CSE410 aka CSE306
Software Quality

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

http://www.cse.buffalo.edu/faculty/alphonce/SP17/CSE410
https://piazza.com/class/iybn33z3aro2p
FIX BAD CODE
# Learning outcomes

<table>
<thead>
<tr>
<th>Learning outcome</th>
<th>Instructional methods</th>
<th>Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employ static and dynamic analysis tools to detect faults in a given piece of software.</td>
<td>Lecture-based instruction</td>
<td>LEX EXP LPR</td>
</tr>
<tr>
<td>Employ profiling tools to identify performance issues (both time and memory) in a given piece of software.</td>
<td>Lab-based hands-on exercises, both individual and group</td>
<td></td>
</tr>
<tr>
<td>Employ testing frameworks to write tests that fail in the presence of software faults, and pass otherwise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Employ a structured, methodical approach to detecting, testing, identifying and correcting software faults.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PRE PST EXP LPR
Learning activities

- (LEX) Ten weekly lab-based exercises, due in the lab session, done in weeks 1 through 10.

- (PRE)/(PST) A “process” team project, done twice, once as a pre-assessment in weeks 1, 2, and 3 of the semester, and a second time as a post-assessment in weeks 10, 11 and 12. Students are required to document their development/debugging process.

- (EXP) Four two-week exploratory projects, done in weeks 4-5, 6-7, 8-9 and 13-14. These projects ask students to apply the tools and techniques they have been taught up to that point in the course to open-source projects found at repositories such as SourceForge, GitHub and BitBucket. Students are required to document their use of the tools and the results they obtained.

- (LPR) A two-part in-lab practical exam, in weeks 13 and 14. Part 1 will cover basic skills, part 2 will cover process.
# Grading

<table>
<thead>
<tr>
<th>INDIVIDUAL</th>
<th>TEAM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LEX</strong></td>
<td><strong>PRE</strong></td>
</tr>
<tr>
<td>1000 (28%)</td>
<td>200 (6%)</td>
</tr>
<tr>
<td><strong>EXP</strong></td>
<td><strong>PST</strong></td>
</tr>
<tr>
<td>800 (22%)</td>
<td>400 (11%)</td>
</tr>
<tr>
<td><strong>LPR</strong></td>
<td></td>
</tr>
<tr>
<td>1200 (33%)</td>
<td></td>
</tr>
</tbody>
</table>
Teams & Recitations

- Form teams this week - I recommend teams of size 2-3.
- Recitations start this week.
- Recommend all team members attend same recitation, but not required.
(PRE) PROCESS PROJECT

- Write a C program that reads from a file (filename.IN), interprets the input as an infix arithmetic expression, and writes to a file (filename.OUT) the prefix form of the expression, the postfix form of the expression, and the value of the expression, as shown in the following examples.

- Assumptions:
  - all numbers are integers
  - in input integers may be expressed either using arabic or roman numerals (up to MMMCMXCIX)
  - in output only arabic numerals may be used
  - usual precedence rules apply, as well as the role of parentheses; operators are +, -, *, and /
EX1 - foo.IN contains
2 + 30 * iv

The program must write the following to foo.OUT:

Prefix: + 2 * 30 4
Postfix: 2 30 4 * +
Value: 122
EX2 - bar.IN contains 

\[(I + X) \times 4\]

The program must write the following to bar.OUT:

Prefix: \(* + 2 \ 30 \ 4\)
Postfix: \(2 \ 30 + 4 \ *)
Value: 128
EX3 - file.IN contains what you don't know.

The program must write the following to file.OUT:

Error!
EX4 - example.IN contains 3 / 0

The program must write the following to file.OUT:

Prefix: / 3 0
Postfix: 3 0 /
Value: error
Activity log (.csv file)

- design discussions (DD)
- coding progress (CP)
- testing done (TD)
- bugs found and fixed (BFF)
- tools used (editor, compiler, etc) (TU)

DATE, UBIT, ACTIVITY, TOOLS, NOTES

20170130, alphonce, CP, compiler, write main function
Due date: Friday, February 17

- Work with 2-3 teammates
- Set up a PRIVATE repo in BITBUCKET for your team’s code
- Include me in the repo: alphonce
- Each team member must keep a log of their project activities as the code is developed
ACADEMIC INTEGRITY

- You and your teammates must write the code by yourselves.
- You may look up (but cite!) resources for the necessary algorithms and data structures.

