Lab 2: C Programming in Linux

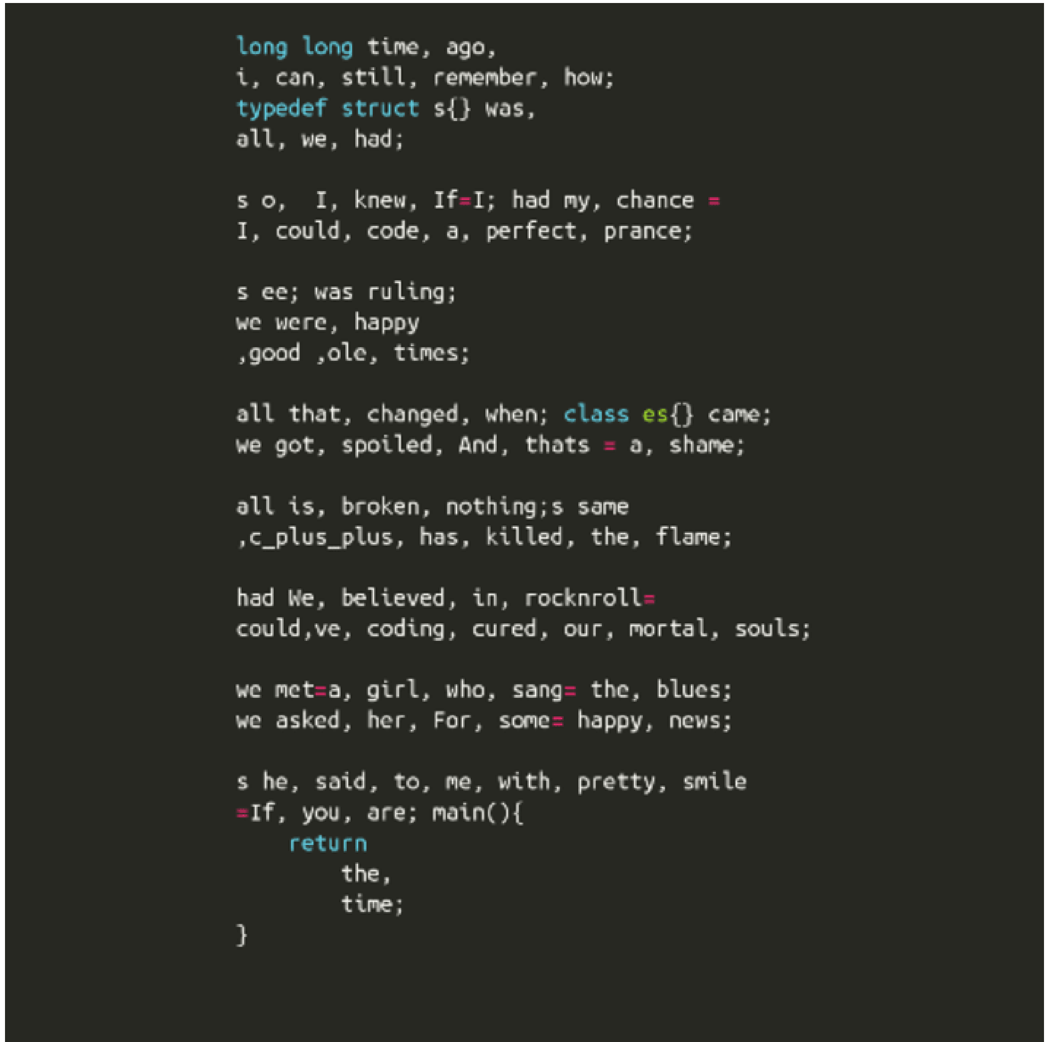
Prof. Zichen Xu

Dear C Programming Experts

- We will do a lab with C programming, which shall be what you have mastered
- You are given a list of tasks to get hands-on experience with C in Linux
- Before that, let’s enjoy a poem

