CS136: Debugging

Tim Brecht
Steps to Help you Debug Your Code

• Step 0: Try to avoid writing code with bugs
• Step 1: Triggering bugs
• Step 2: Determine the source of the problem
• Step 3: Fixing the code
• Step 4: Checking that the bug has been fixed
• Step 5: Checking if new bugs have been added (regression testing)
• Step 6: Submit to Marmoset
• Step 7: Check the results of your submission
• Step 8: Continue to learn
Steps to Help you Debug Your Code

• Step 0: Try to avoid writing code with bugs
• Step 1: Triggering bugs
• Step 2: Determine the source of the problem
• Step 3: Fixing the code
• Step 4: Checking that the bug has been fixed
• Step 5: Checking if new bugs have been added (regression testing)
• Step 6: Submit to Marmoset
• Step 7: Check the results of your submission
• Step 8: Continue to learn
Step 0: Try to avoid writing code with bugs

Start with Tests

Coming up with good test cases is important

- Start with simple cases and work to harder cases
- Helps in understanding the specification (assignment)
  - Refer back to the specification
- Think about corner cases and come up with tests for them
  - Do this first: so your code is designed to handle them
  - Easier than trying to patch code later
Step 0: Try to avoid writing code with bugs

Start with Tests

• How will you test functions?
• How will you test the whole program?
Testing Functions with Assertions

```c
const bool do_tests = true;

void test_helpers(void) {
    assert(sqrt(25) == 5);
    assert(exp(2, 4) == 16);
    // etc.
}

int main(void) {
    if (do_tests) {
        test_helpers();
    }
    // program code goes here
}
```
Testing Functions with Test Harness

public.in:
sqrt 25
exp 2 4

public.expect:
5
16
Testing Programs

I/O based Testing (e.g., compute mean)

```
public.in:
1 2 3 4 5

public.expect:
3
```
Testing Modules

// test-my-module.c
// This program tests the module named my-module

#include "my-module.h"

int main(void) {
    // Add tests here
}

Try to do as much testing as possible that:

1) Does not require you to look at ANY output to know that the tests passed
   ◦ That is the power of assertion-based testing

2) Does NOT require you to create long and/or complicated input and expect files
   ◦ Takes time
   ◦ Is error prone
   ◦ Can be hard to maintain (e.g., modify and expand)

• It is important to learn how to do assertion and I/O based testing
Step 0: Try to avoid writing code with bugs

Come up with good, clean simple design

• Easier to understand
• Less prone to errors
• Easier to debug

• Using helper functions is the key
  ◦ Change longer functions into a sequence of calls to helper functions
Step 0: Try to avoid writing code with bugs

Start Small and Add Small Pieces

• Start with smallest, easiest helper functions
• Test them so you know they work
• Build the next piece, ensure it works, repeat
Steps to Help you Debug Your Code

• Step 0: Try to avoid writing code with bugs
• Step 1: Triggering bugs
• Step 2: Determine the source of the problem
• Step 3: Fixing the code
• Step 4: Checking that the bug has been fixed
• Step 5: Checking if new bugs have been added (regression testing)
• Step 6: Submit to Marmoset
• Step 7: Check the results of your submission
• Step 8: Continue to learn
Step 1: Triggering bugs

• You cannot prove your code is correct by testing alone (for useful programs)
  ◦ But you can build confidence in its correctness
  ◦ Only if you have a set of good, thorough tests

How do you know if your code has bugs?

• To find a bug you need a test that triggers the bug
  1) Test for which you know the expected result
  2) Ability to check if result matches what is expected

See Step 0: and coming up with tests first
Develop Good Tests

Think of yourself as an adversary (enemy / opponent)

• Your goal is to try to break the code
  ◦ Want function call to fail, what parameters (arguments) to use?
  ◦ Want program to fail, what input would I use?
  ◦ This is a way to identify corner (edge) cases
Never, ever, ever, ever ignore a bug

Ignored bugs will come back and bite you (at the worst time)

• Bugs to not magically disappear
• Spend the time to understand your code
• Check your tests cases
Some Testing Terminology

