How do you debug code?

- Use a debugging tool such as **gdb** or something within the IDE
- Add debugging statements within the code
- Pinpoint where the bug occurs
- Ask for help
- Sleep on it
- Start over
- Give up
Why do bugs exist?

Why don’t we get it right the first time around?!

- We often jump into a project with a “code first” approach
- Once we scale up beyond small units of code, the complexity of the code dramatically increases
- There are too many things that could go wrong; we can never be sure we’ve considered (and handled) every error condition
- Regardless, all bugs are caused by human error....

What types of bugs are there?

- At the coding level, there are many types of common bugs:
  - Uninitialized or improperly initialized variables
  - Rounding or precision errors
  - Library misuse errors (so be sure you read the manual!)
  - Errors in libraries (or other code that you rely upon)
  - Mishandled edge cases
  - Off-by-one errors
  - Incomplete error handling
  - Synchronization (i.e., timing) problems, race conditions, deadlocks
  - Memory leaks
Where are bugs introduced?

- As we scale up, overall system complexity rapidly increases
- Larger-scale bugs are caused by:
  - Miscommunication
  - Misunderstanding of requirements
  - Lack of (good) documentation
  - Improper planning and design of the interfaces between the various smaller units of code
  - Lack of testing or documented test plans
  - Lack of regression testing as code continues to evolve
  - Lack of code quality rules/guidelines; without such rules, most of the last-minute changes are not properly tested at all

Debugging rules

- Understand the system
- Make it fail
- Quit thinking and look
- Divide and conquer
- Change one thing at a time
- Keep an audit trail
- Check the plug
- Get a fresh view
- If you didn’t fix it, it ain’t fixed
Understand the system

- First, “you have to understand how things are supposed to work if you want to figure out why they don’t”
- Take the time to read the manual first, “before all else fails”
- The answer is often deep within the manual or between the lines
- Handle all specific return values, error conditions, etc.
- Nonetheless, don’t trust that the manual has all the answers or is always correct

Make it fail

- Mimic as many conditions as you can, then reproduce the error
- Watch what happens when the error occurs
  - Monitor variable values, memory usage, CPU usage, output, etc.
- Pinpoint what you think the problem is, then make it fail again
- Apply a fix, then question whether you’ve fixed the problem
- Use whatever you did to make it fail in the first place as a test to be sure it works after a fix is applied
Quit thinking and look

- As confident coders, we think a lot; this leads to too much thinking (i.e., hypothesizing) when we’re debugging
- “You can think up thousands of possible reasons for a failure. You can see only the actual cause.”
- Look at program output, variable values, program stack traces, memory, etc. to lead you to where the problem is occurring
- Focus on symptoms, not theories

Divide and conquer

- Narrow the search as efficiently as you can
- Use a successive approximation approach
- The “conquer” portion here is eliminating areas of the code that are not causing the specific error you are debugging
- If you find new (seemingly unrelated) bugs along the way, systematically fix them, then keep going
- Be careful not to “fix” too much; i.e., too much refactoring will more likely introduce new bugs instead of fixing the one you’re looking for
Change one thing at a time

- Fix one bug at a time
- Change one parameter of your test at a time
- If you fix or refactor something along the way, you are now obligated to test everything again!
- What’s changed since everything last worked?

Keep an audit trail

- Why is everything suddenly working?!?
- Or... why is everything suddenly broken?!
- Document what you did, in what order, and what the results were
- Be as detailed as you can be, i.e., include as much information as you can, including timestamps, sequences of events, etc.
- “Understand that any detail could be the important one”
- Learn from your mistakes, i.e., these can become future (automated) test cases
Check the plug

- “Question your assumptions”
- “Never trust your assumptions”
- Make sure you start at square zero; i.e., be sure you are executing the system in its true real-world environment to the extent possible
- Look to the tool you’re using (e.g., library function, compiler version, etc.) to see if the problem lies there
  - But only do this if you’ve exhausted all other possibilities
  - The problem is likely caused by your own human error, so never start by thinking this....

Get a fresh view

- Ask for help from other developers, specific experts, etc.
- Practice egoless programming, knowing that you are human
- Focus on symptoms, not theories
- Share your observations, even if you feel they’re unrelated
- Take a break, take a walk, take a nap, do something else, then take a fresh look at the problem
If you didn’t fix it, it ain’t fixed

- Oftentimes with larger-scale systems, making a change can end up fixing the observed problem or bug
- In reality, unless you found the bug and know you fixed it, chances are you only masked it or circumvented it temporarily
- “Check that it’s really fixed”
- “Check that it’s really your fix that fixed it”
- Learn from these errors by improving your future design and coding approaches

```c
#include <stdio.h>
#include <stdlib.h>

int findmin( int a[], int len );

int main()
{
    int a[] = { 5, 10, 0, 4, 5, 15 };
    int min = findmin( a, 6 );
    printf( "min is %d\n", min );  /* should display "min is 0" */
    return EXIT_SUCCESS;
}
```
**Example (ex01.c)** [continued]

```c
int findmin( int a[], int len )
{
    int i = 0, p = a[0], q = a[0];

    while ( i < len )
    {
        q = a[++i];
        if ( p > q ) p = q;
    }

    return q;
}
```

**Example (ex02.py)**

```python
# find all values in list y that are greater than x,
# accumulating results in res, which is returned
def findhighvalues( x = 0, y = [], res = [] ):
    for z in y:
        if z > x:
            res.append( z )
    return res

# test it out...
x = 25
y = [ 30, 20, 40, 10, 25 ]
res = findhighvalues( x, y )
print "res is", res          # should display "res is [30, 40]"
```
Example (ex02.py) [continued]

# test to see if we can accumulate more values in res
y = [ 12, 15, 50, 18 ]
res = findhighvalues(x, y, res)
print "res is", res  # should display
# "res is [30, 40, 50]"

# test a case in which no values are greater than x
# and then all values are greater than x
y = [ 12, 15, 18 ]
res = findhighvalues(x, y)
y = [ 30, 40, 50 ]
res = findhighvalues(x, y, res)
print "res is", res  # should display
# "res is [30, 40, 50]"

Example (ex03.java)

```java
public class ex03 {
    public static void main(String[] args) {
        Integer myLuckyNumber = 13;
        Integer yourLuckyNumber = 13;
        boolean a = (myLuckyNumber == yourLuckyNumber);

        Integer answerOne = 80 * 10;
        Integer answerTwo = 80 * 5 * 2;
        boolean b = (answerOne == answerTwo);

        System.out.println("Lucky numbers? " + a);
        System.out.println("Same answers? " + b);
    }
}
```

Find the answer by reading the Javadoc for Integer....