CS 521: Systems Programming

Beginning C

Lecture 2

Today's Schedule

- Differences: C vs. Other Languages
- Phases of Compilation
- Data Types
- A project-based intro to C with `echo`

Today's Schedule

- **Differences: C vs. Other Languages**
- Phases of Compilation
- Data Types
- A project-based intro to C with `echo`

Architectural Differences

- C is compiled to ***machine code***, unlike Python or Java
  - The compiled ***binary executable*** contains instructions that your CPU understands
  - There are several compilers on the market today (gcc, clang, msvc) that transform your code into machine code
- Java runs on a virtual machine (JVM)
- Python is interpreted (translated to machine code on the fly)
- We can achieve better performance with C, but are also given more responsibility
  - Memory management is up to us (no automatic garbage collection)

Main Advantages

- C is fairly simple: the language does not have a multitude of features
- But coming from Java, the syntax is still familiar
- It's the *lingua franca* of systems programming
  - When we operate close to the hardware, it can be much easier to implement than the equivalent Java/Python/etc.
  - Want to contribute to the Linux kernel? It's written in C (including the drivers)
- Performance

Main Disadvantages

- Much less functionality is available in the standard library than other languages
  - For example: no built in list, hashmap, tree, etc.
- Memory leaks
- Segmentation faults (invalid memory access)
- No objects – if you're used to object-oriented programming, C will make you rethink your program

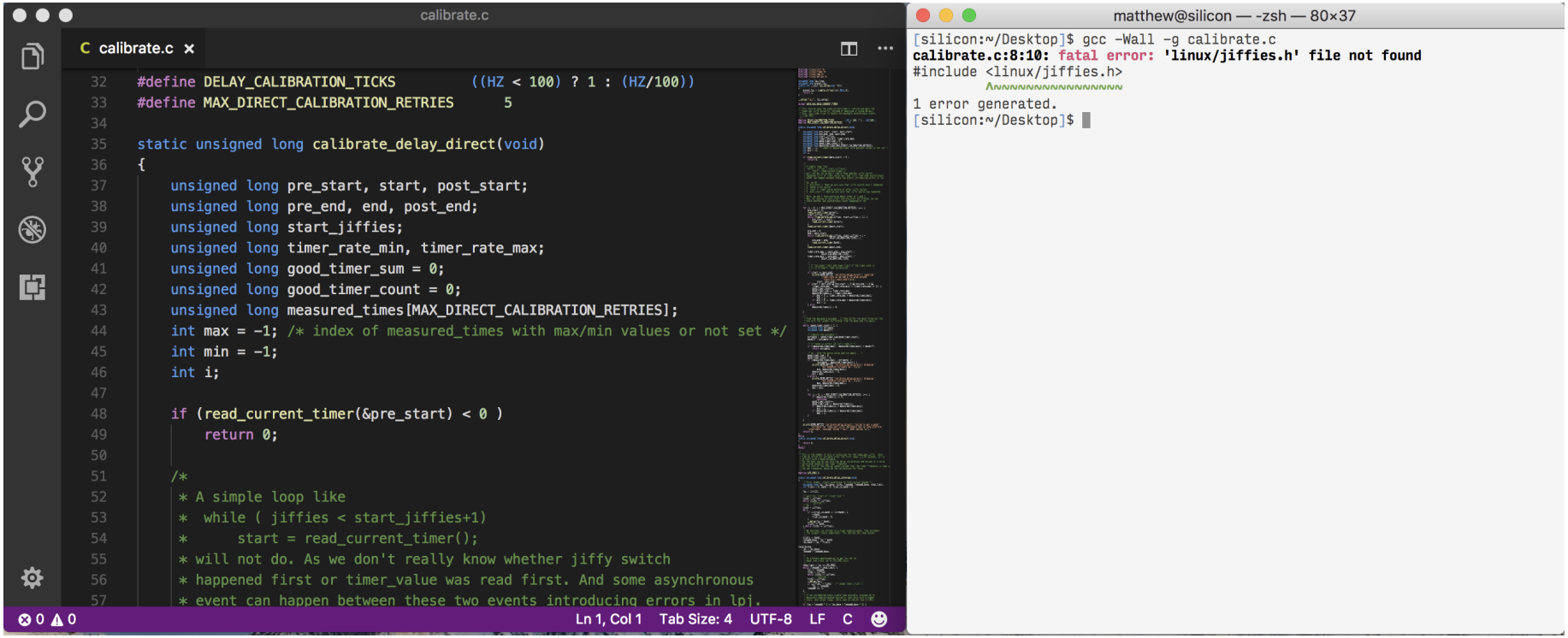
Standardization

- C is not controlled by a single entity; it is a standard
- The standard itself is fairly loose, and allows **undefined behavior** (UB)
  - Basically, the language standard doesn't specify how *everything* should work
    - Compilers can do whatever they want with UB
    - This is why we're making sure we all have the same platform (our VMs) in class 😉

