WE ADDED A NEW PERFORMANCE TEST, BUT LEARNED THAT THE TEST ITSELF IS FLAWED.
Dilbert.com DilbertCartoonist@gmail.com
NOW OUR PRODUCT FAILS OUR OWN TESTS AND OUR CUSTOMERS ARE ASKING TO SEE THE TEST RESULTS.
8-11-10 ©2010 Scott Adams, Inc./Dist. by UFS, Inc.
DO I HAVE PERMISSION TO FAKE THE TEST DATA?
I DIDN'T EVEN KNOW DATA CAN BE REAL.

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**.
- 3-2-1 – due today
- No Quiz this week
- Check the due date! **No late submissions!!**

Recap

- A function is a sequence of instructions with a name
  - Re-usable, easy to read, avoids repetition etc.
- Function definition
  - `return_type functionName(formal parameters){}`
  - Formal parameters will have a data type; int x
- Function call
  - `[variable = ]functionName(function arguments)`
  - Function arguments will only have variable names or values; int x or 10
- Scope indicates the lifetime of a variable. Any variable declared inside a function is only accessible inside that function

Today

- Function prototype
- Unit testing

Function Prototype

Function Declarations (Prototype Statements)

- It is a compile-time error to call a function that the compiler does not know
  - just like using an undefined variable.

- So define all functions before they are first used
  - But sometimes that is not possible, such as when 2 functions call each other

Function Declarations (Prototype Statements)

- Therefore, some programmers prefer to include a definition, aka "prototype" for each function at the top of the program, and write the complete function after `main(){}`

- A prototype is just the function header line followed by a semicolon:

```
double cube_volume(double side_length);
```

- The variable names are optional, so you could also write it as:

```
double cube_volume(double);
```

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

// Declaration of cube_volume
double cube_volume(double side_length);

int main()
{
   double result1 = cube_volume(2); // Use of cube_volume
   double result2 = cube_volume(10);
   cout << "A cube with side length 2 has volume "<< result1<< endl;
   cout << "A cube with side length 10 has volume "<< result2<< endl;
   return 0;
}

// Definition of cube_volume
double cube_volume(double side_length)
{
   return side_length * side_length * side_length;
}
```

Function Declaration (prototype)

Common error: No function declared before encountering function call in `main()`

```
int main()
{
   double volume = cube_volume(2.0);
}


double cube_volume(double side_length)
{
   return side_length * side_length * side_length;
}
```

Steps to Implementing a Function

1. Describe what the function should do.
   - EG: Compute the volume of a pyramid whose base is a square.
2. Determine the function's "inputs".
   - EG: height, base side length
3. Determine the types of the parameters and return value.
   - EG: `double pyramid_volume(double height, double base_length)`
4. Write pseudocode for obtaining the desired result.

```
volume = 1/3 x height x base length x base length
```

5. Implement the function body.

```
{
    double base_area = base_length * base_length;
    return height * base_area / 3;
}
```

6. Test your function
   - Write a `main()` to call it multiple times, including boundary cases

Good Design – Keep Functions Short

- There is a certain cost for writing a function:
  - You need to **design, code, and test** the function.
  - The function needs to be **documented**.
  - You need to spend some effort to make the function **reusable** rather than tied to a specific context.

- So it’s tempting to write long functions to minimize their number 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.

Long functions are hard to understand and to debug

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

/**   Computes the volume of a pyramid whose base is a square.
   @param height the height of the pyramid
   @param base_length length of one side of the pyramid′s base
   @return the volume of the pyramid
*/
double pyramid_volume(double height, double base_length)
{
   double base_area = base_length * base_length;
   return height * base_area / 3;
}

int main()
{
   cout << "Volume: " << pyramid_volume(9, 10) << endl;
   cout << "Expected: 300";
   cout << "Volume: " << pyramid_volume(0, 10) << endl;
   cout << "Expected: 0";
   return 0;
}
```

Unit Testing

First C++ program

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

int main()
{
    cout << "Hello World\n";
}
```

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

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 parameters to the unit,
- determining the range of possible inappropriate parameters,
- 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 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 `int_name` function might look like this:

```
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");
}
```

Note: `int_name()` takes an integer as a parameter and returns a string

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 *int_name* function are:
    - 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 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.

Test-Driven Development

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

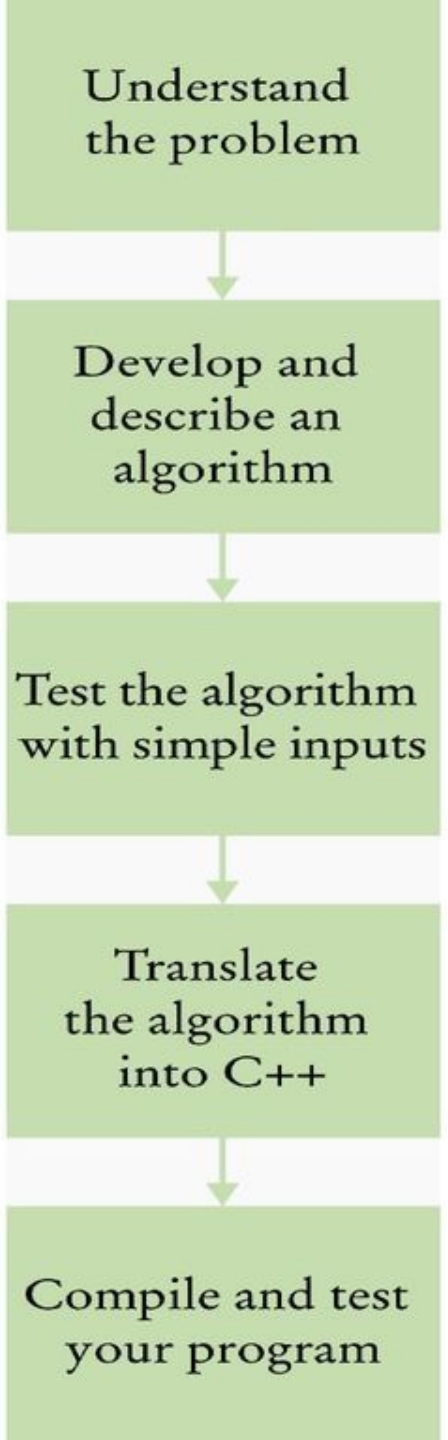
- You know what your program should do, so you can <u>write the unit tests first!</u>

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

