that do not cause the symptoms to appear.
- Organize the data to locate the patterns in the error.

Induction method : Table used to structure the data about the bug.

?	IS	IS NOT
WHAT		
WHERE		
WHEN		
TO WHAT EXTENT		

?	IS	IS NOT
WHAT	The median printed in report 3 is incorrect.	The calculation of mean or standard deviation.
WHERE	Only on report 3.	On the other reports, students grades seem to be calculated correctly.
WHEN	Occurred in test run using 51 students.	Did not occur in test run for 2 and 200 students.
TO WHAT EXTENT	The median printed was 26. It also occurred in test run using one student, the median printed was 1.	

# Debugging by Deduction

---

- Deduction -- Proceed from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion (the location of the error)

- Murder case

- Induction -- Start with set of clues and induce who is the suspect

- Deduction -- Start with set of suspects and, by process of elimination using the clues, deduce who is the suspect

### **1. Enumerate the possible causes or hypotheses**

Develop a list of all conceivable causes of the error

### **2. Use the data to eliminate possible causes**

Carefully examine all of the data, particularly by looking for contradictions, and try to eliminate all but one of the possible causes

If all are eliminated, you need more data gained from additional test cases to devise new theories

If more than one possible cause remains, select the most probable cause -- the prime hypothesis -- first

### **3. Refine the remaining hypothesis**

Possible cause at this point might be correct, but it is unlikely to be specific enough to pinpoint the error

Use the available clues to refine the theory

For example, you might start with the idea that "there is an error in handling the last transaction in the file" and refine it to "the last transaction in the buffer is overlaid with the end-of-file indicator"

### **4. Prove the remaining hypothesis**

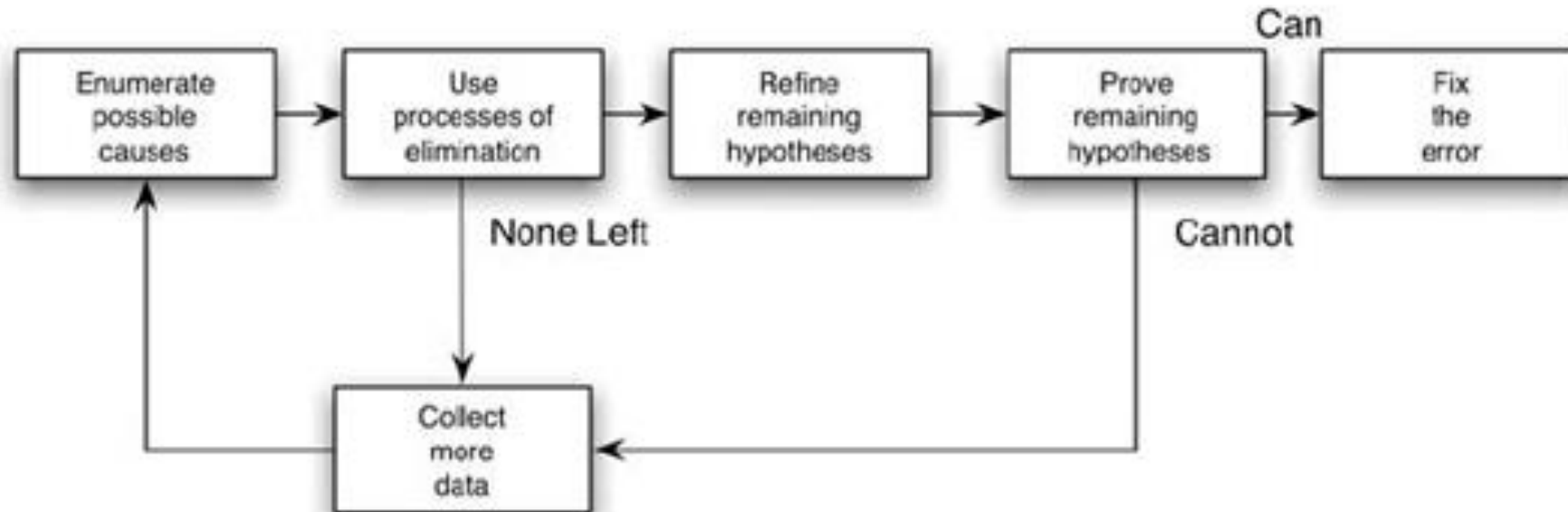
Compare it to the original clues or data, making sure that this hypothesis completely explains the existence of the clues

