Lecture 5:
File I/O, Advanced Unix, Enum/Struct/Union, Subtyping
Project 3

• Time to get going on Project 3
• It is about 1000 lines of code
File I/O
File I/O: streams and file descriptors

- Two ways to access files:
  - File descriptors:
    - Lower level interface to files and devices
      - Provides controls to specific devices
    - Type: small integers (typically 20 total)
  - Streams:
    - Higher level interface to files and devices
      - Provides uniform interface; easy to deal with, but less powerful
    - Type: FILE *

Streams are more portable, and more accessible to beginning programmers. (I teach streams here.)
File I/O

• Process for reading or writing
  – Open a file
    • Tells Unix you intend to do file I/O
    • Function returns a “FILE *
      – Used to identify the file from this point forward
    • Checks to see if permissions are valid
  – Read from the file / write to the file
  – Close the file
Opening a file

- FILE *handle = fopen(filename, mode);

The argument mode points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

```
'`r'` Open text file for reading. The stream is positioned at the beginning of the file.

`r+'` Open for reading and writing. The stream is positioned at the beginning of the file.

```
```
Example: FILE *h = fopen("/tmp/330", "wb");
```

Close when you are done with “fclose”

```
`a+'` Open for reading and writing. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file, irrespective of any intervening fseek(3) or similar.

```

Note: #include <stdio.h>
FREAD(3) BSD Library Functions Manual FREAD(3)

NAME
fread, fwrite — binary stream input/output

LIBRARY
Standard C Library (libc, -lc)

SYNOPSIS
#include <stdio.h>

size_t
fread(void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);

size_t
fwrite(const void *restrict ptr, size_t size, size_t nitems,
       FILE *restrict stream);

DESCRIPTION
The function fread() reads nitems objects, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr.

The function fwrite() writes nitems objects, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr.

RETURN VALUES
The functions fread() and fwrite() advance the file position indicator for the stream by the number of bytes read or written. They return the number of objects read or written. If an error occurs, or the end-of-file is reached, the return value is a short object count (or zero).
Example

C02LN00GFD58:330 hank$ cat rw.c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char *hello = "hello world: file edition\n";
    FILE *f = fopen("330", "w");
    fwrite(hello, sizeof(char), strlen(hello), f);
    fclose(f);
}
C02LN00GFD58:330 hank$ gcc rw.c
C02LN00GFD58:330 hank$ ./a.out
C02LN00GFD58:330 hank$ cat 330
hello world: file edition
File Position Indicator

• File position indicator: the current location in the file

• If I read one byte, the one byte you get is where the file position indicator is pointing.
  – And the file position indicator updates to point at the next byte
  – But it can be changed...
The `fseek()` function sets the file position indicator for the stream pointed to by `stream`. The new position, measured in bytes, is obtained by adding `offset` bytes to the position specified by `whence`. If `whence` is set to SEEK_SET, SEEK_CUR, or SEEK_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the `fseek()` function clears the end-of-file indicator for the stream and undoes any effects of the `ungetc(3)` and `ungetwc(3)` functions on the same stream.
ftell

long ftell(FILE *stream);

The `ftell()` function obtains the current value of the file position indicator for the stream pointed to by `stream`. 
We have everything we need to make a copy command…

• fopen
• fread
• fwrite
• fseek
• ftell

Can we do this together as a class?
argc & argv

- two arguments to every C program
- argc: how many command line arguments
- argv: an array containing each of the arguments
- "./a.out hank childs"
- \( \rightarrow \) argc == 3
```c
#include <stdio.h>
#include <printf.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    FILE *f_in, *f_out;
    int buff_size;
    char *buffer;

    if (argc != 3) {
        printf("Usage: %s <file1> <file2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    f_in = fopen(argv[1], "r");
    fseek(f_in, 0, SEEK_END);
    buff_size = ftell(f_in);
    fseek(f_in, 0, SEEK_SET);

    buffer = malloc(buff_size);
    fread(buffer, sizeof(char), buff_size, f_in);

    printf("Copying %d bytes from %s to %s\n", buff_size, argv[1], argv[2]);

    f_out = fopen(argv[2], "w");
    fwrite(buffer, sizeof(char), buff_size, f_out);

    fclose(f_in);
    fclose(f_out);

    return 0;
}
```
Return values in shells

C02LN00GFD58:330 hank$ ./a.out copy.c copy2.c
Copying 697 bytes from copy.c to copy2.c
C02LN00GFD58:330 hank$ echo $? 0
C02LN00GFD58:330 hank$ ./a.out copy.c
Usage: ./a.out <file1> <file2>
C02LN00GFD58:330 hank$ echo $? 1

$? is the return value of the last executed command
Printing to terminal and reading from terminal

• In Unix, printing to terminal and reading from terminal is done with file I/O
• Keyboard and screen are files in the file system!
  – (at least they were ...)