The Software Development Process

- For each problem the programmer goes through these steps
- ***You MUST write an algorithm in words, pictures, and/or equations before attempting to translate to C++***

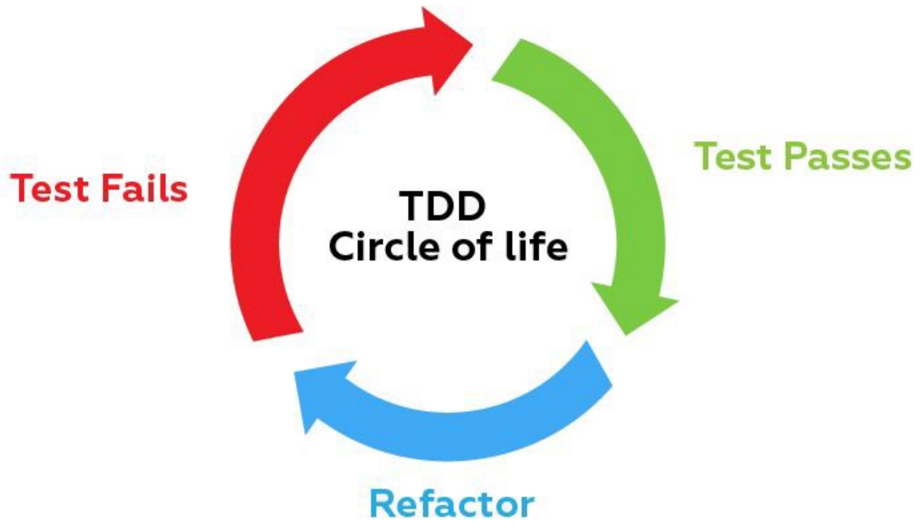

Understand the problem
Develop and describe an algorithm
Test the algorithm with simple inputs
Translate the algorithm into C++
Compile and test your program

Test-Driven Development

TDD cycle:

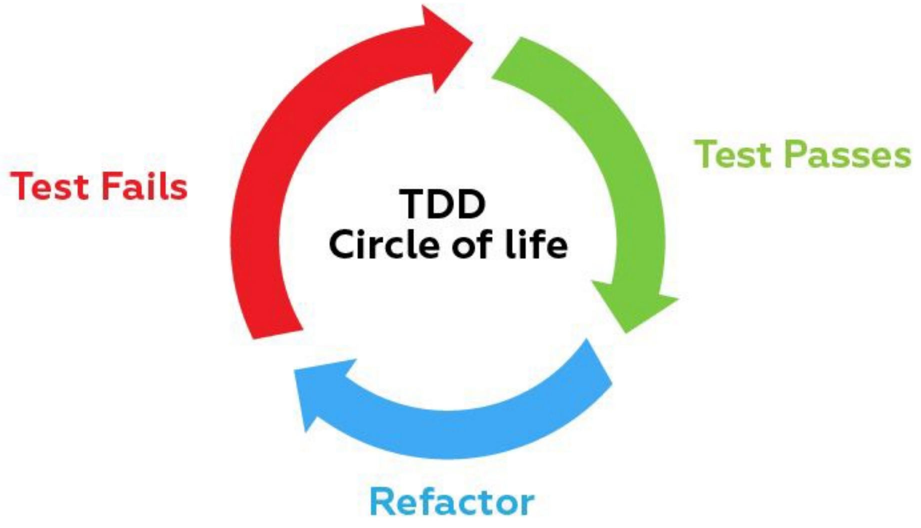
For each test:

- 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:

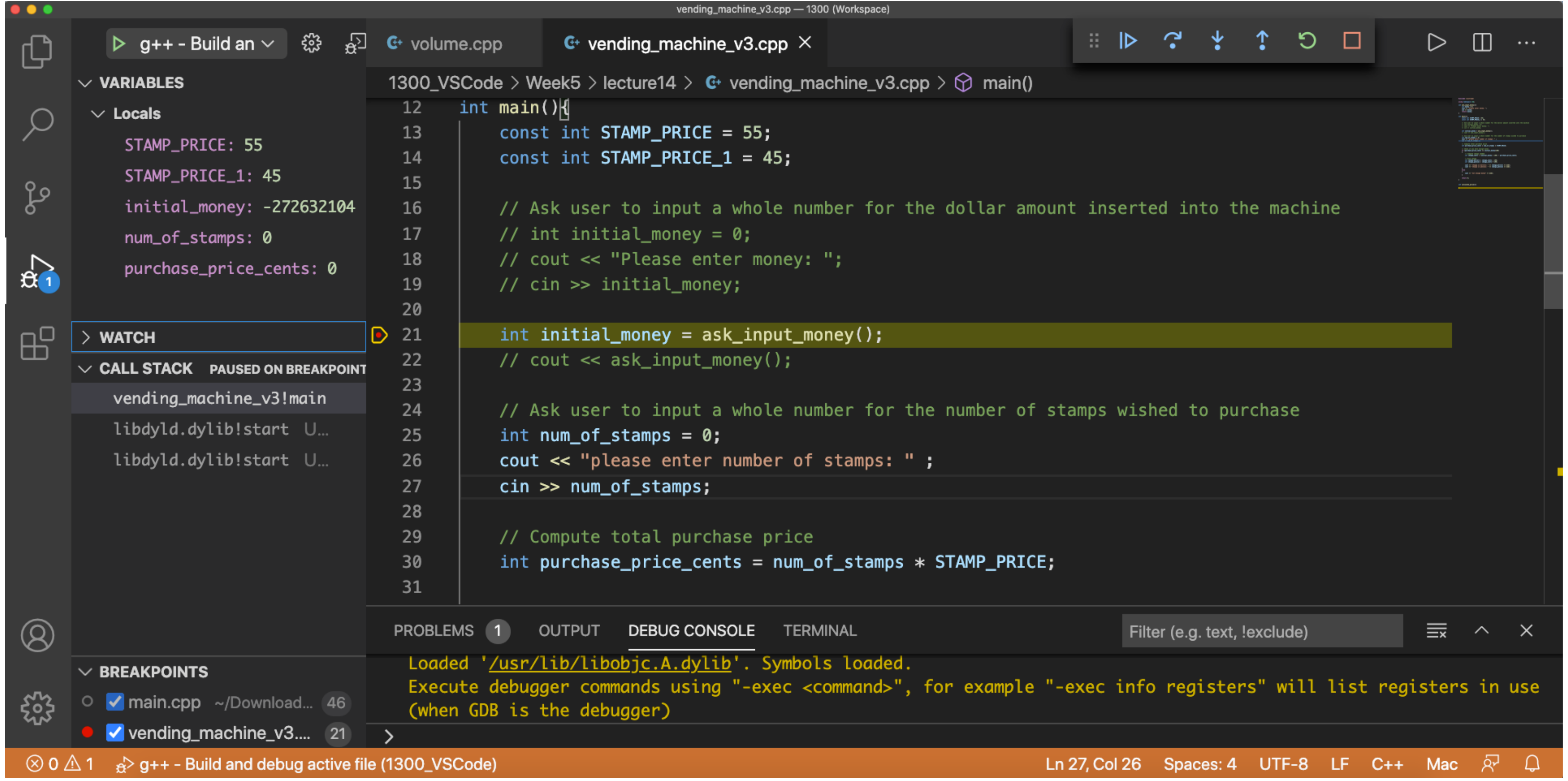
1. Run code
2. Code won’t compile
3. Move the return statement closer and closer to the beginning of the function
    1. Determine where the code breaks by finding out when the code actually compiles and runs
    2. Determine what might be going wrong and correct it
    3. Return to Step 1

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.


vending_machine_v3.cpp — 1300 (Workspace)
g++ - Build an
volume.cpp
vending_machine_v3.cpp
1300_VSCode > Week5 > lecture14 > vending_machine_v3.cpp > main()
VARIABLES
Locals
WATCH
CALL STACK PAUSED ON BREAKPOINT
BREAKPOINTS
PROBLEMS 1
OUTPUT
DEBUG CONSOLE
TERMINAL
Filter (e.g. text, !exclude)