If it does not, the hypothesis is invalid, the hypothesis is incomplete, or multiple errors are present

### **5. Fix the problem**

Then ... perform regression testing to ensure your bug fix did not create problems in other program areas

## The deductive debugging process





# Debugging by Backtracking

---

- I. An effective method for locating errors in small programs is to backtrack the incorrect results through the logic of the program until you find the point where the logic went astray.
- II. You are looking for the location in the program between the point where the state of the program was what it was expected to be and the first point where the state of the program was not what it was expected to be.
- III. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases, becomes unmanageably large for complex programs.
- IV. "Everything appears to work fine until WHEN?"

VI.

- 
- I. While debugging you must be in two states:
    - I. Either you are testing a theory.
    - II. Or you are gathering data to come up with a suitable theory.
  - II. You must have a clear goal in mind.
  - III. DO NOT FUMBLE AROUND RANDOMLY.
  - IV. Every test you run should tell you something you can act on. Do not experiment just like that.

# Example Bug

---

- I. The program sometimes (but not always) saves the user's documents to the root directory when it should instead save them to the user's home directory.
- II. What could be the possible cause?
- III. Maybe the *get user name* function fails to identify the user for some reason, so we don't know which home directory to use, and therefore fall back to the root directory. Possibly that funky code that gets the user's name from his or her web browser connection is the culprit?
- IV. Maybe the *write document* function tries to write to the correct home directory, but fails for some reason, so we fall back to the root directory. Possibly we're not checking whether we have write permissions over that home directory?
- V. Maybe the *save legacy document format* function is screwing up somehow. That function is very integral to this feature, and a coworker recently added a lot of tricky new code, so possibly the bug is somewhere in there?
- VI. The actual cause was permission problem (Second guess).

# Debugging by Testing

---

- I. Consider two types of test cases:
  - I. test cases for testing, where the purpose of the test cases is to expose a previously undetected error,
  - II. test cases for debugging, where the purpose is to provide information useful in locating a suspected error.
- II. Test cases for testing tend to be "fat" because you are trying to cover many conditions in a small number of test cases.
- III. Test cases for debugging, on the other hand, are "slim" since you want to cover only a single condition or a few conditions in each test case.
- IV. After a symptom of a suspected error is discovered, you write variants of the original test case to attempt to pinpoint the error.
- V. Such an approach is often used in conjunction with the induction method to obtain information needed to identify the hypothesis or prove a hypothesis. It is also used in conjunction with deduction method to eliminate suspected causes and refine the hypothesis.

# Some Debugging Techniques

---

- I. Traditional debugging techniques require:
  - I. **Diagnostic output statements:** These can be embedded in the source code as specially formatted comment statement that are activated using a special translator option.
  - II. **Snapshot dumps:** A snapshot dump is a machine level representation of the partial or total program state at a particular point in the execution sequence.
  - III. **Trace facility:** A trace facility lists changes in selected state components. Eg. Toast while debugging Android applications.
  - IV. **Traditional break points:** This facility interrupts program execution and transfers control to the programmer's terminal when execution reaches a specified break instruction in the source code.

# Basic debugging tactics

---

## Commenting out the code

If your program is exhibiting erroneous behavior, one way to reduce the amount of code you have to search through is to comment some code out and see if the issue persists. If the issue remains, the commented out code wasn't responsible.

```
1  int main()
2  {
3      getNames(); // ask user to enter a bunch of names
4      doMaintenance(); // do some random stuff
5      sortNames(); // sort them in alphabetical order
6      printNames(); // print the sorted list of names
7
8      return 0;
9  }
```



# Basic debugging tactics

## Commenting out the code

- If your program is exhibiting erroneous behavior, one way to reduce the amount of code you have to search through is to comment some code out and see if the issue persists.
- If the issue remains, the commented out code wasn't responsible.

```
1  int main()
2  {
3      getNames(); // ask user to enter a bunch of names
4      doMaintenance(); // do some random stuff
5      sortNames(); // sort them in alphabetical order
6      printNames(); // print the sorted list of names
7
8      return 0;
9  }
```

We suspect `doMaintenance()` has nothing to do with the problem, so let's comment it out.



```
1  int main()
2  {
3      getNames(); // ask user to enter a bunch of names
4      // doMaintenance(); // do some random stuff
5      sortNames(); // sort them in alphabetical order
6      printNames(); // print the sorted list of names
7
8      return 0;
9  }
```

Two things can happen when commenting the function

```
graph TD; A[Two things can happen when commenting the function] --> B[Problem goes away.]; A --> C[Problem persists.];
```

**Problem goes away.**

Conclusion : Function is the problem.