Standard Streams

• Wikipedia: “preconnected input and output channels between a computer program and its environment (typically a text terminal) when it begins execution”

• Three standard streams:
  – stdin (standard input)
  – stdout (standard output)
  – stderr (standard error)

What mechanisms in C allow you to access standard streams?
printf

• Print to stdout
  – printf(“hello world\n”);
  – printf(“Integers are like this %d\n”, 6);
  – printf(“Two floats: %f, %f”, 3.5, 7.0);
fprintf

- Just like printf, but to streams
  - `fprintf(stdout, "helloworld\n");`
    - → same as printf
  - `fprintf(stderr, "helloworld\n");`
    - prints to "standard error"
  - `fprintf(f_out, "helloworld\n");`
    - prints to the file pointed to by FILE *f_out.
buffering and printf

• Important: printf is buffered

• So:
  – printf puts string in buffer
  – other things happen
  – buffer is eventually printed

• But what about a crash?
  – printf puts string in buffer
  – other things happen ... including a crash
  – buffer is never printed!

Solutions: (1) fflush, (2) fprintf(stderr) always flushed
Advanced Unix
Unix shells allows you to manipulate standard streams.

• “>” redirect output of program to a file

• Example:
  – ls > output
  – echo “this is a file” > output2
  – cat file1 file2 > file3
Unix shells allows you to manipulate standard streams.

- “<” redirect file to input of program
- Example:
  - python < myscript.py
  - Note: python quits when it reads a special character called EOF (End of File)
  - You can type this character by typing Ctrl-D
  - This is why Python quits when you type Ctrl-D
  - (many other programs too)
Unix shells allows you to manipulate standard streams.

- “>>>” concatenate output of program to end of existing file
  - (or create file if it doesn’t exist)

- Example:
  - `echo “I am starting the file” > file1`
  - `echo “I am adding to the file” >> file1`
  - `cat file1`
    - I am starting the file
    - I am adding to the file
What’s happening here?

C02LN00GFD58:330 hank$ mkdir tmp
C02LN00GFD58:330 hank$ cd tmp
C02LN00GFD58:tmp hank$ touch f1
C02LN00GFD58:tmp hank$ ls f1 f2 > out
ls: f2: No such file or directory
C02LN00GFD58:tmp hank$ cat out
f1

ls is outputting its error messages to stderr
Redirecting stderr in a shell

C02LN00GFD58:Documents hank$ cd ~/330
C02LN00GFD58:330 hank$ mkdir tmp
C02LN00GFD58:330 hank$ cd tmp
C02LN00GFD58:tmp hank$ touch f1
C02LN00GFD58:tmp hank$ ls f1 f2 > out
ls: f2: No such file or directory
C02LN00GFD58:tmp hank$ cat out f1
C02LN00GFD58:tmp hank$ ls f1 f2 > out 2>out_error
C02LN00GFD58:tmp hank$ cat out_error
ls: f2: No such file or directory
Redirecting stderr to stdout

```bash
mkdir tmp
ls f1 f2 > out
ls: f2: No such file or directory
ls f1 f2 > out 2>out_error
ls: f2: No such file or directory
ls f1 f2 > out 2>&1
ls: f2: No such file or directory
```

Convenient when you want both to go to the same stream
c functions: fork and pipe

• fork: duplicates current program into a separate instance
  – Two running programs!
  – Only differentiated by return value of fork (which is original and which is new)

• pipe: mechanism for connecting file descriptors between two forked programs

Through fork and pipe, you can connect two running programs. One writes to a file descriptor, and the other reads the output from its file descriptor

Only used on special occasions. (And one of those occasions is with the shell.)
pipes in Unix shells

- represented with “|”
- output of one program becomes input to another program
Very useful programs

• grep: keep lines that match pattern, discard lines that don’t match pattern
Very useful programs

- **sed**: replace pattern 1 with pattern 2
  - `sed s/pattern1/pattern2/g`
    - `s` means substitute
    - `g` means “global” ... every instance on the line

