CSE 451 Section 1

- C Refresher (or introduction?)
- Project 0

C?

Administrivia

- Office hours?
  - Tentatively: 1:30 – 3:30 on W in 002
- Special C office hours next week.
  - MW 1:30 – 3:30

Motivation: Why C?

- Why couldn't we write an entire OS in, say, Java? Java is interpreted -- the .class files need to be run using the Java VM. What would the Java VM itself run on? This needs to be native code.
- OS needs precise control -- programmer needs to be able to reason precisely about the code execution.
- Things like garbage collection are unpredictable -- they can happen at any time. This is unacceptable inside the OS. Some tasks may run in modes which necessarily must be completed very quickly, such as when interrupts are disabled (you'll learn about this more later).
- Memory management inside the OS is much more constrained; you typically have multiple "pools" of memory from which to allocate objects -- a single simple "new" operator no longer cuts it.
- C was invented at Bell Labs specifically for OS Programming (UNIX) in 1977. Its design goals were to be more portable and readable than assembly, but to compile very simply to assembly in a straightforward manner.
- Plus, since Linux is already written entirely in C (except for a few architecture-specific bits in assembly), any modifications we make to this must be in C as well.

Differences in Compilation

- Java: .java files → .class files, which are run through a VM. Can be "zipped" together in a .jar, but still functionally a set of separate files.
- C: .c files → .o files; header files used to share definitions. .o files are linked together into libraries and/or executables.

Differences in Libraries

- In Java, you had the Class Libraries (java.util, java.String, java.Math, etc..). You would import a namespace such as **java.util.*** into the active file. This let the VM know that names in this file referred to names from that namespace.

- In C, you have a set of standard functions rolled together in "libc." There is no concept of namespaces. To use a function that isn't a part of the current file, you need to predefine it as **extern**. These definitions or **prototypes** are in a set of **header files** which you can include.

- The **linker** will then match up all the library functions used with your uses of them in the final compilation step.

- To include one file inside another, we use a preprocessor directive, of the form:

```
#include<filename.h>
```

- There is no **preprocessor** in Java.

- In C, the **preprocessor** reads the files, looking for lines beginning with '**#**‘ called **preprocessor directives**, and then performs an appropriate action, effectively rewriting the file before it is consumed by the compiler.

Differences w.r.t. OOP

- There are **no classes** in C!
- All methods are "static" in the Java sense -- but don't declare them with the static keyword, unless you know exactly what you're doing; it means something else in C.
- data structures are stored in structs -- which are like classes with **all members public**, and **no functions**.
- You must manipulate these with methods "outside" of the structure.
- In Java, all objects are stored in the heap.
- In C, structs may live in the heap, on the stack, or in global space. If you allocate a struct in the heap, you will receive a **pointer** to the struct. Effectively, this is the same as a **reference** in Java -- it contains the address of the struct, in memory.

Things to get used to about C

- Pointers and pointer arithmetic
- Static vs. dynamic memory allocation
- Call-by-value vs. call-by-reference
- Structures, typedef, unions

… This is what project 0 is here to remind you about (or teach you for the first time?)

Pointers

- Example Use:

```
int a = 5;
int b = 6;
int *pa = &a;  // Declares a pointer to a with value as the // address
   of a.
*pa = b;    // Changes the value of a to b, that is a == 6
pa = &b;    // Changes pa to point to the place where b is on
   // the stack.
```

- Pointer arithmetic:

```
int foo[2]; // foo is a pointer to the beginning of the array
*(foo + 1) = 5; // the second int in the array is set to 5
// How many bytes are shifted in foo + 1?
```

Struct and Typedef

```
struct foo_s {              // defines a type that is referred
   int x;                   // to as a "struct foo_s"
   int y;
};                          // don't forget this ;

struct foo_s foo;           // declares a local foo struct on the stack
                            // of type struct foo_s
foo.x = 1;                  // access the x field

typedef struct foo_s *foo_t;    // use typedef to create an alias
                                // allowing you to now use foo_t
                                // to declare variables instead.

foo_t another_foo = (foo_t)malloc(sizeof(struct foo_s));

another_foo->x = 2;         // another_foo is a pointer to a struct foo_s
                            // on the heap. The "->" operator dereferences
                            // the pointer and accesses the field x.
```

Union

```
union {
  int foo;
  char bar;
  char * string;
} name; // what is the size of name?

name temp;
temp.foo = 4;
temp.string = “foo”;
temp.bar = 'c';
```

// what is the “value” of temp.bar? temp.string?