Systems Culture

- There is a somewhat different culture in the systems world
- Using an IDE (like Eclipse, IntelliJ, etc) is less common
  - The Unix command line provides many of the usual IDE features
- Many developers prefer to use a text editor and a terminal to write their programs
  - Text editor: edit, save
  - Terminal: compile, run

Writing C Programs



Recommendation

- Use whatever is comfortable for you
- If you get a chance, try to learn the basics of a terminal editor (even `nano` counts!)
  - Or `vim`, `emacs`
- (maybe at least know how to quit vim and emacs 😉)
  - By the way, what's the universal "quit" key combination in the terminal?

Revisiting Hello World

```c
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

To run:

```
gcc hello.c -o hello
./hello
```

Slightly More Advanced

```c
#include <stdio.h>

void say_hello(int times);

int main(void) {
    say_hello(6);
    return 0;
}

void say_hello(int times) {
    int i;
    for (i = 1; i <= times; ++i) {
        printf("Hello world! (#%d)\n", i);
    }
}
```

Differences from Java/Python

- Including libraries looks a bit different
- No public/private etc. access modifiers
- Forward declarations (prototypes)
- No objects
- No exceptions
- A **huge** difference: what return types are used for
  - Often error checking!
- But, there are a lot of similarities…

Similarities to Java/Python

- Arithmetic is mostly the same
- We use &&, ||, and != instead of and, or and not
- if, then, else
- Loops
- Switches

Some Advice

- The similarity between C and Java can be deceiving
- In these small programs, there's hardly a difference!
- However, you will soon see that the structure of larger programs ends up being quite different
  - Since there are no classes, the focus shifts to writing functions
  - Organization might seem a bit less natural, but you can still break your functions up into ***modules***

Today's Schedule

- Differences: C vs. Other Languages
- **Phases of Compilation**
- Data Types
- A project-based intro to C with `echo`

Compiling Your Programs

- You might have not cared much about compiling code previously
  - **Compile**: turn code into an executable
- …but with C, it's a bigger deal
- The C compiler goes through a few phases to get from **code** to a finished, ready-to-run **binary executable**

Phases of C Compilation

1. **Preprocessing**: perform text substitution, include files, and define macros. The first pass of compilation.
   - Directives begin with a #
2. **Translation**: preprocessed code is converted to machine language (also known as *object code*)
3. **Linking**: your code likely uses external routines (for example, `printf` from stdio.h). In this phase, libraries are added to your code

Stepping Through Compilation

- When we compile our source code, we get an output binary that is ready to run
  - The steps are mostly invisible to us
- We can ask the compiler to only execute a subset of its compilation phases
  - Let's do just that!

Preprocessing

- We can ask gcc to only perform the preprocessing step using the `-E` flag:
  - `gcc -E my_program.c`
- This will print the preprocessed file to the terminal
- We can write this output to a file by redirecting the stdout (**standard output**) stream:
  - `gcc -E my_program.c > my_program.pre`
- …And view it with a text editor

Translating to Assembly Code

- We can also view the **assembly** code generated by the compiler:
  - `gcc -S my_program.c`
  - Produces my_program.s
- This representation is very close to the underlying **machine code**
- For a reference on x86-64 processor assembly:
  - https://web.stanford.edu/class/cs107/guide_x86-64.html

Producing Object Code

- Finally, we can produce the **machine code** / **object code** representation of the program
- `gcc -c my_program.c`
  - Produces my_program.o
- We can view this with a **hex dump**
  - `hexdump -C my_program.o`

Today's Schedule

- Differences: C vs. Other Languages
- Phases of Compilation
- **Data Types**
- A project-based intro to C with `echo`

C Data Types

- When defining arguments and variables, the following data types are possible in C:
  - `char`
  - `int`
  - `float`
  - `double`
- Wait… that's it?! Yeah! Well, there are a few *modifiers*:
  - `short`, `long`, `signed`, and `unsigned`

Sizing

- `short` and `long` modify the data type's size
- The C standard specifies the *minimum* size for each type. You can determine the sizes (in bytes) with `sizeof`:
  - `sizeof(char) = 1`
  - `sizeof(short int) = 2`
  - `sizeof(int) = 4`
  - `sizeof(long int) = 8`
- …but these can be platform-specific. Don't make assumptions!
  - One thing can be certain: `char` is **guaranteed** to be 1 byte

Demo: Data Type Sizes

(you can do this one on your VM, or local machine if you have a C compiler!)

Signed Data Types

- Integer types can be **signed** or **unsigned**
  - Signed integers use one bit as a ***sign bit*** to determine whether the number is negative or positive
- Java doesn't have unsigned `ints`. What might they be useful for?
  - Enforce a particular variable to always be positive
  - Use that extra bit to store larger positive numbers
- Related: integer overflow is undefined behavior (UB)

Today's Schedule

- Differences: C vs. Other Languages
- Phases of Compilation
- Data Types
- **A project-based intro to C with `echo`**

Creating an Echo Chamber

- We'll do a lot of **project-based** learning in this class
  - Choose something we want to accomplish, figure out what we need to learn, and then actually build it
- Many of our projects will involve building our own versions of common command line tools
- Today's subject: `echo`
- Hear that?
  - `echo`
    - `echo`
      - `echo`
        - `echo`

Echo

- What does the `echo` command do?

