Unit Testing

Due this week

- **Homework 4**
  - Write solutions in VSCode and paste in Autograder, **Homework 4 CodeRunner**.
  - Zip your .cpp files and submit on canvas **Homework 4**. Check the due date! **No late submissions!!**
- No Quiz this week
- 3-2-1 due today

Today

- Function prototype (uhhh we already learned this)
- Unit testing

Warm up Activity (will be shorter this time I promise)

- Create a function called my_sum that has two int parameters and returns an int that is the simply the sum of the two params.
- Create another function, *also* called my_sum that has *three* int parameters and returns an int that is simply the sum of the *three* params.
- Test out both functions in your main function to confirm they work as expected

Unit Testing

First C++ program

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World“ << endl;
    return 0;
}
```

Good introduction:

- program is short
- logic is simple: direct inspection

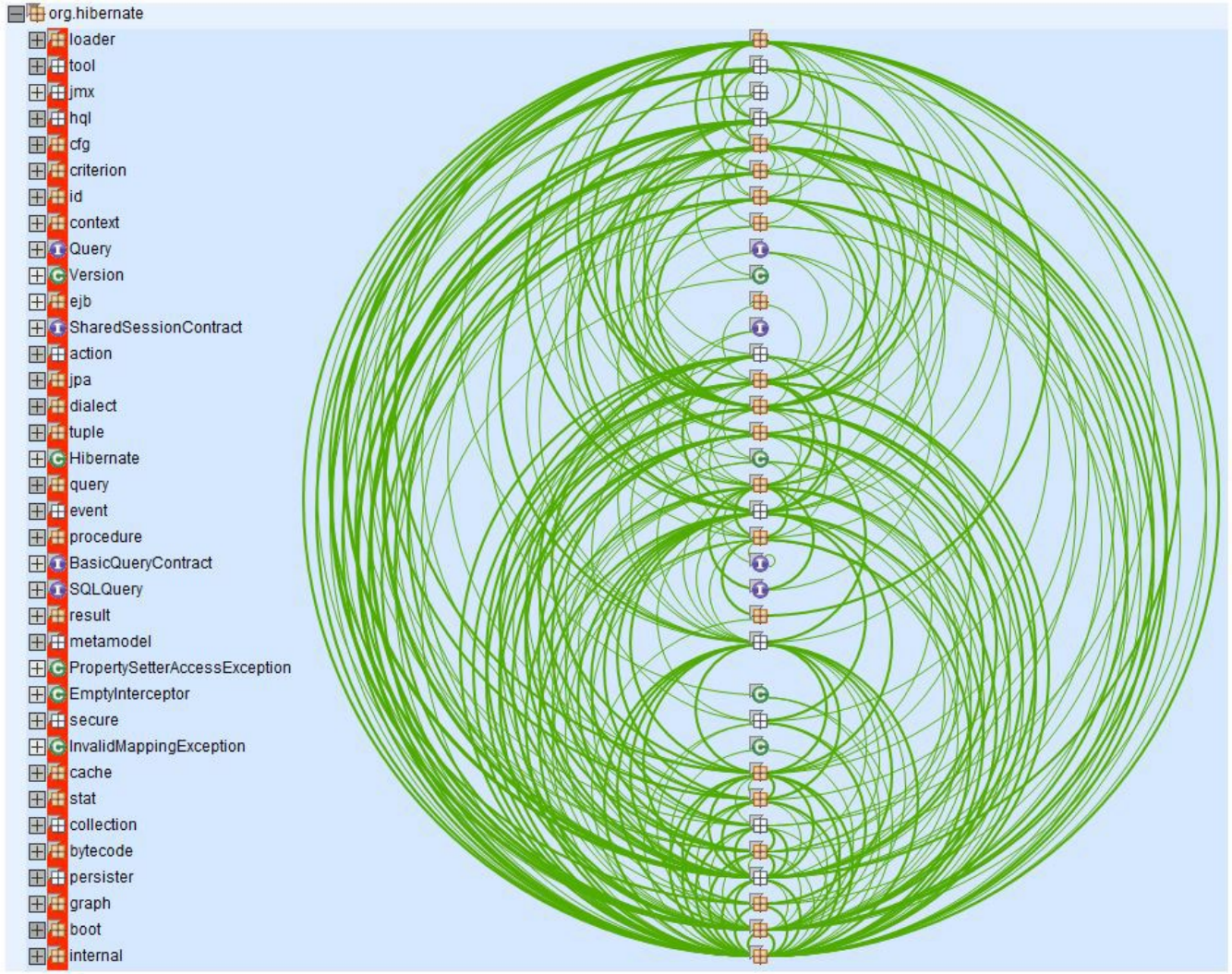
New student: “testing is pointless and adds unneeded complexity”

Real world programs

- There are decisions to be made
  - multiple paths of execution
- Decisions are based on:
  - user input,
  - data from streams, …

The programmer strives to control the inputs and the results of these decisions…

…but … once it gets too big …



Testing approaches

1. Implement, then test:
   - develop test cases
   - run the program with different inputs
   - check output/performance
   - if it fails, fix it

Big improvement already. But …

..if the whole program is tested at once, it is nearly impossible to develop test cases that clearly indicate what the failure is.

Testing approaches

2. Split and simplify
   - test small units
   - one unit tests one job or one concept
   - layered approach – goes hand in hand with the layered approach to the original development

Simplest layer: **unit testing**

A unit is the smallest conceptually whole segment of the program. Examples of basic units might be a single class or a single function.

Unit testing

White-box testing = taking into account the internal structure of the program.

→ are the variables what we think they should be?

1. Test functions in isolation:
   - write a short program, called a ***test harness***, that calls the function to be tested and verifies that the results are correct.
   - When the program completes without an error message, then all the tests have passed.
   - If a test fails, then you get an error message, telling you which test failed.

Unit testing

Example: a unit test for the `removeWhitespace` function might look like this:

```
int main()
{
    assert(removeWhitespace(“  “) == "");
    assert(removeWhitespace(“Hello“) == “Hello");
    assert(removeWhitespace(“Oh Hello“) == “OhHello");
    assert(removeWhitespace(“ Wo lo lo “) == “Wololo");
    assert(removeWhitespace(“”) == "");
}
```

Note: `removeWhitespace()` takes a string as a parameter and returns a string with all spaces removed

What is an *assert* statement?

Assertions are statements used to test assumptions made by programmer.

```
#include <stdio.h>
#include <assert.h>
int main()
{
    int x = 7;
    /*  Some big code in between and let's say x is accidentally changed to 9  */
    x = 9;

    // Programmer assumes x to be 7 in rest of the code
    assert(x==7);

    /* Rest of the code */
    return 0;
}
```

assert

Assertions are statements used to test assumptions made by programmer.

What assumptions?

```
int main()
{
     assert(int_name(19) == "nineteen");
}
```

Expected values:
if I pass 19 to the function, it should return the string “nineteen”

What to test?

Selecting test cases is an important skill.

- test inputs (parameters) that a typical user might supply.
- **boundary cases** *(or edge cases)*.
  - Boundary cases for the *removeWhitespace* function are:
    - the smallest valid input (empty string)
    - White spaces at beginning and/or end
    - No whitespace
- **test coverage** = You want to make sure that each part of your code is exercised at least once by one of your test cases.
  - look at every if/else *branch*

Jargon

- **test suite** = collection of test cases
- **regression testing** = testing against past failures
- **unit test framework** = have been developed for C++ to make it easier to organize unit tests. These testing frameworks are excellent for testing larger programs, providing good error reporting and the ability to keep going when some test cases fail or crash.

Example: *fizzbuzz*

Fizz Buzz Test

