# Week 9 Lecture 1: Testing *Part 2* and Design!

# Review and Preview

- Structures and Functions
- Testing
- More Testing
- Design

**K&R: Chapters 1-5**
**Sections 4.5, 4.11**
**Sections 5.1-5.5**
**Sections 6.1-6.3**


DESTROYED THE HOUSE
The Ultimate Tester
Testing
TIME FOR A NAP
com

# Tricks of the Trade

- **Test boundary conditions**
  - loops and conditional statements should be checked to ensure that loops are executed the correct number of times and that branching is correct
  - if code is going to failure, it usually fails at a boundary
  - check for off-by-one errors, empty input, empty output

# Tricks of the Trade

- **Test pre- and post-conditions**
  - before and after a critical piece of code, check to see if the necessary pre- and post-conditions (value of variables, etc) exist
  - remember a famous divide by zero error

*On the USS Yorktown (guided-missile cruiser), a crew member entered a zero, which resulted in a divide by zero, the error cascaded and shut down the propulsion system and the ship drifted for a couple of hours.*

# Tricks of the Trade

- **Program defensively**
  - Test for cases which “can't occur” just in case!
  - Test for extreme values - negative values, huge values, etc

# Tricks of the Trade

```
if ( grade < 0 || grade > 100 ) /* impossible grades! */

        letter = '?';

else if ( grade >= 80 )

        letter = 'A';

else   …
```

Always check for NULL pointers, out of range subscripts, divide by zero.

# Tricks of the Trade

- **Check returns**
  - Always look at the values returned from library functions and system calls.
  - Remember to check for output errors and failures as well as input failures.

# Testing Strategies

- **Test incrementally.**
  - Write, test, add more code, test again, repeat.  And repeat.  And repeat.

http://3.bp.blogspot.com/-uUuUxmCT-K0/UFThsJghKII/AAAAAAAAAww/Lo0Z_OUj1G4/s1600/repeater6.jpg

# Testing Strategies

- **Test simple parts first**
  - Focus on testing the simplest features and only move on when these are working.
  - Enumerate the possible scenarios for the code and design tests for all of them.
  - Test individual functions (another good reason to modularize) using testing code to rigorously and automatically test a function or piece of code.

```c
#include <stdio.h>
#include <string.h>
#include "textProc.h"

#define MAXSIZE  500  /* max size of input line */

int main ( int argc, char *argv[] ) {
   char line[MAXSIZE];
   int ret;

   while ( fgets(line, MAXSIZE, stdin) != NULL ) {
      ret = replaceDigits ( line );
      printf ( "%d - %s", ret, line );
   }
   return ( 0 );
}
```

# Test 1 - Replace digits with spaces and check return value and text

---

replaceDigitsTestFile.txt

This is test #19 in test
suite 17 on this day in
2019 and for 2019 and 2020.
Test: 0123456789

```
$ cat replaceDigitsTestFile.txt | ./testReplaceDigits
```

2 - This is test #  in test
2 - suite  on this day in
12 -  and for  and  .
10 - Test:

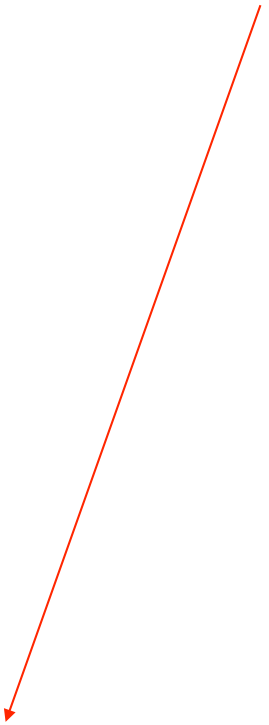
```c
#include <stdio.h>
#include <string.h>
#include "textProc.h"

#define MAXSIZE  500     /* max size of input line */

int main ( int argc, char *argv[] ) {
   char line[MAXSIZE];
   int ret = 0;

   while ( fgets(line, MAXSIZE, stdin) != NULL ) {
      ret = replaceDigits ( line );
      printf ( "%s", line );
   }

   return ( 0 );
}
```

