From week 2....

- Write perfect C code to solve the three problems below.
  - You are **not** allowed to use any library functions from `string.h`
  - Bring a printed copy of your solutions to our next class

- Write a function to determine whether a given string called `needle` is a substring of another string called `haystack`

- Write a function to count how many times a given string called `needle` is a substring of another string called `haystack`

- Write a function to determine the longest palindrome in a given string called `wow`
From week 2....

- Write perfect C code to solve the three problems below.
  - You are not allowed to use any library functions from `string.h`.
  - Bring a printed copy of your solutions to our next class.
- Write a function to determine whether a given string called `needle` is a substring of another string called `haystack`.
- Write a function to count how many times a given string called `needle` is a substring of another string called `haystack`.
- Write a function to determine the longest palindrome in a given string called `wow`.

Code Reviews:
1. Form groups of 3-8 students each.
2. Select (randomly?) someone's code to review.
3. Select (also randomly?) someone to document each identified defect (and possible solution).
4. The developer who wrote the code has ~2 minutes to introduce and describe his or her code.
5. Participants take turns critiquing the code, asking questions, making suggestions, etc.
6. The note-taker summarizes the findings and calculates defect density and code review efficacy.
7. Go back to step (2).

Beyond just improving the quality of the reviewed code, other side effects include:

- Incorporating new developers (or old developers new to your team) into the team, into the code base, into the developer culture, etc.
- Sharing changes and general knowledge of the system, thereby increasing the knowledge of all developers and eliminating single points of failure.
- Encouraging collaboration among team members.
- Leveling the playing field among new and veteran developers.
- Helping to achieve an underlying consistency and uniformity in the (ever-expanding) code base.
What is software quality?

- External qualities are visible to system users, observers, etc.
- Internal qualities are visible only to developers
- Products are anything we produce during the development process; we want all products to be of high quality
- Processes are the actual business procedures we follow to produce the software system; these processes should also be of high quality (e.g., code reviews)
- Tradeoffs are often necessary here...

What is software quality?

- Representative qualities include:
  - Correctness
  - Reliability/Dependability
  - Robustness
  - Usability
  - Verifiability
  - Maintainability
  - Extensibility
  - Reusability
  - Portability
  - Flexibility
  - Readability
  - Understandability
  - Repeatability
  - Predictability
  - Timeliness
  - Efficiency/Performance
  - Visibility
  - Simplicity
  - Functionality
  - Testability

Can we quantify any of these qualities?
How do we test software?

- Our approach to testing must identify:
  - What representative subset of possible test cases do we test?
  - When do we start testing?
    - And when do we feel confident enough to stop?
  - Who does the testing?
    - And how do they communicate successes and failures to the rest of the team?
  - What types of testing do we do?
    - Unit testing (and black-box versus white-box testing)
    - System integration testing
    - User acceptance testing
    - Regression testing

Unit testing

- Quality testing has a high probability of finding yet-undiscovered bugs and defects
- Unit testing focuses on testing individual program units in isolation from one another
- Goals include repeatability and comprehensive coverage
- Repeatability enables automation
- Comprehensive coverage ensures correctness
- Key problem is identifying a representative subset of all possible test cases (since exhaustive testing is usually infeasible)
What cases do we test for?

- Typical and generally expected input values (depends on the program unit being tested)
- Boundary or edge cases, including minimum and maximum input values
- Valid values near the boundary cases (e.g., maximum – 1)
- Invalid values near the boundary cases (e.g., maximum + 1)
- Unusual cases, including zero, the empty string, NULL, etc. (depends on the program unit being tested and data types)
- Does the behavior match the given specifications/expectations? And is anything missing?

Scaling up our verification process

- How do we scale up our verification process beyond unit testing?
- In principle, we need to verify everything, but how?
- And how can we verify the validity of the tests themselves?
- We must also test every aspect of software quality that we (or our users) care about
- Good (i.e., comprehensive) testing requires different people testing at different levels and at different times
Verification results may vary

- Verification (i.e., test) results may vary in a variety of ways:
  - Results may not necessarily be “pass” of “fail”
  - Results may differ on different test runs, which likely implies shortcomings of the tests themselves (but could also be a result of race conditions)
  - What does it mean to pass all of the tests?

![image of a scale with 0%, fail, pass, and 100%]

Verification may be subjective

- Much of what we test is objective
  - We can measure (and confirm) the results of an objective activity (i.e., an individual test case)
  - e.g., verifying specific output, validating response times, etc.
- Not all qualities can be (easily) objectively quantified
  - What are some examples here?
  - In practice, objective measures may be used to develop subjective estimates
  - e.g., lines of code, number of classes/modules/components, etc.
  - e.g., reusability and/or extensibility might be measured in how long it takes to support a new feature, device type, document type, etc.
Verifying implicit requirements

- Many requirements specifications ignore a number of crucial aspects of software quality
  - e.g., maintainability, reusability, extensibility, reliability, etc.
- As an example, consider what performance requirements might be important for a search engine
  - How about for an ATM?
  - For the Starbucks app?
- There is often a “normal” or “normal behavior” defined (and verified) in specifications and test cases
  - How do we ensure that the abnormal (i.e., <5%?) is properly handled?