The "Fizz-Buzz test" is an interview question designed to help filter out the 99.5% of programming job candidates who can't seem to program their way out of a wet paper bag. The text of the programming assignment is as follows:

> *"Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz"."*

Example: *fizzbuzz*

Give me examples of tests you need to run!

1. Can we call the function? Does it compile? Are there are syntax errors?
2. Output “1” when I pass 1
3. Output “2” when I pass 2
4. Output “Fizz” when I pass 3
5. Output “Buzz” when I pass 5
6. Output “Fizz” when I pass 9 (multiple of 3)
7. Output “Buzz” when I pass 10 (multiple of 5)
8. Output “FizzBuzz” when I pass 15 (multiple of 3 and of 5)

Test-Driven Development

TDD: the practice of writing unit tests before you write your code

- You know what your program should do, so you can write the unit tests first!

Benefits:

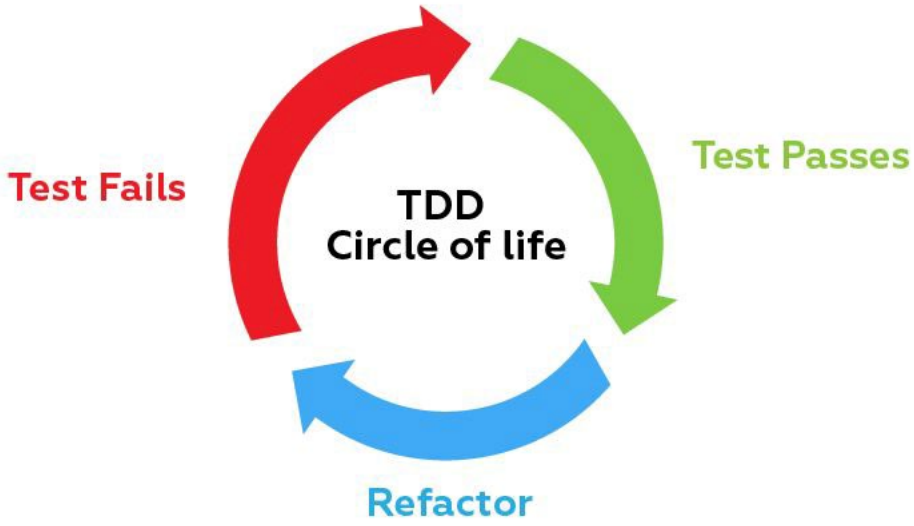
- Every line of code is working as soon as it is written, because you can test it immediately
- If there is a problem, it is easy to track down because you have only written a small amount of code since the last test

Test-Driven Development

TDD cycle:

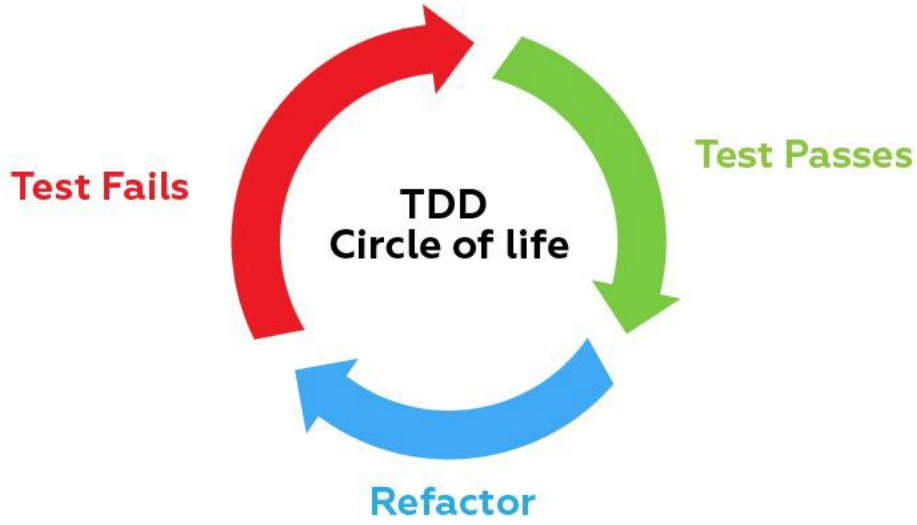
For each test:

- Write the test
- It will probably initially fail
- Fix the implementation (add to it/modify it)
  - Run the test again… and again and again…
  - Stop when the test passes



General Recommendations

- Your test cases should only test one thing
- Test case should be short
- Test should run fast, so it will be possible to run it often
- Each test should work independent of the other tests
  - Broken tests shouldn’t prevent other tests from running
- Tests shouldn’t be dependent on the order of their execution



Debugging your functions -- your code runs but spits out garbage!

Typical debug session:

1. Run code
2. Code does not work
3. Print key variable values out at different points in the source code
    1. Determine where the code breaks by comparing variable values to what you expect