`sed` is also available in “vi”
- `:%s/pattern1/pattern2/g` (% means all lines)
- `:103,133s/p1/p2/g` (lines 103-133)
Wildcards

• ‘*’ is a wildcard with unix shells

```
fawcett:tmp child$ ls
Abe   Chavarria Hebb  Macy            Smith
Alajaji Chen  Jia  Maguire       Steelhammer
Alamoudi Clark Kine  Michlanski  Szczepanski
Anastas Collier Lee  Moreno       Totten
Andrade Costello Legge  Olson       Vega-Fujioka
Ballarche Donnelly Li   Owen          Wang
Brennan Etzel   Lin   Pogrebinsky Whiteley
Brockway Friedrich Liu   Qin           Woodruff
Brogan Garvin   Lopes Rhodes        Xu
Brooks Gonzales Luo  Roberts       Yaconelli
Bruce Guo   Lyon    Rodriguez      Young
Carlton Hampton Machado Rozenboim  Zhang
carlton Chavarria  Clark  Costello
Chalmers Chen  Collier
```

‘?’ is a wildcard that matches exactly one character
Backgrounding

• “&”: tell shell to run a job in the background
  – Background means that the shell acts as normal, but the command you invoke is running at the same time.

• “sleep 60” vs “sleep 60 &”

When would backgrounding be useful?
Suspending Jobs

• You can suspend a job that is running
  Press “Ctrl-Z”
• The OS will then stop job from running and not schedule it to run.
• You can then:
  – make the job run in the background.
    • Type “bg”
  – make the job run in the foreground.
    • Type “fg”
      – like you never suspended it at all!!
Other useful shell things

- ‘tab’: auto-complete
- esc=: show options for auto-complete
- Ctrl-A: go to beginning of line
- Ctrl-E: go to end of line
- Ctrl-R: search through history for command
Enum, structs, and unions
Enums

• Enums make your own type
  – Type is “list of key words”

• Enums are useful for code clarity
  – Always possible to do the same thing with integers

• Be careful with enums
  – ... you can “contaminate” a bunch of useful words
enum example

C keyword “enum” – means enum definition is coming

```c
enum StudentType {
    HighSchool,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    GradStudent
};
```

This enum contains 6 different student types

semi-colon!!!
enum example

int AverageAge(enum StudentType st)
{
    if (st == HighSchool)
        return 16;
    if (st == Freshman)
        return 18;
    if (st == Sophomore)
        return 19;
    if (st == Junior)
        return 21;
    if (st == Senior)
        return 23;
    if (st == GradStudent)
        return 26;

    return -1;
}
enums translate to integers ... and you can set their range

```c
128-223-223-72-wireless:330 hank$ cat enum2.c
#include <stdio.h>

define enum StudentType
{
    HighSchool = 105,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    GradStudent
};

int main()
{
    printf("HighSchool = %d, GradStudent = %d\n", HighSchool, GradStudent);
}
```

```bash
128-223-223-72-wireless:330 hank$ gcc enum2.c
128-223-223-72-wireless:330 hank$ ./a.out
HighSchool = 105, GradStudent = 110
```
But enums can be easier to maintain than integers

```cpp
enum StudentType {
    HighSchool,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    PostBacc,
    GradStudent
};

int AverageAge(enum StudentType st) {
    if (st == HighSchool) return 16;
    if (st == Freshman) return 18;
    if (st == Sophomore) return 19;
    if (st == Junior) return 21;
    if (st == Senior) return 23;
    if (st == PostBacc) return 24;
    if (st == GradStudent) return 26;
    return -1;
}
```

If you had used integers, then this is a bigger change and likely to lead to bugs.
Structs
Data types

- float
- double
- int
- char
- unsigned char

All of these are simple data types
Structs: a complex data type

• Structs: mechanism provided by C programming language to define a group of variables
  – Variables must be grouped together in contiguous memory

• Also makes accessing variables easier ... they are all part of the same grouping (the struct)
C keyword “struct” – means struct definition is coming

```
struct Ray
{
  double origin[3];
  double direction[3];
};
```

This struct contains 6 doubles, meaning it is 48 bytes

```
int main()
{
  struct Ray r;
  r.origin[0] = 0;
  r.origin[1] = 0;
  r.origin[2] = 0;
  r.direction[0] = 1;
  r.direction[1] = 0;
  r.direction[2] = 0;
}
```

Declaring an instance

“." accesses data members for a struct
Nested structs

```c
struct Origin {
    double originX;
    double originY;
    double originZ;
};

struct Direction {
    double directionX;
    double directionY;
    double directionZ;
};

struct Ray {
    struct Origin ori;
    struct Direction dir;
};

int main() {
    struct Ray r;
    r.ori.originX = 0;
    r.ori.originY = 0;
    r.ori.originZ = 0;
    r.dir.directionX = 0;
    r.dir.directionY = 0;
    r.dir.directionZ = 0;
}
```

- accesses `dir` part of `Ray`
- accesses `directionZ` part of `Direction` (part of `Ray`)
typedef

- typedef: tell compiler you want to define a new type

```c
struct Ray
{
    double origin[3];
    double direction[3];
};

int main()
{
    struct Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

```c
typedef struct
{
    double origin[3];
    double direction[3];
} Ray;

typedef struct Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
```

saves you from having to type “struct” every time you declare a struct.
Other uses for typedef

• Declare a new type for code clarity
  – typedef int MilesPerHour;
    • Makes a new type called MilesPerHour.
    • MilesPerHour works exactly like an int.

• Also used for enums & unions
  – same trick as for structs ... typedef saves you a word
Unions

• Union: special data type
  – store many different memory types in one memory location

```c
typedef union
{
    float x;
    int   y;
    char  z[4];
} cis330_union;
```

When dealing with this union, you can treat it as a float, as an int, or as 4 characters.

This data structure has 4 bytes
Unions

128-223-223-72-wireless:330 hank$ cat union.c
#include <stdio.h>

typedef union
{
    float x;
    int y;
    char z[4];
} cis330_union;

int main()
{
    cis330_union u;
    u.x = 3.5; /* u.x is 3.5, u.y and u.z are not meaningful */
    u.y = 3;   /* u.y is 3, now u.x and u.z are not meaningful */
    printf("As u.x = %f, as u.y = %d\n", u.x, u.y);
}
128-223-223-72-wireless:330 hank$ gcc union.c
128-223-223-72-wireless:330 hank$ ./a.out
As u.x = 0.000000, as u.y = 3
Unions Example

typedef struct {
    int firstNum;
    char letters[3];
    int endNums[3];
} CA_LICENSE_PLATE;

typedef struct {
    char letters[3];
    int nums[3];
} OR_LICENSE_PLATE;

typedef struct {
    int nums[6];
} WY_LICENSE_PLATE;

typedef union {
    CA_LICENSE_PLATE ca;
    OR_LICENSE_PLATE or;
    WY_LICENSE_PLATE wy;
} LicensePlate;
Unions Example

typedef struct
{
    int firstNum;
    char letters[3];
    int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
    char letters[3];
    int nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
    int nums[6];
} WY_LICENSE_PLATE;

typedef union
{
    CA_LICENSE_PLATE ca;
    OR_LICENSE_PLATE or;
    WY_LICENSE_PLATE wy;
} LicensePlate;

typedef enum
{
    CA,
    OR,
    WY
} US_State;

typedef struct
{
    char *carMake;
    char *carModel;
    US_State state;
    LicensePlate lp;
} CarInfo;

int main()
{
    CarInfo c;
    c.carMake = "Chevrolet";
    c.carModel = "Camaro";
    c.state = OR;
    c.lp.or.letters[0] = 'X';
    c.lp.or.letters[1] = 'S';
    c.lp.or.letters[2] = 'Z';
    c.lp.or.nums[0] = 0;
    c.lp.or.nums[1] = 7;
    c.lp.or.nums[2] = 5;
}
Function Pointers
Function Pointers

• Idea:
  – You have a pointer to a function
  – This pointer can change based on circumstance
  – When you call the function pointer, it is like calling a known function
Function Pointer Example

```
#include <stdio.h>

int doubler(int x) { return 2*x; }
int tripler(int x) { return 3*x; }
int main()
{
    int (*multiplier)(int);
    multiplier = doubler;
    printf("Multiplier of 3 = %d\n", multiplier(3));
    multiplier = tripler;
    printf("Multiplier of 3 = %d\n", multiplier(3));
}
```

```
128-223-223-72-wireless:cli hank$ gcc function_ptr.c
128-223-223-72-wireless:cli hank$ ./a.out
Multiplier of 3 = 6
Multiplier of 3 = 9
```
What are the pros and cons of each approach?