# Test 2: Check that the digits are replaced with spaces

---

```
$ cat replaceDigitsTestFile.txt | ./test2ReplaceDigits > testfile.txt
```

```
$ wc replaceDigitsTestFile.txt testfile.txt
  4  20  94 replaceDigitsTestFile.txt
  4  16  94 testfile.txt
  8  36 188 total
```



**>** is called **redirection**

The output on stdout is put into the file testfile.txt

# Test 2: Check that the digits are replaced with spaces

```
$ od -c testfile.txt
0000000  T   h   i   s       i   s       t   e   s   t       #
0000020      i   n       t   e   s   t  \n   s   u   i   t   e
0000040          o   n       t   h   i   s       d   a   y       i   n
0000060 \n                   a   n   d       f   o   r
0000100              a   n   d                   .  \n   T   e   s
0000120  t   :                                   \n
0000136
```

od (octal dump) is a UNIX command that “dumps” the contents of a file using different interpretations of the bytes. In this case, the -c flag means to print bytes as printable ASCII characters (1 byte at a time).

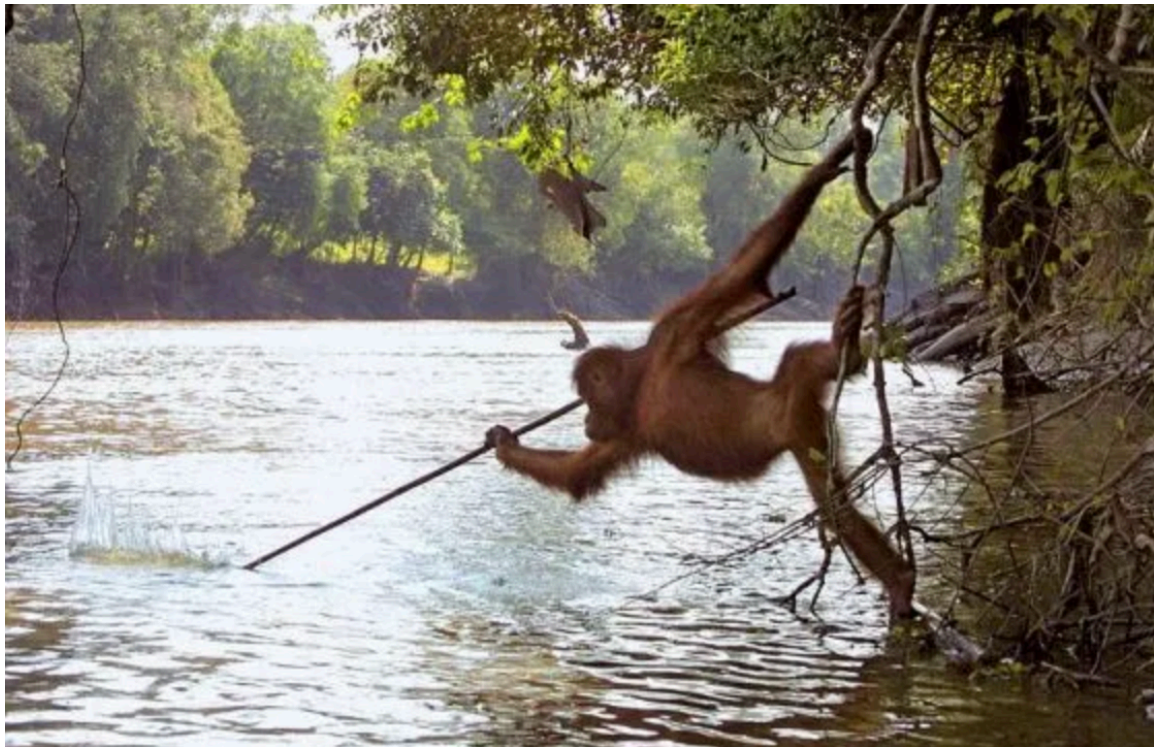
# Testing Strategies

- **Know what output to expect**
  - It is obvious that you cannot know if your program is correct unless you know what output it should produce in all situations.
  - As programs become bigger and more complicated, this becomes harder and harder to do.

