CS 449 – Lecture 2

Dr. Jack Lange

tar

- tar is an archival tool for bundling multiple files
  - Will be used for turning in projects/labs
- Create an archive
  - `tar -czf <archive name>.tar.gz [files to include]`
- Extract an archive
  - `tar -xzf <archive name>.tar.gz`

C syntax parsing

- C is parsed **top down**
- This means variables/functions must be declared before they are used
  - **Before** means **above**
- In Java this is true for variables inside a method
  - But not for objects or methods
- In C this is true for **everything**
  - Functions must be declared before use
  - Variables must be declared before use
  - Data structures must be defined before use

Getting Input for a C program

- 1st mechanism: Command line arguments
  - List of words you typed into command line
  - Available as an array of strings named **`argv`**
    - `char * argv[]`
  - `argv[0]` is always the name of the executable you are running
- Example: cmdargs.c

Interactive Input

- 2nd Mechanism: Console Input
  - Example: **hello-1.c**
- scanf()
  - `int scanf(const char *format, ...);`
  - Waits for input from the console and parses it
    - According to a format string that specifies how to parse input
  - Copies resulting values into memory locations
    - Uses location of variables in memory (& operator)
  - Returns the number of successful matches
    - Used to detect errors (0 == No input parsed)

Interactive Input - Streams

- 3rd mechanism: FIFO Streams
  - **FIFO** = First In, First Out (in order processing)
- Console streams
  - stdin – standard input
  - stdout – standard output
  - stderr – standard error
- Other streams
  - Files, Devices, Network Sockets
  - We'll get to block interfaces later

Streams (2)

- Streams are buffered by the OS
  - Must be flushed to transfer data
  - Buffers can be configured
- Implicit/Explicit streams
  - printf() and other functions implicitly use stdin/stdout
  - Variants exist that allow explicit stream selection
    - E.g. fprintf() allows printing to a specific stream

- `int printf(const char *format, ...)`
- `int fprintf(FILE *stream, const char *format, ...);`

Easy File I/O

- The UI shell can be used for file I/O
  - Files and streams are the same in C
    - Unix: *Everything is a file*
  - …so we can mix and match easily
- Shell Redirection
  - Redirect console streams
    1. Redirect *stdout* to a file ('>' and '>>')
       - `ls > files.out` -- creates new file "files.out"
       - `ls >> all_files.out` -- appends to file "all_files.out"
    2. Read *stdin* from a file ('<')
       - `wc < files.out`
    3. Pipe *stdout* of one program to *stdin* of another **('|')**
       - `ls -l | sort`
       - `ls | grep .c | wc`

Special Streams

- Streams are everywhere
- OS uses streams to provide special features
  - Exposed as special files in the file system
- Examples
  - /dev/null
    - A black hole of zero size but infinite capacity
    - Reads always returns nothing
    - Writes disappear
  - /dev/zero
    - Reads: infinite number of zeroes
    - Different than /dev/null
  - /dev/random
    - Reads: a stream of random numbers

File I/O in programs

- Command line and streams handled by system
- Files you have to manage on your own
  - `FILE * file = fopen(name, <mode>);`
    - On error NULL is returned
  - `void fclose(FILE * file)`
- Text files
  - Once open, a text file **CAN** be treated like a console
  - `fprintf, fscanf` – variants that can work on files

ASCII Files

- ASCII: Encoding that maps bytes to text characters
  - Translates characters from human readable to machine readable
  - ”7” is a human readable string representation of the number 7
  - 7 is a raw binary number stored in memory

- Examples

```
int i = 7
fprintf("%d\n", i);

Memory: 0x00000007
File: 0x37, 0x0a
```

```
int i = -7;
fprintf("%d\n", i);

Memory: 0xfffffff9
File: 0x2d, 0x37, 0x0a
```

Explicit File I/O

- An application can only operate on data in memory
  - To analyze and modify data it must be resident in memory first
  - Programmer must **copy** data to/from files and memory

- File accesses are similar to memory
  - Files are just arrays of bytes
  - BUT, we accesses are **explicit** not **implicit**

- Explicit
  - Handled by the programmer explicitly (in code)
- Implicit
  - Handled by the compiler/HW automatically

File I/O

- **Reading raw data from a file to memory**

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

- **Writing raw data from memory to a file**

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

File I/O behavior

```
int x = 5;
File * outfile = fopen(argv[1], "wb");
fwrite(&x, sizeof(x), 1, outfile);

memory: 0x00000007
file: 0x00000007
```

File Offsets

- Files are arrays of bytes
  - BUT the fread/fwrite don't have file offset args
- File offsets are implicitly tracked by the system
  - I/O operations always occur at the current "file position"
  - The *file position* is the offset at which you are currently at
  - The location changes as you read/write
    - Reading 5 bytes from a file, moves the current position forward by five bytes
- Explicitly modifying the file position
  - `int fseek(FILE * stream, long offset, int whence);`
  - fseek(fstream, 16, SEEK_SET); -- Moves the file position to byte 16 of the file