So, we explore its implementation in the codebase.

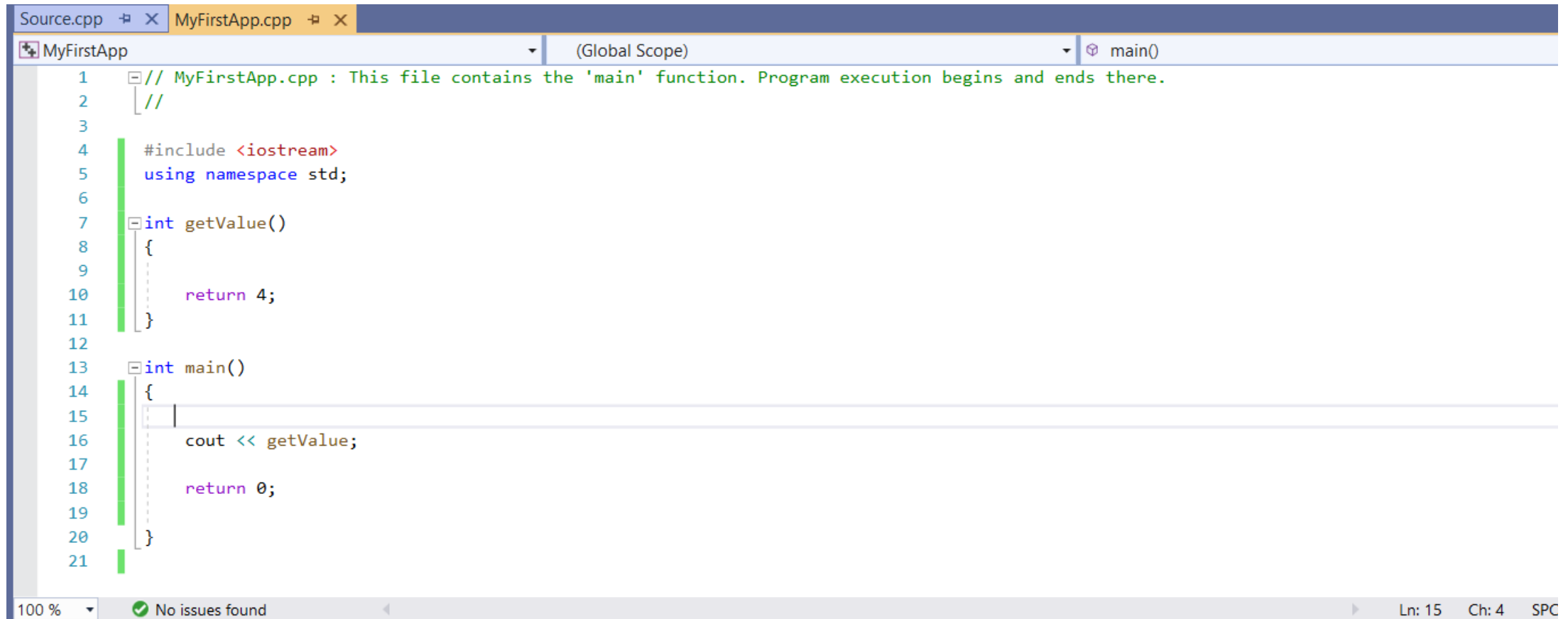
**Problem persists.**

Conclusion : Function is not the problem.  
BUT, now we have excluded a particular case and know It was not causing the issue.

# Basic debugging tactics

## Validating your code flow

- A program may be calling a function multiple times.
- You may place print statements at the top of your function, every time a function is called. Thus, when program runs you know which function is being called.



```
1 // MyFirstApp.cpp : This file contains the 'main' function. Program execution begins and ends there.
2 //
3
4 #include <iostream>
5 using namespace std;
6
7 int getValue()
8 {
9
10     return 4;
11 }
12
13 int main()
14 {
15
16     cout << getValue;
17
18     return 0;
19 }
20
21
```

100 % No issues found Ln: 15 Ch: 4 SPC

# Basic debugging tactics

## Printing Values

- With some types of bugs, the program may be calculating or passing the wrong value.
- We can also output the value of variables (including parameters) or expressions to ensure that they are correct.