# Testing Strategies - Use Tools

- Use programs like **`cmp`** (compare files for identity) and **`diff`** (report differences) to compare against known results.

- Get to know shell commands, shell scripting, and the "little languages" that can help you construct test scripts.




Program
Design

# Program Design - Is there more than one way to..?

- When you are thinking about writing a program, how do you approach the problem?
- Do you immediately start coding? Obviously not!
- Do you sketch out your solution idea(s) on paper and go over the advantages and disadvantages? A good first step!
- Do you identify the problem and then divide it into sub-problems? But how do you divide a problem?

Let's look at different approaches...


The Monolith
Approach

# The Monolith Approach

- Everything will be done in one main routine - the only functions used will be ones supplied by the system.
- Step by step solution of the problem - like a long story that you are telling. Easy to follow for the reader.
- Will it get the job done? Yes but…
- Are there disadvantages to this approach? Let me count the ways…

# Disadvantages of the Monolith Approach

- Can become very long and then it can start to become very hard to read. **Our memories are limited.**
- Parts of the code can be repetitions of previous code. **The world is telling you that you need functions!**
- Length and the repetitive nature of the solution will make it very hard to modify in the future. **And code is always going to be modified...**


Stolen from The Oatmeal

**The ability to adapt to change (requirements, environment, etc.) will always be a challenge for a developer!**

DIVIDING


THE PROBLEM

# Dividing the Problem

- Unfortunately there is not just one way to divide a problem. What is the criteria that you will use to divide the code into functions?

- There are two obvious criteria:

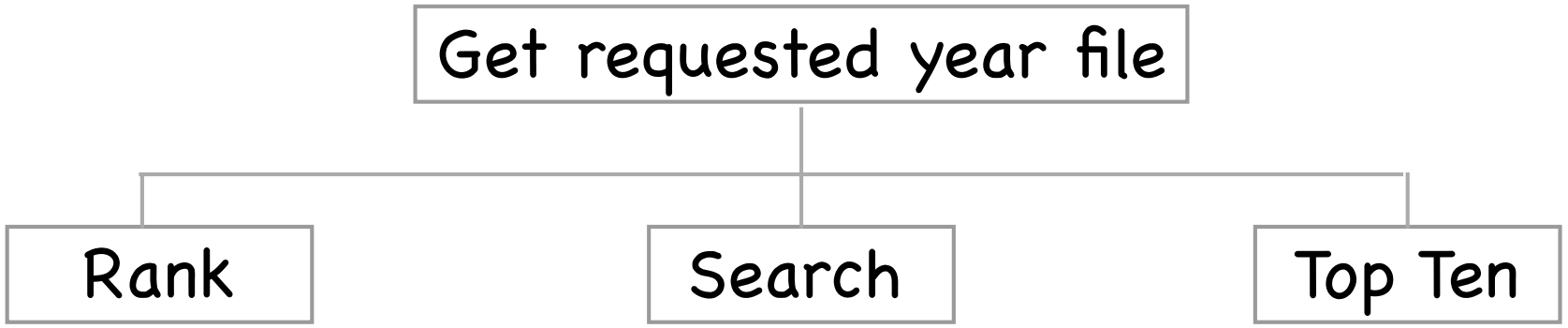
  - **Form**/Order

  - **Function**

# Form

- If you are using *Form* as the criteria you will probably look at the *order* of your operations and think about divisions/ functions based on **branching**.

- Let's look at Assignment 3 as an example:




Rank, Search and Top Ten functions will do all their own input and output and the mainline only controls the Names file input and controlling the main loops (same/different year, termination).

# **Form** - Advantages and Disadvantages

- The mainline is now very easy to ready and understand.
- There is almost no input and output in the mainline.
- If you need to change the I/O, for example, change from a text-based interface to a graphical interface (GUI) then the I/O is spread throughout your code (mainline, all functions) and so changing this will require everything to be changed.
- Good for a problem that is self-contained and has little chance of greatly changing particularly with regards to I/O.

