Announcements

- Syllabus: posted on website
- Academic Integrity
- Team formation - make sure to form teams and give composition in a private Piazza message.
- PRE will be posted once teams are formed.
Compiler

- use /util/bin/gcc compiler (this is 6.4.0)
- use -std=c11 (you can use other options too)
- test on timberlake.cse.buffalo.edu (that's our reference system)
## The 13 Golden Rules of Debugging

1. Understand the requirements
2. Make it fail
3. Simplify the test case
4. Read the right error message
5. Check the plug
6. Separate facts from interpretation
7. Divide and conquer
8. Match the tool to the bug
9. One change at a time
10. Keep an audit trail
11. Get a fresh view
12. If you didn’t fix it, it ain’t fixed
13. Cover your bugfix with a regression test
1. Understand the requirements

- Is it a bug or a misunderstanding of expected behavior?
- Requirements will tell you.
2. Make it fail

- Write test cases to isolate bug and make it reproducible.
- This will increase confidence that bug is fixed later.
- These tests will be added to the suite of regression tests ("does today’s code pass yesterday’s tests?")
3. Simplify the test case

- Ensure there is nothing extraneous in the test case.
- Keep it simple! Whittle it down until you get at the essence of the failure.
4. Read the right error message

“Everything that happened after the first thing went wrong should be eyed with suspicion. The first problem may have left the program in a corrupt state.” [p. 9]
5. Check the plug

- Don't overlook the obvious - things like permissions, file system status, available memory.

- “Think of ten common mistakes, and ensure nobody made them.” [p. 9]
6. Separate fact from fiction

- "Don't assume!"
- Can you prove what you believe to be true?
7. Divide and conquer

- Beware bugs caused by interactions amongst components.
- Develop a list of suspects (source code, compiler, environment, libraries, machine, etc).
- Each component alone may work correctly, but in combination bad things happen.
- Can be especially tricky with multithreaded programs.
8. Match the tool to the bug

- If all you have is a hammer ... you'll end up with a very sore thumb.
- Build a solid toolkit to give you choices.
- Use multiple tools/approaches (e.g. testing and debugging work better together than either alone)
9. One change at a time

- Be methodical. If you make multiple changes at one you can't tease apart which change had which effect.
- With your list of suspects, document what you predict the outcome of a change will be.
- Document the changes you make, and the results.
- Did results match predictions?
10. Keep an audit trail

- Make sure you can revert your code: use a code repository! This lets you back out changes that were not productive.
11. Get a fresh view

- Ask for someone else to have a look — but not before having done steps 1 - 10!
- Even just explaining the situation can help you better understand what is happening.
12. If you didn't fix it, it aint fixed

- Intermittent bugs will recur.
- If you make a change to the code and the symptom goes away, did you really fix it? You must convince yourself that the fix you applied really did solve the problem!
13. Cover your bug fix with a regression test

- Make sure the bug doesn't come back! Just because it worked yesterday doesn't mean it still works today. This is especially important in team environments where you are not the only person touching the code.
Essential tools

- compiler (e.g. gcc)
- debugger (e.g. gdb)
- memory checker (e.g. memcheck)
- runtime profiler (e.g. gprof)
- automated testing framework (e.g. cunit)
- build tool (e.g. make, gradle)
- code repository (e.g. git)
- organization/collaboration tool (e.g. ZenHub, Trello)
- pad of paper / whiteboard
Classification of bugs

- **Common bugs** (source code, predictable)
- **Sporadic bugs** (intermittent)
- **Heisenbugs** (averse to observation)
  - race conditions
  - memory access violations
  - (programmer) optimizations
- **Multiple bugs** - several must be fixed before program behavior changes - consider violating rule #9 "one change at a time"
uncertainty principle

...the uncertainty principle, also known as Heisenberg's uncertainty principle, is any of a variety of mathematical inequalities[1] asserting a fundamental limit to the precision with which certain pairs of physical properties of a particle, known as complementary variables, such as position x and momentum p, can be known.

The term observer effect refers to changes that the act of observation will make on a phenomenon being observed. This is often the result of instruments that, by necessity, alter the state of what they measure in some manner.

debugging tools

- instrument code during compilation
- instrumented code may behave differently than uninstrumented code
- in other words: the act of using a debugger may mask a bug, causing its symptoms to disappear, only to reappear when run without instrumentation
Essential tools

- compiler (e.g. gcc)
- debugger (e.g. gbd)
- memory checker (e.g. memcheck)
- runtime profiler (e.g. gprof)
- automated testing framework (e.g. cunit)
- build tool (e.g. make, gradle)
- code repository (e.g. git)
Memory organization

Each process (a running program) has a chunk of memory at its disposal.

This memory is divided into "static" memory (allocated/structured before execution begins) and "dynamic" memory (allocated while the program executes.)
Memory organization

The static segment is divided into a TEXT segment (holding the machine language instructions of the program), and a DATA segment (which has space for statically allocated memory, constants, literal values, etc).
Memory organization

The dynamic segment is divided into STACK and a HEAP areas.

The HEAP is generally located adjacent to the STATIC segment, and grows "down" (to higher memory addresses).
The STACK is generally located at the far end of memory and grows "up" (to lower memory addresses).

The area between the HEAP and the STACK represents available (free) memory.

If the HEAP and STACK collide we have an out-of-memory error.
The STACK holds invocation records (also called stack frames).

An invocation record is created whenever a function is called. It has space for the function's parameters, local variables, any return value, as well as bookkeeping information related to the call itself (e.g., where to return to).
Consider this code:

```c
void g() { ... }
void f() { ... g(); ... }
int main() { ... f() ... }
```

The invocation record for `main` is pushed on the stack as soon as execution begins. `main`'s record is the current/active one.
Consider this code:

```c
void g() { ... }
void f() { ... g(); ... }
int main() { ... f() ... }
```

When `f()` is called, an invocation record for `f` is pushed to the top of the stack.

`f`'s record is the current/active one.
Consider this code:

```c
void g() { ... }
void f() { ... g(); ... }
int main() { ... f() ... }
```

When `g()` is called, an invocation record for `g` is pushed to the top of the stack.

g's record is the current/active one.
Consider this code:

```c
void g() { ... }
void f() { ... g(); ... }
int main() { ... f() ... }
```

When `g()` returns its invocation record is removed from the stack, an `f`'s invocation record becomes the current/active one.
Consider this code:

```c
void g() { ... }

void f() { ... g(); ... }

int main() { ... f() ... }
```

When `f()` returns its invocation record is removed from the stack, an `main`'s invocation record becomes the current/active one.