In case of Android applications,  
one generally uses Toast feature.

```
1  #include <iostream>
2
3  int add(int x, int y)
4  {
5      return x + y;
6  }
7
8  void printResult(int z)
9  {
10     std::cout << "The answer is: " << z << '\n';
11 }
12
13 int getUserInput()
14 {
15     std::cout << "Enter a number: ";
16     int x{};
17     std::cin >> x;
18     return x;
19 }
20
21 int main()
22 {
23     int x{ getUserInput() };
24     int y{ getUserInput() };
25
26     std::cout << x << " + " << y << '\n';
27
28     int z= add(x, 5) ;
29     printResult(z);
30
31     return 0;
32 }
```

# Cons of using print statements while debugging

---

1. Debug statement can clutter your code.
2. Debug statements can clutter the output of your program .
3. Debug statements must be removed after you're done with debugging, which makes them nonreusable.
4. Debugging statement cause you to both add and remove through code modification, which may introduce new bugs.

# Infinite Recursion

---

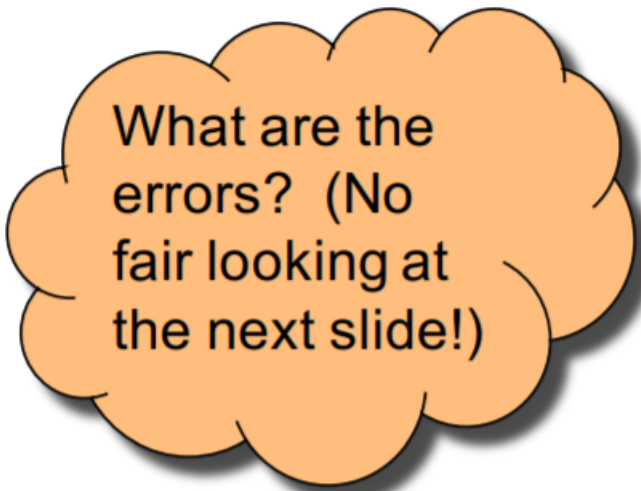
- I. *Maximum recursion depth exceeded* error.
- II. If you suspect a methods/function is causing an infinite recursion, make sure there is a base case.
- III. If there is a base case, but the program is not reaching it. Then what do you do?
- IV. Print
- V. If the program/parameters do not seem to be moving towards the base case, try to find why.

# Things to Remember while Debugging

# Understand the error messages!

---

```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0.
int main(void)
{ printf("hello, world\n");
  return 0;
}
```

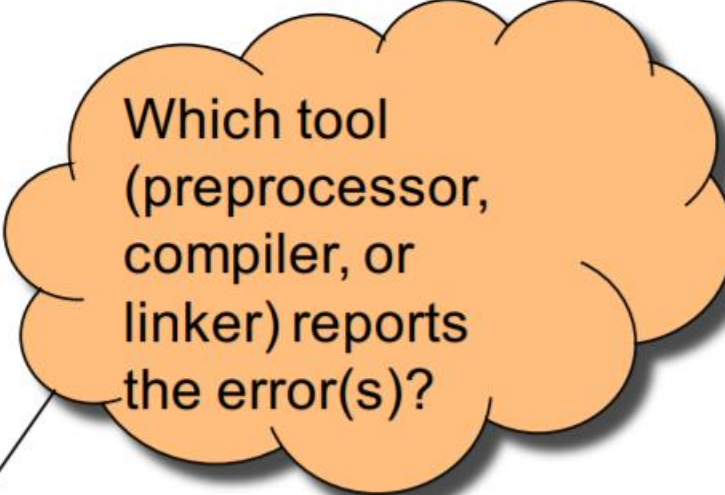


What are the errors? (No fair looking at the next slide!)



# Understand the error messages!

```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0.
int main(void)
{ printf("hello, world\n");
  return 0;
}
```

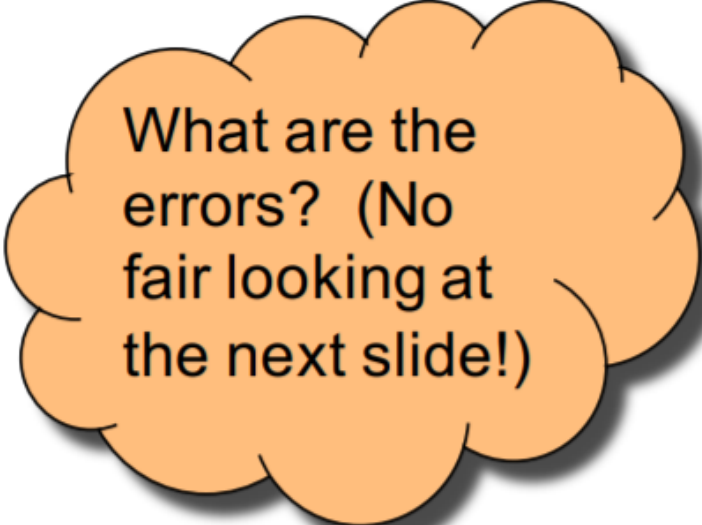


Which tool  
(preprocessor,  
compiler, or  
linker) reports  
the error(s)?

```
$ gcc217 hello.c -o hello
hello.c:1:20: error: stdio.h: No such file or
directory
hello.c:2:1: error: unterminated comment
hello.c:7: warning: ISO C forbids an empty
translation unit
```

# Understand the error messages!

```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{ printf("hello, world\n")
  return 0;
}
```



What are the errors? (No fair looking at the next slide!)

# Understand the error messages!

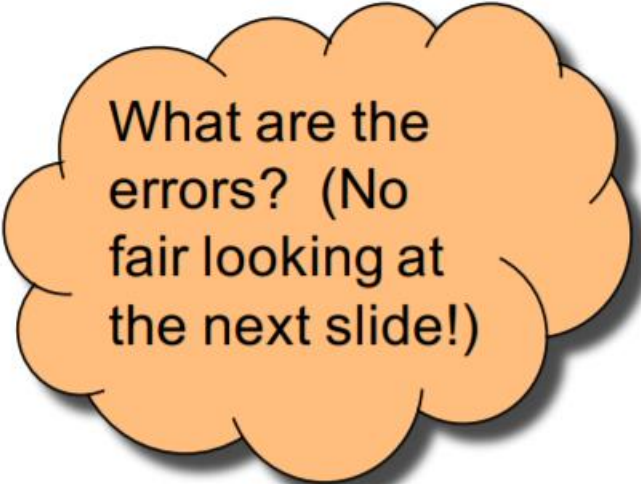
```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{ printf("hello, world\n")
  return 0;
}
```

Which tool  
(preprocessor,  
compiler, or  
linker) reports  
the error?

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:6: error: expected ';' before 'return'
```

# Understand the error messages!

```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{   printf("hello, world\n");
    return 0;
}
```



What are the errors? (No fair looking at the next slide!)

# Understand the error messages!

```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{ printf("hello, world\n")
  return 0;
}
```

Which tool  
(preprocessor,  
compiler, or  
linker) reports  
the error?

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:5: warning: implicit declaration of function
'printf'
/tmp/ccLSPMTR.o: In function `main':
hello.c:(.text+0x1a): undefined reference to `printf'
collect2: ld returned 1 exit status
```

# Run-time Errors

---

- I. So called because the error does not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.
- II. Are much more serious.
  - I. Array index out of bounds.
  - II. Improper use of pointers (\*, \*\*, \*\*\*)
  - III. Accessing null pointers (addresses)
  - IV. Invalid passing of arguments to functions

# Exception Errors

## Name Error

You are trying to use a variable that doesn't exist in the current environment. Remember that local variables are local. You cannot refer to them from outside the function where they are defined.

## Attribute Error

You are trying to access an attribute or method that does not exist.

## Type Error

Mis-match in the variable type and value assigned.  
OR  
Passing wrong number of arguments to a function/method.

## Key Error

You are trying to access an element of a dictionary using a key value that the dictionary does not contain.