# Function

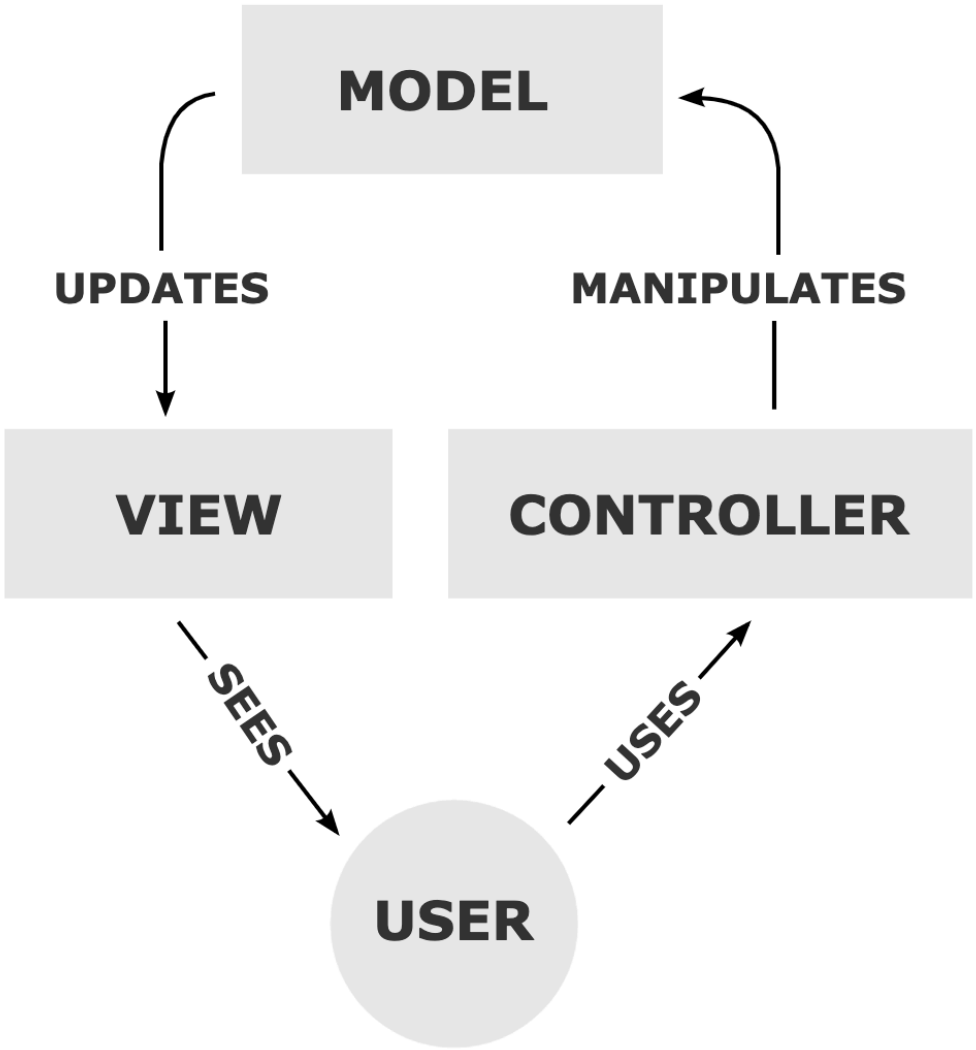
- If you are using *Function* as the criteria you will probably look at the *operations* that you need to perform and construct functions based on **results**.

- Once again let's look at Assignment 3 as an example:


Rank, Search and Top Ten functions only calculate the requested results and the mainline controls all input, output, and program flow.

# **Function** - Advantages and Disadvantages

- More code is left in the mainline (all of the I/O) and the functions tend to be light-weight.
- Light-weight functions tend to be easier to reuse and adapt.
- Functions do not handle I/O.  I/O is concentrated in the mainline where is will be easier to **modify**.
    - Knowing where all of the I/O is done is very important, particularly as code gets larger and more complex.




**Model–View–Controller** is a software *design pattern* commonly used for developing user interfaces which divides the related program logic into three interconnected elements.

**Model**

The central component of the pattern. It is independent of the user interface. It directly manages the data, logic and rules of the application.

**View**

Any representation of information. Multiple views of the same information are possible.

**Controller**

Accepts input and converts it to commands for the model or view.

https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller