CS246
Unix: processes
C: more defines, structs

March 18
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

/**
 * A stupid program that just runs for the number of seconds given
 * on the command line
 * Prints a ping every 10 seconds
 * **/

int main(int argc, char const *argv[]) {
    int tim = atoi(argv[1]);
    tim++;
    for (int i = 1; i < tim; i++) {
        if (0==(i%10)) {
            printf("%d\n", i);
        }
        sleep(1);
    }
    return 0;
}
Background and foreground

- foreground
  - program runs and you get back the cursor on completion
- “background”
  - program runs, but you get cursor immediately.
    - can do other things in the shell.
- Start in background
  - & at end of line starts program “in background”
    - ./longrunner &
      - problem it is still putting info to the console
    - ./longrunner > longrun.out &
- Note all jobs are tied to their shell.
  - So if shell dies, jobs dies.
    - Can be avoided with some extra work
Stopping (killing) processes

• Foreground
  • CTRL-C

• Background
  • need to know “process id” or “job id”

• Job id is shell specific.
  • Each shell knows what processes are running under its aegis
    • UNIX> jobs
      • job id are on left in []
      • typically small integers (1,2,3,...)
  • UNIX> kill %jid — kills well behaved jobs

[gtowell@powerpuff L10]$ ./a.out 600 > aa &
[1] 3805534
[gtowell@powerpuff L10]$ ./a.out 600 > bb &
[2] 3805543
[gtowell@powerpuff L10]$ jobs
[1]- Running ./a.out 600 > aa &
[2]+ Running ./a.out 600 > bb &
[gtowell@powerpuff L10]$
Process id

• Process id is across entire machine
  • usually a largeish integer
• UNIX> ps
  • default — all processes in the current shell
  • ps aux
    • all processes on device
  • ps aux | grep myunixname
    • all processes that are running for me
• UNIX> kill pid

[gtowell@powerpuff L10]$ ./a.out 600 > bb &
[1] 3807567
[gtowell@powerpuff L10]$ ./a.out 600 > aa &
[2] 3807740
[gtowell@powerpuff L10]$ ps
  PID  TTY       TIME CMD
 3802628 pts/13   00:00:00 bash
 3807567 pts/13   00:00:00 a.out
 3807740 pts/13   00:00:00 a.out
 3807858 pts/13   00:00:00 ps
Pausing and restarting

• Pause a foreground process
  • CRTL-z
• Restart a paused process
  • fg [%jid][pid]— restart in foreground
  • bg [%jid][pid]— restart in background
  • realistically, I only ever use bg and fg on the “current” process
More with processes

• pid also appears in /proc directory
• UNIX> top
  • a continually updating view on what is using resources on computer
  • pid in far left of top

```
100% □ 10%

top - 11:25:44 up 381 days, 16:24, 3 users, load average: 0.56, 0.61, 0.57
Tasks: 500 total, 1 running, 494 sleeping, 5 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.0 sy, 0.0 ni, 99.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 32029.8 total, 25170.1 free, 1325.2 used, 5534.6 buff/cache
MiB Swap: 9001.0 total, 7207.5 free, 1793.5 used, 30242.0 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
574 root 20 0 258108 60460 51736 S 0.3 0.2 305:59.64 systemd+
3828102 gtowell 20 0 209028 4132 3316 R 0.3 0.0 0:00.16 top
3828115 root 20 0 9988 6820 5828 S 0.3 0.0 0:00.03 sshd
1 root 20 0 172868 6892 4344 S 0.0 0.0 78:21.52 systemd
```
Yet more

- Pause a process
  - foreground — CTRL-z
  - background — UNIX> kill -SIGSTOP pid
- Resume a process
  - foreground — fg [%jid][pid]
  - background — UNIX> kill -SIGCONT pid
- Killing zombies
  - UNIX> kill -SIGKILL pid
    - this is super aggressive
uptime

- UNIX>uptime
  - Stats about how heavily used computer is. Last 3 numbers give number of CPUs in use.
- To really understand, need to know # of CPUs and maybe info about the CPUs.
- https://cs.brynmawr.edu/~gtowell/chks.html

```
[gtowell@powerpuff ~]$ uptime
12:51:29 up 381 days, 17:50, 3 users, load average: 0.69, 0.63, 0.58
[gtowell@powerpuff ~]$ cat /proc/stat | grep cpu[0-9] | wc
 32 363 2088
model name: Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
```

```
[gtowell@benz:~]$ uptime
12:55:21 up 23:11, 1 user, load average: 0.18, 0.08, 0.03
[gtowell@benz:~]$ cat /proc/stat | grep cpu[0-9] | wc
 8 88 343
[gtowell@benz:~]$ cat /proc/cpuinfo | grep 'model name' | uniq
model name: Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz
```
Lab from Thursday

```c
#include <stdio.h>

long factorial(int factorialOf) {
    if (factorialOf <= 1) {
        return (long)1;
    }
    return factorialOf * factorial(factorialOf - 1);
}

long factorialTail(int factorialOf, long product) {
    if (factorialOf <= 1) {  
        return product;
    }
    return factorialTail(factorialOf - 1, product * factorialOf);
}

void main(void) {
    int factorialOf = 0;
    printf("Enter an int to find factorial of: ");
    scanf("%d", &factorialOf);

    printf("No tail recursion: %ld\n", factorial(factorialOf));
    printf("Tail recursion: %ld\n", factorialTail(factorialOf, 1));
}
```
More with \#define

- define useful functions
- Have parts of code that get “commented out” depending on presence of a define

```c
#define SIZE 20
#define RAND_RANGE(min, max) min + rand() / (RAND_MAX - (max - min) + 1)
#define SWAP_INT(a, b) {int t=a; a=b; b=t;}
#ifndef max
    #define max(a,b) (a > b ? a : b)
#endif
#ifndef min
    #define min(a,b) (a > b ? (b) : (a))
#endif
#define MEDIAN(a,b,c) ( (a > b) ? max(b, min(a,c)) : min(b, max(a,c)) )
#define DO_MEDIAN 1
#define DO_SORT 1
#if DOSORT
    void iSort(int *arr, int lo, int hi) {
        printf("iSort %d %d\n", lo, hi);
        SWAP_INT(arr[lo], arr[hi]);
    }
#endif
// more here
```
Yet more
#define

• can add defines at compile time via gcc
  • NO code change
  • gcc -D LOG_LEVEL=5 xx.c

• Most production shops are paranoid (rightly) about code changes

```c
#ifndef LOG_LEVEL
#define LOG_LEVEL 0
#endif
#define LOG_VERBOSE 1
#define LOG_INFO 5
#define LOG_ERROR 10

// basic logging comment
#define LOG(level, ...) if (LOG_LEVEL <= level) fprintf(LOG_LEVEL>=LOG_ERROR ? stderr : stdout, __VA_ARGS__)

// log the start of a function ... arguably should include the args
#define FUNC_START() if (LOG_LEVEL <= LOG_INFO) fprintf(LOG_LEVEL>=LOG_ERROR ? stderr : stdout, "start %s
", __func__)

// log the end of a function
#define FUNC_END() if (LOG_LEVEL <= LOG_INFO) fprintf(LOG_LEVEL>=LOG_ERROR ? stderr : stdout, "finish %s
", __func__)

void v1() {
    FUNC_START();
    for (int i=0; i<10; i++) {
        LOG(LOG_VERBOSE, "%d %c %c\n", i, 'a'+i, TO_UPPER('a'+i));
    }
    FUNC_END();
}

int main(int argc, char * argv[]) {
    for (int i=0; i<argc; i++) {
        LOG(LOG_INFO, "%s\n", argv[i]);
    }
    v1();
}
```
Defining new types

- C allows fairly arbitrary definition of new data types
  - typedef type name
- Look in stdio.h — used a lot
- Allows compile time checking for some error types

```c
#define DOLLAR_FORMAT "%.2f"

// define a new type named "dollar"
// typedefs should be global although you can make them local
// but is it reasonable to do so?
typedef float dollar;

void pdollars(dollar d) {
    printf(DOLLAR_FORMAT, d);
}

int main(int argc, char * argv[]) {
    dollar money = 5.5;
pdollars(money);
    printf("\n");
}
```
Structs

- Way of grouping disparate data types
  - NOT objects
  - No methods
  - No access controls
- Two ways of defining
  - recommendation: use typedef

```c
// define a struct
struct p {
    int a;
    int b;
};

// define a struct using typedef
typedef struct {
    int a;
    int b;
} pType;
```
More on Structs

- Pass by Value!
- When working with a pointer to a struct can use -> to access components

```c
void printit(struct p pa) {
    printf("struct p %d %d %d\n", &pa, pa.a, pa.b);
}
void printitPT(pType pa) {
    printf("pType %d %d %d\n", &pa, pa.a, pa.b);
}
void printitPoint(pType * paP) {
    // note use of two ways to get to pointed to struct contents
    // -> is just shorthand and is supposed to be easier.
    printf("pType* %d %d %d\n", paP, (*paP).a, paP->b);
}

int main(int argc, char const *argv[])
{
    struct p aa;
    aa.a = 5; aa.b = 10;
pType bb = {.a=6, .b=12};
    printf("Pointers %d %d\n", &aa, &bb);
    printit(aa);
    printitPT(bb);
    printitPoint(&bb);
    return 0;
}
```
More Structs

- strtok and the weather data from thursday
- strcpy!
- strtol is equivalent to atoi

```c
typedef struct {
    char time[10];
    int temp;
    int dewPoint;
    int relHum;
    char windDir[10];
    int windSpeed;
} WeatherData;

void parseA(char* line, WeatherData w) {
    //printf("WaP %d\n", &w);
    char* c = strtok(line, " \	");
    strcpy(w.time, c);
    strtok(NULL, " \	"); // AM / PM skipped
    c = strtok(NULL, " \	");
    w.temp = (int)strtol(c, NULL, 10);
    c = strtok(NULL, " \	");
    c = strtok(NULL, " \	");
    w.dewPoint = (int)strtol(c, NULL, 10);
    c = strtok(NULL, " \	");
    c = strtok(NULL, " \	");
    w.relHum = (int)strtol(c, NULL, 10);
    c = strtok(NULL, " \	");
    strcpy(w.windDir, c);
    c = strtok(NULL, " \	");
    c = strtok(NULL, " \	");
    w.windSpeed = (int)strtol(c, NULL, 10);
}
```

// this does not work – quite
Structs again!

- Both of these work.
- You can return a struct from a function!
Weather end

• Look at differing calls to parseA, parseB and parseC
  
  • A
    • does not work
  
  • B
    • return by value so the struct created in function gets copied into weather[c]
  
  • C
    • pass the reference to the array object
    • fastest and cleanest.
    • recommend: always (almost) pass pointers to structs.

```c
void wprinter(WeatherData* w) {
    printf("%d Time:%s  Temp:%d F\n", w, w->time, w->temp);
}
WeatherData weather[100];
int main(void) {
    char line[256];
    FILE* f = fopen(WFILE, "r");
    if (f == NULL) {
        fprintf(stderr, "Could not open %s -- quitting\n", WFILE); return 1;
    }
    int c = 0;
    while (NULL != fgets(line, 256, f)) {
        switch (METHOD) {
        case 1:
            parseA(line, weather[c]); break;
        case 2:
            weather[c] = parseB(line); break;
        case 3:
            default:
            parseC(line, &weather[c]); break;
        }
        c++;
    }
    for (int i=0; i<c; i++) {
        wprinter(&weather[i]);
    }
}
```
Lab

• Define a struct that holds one integer and one character
• in main, create an array holding 51 instances of your struct
• in a separate function called from main, fill those 51 instances with a random character and a random integer. The loop for doing things 51 times should be in main.
• In a separate function called from main (and not the same as your previous function), print the 51 instances.