Week 12 - Wednesday

# COMP 2400

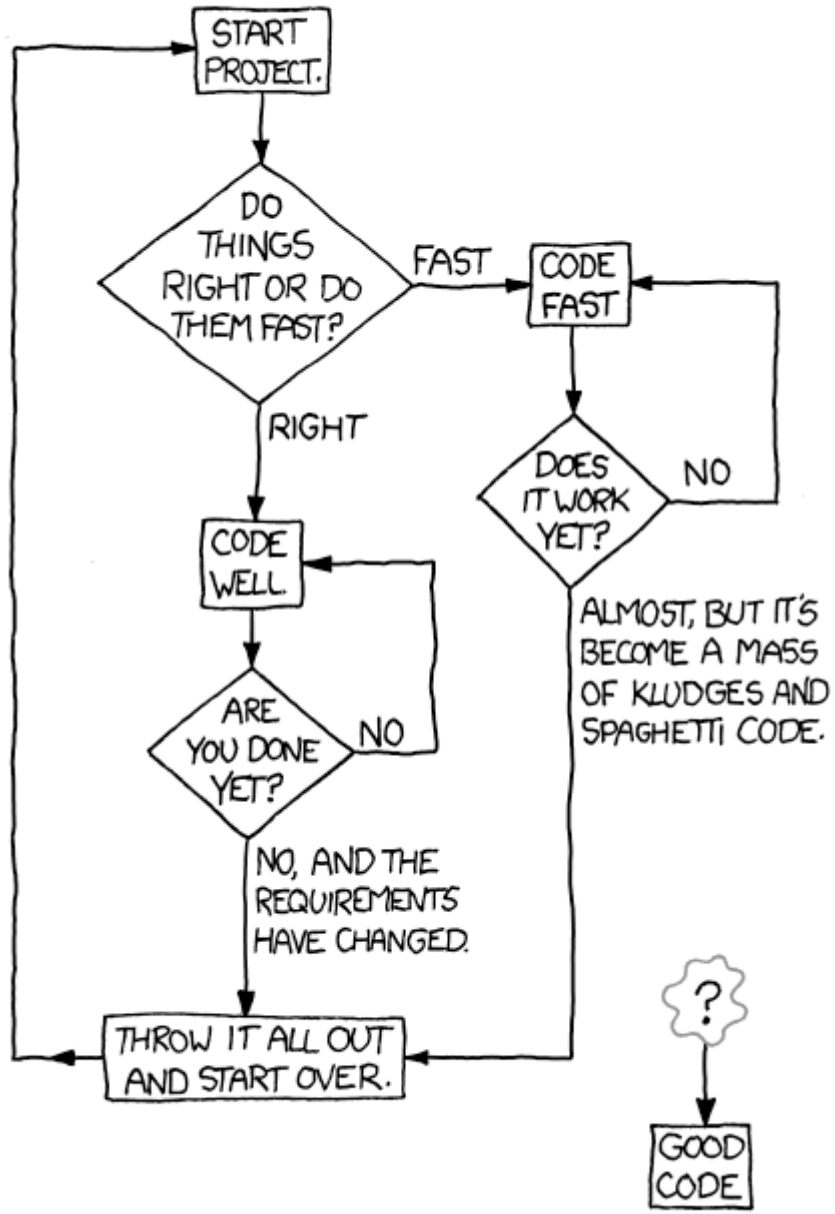
# Last time

- What did we talk about last time?
- Socket practice
- File systems

# Questions?

# Project 5

# HOW TO WRITE GOOD CODE:



# More on File Systems

# Journaling file systems

- If a regular file system (like ext2) crashes, it might be in an inconsistent state
- It has to look through all its i-nodes to try to repair inconsistent data
- A **journaling file system** (like ext3, ext4, and Reiserfs) keeps metadata about the operations it's trying to perform
- These operations are called **transactions**
- After a crash, the file system only needs to repair those transactions that weren't completed

# File attributes

- Files have many attributes, most of which are stored in their i-node
- These attributes include:
  - Device (disk) the file is on
  - i-node number
  - File type and permissions
  - Owner and group
  - Size
  - Times of last access, modification, and change
- There are functions that will let us retrieve this information in a C program
  - **`stat()`**, **`lstat()`**, and **`fstat()`**

# stat structure

- Attributes can be stored in a **stat** structure

```
struct stat {
    dev_t st_dev; /* IDs of device on which file resides */
    ino_t st_ino; /* I-node number of file */
    mode_t st_mode; /* File type and permissions */
    nlink_t st_nlink; /* Number of (hard) links to file */
    uid_t st_uid; /* User ID of file owner */
    gid_t st_gid; /* Group ID of file owner */
    dev_t st_rdev; /* IDs for device special files */
    off_t st_size; /* Total file size (bytes) */
    blksize_t st_blksize; /* Optimal block size for I/O (bytes)*/
    blkcnt_t st_blocks; /* Number of (512B) blocks allocated */
    time_t st_atime; /* Time of last file access */
    time_t st_mtime; /* Time of last file modification */
    time_t st_ctime; /* Time of last status change */
};
```

