CSE306
SOFTWARE QUALITY IN PRACTICE

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

www.cse.buffalo.edu/faculty/alphonce/SP23/CSE306
LATE JOINERS

• I update the rosters in Piazza and AutoLab each day from the UBLearns classlist.

• If you joined the recently it may take a day for the changes to propagate through all the systems.

• We will NOT be strict on the deadlines for LEX01 and LEX02 (to accommodate students registering through end of add/drop)

• If you missed your lab session, do the LEX as soon as you can on your own time: post questions and requests for assistance in Piazza.
ANNOUNCEMENTS

• PRE (the first team project) is posted on the website.

• Team formation was finalized last night.

• Every team has a Piazza group useful for intra-team communication, necessary for team-staff communication.
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"
WHY HEISENBUGS?
THE 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.

OBSERVER EFFECT

…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
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.)
The static segment is divided into a TEXT segment (holding the machine language instructions of the program), a DATA segment (for initialized variables), and a BSS segment (for uninitialized but implicitly zero-assigned values).
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.
MEMORY ORGANIZATION

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

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

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

When `f()` returns its invocation record is removed from the stack, and `main`'s invocation record becomes the current/active one.
The HEAP is used for dynamic allocation of non-local data.

In Java allocation is done using 'new', as in

px = new Foo();

Java's garbage collector frees heap-allocated memory when it is no longer in use.
MEMORY ORGANIZATION

The HEAP is used for dynamic allocation of non-local data.

In Java allocation is done using 'new', as in

px = new Foo();

Java's garbage collector frees heap-allocated memory when it is no longer in use.
In either case the (local) variable px holds the address of the chunk of memory, allocated on the heap, which holds some data.
A local variable, like x in the code shown, has memory for its value set aside in the function's invocation record.

The name of the variable, x in this case, does not exist at runtime.
Memory Organization

Any read from `x` or write to `x` is translated into a memory access at some offset from the current Stack Pointer (SP). SP refers to a known point within an invocation record.

```c
int main() {
    int x = 0;
    .
    .
    .
    return 0;
}
```
https://gcc.gnu.org/onlinedocs/9.4.0/

https://gcc.gnu.org/onlinedocs/gcc-9.4.0/gcc/Standards.html#C-Language


http://releases.llvm.org/10.0.0/tools/clang/docs/UsersManual.html#c
COMMON OPTIONS

-std set language standard
-o set output file name
-g include debugging information in object file
-c compile/assemble do not link
-Wall report "all" warnings
-L library path
-I include path
Inspect this program and describe as best you can what this program will do when it runs.

Also discuss where in memory space for variable x will be allocated.
ACTIVITY

• Visit the course website: https://cse.buffalo.edu/faculty/alphonce/SP23/CSE306/

• Click on the "In-class activity" button for Feb 06.

• Answer the question on the "What does it do?" page.
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 0;
    while (x < 10) {
        printf("x has value %d\n",x);
        x = x + 1;
    }
    exit(EXIT_SUCCESS);
}
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 0;
    while (x < 10) {
        printf("x has value %d\n", x);
        x = x + 1;
    }
    exit(EXIT_SUCCESS);
}
• Answer the question on the "Rewrite" page.
ACTIVITY

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

int main(void) {
    int x = 0;
    while (x < 10) {
        printf("x has value %d\n", x);
        x = x + 1;
    }
    exit(EXIT_SUCCESS);
}