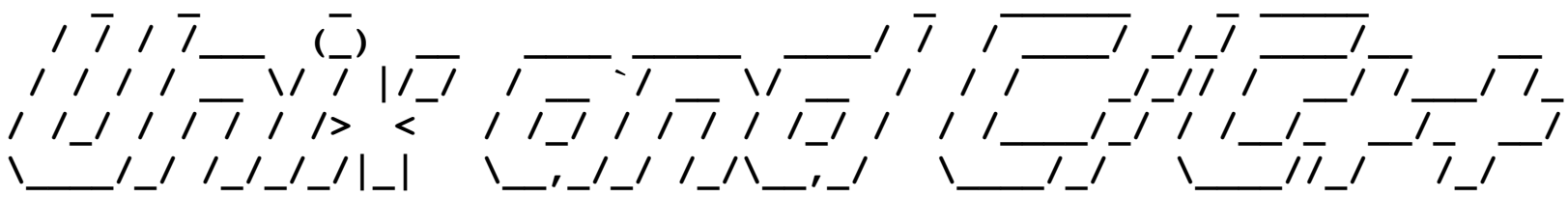
UNIVERSITY OF OREGON

CIS 330:



Lecture 6: file I/O, finish Unix

April 15th, 2016

Hank Childs, University of Oregon

Announcements

- Projects
  - 4A assigned: due in class on Weds
  - 2B prompt available Saturday AM, due on Friday
- No class on Weds April 27
  - likely a short YouTube replacement lecture

Outline

- Review
- File I/O
- Project 2B
- Redirection
- Pipes

Outline

- Review
- File I/O
- Project 2B
- Redirection
- Pipes

Makefile example: multiplier lib

code — bash — 80×26

```
C02LN00GFD58:code hank$ cat Makefile
lib: doubler.o tripler.o
        ar r libmultiplier.a doubler.o tripler.o
        cp libmultiplier.a ~/multiplier/lib
        cp multiplier.h ~/multiplier/include

doubler.o: doubler.c
        gcc -c doubler.c

tripler.o: tripler.c
        gcc -c tripler.c
C02LN00GFD58:code hank$
C02LN00GFD58:code hank$
C02LN00GFD58:code hank$ make
ar r libmultiplier.a doubler.o tripler.o
cp libmultiplier.a ~/multiplier/lib
cp multiplier.h ~/multiplier/include
C02LN00GFD58:code hank$
C02LN00GFD58:code hank$ touch doubler.c
C02LN00GFD58:code hank$ make
gcc -c doubler.c
ar r libmultiplier.a doubler.o tripler.o
cp libmultiplier.a ~/multiplier/lib
cp multiplier.h ~/multiplier/include
C02LN00GFD58:code hank$
```

Unix command: tar

- tar cvf 330.tar file1 file2 file3
  - puts 3 files (file1, file2, file3) into a new file called 330.tar
- scp 330.tar @ix:~
- ssh ix
- tar xvf 330.tar
- ls

  file1 file2 file

Memory Errors

- Array bounds read

```
int main()
{
    int var;
    int arr[3] = { 0, 1, 2 };
    var=arr[3];
}
```

- Array bounds write

```
int main()
{
    int var = 2;
    int arr[3];
    arr[3]=var;
}
```

Memory Errors

- Free memory read / free memory write

```
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    var[0] = var[1];
}
```

Vocabulary: “dangling pointer”: pointer that points to memory that has already been freed.

Memory Errors

- Freeing unallocated memory

```
int main()
{
    int *var = malloc(sizeof(int)*2);
    var[0] = 0;
    var[1] = 2;
    free(var);
    free(var);
}
```

Memory Errors

- Freeing non-heap memory

```
int main()
{
    int var[2]
    var[0] = 0;
    var[1] = 2;
    free(var);
}
```

Memory Errors

- NULL pointer read / write

```
int main()
{
    char *str = NULL;
    printf(str);
    str[0] = 'H';
}
```

- NULL is never a valid location to read from or write to, and accessing them results in a "segmentation fault"
  - …. remember those memory segments?

