CSE306
SOFTWARE QUALITY IN PRACTICE

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

www.cse.buffalo.edu/faculty/alphonce/FA24/CSE306

LATE JOINERS AND MISSED SUBMISSIONS

- I update the rosters in Piazza/AutoLab/TopHat regularly from the UBLearns classlist (last update to TopHat was ~5 minutes ago).

- If you joined the recently it may take a day (possibly two) for the changes to propagate through all the systems.

- We will NOT be strict on the deadlines for LEX01, LEX02, and LEX03 (to accommodate students registering through end of add/drop): we will allow submissions until 11:59 PM Friday of next week.

- 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.

- Remember to not only submit your code, but also submit the form: 57 students submitted code to AutoLab, but only 52 students submitted the form.

REMINDERS

- 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.
- If necessary I will step in and assign students to teams.

COMPILER

- On cerf use /usr/bin/gcc compiler (this is 11.4.0, and should be your default)

- use -std=c11   (you can use other options too)

- test on cerf.cse.buffalo.edu (that's our reference system)

STATIC VS DYNAMIC PROGRAM ANALYSIS

- static analysis - done on program without executing it
- dynamic analysis - done on program by executing it

THE COMPILER:
A STATIC ANALYSIS TOOL

- We will explore what a compiler can and can’t tell us about our code.

