Lecture 8: Unit Testing

Announcements and reminders

Submissions:

- HW 3 -- due Saturday at 6 PM

Course reading to stay on track:

- 5.6 - 5.9 today
- 3.1-3.2, 3.7-3.8 by Monday
- 3.3-3.6 by Wednesday

Last time on *Starting Computing*…

We learned how to pass parameters into a function and send return values back out!

We learned about functions to perform sets of tasks without return values!

→ `void` functions

Function design -- keep it short and sweet!

There is a cost to writing a function

- You need to **design, code and test** the function
- The function needs to be **documented**
- You spend some effort to make the function **reusable** rather than tied to a specific context

➪ Tempting to write long functions to minimize the number of functions needed and the overhead

➪ ***BUT*** as a rule of thumb, a function that is too long to fit on a single screen should be broken up

First C++ program

```
#include <iostream>
using namespace std;
int main()
{
    cout << “Hello world!” << endl;
    return 0;
}
```

Good introduction:

- Program is **short**
- Logic is **simple**

But… sends the message that testing is pointless
and adds needless extra work

Real-world programs

There are decisions to be made

- Multiple paths of execution

Decisions are based on…

- User input
- Data from streams (files, user, etc…)

The programmer strives to control the inputs and 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 entire 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 test -- 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.

Testing approaches

2. Split and simplify

- Test small units
- One unit test -- 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.

Example: program to compute volume

Unit testing

For each unit, the tester (who may or may not be the programmer) attempts to determine what states the unit can encounter while executing as part of the program.

- Determining the range of appropriate inputs to the unit
- Determining the range of possible inappropriate inputs
- Recognizing any ways the state of the rest of the program might affect execution in this unit

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 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 `int_name` function might look like this: (can you tell from these tests what `int_name` should do?)

```
int main():
{
     assert (int_name(19) == "nineteen");
     assert (int_name(29) == "twenty nine");
     assert (int_name(1091) == "one thousand ninety one");
     assert (int_name(30000) == "thirty thousand");
     return 0;
}
```

assert(...)

→ **Assertions** are statements used to test assumptions made by the programmer

```
#include <iostream>
#include <assert.h>
int main()
{
    int x = 7;
```

Suppose in some intermediate code here, x is accidentally changed to 9...

```
    x = 9;
    // Programmer assumes x to be 7 in the rest of the code
    assert(x == 7);
    //  Rest of the code ...
    return 0;
}
```

assert(...)

→ **Assertions** are statements used to test assumptions made by the programmer

What assumptions?

```
int main():
{
    assert(int_name(19) == "nineteen");
    return 0;
}
```

Expected values:

“If I pass 19 into the function, it should return the string “nineteen”

assert(...)

→ **Assertions** are statements used to test assumptions made by the programmer

What assumptions?

```
int main():
{
    assert(int_name(19) == "nineteen");
    return 0;
}
```

Expected values:

"If I pass 19 into the function, it should return the string "nineteen"

Example: Let's refactor our tests for `cube_volume()` from last time using `assert(...)`. WOO!!

What to test?

Selecting test cases is an important skill

- Inputs that a typical user might supply
- **Boundary cases** (or **edge cases**)
  - Boundary cases for the `cube_volume` might include:
    - The smallest valid input
    - The largest valid input?
    - Non-integer input
    - Negative input?
- 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 possible branch within your code

Some techno-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

The fizzbuzz test

The **fizz-buzz test** is an interview question designed to help filter out the 99.5% of programming job applicants who can't 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 3 print "Fizz" instead of the number and for multiples of 5 print "Buzz". For numbers that are multiples of both 3 and 5 print "FizzBuzz".

What are some tests we need to run for our FizzBuzz program?

The fizzbuzz test

What are some tests we need to run for our FizzBuzz program?

The fizzbuzz test

What are some tests we need to run for our FizzBuzz program?

1. Can we call the function? Does it compile? Are there 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 5)

The fizzbuzz test

What are some tests we need to run for our FizzBuzz program?