```
[mmalensek@mmalensek-vm ~]$ echo
```

Wow!!!

```
[mmalensek@mmalensek-vm ~]$ echo Hello World!
Hello World!
```

Going to the Documentation

- You probably already have a good grasp of what echo does, but let's go to the *real* authority: the documentation!
- To access the **manual pages**, use the `man` command
- `man echo`

Gathering Requirements

- What do we need to be able to do to build our own copy of `echo`?
- The **GNU** version of `echo` supports a ton of features… Maybe we can copy the **BSD** version instead
  - (command line tools have a standard set of features, but there are several different implementations!)
  - Check `man echo` on a Mac
- Take a few minutes to discuss with the people around you…

Requirements

Here's what I came up with.

- A way of accessing the *command line arguments* passed to the program (e.g., `./prog arg1 arg2 arg3`)
- A loop so we can iterate through each one
- We already know how to print… sort of. More detail there would be good
- We need to handle the `-n` *command line flag*

Command Line Arguments

- In Java, the `main` method has one argument: an array of strings that contain the command line args
- So far we've seen one way of declaring `main` in C:

  ```
  int main(void)
  ```

- There is **another way** to do it!

  ```
  int main(int argc, char *argv[])
  ```

  - `argc` : argument count
  - `argv` : argument values (as an array of `char *` … what's that?)

The First Argument

- The **first** argument will always be the *program name*
- i.e., if you run `./some_prog` then

  `argv[0] = "./some prog"`
- This also means that `argc` will always be at least **1**

Next Requirement: A Loop

- We can use a `for` loop with the `argc` count to loop through all the arguments
- We haven't fully discussed arrays yet, but let's just pretend we know what we're doing!
- If I access `argv[i]` I will get the i^{th} value of the array of… `char *`?

What the $%*@ is char star?

- In C, the `*` indicates a *pointer*. So a `char *` type is a **pointer to a character**.
- C does not have strings… instead, we work with *character arrays* instead
- So `char *argv[]` is an array of pointers to characters
  - geez
- Understanding that seems like it might take work, so let's save that for another day…

Printing

- We can google how to use `printf`, and we'll get some great answers
- But we can also look at the documentation:
- `man 3 printf`
  - `man 3` means use the 3rd section of the manual – the C documentation.
  - `man printf` will actually give you information about something else – the `printf` command line utility

Printing a String

- We can use `printf("%s", some_string);` to print a string
- If we use `puts(some_string)` it will include a **newline** character (`\n`) at the end, and we don't want that

Handling Flags

- Most command line utilities support **flags** to make them behave in different ways
- When `echo` receives a `-n` flag, it doesn't print a trailing newline
- How can we handle this? With a conditional!
  - ```
    if (argv[i][0] == '-') {
    /* First letter is a - character! */
    /* What do we check for next? */
    }
    ```
    ```

WAIT!

- I thought `argv` was an array of pointers to **A** character, right?
  - How are we indexing into it twice like a 2D array?
- Well…
- This is because in C, strings are arrays of characters.
  - When you create a string, it is represented as a pointer to the first character in that string
- When we do `argv[i][0]` we are accessing the first character in the string
- Weird, but don't worry yet. We will talk about Strings a LOT more

Ok!

We are ready to build our own copy of `echo`. Let's get started – this is **Lab 1**!