```
STAMP_PRICE: 55
STAMP_PRICE_1: 45
initial_money: -272632104
num_of_stamps: 0
purchase_price_cents: 0
```

```
vending_machine_v3!main
libdyld.dylib!start  U...
libdyld.dylib!start  U...
```

```
main.cpp  ~/Download...  46
vending_machine_v3....  21
```

```
12  int main(){
13      const int STAMP_PRICE = 55;
14      const int STAMP_PRICE_1 = 45;
15
16      // Ask user to input a whole number for the dollar amount inserted into the machine
17      // int initial_money = 0;
18      // cout << "Please enter money: ";
19      // cin >> initial_money;
20
21      int initial_money = ask_input_money();
22      // cout << ask_input_money();
23
24      // Ask user to input a whole number for the number of stamps wished to purchase
25      int num_of_stamps = 0;
26      cout << "please enter number of stamps: " ;
27      cin >> num_of_stamps;
28
29      // Compute total purchase price
30      int purchase_price_cents = num_of_stamps * STAMP_PRICE;
31
```

```
Loaded '/usr/lib/libobjc.A.dylib'. Symbols loaded.
Execute debugger commands using "-exec <command>", for example "-exec info registers" will list registers in use
(when GDB is the debugger)
```

```
⊗ 0 ⚠ 1   g++ - Build and debug active file (1300_VSCode)   Ln 27, Col 26   Spaces: 4   UTF-8   LF   C++   Mac
```

- 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

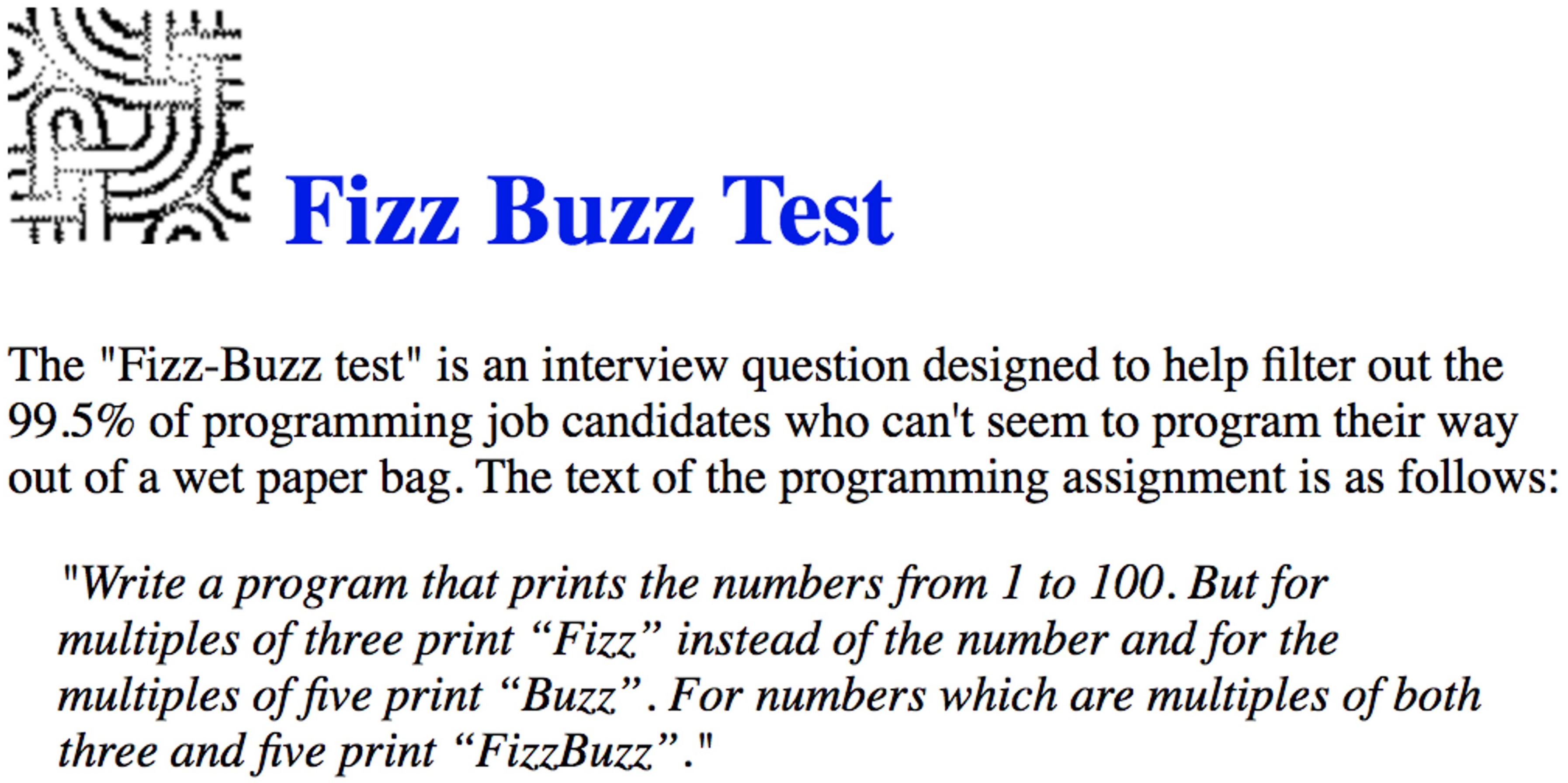
Best practice to avoid needing those last few slides

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

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)