1. Can we call the function? Does it compile? Are there 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 5)

→ Let’s look at a solution (possibly buggy) and run our **batch of tests**

fizzBuzzBroken.cpp

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

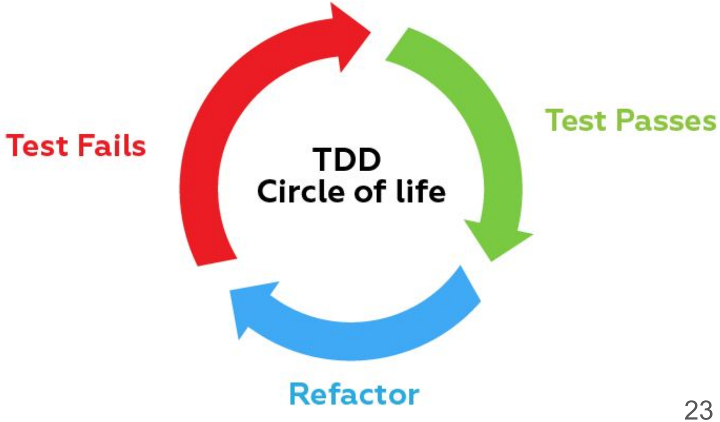
TDD: the practice of writing unit tests before you write your code

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

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



Test-Driven Development

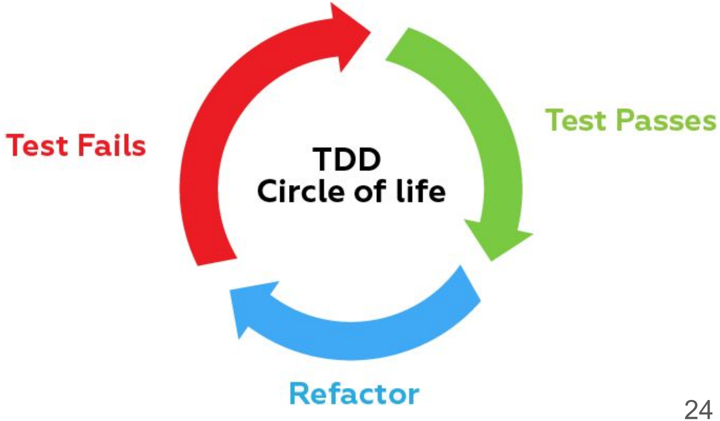
TDD: the practice of writing unit tests before you write your code

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

TDD cycle:

For each test:

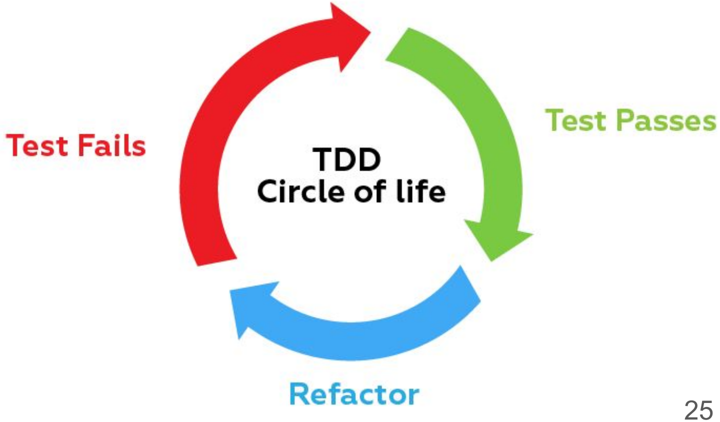
- Write the test
- It will probably initially fail
- Fix the implementation (add to it/modify it)
  - Run the test again… and again and again…
  - Stop when the test passes



Example: ***volumes.cpp***

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
    a) Determine where the code breaks by comparing variable values to what you expect
    b) Determine what might be going wrong and correct it
    c) 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
    a) Determine where the code breaks by finding out when the code actually compiles and runs
    b) Determine what might be going wrong and correct it
    c) Return to Step 1