Pass-By-Value, Pass-By-Reference

```
int doSomething(int x) {
   return x+1;            // x is copied into the stack frame of
                          // this function, and the local copy is
                          // modified and returned
}

void doSomethingElse(int *x) {
   *x += 1;               // The pointer is copied into the stack
                          // frame, then dereferenced and the
                          // memory being pointed to is changed
}

void foo() {
   int x = 5;
   int y = doSomething(x);    // y will get 6, but x will be unchanged
   doSomethingElse(&x);       // changes the value of x to 6
}
```

Common C Pitfalls (1)

- What's wrong and how to fix it?

```
char* get_city_name(double latitude, double longitude) {
  char city_name[100];
  …
  return city_name;
}
```

Common C Pitfalls (1)

- Problem: return pointer to statically allocated memory.
- Solution: allocate on heap:

```
char* get_city_name(double latitude, double longitude)
{
  char* city_name = (char*)malloc(100);
  …
  return city_name;
}
```

Common C Pitfalls (2)

- What's wrong and how to fix it?

```
char* buf = (char*)malloc(32);
strcpy(buf, argv[1]);
```

Common C Pitfalls (2)

- Problem: Buffer overflow
- Solution:

```
int buf_size = 32;
char* buf = (char*)malloc(buf_size);
strncpy(buf, argv[1], buf_size);
```

- Are buffer overflow bugs important?

Common C Pitfalls (3)

- What's wrong and how to fix it?

```
char* buf = (char*)malloc(32);
strncpy(buf, "hello", 32);
printf("%s\n", buf);

buf = (char*)malloc(64);
strncpy(buf, "bye", 64);
printf("%s\n", buf);

free(buf);
```

Common C Pitfalls (3)

- Problem: Memory leak
- Solution:

```
char* buf = (char*)malloc(32);
strncpy(buf, "hello", 32);
printf("%s\n", buf);
free(buf);
buf = (char*)malloc(64);
…
```

- Are memory leaks important?
  - OS, web server, web browser, your projects?

Common C Pitfalls (4)

- What's wrong (besides ugliness) and how to fix it?

```
char foo[2];
foo[0] = 'H';
foo[1] = 'i';
printf("%s\n", &foo);
```

Common C Pitfalls (4)

- Problem: String is not NULL-terminated
- Solution:

```
char foo[3];
foo[0] = 'H';
foo[1] = 'i';
foo[2] = '\0';
printf("%s\n", &foo);
```

- Or, "more better":

```
char * foo = "Hi";
```

  – Double-quoted string literal syntax gets NULL-terminated automatically.

Project 0

- Work Individually
  - For the later projects you will be in a group, start planning who you will work with now.

Project 0

For you to brush up (or learn) your C skills

- Part 1:
  - Find and fix two bugs in queue implementation
  - Implement queue reverse and queue sort
- Part 2:
  - Implement (.c) a hash table.
- Part 3:
  - Setup and use a table of function pointers containing math calls
- Read the project writeup on the web.

Part 1

- Queue source gives a good example for function pointers.
- Make sure:
  - You comment the bug fixes so they are easy to find.
  - Implement sort in non-descending order.
  - Implement sort and reverse in-place.
  - You apply pfcompare correctly.

Pfcompare

e1 < e2 // -1

e1 == e2 // 0

e1 > e2 // 1

Part 2

- This has been simplified from previous quarters since the interface is defined.
- Think about:
  - ~~What happens if your hash table is full? (answer: resize)~~
  - Which implementation you want to use (separate chaining or open addressing)

Part 3

- Again, see queue_size for an example of function pointer use.
- READ THE ADDENDUM ATTACHED TO THE PROJECT PAGE. You are expected to use unions and not void *
- Remember:
  - math_call's return value is an error code, not the evaluated result
  - The point is to create/use a function pointer table. math_call should not explicitly call add, substract, etc.