Memory Errors

- Uninitialized memory read

```
int main()
{
    int *arr = malloc(sizeof(int)*10);
    int V2=arr[3];
}
```

Memory error in action

```
fawcett:error childs$ cat t.c
#include <stdio.h>

int main()
{
    int *X = NULL;
    printf("X is %d\n", *X);
}
fawcett:error childs$ gcc t.c
fawcett:error childs$ ./a.out
Segmentation fault
fawcett:error childs$
```

Project 4A

- Posted now
- You will practice debugging & using a debugger
  - There are 3 programs you need to debug
    - In this case, “debug” means identify the bug
      - Does not mean fix the bug
    - Can use gdb or lldb
    - May want to run on ix
- Worksheet due in class next week

ASCII Character Set

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

There have been various extensions to ASCII …
now more than 128 characters
Many special characters are handled outside this convention

image source: granneman.com

signed vs unsigned chars

- signed char (“char”):
  - valid values: -128 to 127
  - size: 1 byte
  - used to represent characters with ASCII
    - values -128 to -1 are not valid
- unsigned char:
  - valid values: 0 to 255
  - size: 1 byte
  - used to represent data

character strings

- A character “string” is:
  - an array of type “char”
  - that is terminated by the NULL character
- Example:

  char str[12] = “hello world”;
  - str[11] = ‘\0’ (the compiler did this automatically)
- The C library has multiple functions for handling strings

Character strings example

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

int main()
{
    char str[12] = "hello world";
    char *str2 = str+6;

    printf("str is \"%s\" and str2 is \"%s\"\n",
           str, str2);

    str[5] = '\0';

    printf("Now str is \"%s\" and str2 is \"%s\"\n",
           str, str2);
}
128-223-223-72-wireless:330 hank$ gcc string.c
128-223-223-72-wireless:330 hank$ ./a.out
str is "hello world" and str2 is "world"
Now str is "hello" and str2 is "world"
```

memcpy

```
MEMCPY(3)                 BSD Library Functions Manual                 MEMCPY(3)

NAME
     memcpy -- copy memory area

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <string.h>

     void *
     memcpy(void *restrict dst, const void *restrict src, size_t n);

DESCRIPTION
     The memcpy() function copies n bytes from memory area src to memory area
     dst.  If dst and src overlap, behavior is undefined.  Applications in
     which dst and src might overlap should use memmove(3) instead.

RETURN VALUES
     The memcpy() function returns the original value of dst.
```

I mostly use C++, and I still use memcpy all the time

if-then-else

```
int val = (X < 2 ? X : 2);
```

←→

```
if (X<2)
{
   val = X;
}
else
{
   val = 2;
}
```

Outline

- Review
- File I/O
- Project 2B
- Redirection
- Pipes

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 begin-ning of the file.

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

exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

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.

racter or
rings
99:1990

Note: #include <stdio.h>

Reading / Writing

Promotion — less — 98×35

```
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
```

fseek

```
int
fseek(FILE *stream, long offset, int whence);
```

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”
- → argc == 3
- → argv[0] = “a.out”, argv[1] = “hank”, argv[2] = “childs”

```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

Outline

- Review
- File I/O
- Project 2B
- Redirection
- Pipes

Project 2B

(which means submitted by 6am on April 24th, 2015)
Worth 4% of your grade

Assignment: Write a program that reads the file "2E_binary_file". This file contains a two-dimensional array of integers, that is 10x10. You are to read in the 5x5 bottom left corner of the array. That is, the values 0-4, 10-14, 20-24, 30-34, and 40-44. You may only read 25 integers total. Do not read all 100 and throw some out. You will then write out the new 5x5 array. Please write this as strings, one integer per line (25 lines total). You should be able to "cat" the file afterwards and see the values.

Use Unix file streams for this project (i.e., fopen, fread, fseek, fprintf). Your program will be checked for good programming practices. (Close your file streams, use memory correctly, etc. I am not referring to style, variable initialization, etc.)

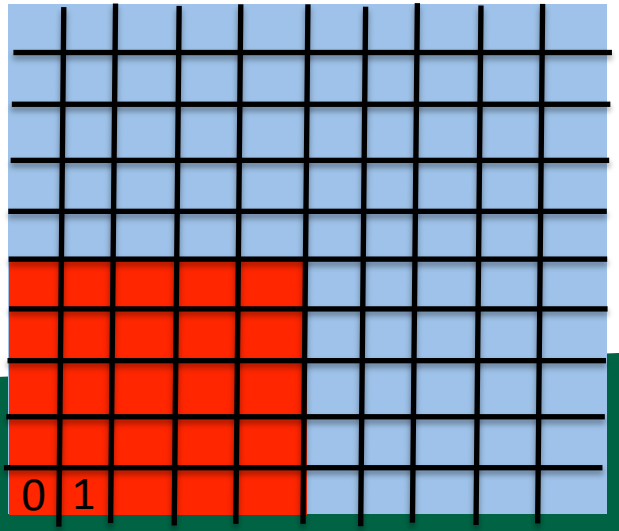
Also, add support for command line arguments (argc and argv).

Your program should run as:
./<prog_name> <input_name> <output_name>

(The input_name will be 2E_binary_file, unless you change it.)