Debugging your functions -- using the IDE debugger

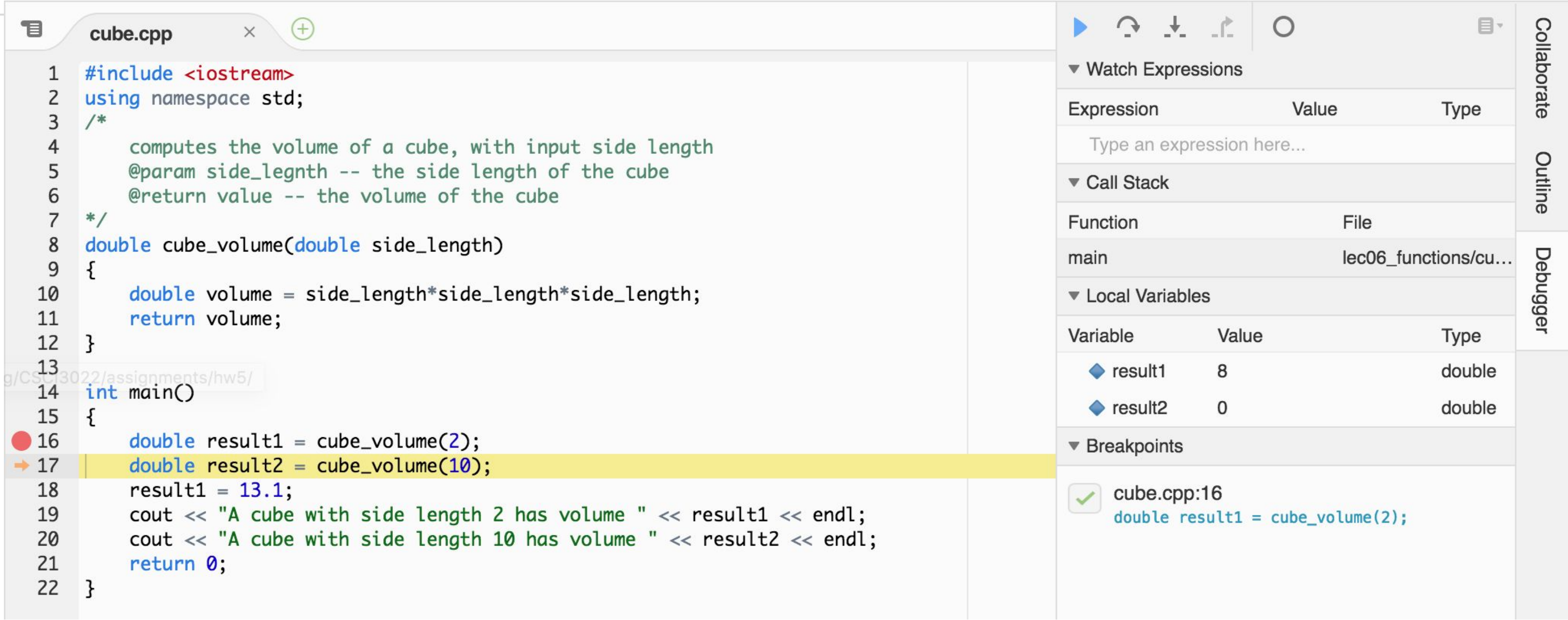
Your Cloud9 IDE includes a debugger that:

- 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 the variables

→ these nice features greatly speed up correcting code bugs!

Documentation: https://docs.c9.io/docs/debugging-your-code

Debugging your functions -- using the IDE debugger



- There is a **<u>breakpoint</u>** at line 16
- **<u>Yellow arrow/highlight</u>** shows next line to be executed
- **<u>Debugger panel</u>** at right shows the **<u>Local Variables</u>**
  -- `result1` has a value already, but `result2` does not (it just happens to be 0)

Debugging your functions -- using the IDE debugger

Typical debug session:

1) Set a breakpoint early in the program by clicking on a line in the source code (in ***gutter***)

2) Click the wee bug to turn on debugging → 

3) Start execution with the “Run” button → 

4) When the code stops at the breakpoint, examine variable values in the variables window

5) Step through the code one line at a time, or one function at a time, continuing to compare variable values to what you expected

6) Determine any errors in the code and correct them, 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

What just happened…?

We learned how to build our functions from the bottom-up!

We learned how to test our software!

… and how to build our software and test it as we go!