# Example with `stat()`

- Let's say you need to find out the size of a file
  - Which you need to do for Project 6
- Technically, the type for **`st_size`** is **`off_t`**
  - Files can be large (bigger than **`INT_MAX`** bytes)
  - Since it's not clear what **`off_t`** is, you can cast to **`long`** (or if you're really worried, **`long long`**)
- Use **`stat()`** if you have a file name and **`fstat()`** if you have a file descriptor

```
struct stat information;
stat (filename, &information);
printf ("The size of %s is %ld bytes.\n", filename, (long)information.st_size);
```

# Function Pointers

# Function pointers

- C can have pointers to functions
- You can call a function if you have a pointer to it
- You can store these function pointers in arrays and structs
- They can be passed as parameters and returned as values
- Java doesn't have function pointers
  - Instead, you pass around objects that have methods you want
  - C# has delegates, which are similar to function pointers

# Why didn't we cover these before?

- K&R group function pointers in with other pointers
- I put them off because:
  - They are confusing
  - The syntax to declare function pointer variables is awful
  - They are not used very often
  - They are not type-safe
- But you should still know of their existence!

# Declaring a function pointer

- The syntax is a bit ugly
- Pretend like it's a prototype for a function
  - Except take the name, put a * in front, and surround that with parentheses

```
#include <math.h>
#include <stdio.h>

int main()
{
	double (*root) (double); // pointer named root
	root = &sqrt; // note there are no parentheses
	printf( "Root 3 is %lf", root(3) );
	printf( "Root 3 is %lf", (*root)(3) ); // also legal


	return 0;
}
```

# A more complex example

- Some function's prototype:

```
int** fizbin(char letter, double length, void* thing);
```

- Its (worthless) definition:

```
int** fizbin(char letter, double length, void* thing)
{
    return (int**)malloc(sizeof(int*)*50);
}
```

- A compatible function pointer:

```
int** (*pointer)(char, double, void*);
```

- Function pointer assignment:

```
pointer = fizbin;
```

# Two styles

- Just to be confusing, C allows two different styles for function pointer assignment and usage

```
#include <math.h>
#include <stdio.h>

int main()
{
      int (*thing) (); // pointer named thing
      thing = &main; // looks like regular pointers
      thing = main; // short form with & omitted

      (*thing)();  // normal dereference
      thing();     // short form with * omitted

      return 0;
}
```

# Motivation

Why would we want function pointers?

# Motivation

- Consider a bubble sort that sorts an array of strings
  - The book uses quicksort as the example, but I don't want to get caught up in the confusing parts of quicksort

```
void bubbleSort(char* array[], int length)
{
      for(int i = 0; i < length - 1; i++ )
            for(int j = 0; j < length - 1; j++ )
                  if(strcmp(array[j],array[j+1]) > 0)
                  {
                        char* temp = array[j];
                        array[j] = array[j + 1];
                        array[j + 1] = temp;
                  }
}
```

# Motivation

- Now consider a bubble sort that sorts arrays of pointers to single **`int`** values

```
void bubbleSort(int* array[], int length)
{
      for(int i = 0; i < length - 1; i++ )
            for(int j = 0; j < length - 1; j++ )
                  if(*(array[j]) > *(array[j+1]))
                  {
                        int* temp = array[j];
                        array[j] = array[j + 1];
                        array[j + 1] = temp;
                  }
}
```

# A rectangle struct

- Let's pause for a moment in our consideration of sorts and make a struct that can contain a rectangle

```
typedef struct
{
     double x;          //x value of upper left
     double y;          //y value of upper left
     double length;
     double height;
} Rectangle;
```

# Motivation

- Now consider a bubble sort that sorts arrays of pointers to **`Rectangle`** structs
  - Ascending sort by *x* value, tie-breaking with *y* value

```
void bubbleSort(Rectangle* array[], int length)
{
      for(int i = 0; i < length - 1; i++ )
            for(int j = 0; j < length - 1; j++ )
                  if(array[j]->x > array[j+1]->x ||
                  (array[j]->x == array[j+1]->x &&
                  array[j]->y > array[j+1]->y))
                  {
                        Rectangle* temp = array[j];
                        array[j] = array[j + 1];
                        array[j + 1] = temp;
                  }
}
```

# Universal sort

- We can write a bubble sort (or ideally an efficient sort) that can sort anything
  - We just need to provide a pointer to a comparison function

```
void bubbleSort(void* array[], int length,
      int (*compare)(void*, void*))
{
      for(int i = 0; i < length - 1; i++ )
            for(int j = 0; j < length - 1; j++ )
                  if(compare(array[j],array[j+1]) > 0)
                  {
                        void* temp = array[j];
                        array[j] = array[j + 1];
                        array[j + 1] = temp;
                  }
}
```

# Typechecking

- Function pointers don't give you a lot of typechecking
- You might get a warning if you store a function into an incompatible pointer type
- C won't stop you
- And then you'll be passing who knows what into who knows where and getting back unpredictable things

# Simulating OOP

- C doesn't have classes or objects
- It's possible to store function pointers in a struct
- If you always pass a pointer to the struct itself into the function pointer when you call it, you can simulate object-oriented behavior
- It's clunky and messy and there's always an extra argument in every function (equivalent to the **`this`** pointer)
- As it turns out, Java works in a pretty similar way
  - But it hides the ugliness from you
  - Python doesn't hide as much ugliness, always requiring **`self`**

# Ticket Out the Door

# Upcoming

# Next time…

- Introduction to C++

# Reminders

- Start on Project 6