    2. Determine what might be going wrong and correct it
    3. Return to Step 1

Debugging your functions -- your code does not even run!

Typical debug session:

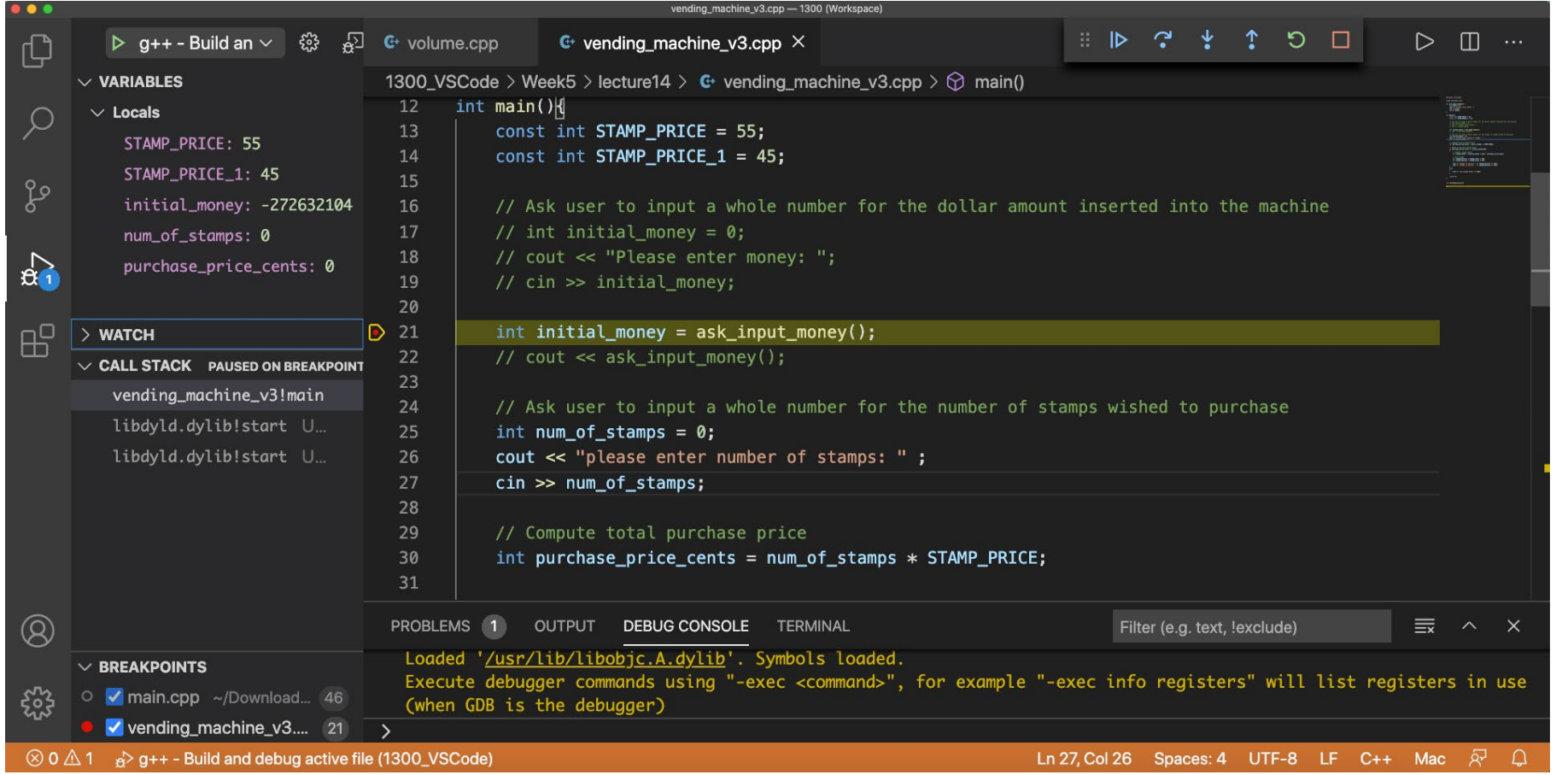
1. Run code
2. Code won’t compile
3. Move the return statement closer and closer to the beginning of the function
    1. Determine where the code breaks by finding out when the code actually compiles and runs
    2. Determine what might be going wrong and correct it
    3. Return to Step 1

Using the IDE Debugger

Your VS Code IDE can be set up to use a debugger that:

- Allows execution of the program one statement at a time
- Shows intermediate values of local function variables
- Sets “breakpoints” to allow stopping the program at any line to examine variables

- These features greatly speed up correcting your code.



- There is a breakpoint on line 21.
- Next line to be executed is shown by yellow arrow in the Breakpoint margin at left.
- The Debugger panel at right shows the Local Variables: STAMP_PRICE, STAMP_PRICE_1, num_of_stamps, purchase_price_cents
- initial_money has not been initialized yet.

Using the IDE Debugger

Typical debug session:

1. Set a breakpoint early in the program, by clicking on a line in the source code
2. Start execution with the green “Run” triangle, *from the Debug panel*
3. When the code stops at the breakpoint, examine variable values in the variables window
4. Step through the code one line at a time or one function at a time, continuing to compare variable values to what you expected
5. Determine the error in the code and correct it, then go back to step 1

Best practice to avoid needing those last few slides

1. Start with the **simplest possible case** for your function
2. Add in layers of complexity **incrementally**
3. **Test your work** frequently

    Do this as you are adding in these layers of complexity
4. **Save your work** frequently