Linux Poem: The American C

By Done Lean



```
long long time, ago,
i, can, still, remember, how;
typedef struct s{} was,
all, we, had;

s o,  I, knew, If=I; had my, chance =
I, could, code, a, perfect, prance;

s ee; was ruling;
we were, happy
,good ,ole, times;

all that, changed, when; class es{} came;
we got, spoiled, And, thats = a, shame;

all is, broken, nothing;s same
,c_plus_plus, has, killed, the, flame;

had We, believed, in, rocknroll=
could,ve, coding, cured, our, mortal, souls;

we met=a, girl, who, sang= the, blues;
we asked, her, For, some= happy, news;

s he, said, to, me, with, pretty, smile
=If, you, are; main(){
    return
        the,
        time;
}
```

Lab Objectives

- Objectives:
  - the C programming language
  - the development toolchain (pre-processor, compiler, assembler, linker)
  - the automating the compilation process using Makefiles

Use C and Makefile for Coding

1. Compile and run a C problem, on slide 3
2. Separate the C code on slide 3 into two files: source.c and main.c
3. Try write a makefile to repeat the compilation process (based on the template in the course website), with different flags (-g, -ggdb, -Wall, -O, etc.)

Now I’m going to hit you harder

- Linux has a list of header to provide additional access to process and file statistics

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int chmod(const char *path, mode_t mode); int
fchmod(int fd, mode_t mode);
```

Use C and Makefile for Coding

- The stat structure is designed as follows:

```
struct stat {
            dev_t st_dev; /* ID of device containing file */
            ino_t st_ino; /* inode number */
            mode_t st_mode; /* protection */
            nlink_t st_nlink; /* number of hard links */        uid_t st_uid; /* user ID of owner */
            gid_t st_gid; /* group ID of owner */
            dev_t st_rdev; /* device ID (if special file) */
            off_t st_size; /* total size, in bytes */ blksize_t st_blksize; /* blocksize for file system
I/O */
            blkcnt_t st_blocks; /* number of 512B blocks allocated */
            time_t st_atime; /* time of last access */          time_t st_mtime; /* time of last
modification */
            time_t st_ctime; /* time of last status change */ };
```

Tasks

- You are asked to write a C code to check whether an input string is a file, or directory, or else.
- Print the mode of the file, if it is a file. If you are the owner of the file, chmod it into 777, using C code.
- If this is not a file or a folder/directory, provide a mechanism to handle the error.
- Write a makefile for the above three codes and make a successful compilation

GDB

- A little bit tryout of using GDB
- Make sure you compiled to previous code using debug mode and allow GDB to provide user-friendly information
- Set a break to check the value of time variable change in the first and your input argument in the second program
- Check where is your code at and print the current stack information of the target code

Reference gdb commands

- General Commands:

`file [<file>]`	selects `<file>` as the program to debug
`run [<args>]`	runs selected program with arguments <`args>`
`attach <pid>`	attach gdb to a running process <`pid>`
`kill`	kills the process being debugged
`quit`	quits the gdb program
`help [<topic>]`	accesses the internal help documentation

- Stepping and Continuing:

`c[ontinue]`	continue execution (after a stop)
`s[tep]`	step one line, entering called functions
`n[ext]`	step one line, without entering functions
`finish`	finish the function and print the return value

- Useful breakpoint commands:

`b[reak] [<where>]`	sets breakpoints. `<where>`
`[r]watch <expr>`	sets a watchpoint, which will break when `<expr>` is written to [or read]
`info break[points]`	prints out a listing of all breakpoints
`clear [<where>]`	clears a breakpoint at `<where>`
`d[elete] [<nums>]`	deletes breakpoints by number

- Commands for looking around:

`list [<where>]`	prints out source code at `<where>`
`search <regexp>`	searches source code for `<regexp>`
`backtrace [<n>]`	prints a backtrace `<n>` levels deep
`info [<what>]`	prints out info on `<what>` (like local variables or function args)
`p[rint] [<expr>]`	prints out the evaluation of `<expr>`

- Commands for altering data and control path:

`set <name> <expr>`	sets variables or arguments
`return [<expr>]`	returns `<expr>` from current function
`jump <where>`	jumps execution to `<where>`