Main point:
- Warm-up on C programming
  - Document your programming process at this point in the course
Compiler

- use gcc compiler with C11 standard
- test on timberlake.cse.buffalo.edu (that's our reference system)
Grading Rubric

- Each piece of student work will be assessed using performance indicators with associated rubrics with performance levels:
  - “unsatisfactory” (0 points)
  - “developing” (50 points)
  - “satisfactory” (75 points)
  - “exemplary” (100 points)
- Each performance indicator is assessed independently of the others.
- I will post the rubric for the project this week.
Code portfolio

- I will be asking each of you to contribute a coding project you did for CSE250.

- Please anonymize it (i.e. remove/change anything that can personally identify you from any comments, identifiers, etc in the code)

- DON’T WORRY IF THE CODE ISN’T PERFECT: code with bugs is preferred!
Is this code buggy?

```c
#include <stdio.h>

int main() {
    printf("Hello, world.");
}
```
It depends - what was the specification for the program?
Should the program check the return value of printf?

What is the return value of printf?
Table 1. Single Taxable Income Brackets and Rates, 2017

<table>
<thead>
<tr>
<th>Rate</th>
<th>Taxable Income Bracket</th>
<th>Tax Owed</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>$0 to $9,325</td>
<td>10% of Taxable Income</td>
</tr>
<tr>
<td>15%</td>
<td>$9,325 to $37,950</td>
<td>$932.50 plus 15% of the excess over $9,325</td>
</tr>
<tr>
<td>25%</td>
<td>$37,950 to $91,900</td>
<td>$5,226.25 plus 25% of the excess over $37,950</td>
</tr>
<tr>
<td>28%</td>
<td>$91,900 to $191,650</td>
<td>$18,713.75 plus 28% of the excess over $91,900</td>
</tr>
<tr>
<td>33%</td>
<td>$191,650 to $416,700</td>
<td>$46,643.75 plus 33% of the excess over $191,650</td>
</tr>
<tr>
<td>35%</td>
<td>$416,700 to $418,400</td>
<td>$120,910.25 plus 35% of the excess over $416,700</td>
</tr>
<tr>
<td>39.60%</td>
<td>$418,400+</td>
<td>$121,505.25 plus 39.6% of the excess over $418,400</td>
</tr>
</tbody>
</table>
double f(double x) {
    if (x < 9325) { return 0.1 * x; }
    if (x < 37950) { return 932.50 + 0.15 * (x - 9325); }
    if (x < 91900) { return 5225.25 + 0.25 * (x - 37950); }
    if (x < 191650) { return 18713.75 + 0.28 * (x - 91900); }
    if (x < 416700) { return 46643.75 + 0.33 * (x - 196150); }
    if (x < 418400) { return 120910.25 + 0.35 * (x - 416700); }
    return 131505.25 + 39.6 * (x - 418400);
}
Is this code buggy?

double f(double x) {
    if (x < 9325) { return 0.1 * x; }
    if (x < 37950) { return 932.50 + 0.15 * (x - 9325); }
    if (x < 91900) { return 5225.25 + 0.25 * (x - 37950); }
    if (x < 191650) { return 18713.75 + 0.28 * (x - 91900); }
    if (x < 416700) { return 46643.75 + 0.33 * (x - 196150); }
    if (x < 418400) { return 120910.25 + 0.35 * (x - 416700); }
    return 131505.25 + 39.6 * (x - 418400);
}
What we’ll be doing in this course

- Learn tools to explore program structure and behavior.
- Consider correctness relative to a specification and performance relative to a requirement.
- Employ a methodical approach to tracking down, identifying, documenting and fixing problems with code.
static vs dynamic program analysis

static analysis - done on program without executing it

dynamic analysis - done on program by executing it
Compiler
a static analysis tool

- In recitation this week we will explore what a compiler can and can't tell us about our code.
Fig. 2.1 Simplified build and test flow
# 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
2. Make it fail
3. Simplify the test case
4. Read the right error message
5. Check the plug
6. Separate fact from fiction
7. Divide and conquer
8. Match the tool to the bug
9. One change at a time