• **White box testing:** you can see the code being tested
• **Black box testing:** you can not see the code being tested
• **Unit testing:** testing one piece at a time (e.g., having separate tests for each module)
• **Regression testing:** rerunning ALL tests to check if everything still works
  (after a change to any part of the code)
Syntax errors are not considered bugs

```c
/*
File: error-msgs.c
Used for debugging video.
*/

#include "cs136.h"

const bool dbg = false;

int main(void) {
  int x = 3;
  int y = 1;
  int z = 10;
  if (dbg) {
    trace_sync();
  } else {
    trace_off();
  }
}
```

Running 'MyProjects/expected-output-demo':
Compilation generated warnings:
expected-output-demo/error-msgs.c:11:7: unused variable 'x' [-Wunused-variable]
Program finished with exit code 0.
Runtime errors are bugs

```c
int main(void) {
    int y = 1;
    int z = 10;

    if (dbg) {
        trace_sync();
    } else {
        trace_off();
    }

    for (int i = 3; i >= 0; --i) {
        trace_int(i);
        trace_int(z);
        z = z + (y / i);
    }
    printf("%d\n", z);
}
```

Running 'MyProjects/expected-output-demo':
AddressSanitizer:DEADLYSIGNAL
Memory error occurred! Type of error: floating-point-exception
current framelist: 0
    frame 0: function main in file error-msgs.c at line 23, column 16
Program finished with exit code 1 (An error occurred).
Runtime errors are bugs

```c
1 /*
2 File: error-msgs.c
3 Used for debugging video.
4 */
5
6 #include "cs136.h"
7
8 const bool dbg = false;
9
10 int main(void) {
11    int y = 1;
12    int z = 10;
13
14    if (dbg) {
15        trace_sync();
16    } else {
17        trace_off();
18    }
19
20    for (int i = 3; i > 0; --i) {
21        trace_int(i);
22        trace_int(i);
23    }
24
25    return 0;
26}
```

Running 'MyProjects/expected-output-demo':

```
11
AddressSanitizer:DEADLYSIGNAL
Memory error occurred! Type of error: segmentation-fault-on-null-address
```

Program finished with exit code 1 (An error occurred).
Steps to Help you Debug Your Code

• Step 0: Try to avoid writing code with bugs
• Step 1: Triggering bugs
• Step 2: Determine the source of the problem
• Step 3: Fixing the code
• Step 4: Checking that the bug has been fixed
• Step 5: Checking if new bugs have been added (regression testing)
• Step 6: Submit to Marmoset
• Step 7: Check the results of your submission
• Step 8: Continue to learn
Step 2: Determining the source of the problem

This is often the hardest part

• It can be avoided or will be easier if you follow steps Step 0 and Step 1 first!!
Do not try to fix the bug without first understanding it

No matter how tempting **DO NOT** randomly change code until it works

• This is Step 2: Determining the source of the bug

  **DO NOT SKIP THIS STEP** (i.e., finding the source of the problem)!!
Be organized and methodical (systematic)

- Use logic: there is a logical reason for bug
- Try to find a simple test case that triggers the bug
- Develop a systematic approach to finding the source of the problem
Hyphothesis-based approach

Come up with hypotheses (educated guess / ideas) about what might be wrong
• Based on your knowledge of the problem and test case that fails
• Write them down (e.g., off by one error, loop exit/entry conditions incorrect, etc.)
• Choose the hypothesis you think is most likely
• Come up with a way to test your hypothesis
  ◦ Sometimes test may be simple
    - trace statements \texttt{trace\_int(n)};
    - adding a bunch of assertions (maybe the function fails when \( n < 0 \))
• Rule out hypotheses one by one (update your notes)
Understand your code

What is the path through the code when the bug is triggered?

• trace tools can be very helpful  

\#include "cs136-trace.h"

• Is the problem with the code on that path?
• Or is the problem that the code is following the wrong path?
const bool dbg = true;

int main(void) {
    if (dbg) {
        trace_sync();
    } else {
        trace_off();
    }
    int x = 10;
    // ...
    print_int(x);
    // ...
    print_int(x);
}
Use the Computer (to help)

```python
def main():
    i = 3
    i = 2
    z = 10

def main():
    i = 3
    i = 2
    z = 10

>>> [error-msgs.c|main|21] >> i => 3
>>> [error-msgs.c|main|22] >> z => 10
>>> [error-msgs.c|main|21] >> i => 2
>>> [error-msgs.c|main|22] >> z => 10
```
Using Trace Tools

Running 'MyProjects/expected-output-demo':

AddressSanitizer:DEADLYSIGNAL

>>> [error-msgs.c|main|21] >> i => 3
>>> [error-msgs.c|main|22] >> z => 10
>>> [error-msgs.c|main|21] >> i => 2
>>> [error-msgs.c|main|22] >> z => 10
>>> [error-msgs.c|main|21] >> i => 1
>>> [error-msgs.c|main|22] >> z => 10

11

>>> [error-msgs.c|main|27] >> "A"
>>> [error-msgs.c|main|29] >> "B"

Memory error occurred! Type of error: segmentation-fault-on-null-address

Program finished with exit code 1 (An error occurred).
Using Trace Tools

11

14 if (dbg) {
15     trace_sync();
16 } else {
17     trace_off();
18 }

20 for (int i = 3; i > 0; --i) {
21     trace_int(i);
22     trace_int(z);
23     z = z + (y / i);
24 }
25 printf("%d\n", z);

27 trace_msg("A");
28 int *p = NULL;
29 trace_msg("B");
30 *p = 10;
31 trace_msg("C");
32

Running 'MyProjects/expected-output-demo':
AddressSanitizer:DEADLYSIGNAL

>>> [error-msgs.c|main|21] >> i => 3
>>> [error-msgs.c|main|22] >> z => 10
>>> [error-msgs.c|main|21] >> i => 2
>>> [error-msgs.c|main|22] >> z => 10
>>> [error-msgs.c|main|21] >> i => 1
>>> [error-msgs.c|main|22] >> z => 10

>>> [error-msgs.c|main|27] >> "A"
>>> [error-msgs.c|main|29] >> "B"

Memory error occurred! Type of error: segmentation-fault-on-null-address
Program finished with exit code 1 (An error occurred).
Using Trace Tools

```c
int y = 1;
int z = 10;

if (dbg) {
    trace_sync();
} else {
    trace_off();
}

for (int i = 3; i > 0; --i) {
    trace_int(i);
    trace_int(z);
    z = z + (y / i);
}

printf("%d\n", z);

trace_msg("A");
int *p = NULL;
trace_msg("B");
*p = 10;
trace_msg("C");
```
Using Trace Tools

```c
int y = 1;
int z = 10;

if (dbg) {
    // trace_sync();
} else {
    trace_off();
}

for (int i = 3; i > 0; --i) {
    trace_int(i);
    trace_int(z);
    z = z + (y / i);
}

printf("%d\n", z);

trace_msg("A");
int *p = NULL;
trace_msg("B");
*p = 10;
trace_msg("C");
```

Running 'MyProjects/expected-output-demo':
```plaintext
>>> [error-msgs.c|main|21] >> i => 3
>>> [error-msgs.c|main|22] >> z => 10
>>> [error-msgs.c|main|21] >> i => 2
>>> [error-msgs.c|main|22] >> z => 10
>>> [error-msgs.c|main|21] >> i => 1
>>> [error-msgs.c|main|22] >> z => 10
>>> [error-msgs.c|main|27] >> "A"
>>> [error-msgs.c|main|29] >> "B"
```

AddressSanitizer:DEADLYSIGNAL
11
Memory error occurred! Type of error: segmentation-fault-on-null-address
current framelist: 0
Program finished with exit code 1 (An error occurred).
```
Using Trace Tools

```c
11 int y = 1;
12 int z = 10;

14 if (dbg) {
15    // trace_sync();
16 } else {
17    trace_off();
18 }

20 for (int i = 3; i > 0; --i) {
21    trace_int(i);
22    trace_int(z);
23    z = z + (y / i);
24 }
25 printf("%d\n", z);
26
27 trace_msg("A");
28 int *p = NULL;
29 trace_msg("B");
30 *p = 10;
31 trace_msg("C");
32 }
```

Running 'MyProjects/expected-output-demo':

```
>>> [error-mgs.c main|21] >> i => 3
>>> [error-mgs.c main|22] >> z => 10
>>> [error-mgs.c main|21] >> i => 2
>>> [error-mgs.c main|22] >> z => 10
>>> [error-mgs.c main|21] >> i => 1
>>> [error-mgs.c main|22] >> z => 10

>>> [error-mgs.c main|27] >> "A"
>>> [error-mgs.c main|29] >> "B"

AddressSanitizer:DEADLYSIGNAL

11

Memory error occurred! Type of error: segmentation-fault-on-null-address
current framelist: 0
Program finished with exit code 1 (An error occurred).
```
Using Trace Tools

```c
int y = 1;
int z = 10;

if (dbg) {
    trace_sync();
} else {
    trace_off();
}

for (int i = 3; i > 0; --i) {
    trace_int(i);
    trace_int(z);
    z = z + (y / i);
}
printf("%d\n", z);

trace_msg("A");
int *p = NULL;
trace_msg("B");
p = 10;
trace_msg("C");
```

Running 'MyProjects/expected-output-demo':
AddressSanitizer:DEADLYSIGNAL

```plaintext
>>> [error-mgs.c|main|21] >> i => 3
>>> [error-mgs.c|main|22] >> z => 10
>>> [error-mgs.c|main|21] >> i => 2
>>> [error-mgs.c|main|22] >> z => 10
>>> [error-mgs.c|main|21] >> i => 1
>>> [error-mgs.c|main|22] >> z => 10
```

Memory error occurred! Type of error: segmentation-fault-on-null-address
current framelist: 0
Program finished with exit code 1 (An error occurred).
Using Trace Tools

```c
int y = 1;
int z = 10;

if (dbg) {
    trace_sync();
} else {
    trace_off();
}

for (int i = 3; i > 0; --i) {
    trace_int(i);
    trace_int(z);
    z = z + (y / i);
}

printf("%d\n", z);

trace_msg("A");
int *p = NULL;
trace_msg("B");
*p = 10;
trace_msg("C");
}
```

Running 'MyProjects/expected-output-demo':

AddressSanitizer:DEADLYSIGNAL

```bash
>>> [error-msgs.c|main|21] >> i => 3
>>> [error-msgs.c|main|22] >> z => 10
>>> [error-msgs.c|main|21] >> i => 2
>>> [error-msgs.c|main|22] >> z => 10
>>> [error-msgs.c|main|21] >> i => 1
>>> [error-msgs.c|main|22] >> z => 10
```

```
11
```

```
>>> [error-msgs.c|main|27] >> "A"
>>> [error-msgs.c|main|29] >> "B"
```

Memory error occurred! Type of error: segmentation-fault-on-null-address

current framelist: 0

Program finished with exit code 1 (An error occurred).
Use trace tools NOT `printf`

Debugging statements with `printf` must be removed / commented out for submission
Don’t assume your code implements what you think it does

```c
if (!(x > 10 && x < 15) || (!(y > 20 && y <= 30))) {

• What is your intention with your code?
• Does it actually do what you think it is supposed to do?
• Can you test / determine that it does? How?
• Can your code be simplified
  ◦ So you can be more certain it does what you intend?
  ◦ Don’t forget to use lots of helper functions
```
Is your code too complex to find the bug?

Does your code need to be simplified in order to find the bug?

• To help you be more certain it does what you intend?
• Don’t forget to use helper functions
  ◦ Frequent use of small helper functions makes the code
    - Easier to read and understand
    - Easier to check for correctness
    - Easier to test and debug
Be careful of (watch out for) copy and paste errors

assert(sqr(2) == 4);
Be careful of (watch out for) copy and paste errors

```javascript
assert(sqr(2) == 4);
assert(sqr(2) == 4);
```
Be careful of (watch out for) copy and paste errors

```javascript
assert(sqr(2) == 4);
assert(sqr(3) == 4);
```
Be careful of (watch out for) copy and paste errors

```
assert(sqr(2) == 4);
assert(sqr(3) == 9);
```
Be careful of (watch out for) copy and paste errors

```cpp
assert(sqr(2) == 4);
assert(sqr(3) == 9);
assert(sqr(4) == 16);
assert(sqr(5) == 25);
assert(sqr(6) == 25);
assert(sqr(7) == 49);
assert(sqr(8) == 81);
assert(sqr(10) == 100);
```
Be careful of (watch out for) copy and paste errors

```python
assert(sqr(2) == 4);
assert(sqr(3) == 9);
assert(sqr(4) == 16);
assert(sqr(5) == 25);
assert(sqr(6) == 25);
assert(sqr(7) == 49);
assert(sqr(8) == 81);
assert(sqr(10) == 100);
```
Be careful of (watch out for) typos

assert(sqr(2) == 4);
assert(sqr(3) == 9);
assert(sqr(4) == 16);
assert(sqr(5) == 25);
assert(sqr(6) == 25);
assert(sqr(7) == 49);
assert(sqr(8) == 81);
assert(sqr(10) == 100);
Be careful of (watch out for) typos

```plaintext
assert(sqr(2) == 4);
assert(sqr(3) == 9);
assert(sqr(4) == 16);
assert(sqr(5) == 25);
assert(sqr(6) == 25);
assert(sqr(7) == 49);
assert(sqr(8) == 81);
assert(sqr(10) == 100);
```

Be careful of (watch out for) typos

```javascript
assert(sqr(2) == 4);
assert(sqr(3) == 9);
assert(sqr(4) == 16);
assert(sqr(5) == 25);
assert(sqr(6) == 25);
assert(sqr(7) == 49);
assert(sqr(9) == 81);
assert(sqr(10) == 100);
```
Check for bugs in your tests

Hardest bugs to find are those that do not exist

• Sometimes the problem is with the test(s) and not the code
• Check your assertions
• Check your .in and .expect files (really carefully)
Check for bugs in your tests

Hardest bugs to find are those that do not exist

• Sometimes the problem is with the test(s) and not the code
• Check your assertions
• Check your .in and .expect files (really carefully)