## Index Error

The index you are using to access a list, string, or tuple is greater than its length minus one. Immediately before the site of the error, add a print statement to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

# Semantic Errors

---

- I. Semantic errors are problems with a program that runs without producing error messages but doesn't do the right thing.
- II. Example: An expression may not be evaluated in the order you expect, yielding an incorrect result.
- III. Semantic errors are the hardest to debug, because the compiler and the runtime system provide no information about what is wrong.
- IV. Only you know what the program is supposed to do, and only you know that it isn't doing it.
- V. As a first step, make a connection between the program text and the behaviour you are seeing.
- VI. Try to walk through the program with *print* comments.



# My program does not work...

---

- I. Ask yourself the following questions:
  - I. Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.
  - II. Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.
  - III. Is a section of code producing an effect that is not what you expected? If you invoke some functions, read the documentation for the functions you invoke. Try them out by writing simple test cases and checking the results.

- I. In order to program, you need to have a mental model of what the program is supposed to do.
- II. If you write a program that does not work as you expected; chances are the problem is with your mental model and not the program.
- III. To correct the mental model, break down the program into components (usually functions and methods) and test each component independently.
- IV. Post identification of the discrepancy, you can solve it.
- V. Build and test the components as you build the system.

# Example 1

---

- I. You write a big expression but it does not do what you expect.
- II. `getData.Year[i].addVariable (getData.Year(getData.Year[find.Year(i)].popVariable())`
- III. This can be re-written as:
- IV. 

```
year = find.Year(i)
pickedVariable = getData.Year[year].popVariable()
getData.Year[i].addVariable (pickedVariable)
```
- V. The explicit version is easier to read because the variable names provide additional documentation.
- VI. It is easier to debug because you can check the types of the intermediate variables and display their values.

# Function not returning what you expect...

---

- I. If you have a return statement with a complex expression, you don't have a chance to
- II. print the return value before returning. Again, you can use a temporary variable.

For example, instead of:

```
return getData.Year[i].removeRecords()
```

you could write:

```
count = getData.Year[i].removeRecords()  
return count
```

Now you have the opportunity to display the value of `count` before returning.

# Really, really stuck, and need help....

---



Remember Master Shifu

Try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these effects:

- Frustration and/or rage.
- Superstitious beliefs ("the computer hates me") and magical thinking ("the program only works when I wear my hat backward").
- Random-walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

- **Get up and go for a walk.**
- When you are calm, think about the program. What is it doing? What are some possible causes of that behavior?
- When was the last time you had a working program, and what did you do next?
- Sometimes it just takes time to find a bug.
- We often find bugs when we are away from the computer and let our minds wander.
- Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.



# No, I really need help...

---

- I. Even the best programmers get occasionally stuck.
- II. Before you bring someone, make sure you have exhausted all the techniques.
- III. Your program should be as simple as possible, and should be working on the smallest possible input that causes error.
- IV. You should have `print` statements at appropriate places and their output should be comprehensible.
- V. Explain the problem to the person.
- VI. When the bug is found, see what you learnt and why you missed it.

# Experimental Debugging

- I. In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.
- II. Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again.
- III. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program.
- IV. If your hypothesis was wrong, you have to come up with a new one.
- V. As Sherlock Holmes pointed out, When you have eliminated the impossible, whatever remains, however improbable, must be the truth.





# Modern IDEs

---

- I. Modern IDEs have internal debuggers and error is shown during compilation.
- II. Learn to make sense of the errors.
- III. The goal is not to make the program work, but LEARN how to make a program work.

# Defensive Programming

---

- I. Think of defensive driving : driving under the worst-case scenario ((e.g, other drivers violating traffic laws, unexpected events or obstacles, etc)
- II. Similarly, defensive programming means developing code such that it works correctly under the worst-case scenarios from its environment. (Remember robustness?)
- III. For instance, when writing a function, one should assume worst-case inputs to that function, i.e., inputs that are too large, too small, or inputs that violate some property, condition, or invariant; the code should deal with these cases, even if the programmer doesn't expect them to happen under normal circumstances.
- IV. The goal is not to become an expert at fixing bugs, but rather to get better at writing robust, (mostly) error-free programs in the first place.



*As a matter of attitude,  
programmers should not feel  
proud when they fix bugs, but  
rather embarrassed that their  
code had bugs.*

*If there is a bug in the  
program, it is only because the  
programmer made mistakes.*

# Summary

---

- I. Debugging is a learned skill, none of us evolved to be debuggers.
- II. It is not hard to learn.
- III. Largely transferable skill.