Classify each quality based on whether they are:

1. Objective or subjective
2. Binary or not (i.e., “pass” and “fail” only)

Come up with examples for each (in relation to your search engine project components):

- Representability
  - Correctness
  - Reliability/Dependability
  - Robustness
  - Usability
  - Verifiability
  - Maintainability
  - Extensibility
  - Reusability
  - Portability
  - Flexibility

- Readability
  - Understandability
  - Repeatability
  - Predictability
  - Timeliness
  - Efficiency/Performance
  - Visibility
  - Simplicity
  - Functionality
  - Testability
Software verification

- Two fundamental approaches to software verification:
  - Experimentation (i.e., dynamic approach)
  - Analysis (i.e., static approach)
- Experimentation is essentially well-organized trial and error
- Analysis is a structured approach to deduce and prove (or disprove) the correctness of software
  - How can be applied to other aspects of software quality?

Why do bugs exist?

- We often jump into a project with a “code first” approach
- Once we scale up beyond small units of code, the complexity of the code dramatically increases
- There are too many things that could go wrong; we can never be sure we’ve considered (and handled) every error condition
- Regardless, all bugs are caused by human error....
What types of bugs are there?

- At the coding level, there are many types of common bugs:
  - Uninitialized or improperly initialized variables
  - Rounding or precision errors
  - Library misuse errors (so be sure you read the manual!)
  - Errors in libraries (or other code that you rely upon)
  - Mishandled edge cases
  - Off-by-one errors
  - Incomplete error handling
  - Synchronization (i.e., timing) problems, race conditions, deadlocks
  - Memory leaks

How do we find the last bug?

- Testing is essentially an approximation of exhaustively testing all possibilities
- One possible approach:
  - Attempt to define classes of the given input domain
  - Identify a minimal number of representative test cases
  - What are advantages of this approach?
  - What are disadvantages or risks of this approach?
Small-scale testing

- Small-scale testing typically consists of:
  - White-box (implementation-based) testing
  - Black-box (specification-based) testing

- What approaches are there to small-scale testing?

---

Statement coverage

- Statement coverage ensures that for test set \( T \), executing program \( P \) for each test case input \( d \in T \), each elementary statement of \( P \) is executed at least once.

```c
int f( int x, int y )
{
    while ( x != y )
    {
        if ( x > y ) x -= y;
        else y -= x;
    }
    return x;
}
```
Edge and path coverage

- Edge coverage ensures that for test set $T$, executing program $P$ for each test case input $d \in T$, each edge of $P$'s control flow graph (CFG) is executed at least once.

- Path coverage tests all possible paths from the initial node to the final node of the given CFG.

- Since there typically are too many paths, for loops, consider cases in which the loop executes zero times, only once, a maximum number of times, a typical number of times, etc.

```c
int f( int x, int y )
{
    while ( x != y )
    {
        if ( x > y ) x -= y;
        else y -= x;
    }
    return x;
}
```

Condition coverage

- Condition coverage extends edge coverage by also ensuring that all possible values of the constituents of compound conditions are exercised at least once.

```c
if ( C_1 )
{
    if ( C_2 )
    { S_{true} 
        else
        S_{false}
    }
    else
    S_{false}
}
```
Example

- Apply statement, edge, condition, and path coverage to develop test cases for the given code

```java
while ( z > x )
{
    if ( x != 0 )
    {
        z += 5;
    }
    else
    {
        z -= x;
    }
}
if ( z > 1 )
{
    z /= x;
}
else
{
    z = 0;
}
```

Large-scale testing

- As we scale up, some of the same small-scale testing techniques can be applied to increasingly larger components/modules
- If we have a good design (i.e., high cohesion, low coupling), then there are far fewer components to integrate together
  - Good design techniques lead to higher levels of verifiability
- For system integration, modules are often integrated gradually
  - Developing a “roadmap” of dependencies can help determine the order in which modules should be integrated
  - e.g., if module $M_i$ uses module $M_j$, and $M_j$ uses module $M_k$, then the correct execution of $M_i$ requires correct versions of both $M_j$ and $M_k$ to be available
Large-scale testing

- For dependencies on external modules, you might use a stub, which is a simplified implementation of the given module
  - This implementation has the same input(s) and output(s), but the behavior or translation from input(s) to output(s) is hardcoded or greatly simplified
- Similarly, a driver is a small program that simulates the use of the given module being tested

Integration testing

- Integrating all components at once is called big-bang testing
- Incremental testing makes more sense:
  - A goal here is to incrementally eliminate stubs and drivers
  - We can integrate modules as they reach completion (i.e., even if other modules are still under development)
  - Determining where errors exist is substantially easier
  - We can also identify important groups of modules that might constitute larger subsystems within our entire system
Bottom-up and top-down integration

- What order should we integrate modules for testing?
- Bottom-up aggregation starts at the leaf nodes of the module dependencies tree
  - Requires drivers to simulate higher-level modules
  - Not necessarily a tree, as there could be cycles
- Top-down aggregation starts at the top-level module(s)
  - Requires stubs to simulate lower-level modules