CSE306
SOFTWARE QUALITY IN PRACTICE

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

www.cse.buffalo.edu/faculty/alphonce/SP22/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) will be posted on the website this afternoon.

• If you have not yet formed a team, try to do that today.

• If there are students not on teams on Tuesday I will assign students to teams. This *may* impact existing teams.
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"
...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
MEMORY ORGANIZATION
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), and a DATA segment (which has space for statically allocated memory, constants, literal values, etc).
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.
MEMORY ORGANIZATION

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.
In C allocation is done using 'malloc' (memory allocate):

\[ px = \text{malloc}(\text{sizeof}(\*px)); \]

C is not garbage collected. 'free' must be called explicitly to release unused memory and make it available for re-allocation:

\[ \text{free}(px); \]
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.

```c
int main() {
    int x = 0;
    
    return 0;
}
```
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.
COMPILER VERSIONS
TIMBERLAKE.CSE.BUFFALO.EDU

• /usr/bin/gcc               4.4.7
• /util/bin/gcc               6.4.0
• /util/gcc-7.2.0/bin/gcc    7.2.0
• /util/llvm/bin/clang       3.5.0
COMPILER VERSIONS
TIMBERLAKE.CSE.BUFFALO.EDU

• /usr/bin/gcc               4.4.7
• /util/bin/gcc               6.4.0
• /util/gcc-7.2.0/bin/gcc    7.2.0
• /util/llvm/bin/clang      3.5.0

We want to use the more up-to-date versions.
COMPILER VERSIONS
TIMBERLAKE.CSE.BUFFALO.EDU

- /util/gcc-7.2.0/bin/gcc    7.2.0
- /util/llvm/bin/clang      3.5.0

ssh timberlake.cse.buffalo.edu
# COMMON OPTIONS

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-std</td>
<td>set language standard</td>
</tr>
<tr>
<td>-o</td>
<td>set output file name</td>
</tr>
<tr>
<td>-g</td>
<td>include debugging information in object file</td>
</tr>
<tr>
<td>-c</td>
<td>compile/assemble do not link</td>
</tr>
<tr>
<td>-Wall</td>
<td>report &quot;all&quot; warnings</td>
</tr>
<tr>
<td>-L</td>
<td>library path</td>
</tr>
<tr>
<td>-l</td>
<td>include path</td>
</tr>
</tbody>
</table>
#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);
}
ACTIVITY

• Visit the course website: https://cse.buffalo.edu/faculty/alphonce/SP22/CSE306/

• Click on the "Activity Form" button for Feb 07.

• Answer the question on the "What does it do?" page.
ACTIVITY

```c
#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);
}
```

It prints the values 0 through 9 in this format:

- x has value 0
- x has value 1
- x has value 2
- x has value 3
- x has value 4
- x has value 5
- x has value 6
- x has value 7
- x has value 8
- x has value 9

Memory for x will allocated in main's invocation record.
#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);
}

It prints the values 0 through 9 in this format:

x has value 0
x has value 1
x has value 2
x has value 3
x has value 4
x has value 5
x has value 6
x has value 7
x has value 8
x has value 9

In reality the value of x may exist just in a register at runtime, but for our purposes right now we don't need to be concerned about that.
• Answer the question on the "Rewrite" 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);
}
The next several slides show the code written on the boards.
To answer this Q: yes, it is valid. Declarations are handled at compile time. The compiler therefore knows what type *px refers to.

The declaration should be of *px, not px, however.
A: What does `const` do?

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

const int x = 0;

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

x is not on heap.
int main(void) {
    int *x = malloc(sizeof(int));
    *x = 0;
    while (*x < 10) {
        printf("x has value of %d/n", *x);
        *x = *x + 1;
    }
    exit(EXIT_SUCCESS);
}

Should we free x at end?
```c
struct NUM {
    int x;
    int n;
}

int main() {
    struct NUM *num = malloc(sizeof(num));
    num->x = 0;
    while (num->x < 10) {
        printf("--- %d\n", num->x);
        num->x++;
    }
    free(num);    // free?
    return 0;
}
```