Finally, note that I am handing you a binary file. I think we are all little endian, and so it will be fine. But, if it is big endian, then we will have a problem. You can check if it is little endian by printing the first two values of the file. They should be "0" and "1".

Please submit a tarball with (1) a Makefile (should be simple), (2) your source code, and (3) the output ASCII file from running your program, with the name "ASCII_output".



Outline

- Review
- File I/O
- Project 2B
- Redirection
- Pipes

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

```
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
C02LN00GFD58:tmp hank$ ls f1 f2 > out 2>&1
C02LN00GFD58:tmp hank$ cat out
ls: f2: No such file or directory
f1
```

Convenient when you want both to go to the same stream

Outline

- Review
- File I/O
- Project 2B
- Redirection
- Pipes

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 descript

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

pipes in Unix shells

```
C02LN00GFD58:tmp hank$ cat printer.c
#include <stdio.h>
int main() { printf("Hello world\n"); }
C02LN00GFD58:tmp hank$ cat doubler.c
#include <stdio.h>
int main()
{
    int ch = getc(stdin);
    while (ch != EOF)
    {
        printf("%c%c", ch, ch);
        ch = getc(stdin);
    }
}
C02LN00GFD58:tmp hank$ gcc -o printer printer.c
C02LN00GFD58:tmp hank$ gcc -o doubler doubler.c
C02LN00GFD58:tmp hank$ ./printer | ./doubler
HHeellllloo  wwoorrlldd

C02LN00GFD58:tmp hank$
```

- 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

```
C02LN00GFD58:Documents hank$ ls -l | grep ppt
-rw-r--r--@ 1 hank  staff   3278589 Apr  5 11:40 CIS330_Lec2.pptx
-rw-r--r--@ 1 hank  staff   2220104 Apr  8 20:57 CIS330_Lec3.pptx
-rw-r--r--  1 hank  staff   3899863 Jan 21 09:26 CIS610_lec2.pptx
-rw-r--r--  1 hank  staff   4629257 Jan 30 10:24 CIS610_lec3.pptx
-rw-r--r--  1 hank  staff  21382185 Mar 25 12:40 CIS_colloquium2013.pptx
-rw-r--r--  1 hank  staff  21382185 Jan  7 12:21 CIS_colloquium_2013.pptx
-rw-r--r--@ 1 hank  staff   2172179 Dec 20 15:24 ICS_results.pptx
-rw-r--r--@ 1 hank  staff   4841050 Nov 13 10:10 MBTI.pptx
-rw-r--r--@ 1 hank  staff   2031749 Apr  5 16:20 SC14_flow.pptx
-rw-r--r--  1 hank  staff  17972476 Mar 25 12:43 VMV_2013.pptx
-rw-r--r--@ 1 hank  staff  98149068 Apr  1 10:25 aachen.pptx
-rw-r--r--  1 hank  staff   9815146 Feb 24 07:00 childs_poster_SDAV_AHM_2014.pptx
-rw-r--r--@ 1 hank  staff    592243 Feb 26 04:09 childs_sdav_slides.pptx
-rw-r--r--@ 1 hank  staff  15765504 Feb 13 14:57 cig_exascale.ppt
-rw-r--r--@ 1 hank  staff  16699392 Jan  7 12:14 cis610_Lec1.ppt
-rw-r--r--  1 hank  staff   3159872 Jan  7 11:15 egpgv_cgf.pptx
-rw-r--r--@ 1 hank  staff  15767552 Mar 23 02:48 eu_regional_school.ppt
-rw-r--r--@ 1 hank  staff  35099136 Mar 25 09:42 eu_regional_school_part1.ppt
-rw-r--r--@ 1 hank  staff  10775552 Mar 25 04:49 eu_regional_school_part1B.ppt
-rw-r--r--@ 1 hank  staff  72966144 Mar 26 08:43 eu_regional_school_part2.ppt
-rw-r--r--  1 hank  staff   7571317 Mar 25 12:53 ilm_booth_talk.pptx
```

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)

Bonus topic: wildcards

- ‘*’ is a wildcard with unix shells

```
fawcett:tmp childs$ 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             Lynch           Rodriguez       Young
Carlton         Hampton         Lyon            Roush           Zhang
Chalmers        Harris          Machado         Rozenboim       de
fawcett:tmp childs$ ls C*
Carlton         Chavarria       Clark           Costello
Chalmers        Chen            Collier
fawcett:tmp childs$ ls *z
Rodriguez
fawcett:tmp childs$ ls *ee*
Lee             Steelhammer
fawcett:tmp childs$ ls *e*e*
Lee             Legge           Steelhammer     Whiteley
```

‘?’ is a wildcard that matches exactly one character

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