```c
#include <stdio.h>
int doubler(int x) { return 2*x; }
int tripler(int x) { return 3*x; }
int main()
{
    int val;
    if (condition)
        val = doubler(3);
    else
        val = tripler(3);
    printf("Multiplier of 3 = %d\n", val);
}
```

```c
#include <stdio.h>
int doubler(int x) { return 2*x; }
int tripler(int x) { return 3*x; }
int main()
{
    int (*multiplier)(int);
    int condition = 1;
    if (condition)
        multiplier = doubler;
    else
        multiplier = doubler;
    printf("Multiplier of 3 = %d\n", multiplier(3));
}
```
Function Pointer Example #2

```
#include <stdio.h>
void doubler(int *X) { X[0] *= 2; X[1] *= 2; }
void tripler(int *X) { X[0] *= 3; X[1] *= 3; }
int main()
{
    void (*multiplier)(int *);
    multiplier = doubler;
    multiplier(A);
    printf("Multiplier of 3 = %d, %d\n", A[0], A[1]);
    multiplier = tripler;
    multiplier(A);
    printf("Multiplier of 3 = %d, %d\n", A[0], A[1]);
}
```

Don’t be scared of extra ‘*’s ... they just come about because of pointers in the arguments or return values.
Simple-to-Exotic Function Pointer Declarations

```c
void (*foo)(void);
void (*foo)(int **, char ***);
char ** (*foo)(int **, void (*)(int));
```

These sometimes come up on interviews.
Callbacks

• Callbacks: function that is called when a condition is met
  – Commonly used when interfacing between modules that were developed separately.
  – ... libraries use callbacks and developers who use the libraries “register” callbacks.
Callback example

```c
128-223-223-72-wireless:callback hank$ cat mylog.h
void RegisterErrorHandler(void (*eh)(char *));
double mylogarithm(double x);

128-223-223-72-wireless:callback hank$ cat mylog.c
#include <mylog.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* NULL is an invalid memory location.
 * Useful for setting to something known, rather than leaving uninitialized */
void (*error_handler)(char *) = NULL;

void RegisterErrorHandler(void (*eh)(char *))
{
    error_handler = eh;
}

void Error(char *msg)
{
    if (error_handler != NULL)
        error_handler(msg);
}

double mylogarithm(double x)
{
    if (x <= 0)
    {
        char msg[1024];
        sprintf(msg, "Logarithm of a negative number: %f !!", x);
        Error(msg);
        return 0;
    }
    return log(x);
}
```
Callback example

```c
#include <mylog.h>
#include <stdio.h>

FILE *F1 = NULL;
void HanksErrorHandler(char *msg)
{
    if (F1 == NULL)
    {
        F1 = fopen("error", "w");
    }
    fprintf(F1, "Error: %s\n", msg);
}

int main()
{
    RegisterErrorHandler(HanksErrorHandler);

    mylogarithm(3);
    mylogarithm(0);
    mylogarithm(-2);
    mylogarithm(5);
    if (F1 != NULL)
        fclose(F1);
}
```

```bash
128-223-223-72-wireless:callback hank$
cat program.c

128-223-223-72-wireless:callback hank$
#include <mylog.h>
#include <stdio.h>

FILE *F1 = NULL;
void HanksErrorHandler(char *msg)
{
    if (F1 == NULL)
    {
        F1 = fopen("error", "w");
    }
    fprintf(F1, "Error: %s\n", msg);
}

int main()
{
    RegisterErrorHandler(HanksErrorHandler);

    mylogarithm(3);
    mylogarithm(0);
    mylogarithm(-2);
    mylogarithm(5);
    if (F1 != NULL)
        fclose(F1);
}
```
Function Pointers

• We are going to use function pointers to accomplish “sub-typing” in Project 2D.
Subtyping
Subtyping

• Type: a data type (int, float, structs)

• Subtype / supertype:
  – Supertype: the abstraction of a type
    • (not specific)
  – Subtype: a concrete implementation of the supertype
    • (specific)

The fancy term for this is “subtype polymorphism”
Subtyping: example

• Supertype: Shape

• Subtypes:
  – Circle
  – Rectangle
  – Triangle
Subtyping works via interfaces

• Must define an interface for supertype/subtypes
  – Interfaces are the functions you can call on the supertype/subtypes

• The set of functions is fixed
  – Every subtype must define all functions
Subtyping

• I write my routines to the supertype interface
• All subtypes can automatically use this code
  – Don’t have to modify code when new supertypes are added

• Example:
  – I wrote code about Shapes.
  – I don’t care about details of subtypes (Triangle, Rectangle, Circle)
  – When new subtypes are added (Square), my code doesn’t change