```
assert(sqr(7) == 48);
```
Check for bugs in your tests

Hardest bugs to find are those that do not exist

- Sometimes the problem is with the test(s) and not the code
- Check your assertions
- Check your .in and .expect files (really carefully)

<table>
<thead>
<tr>
<th>test-sqr.in</th>
<th>test-sqr.expect</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>48</td>
</tr>
<tr>
<td>8</td>
<td>81</td>
</tr>
</tbody>
</table>
Check for bugs in your tests

Hardest bugs to find are those that do not exist
• Sometimes the problem is with the test(s) and not the code
• Check your assertions
• Check your .in and .expect files (really carefully)

<table>
<thead>
<tr>
<th>test-sqr.in</th>
<th>test-sqr.expect</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>48</td>
</tr>
<tr>
<td>8</td>
<td>81</td>
</tr>
</tbody>
</table>

• Spend time developing good tests
• Start with tests!
Steps to Help you Debug Your Code

- Step 0: Try to avoid writing code with bugs
- Step 1: Triggering bugs
- Step 2: Determine the source of the problem
- Step 3: Fixing the code
- Step 4: Checking that the bug has been fixed
- Step 5: Checking if new bugs have been added (regression testing)
- Step 6: Submit to Marmoset
- Step 7: Check the results of your submission
- Step 8: Continue to learn
Step 3: Fixing the code

Beware of the quick and easy fix

Do not try randomly change code until it works

• Might work for small programs in CS 135 and CS 136 (at the start)
• Won’t work for larger, more complex programs
• Won’t work in higher-level courses
• Won’t work in a job
• This is NOT the way to learn to and understand how to program
Fix one bug at a time

If you try to fix more than one thing at a time

• Maybe one fix (change) is correct and the other isn’t
  ◦ Which one is correct and which one isn’t?
  ◦ In the end it is usually faster to fix one problem and then move on
Keep track of the things you have tried

- Make notes
- Refer back to those notes
Don’t rule out a fairly radical change

• Sometimes the best fix is to change the approach
• If the code is really complex the fix may also be really complex
• Really complex fixes can be hard to get right
• May need to simplify the code in order to get it working
  ◦ Remember the frequent use of helper functions has many upsides
Things to check (a few common mistakes)

• Using $=$ instead of $==$  
• Which is right $<=$ or $<$  
• Which is right $=>$ or $>$  
• Using $|$ instead of $||$  
• Using $\&$ instead of $\&\&$  
• Not checking return values `scanf("%d", &i);`  
• Off by one errors  
• Dereferencing a `NULL` pointer  
• Dereferencing an incorrect or dangling pointer
Is it possible the problem is more pervasive (wide spread)

- Have you possibly made the same mistake elsewhere?
  - In other functions in the program?
  - In other functions in other programs (e.g., for a different question)?
  - Possibly in questions you think you have completed and have already submitted?
Keep a log of your bugs (mistakes) and fixes

• Write them down in a notebook / file that you keep with you
• Refer back to them from time to time
• Sometimes helps you to remember the problem and the fix
  ◦ Or this seems a bit like a problem I had in the past -- might have a starting point
• Mostly sometimes helps to avoid creating the same problems in the future
Steps to Help you Debug Your Code

• Step 0: Try to avoid writing code with bugs
• Step 1: Triggering bugs
• Step 2: Determine the source of the problem
• Step 3: Fixing the code
• Step 4: Checking that the bug has been fixed
• Step 5: Checking if new bugs have been added (regression testing)
• Step 6: Submit to Marmoset
• Step 7: Check the results of your submission
• Step 8: Continue to learn
Step 4: Checking that the bug has been fixed

- Start with the test that previously triggered the bug / problem
- Are there other tests that could trigger the same or similar problem?
- Are there other tests that will exercise the changed code?
- Do they pass?
- Add some new tests to better exercise (test) the changed code?
Steps to Help you Debug Your Code

• Step 0: Try to avoid writing code with bugs
• Step 1: Triggering bugs
• Step 2: Determine the source of the problem
• Step 3: Fixing the code
• Step 4: Checking that the bug has been fixed
• Step 5: Checking if new bugs have been added (regression testing)
• Step 6: Submit to Marmoset
• Step 7: Check the results of your submission
• Step 8: Continue to learn
Step 5: Check if new bugs have been added

- This is called regression testing
- Sometimes the changes fix the original bug but introduce other bugs
  - Breaking code that previously worked
  - In this case the code is said to have “regressed” (gotten worse)
- Regression testing requires rerunning all tests that previously passed
  - Ensure that changes made don’t introduce new bugs
- If new bugs are found go back to Step 2 (or possibly 1)
Steps to Help you Debug Your Code

• Step 0: Try to avoid writing code with bugs
• Step 1: Triggering bugs
• Step 2: Determine the source of the problem
• Step 3: Fixing the code
• Step 4: Checking that the bug has been fixed
• Step 5: Checking if new bugs have been added (regression testing)
• Step 6: Submit to Marmoset
• Step 7: Check the results of your submission
• Step 8: Continue to learn
Step 6: Submit to Marmoset

• If you changed other questions remember to resubmit them too

• Remember that your goal is to find and fix all bugs before submitting to Marmoset
  ◦ In a job your goal should be to find and fix all bugs before
    a) You release the code for others to test
    b) You release the code for others to use
    c) You release the product

  The later bugs are found the more costly they are to fix!!!
  Fixing code after it has been released to production is not cool
Steps to Help you Debug Your Code

• Step 0: Try to avoid writing code with bugs
• Step 1: Triggering bugs
• Step 2: Determine the source of the problem
• Step 3: Fixing the code
• Step 4: Checking that the bug has been fixed
• Step 5: Checking if new bugs have been added (regression testing)
• Step 6: Submit to Marmoset
• Step 7: Check the results of your submission
• Step 8: Continue to learn
Step 7: Check the results of your submission

- After submitting check that all of the public Marmoset tests pass
- If the public tests fail there is a high chance that the secret (private) tests also fail
- If you have changed and submitted code in multiple questions don’t forget to check the results of ALL new submissions.
Steps to Help you Debug Your Code

• Step 0: Try to avoid writing code with bugs
• Step 1: Triggering bugs
• Step 2: Determine the source of the problem
• Step 3: Fixing the code
• Step 4: Checking that the bug has been fixed
• Step 5: Checking if new bugs have been added (regression testing)
• Step 6: Submit to Marmoset
• Step 7: Check the results of your submission
• Step 8: Continue to learn
Step 8: Continue to learn

• Create a log of your bugs and fixes
  ◦ Refer back to them from time to time

• Learn from the others (read some books)
  ◦ e.g., “Code Complete” (2nd edition), by Steve McConnell