COMPILING AND RUNNING CODE

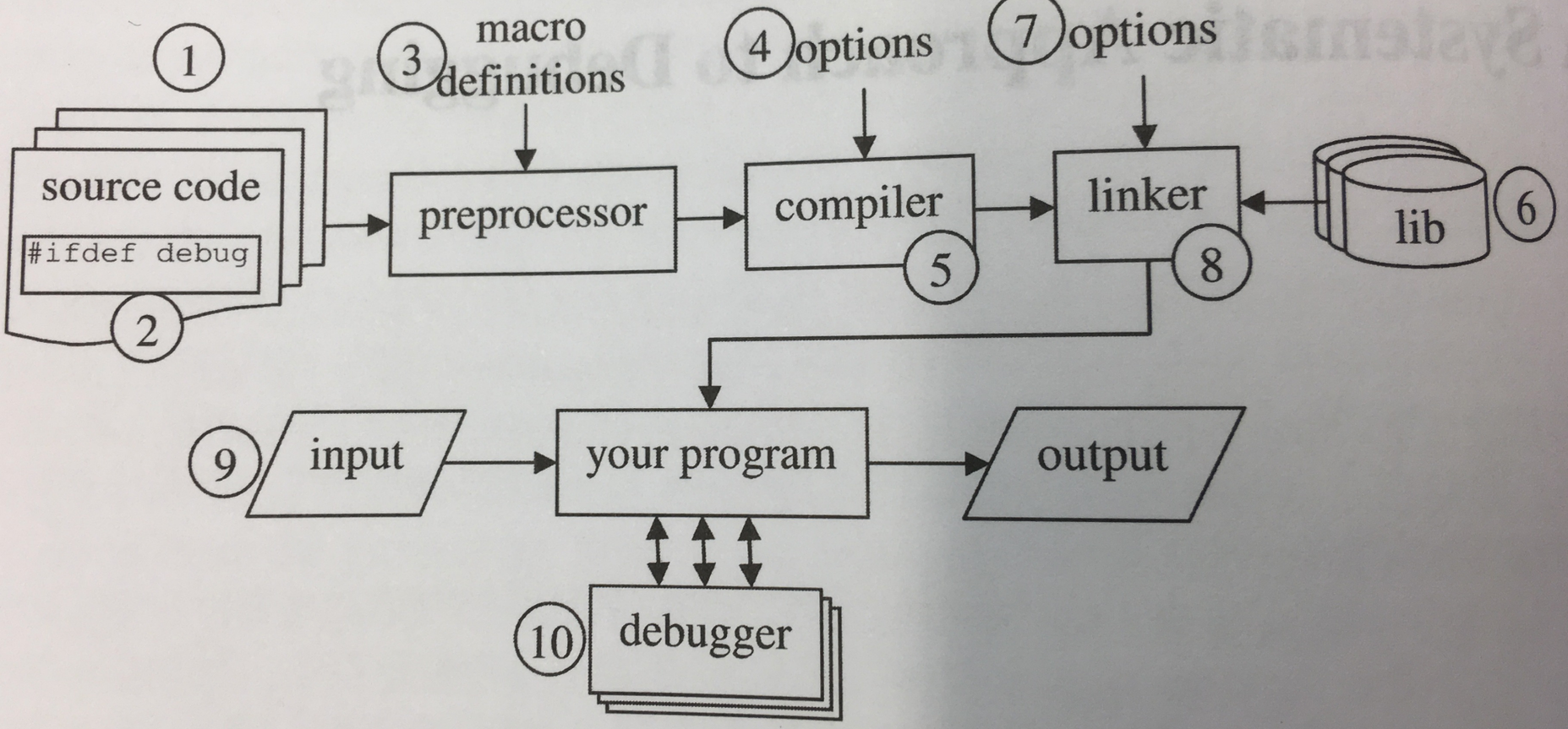

Fig. 2.1 Simplified build and test flow

Textbook, page 6

COMPILING AND RUNNING CODE

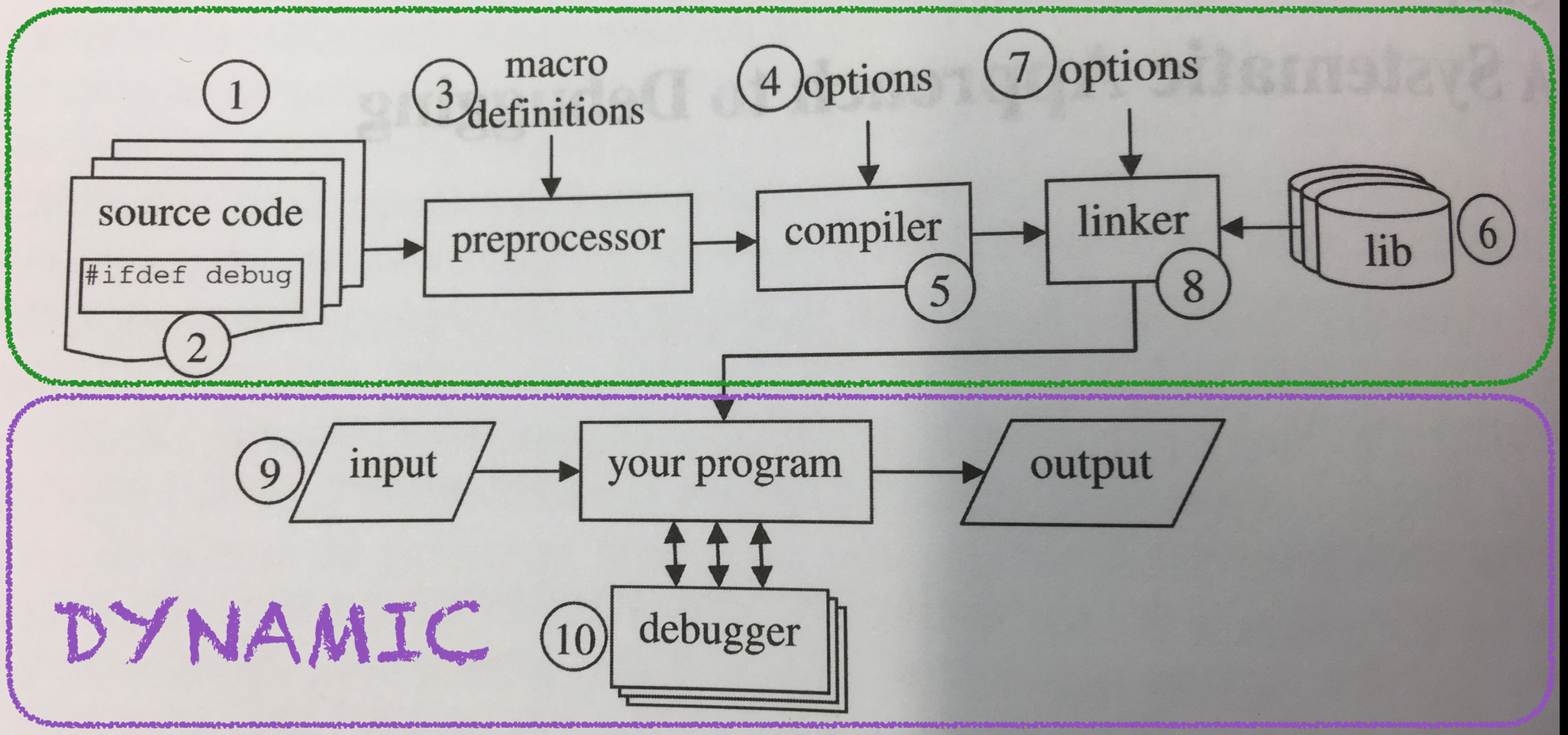

Fig. 2.1 Simplified build and test flow

Textbook, page 6

TEXT, PG 8

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

TOPHAT (PRACTICE) QUESTIONS

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 AIN'T 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. gbd)
- memory checker (e.g. memcheck)
- runtime profiler (e.g. gprof)
- automated testing framework (e.g. criterion or cunit)
- build tool (e.g. make)
- code repository (e.g. git)
- organization/collaboration tool (e.g. Trello or ZenHub)
- pad of paper / whiteboard