COSC 3750

Programming

Kim Buckner

University of Wyoming

Feb. 08, 2022

Last time

- Started talking about file I/O
- Using the buffered file functions like fopen, fread, fwrite
- Must stress, if you opened it, CLOSE IT. Do not rely on the operating system or some cleanup function to do it correctly.

YOU MUST

- NOT USE TABS FOR INDENTATION!
- fix your editor so that it uses spaces instead.
- NOT COPY files to Windows editors. They put in bad characters. Those characters keep scripts from running and scripts from compiling.

Writing a program

- This will be in C
- This is NOT C++.
- You will create a Makefile for each program you write.
- Good style, reasonable comments, and clean formatting are all important.

(more . . .)

- Read the Style Guide.
- Keep it simple.
- You will NOT need anything like "-std=c90."
- C is getting more C++-like in that it now allows things like

```
for(int i=0;...)
```

“wycat”

- The first C programming assignment is to create a version of the utility “cat”.
- This requires two sets of interrelated operations.
- First accessing the program arguments and
- Then opening, reading from, writing to, and closing files.

(more . . .)

- You WILL do what the assignment document requires.
- You WILL NOT do what Sharon on Stackoverflow, or Tim on Stackexchange says.
- You WILL NOT do things because you saw them in the **cat** manpage.

(more . . .)

- IF you have ANY questions ask ME. I am the final arbiter. I have over thirty of these to grade so am unlikely to be lenient about problems/errors/misinterpretations.

The arguments

- All C/C++ programs have available to them the command line arguments AND the environment variables.
- Most of you have seen
  `int main(int argc, char **argv)`
- But what does that mean?
- "argc" is the argument count. The name of the executable is always first. Means the smallest value argc can contain is one (1).

argv

- This is the argument vector, an array.
- BUT it is an array of pointers to what are usually called “c-strings”.
- Often referred to as “a NULL terminated array of pointers to null terminated strings”.
- Will often see code that iterates over argv until the value is NULL.

envp

- The environment pointer.
- It is the one you do not normally see, or usually even need.
- Looks like

```
int main(int argc, char **argv,
char **envp)
```

- Similar to argv but contains externally defined variables that become part of the process's environment.

NULL

- This is not a be-all, end-all value.
- It MAY be
  - #define NULL ((void *)0)
  - #define NULL 0L
- ONLY USE IT FOR POINTERS!!!!!

null

- This is not a real thing in C or C++.
- If you look at an ASCII table, you will see the first value is NUL.
- So when you see a statement like "a NULL terminated array of null terminated strings" it means
  - it is an array of pointers to characters,
  - the pointers are to character arrays,
  - the last character in each array is the character '\0',
  - the last array pointer is NULL (no array).

Accessing

- *envp* can be accessed with *getenv(3)*, *setenv(3)*, *putenv(3)*, etc.
- *argc* is just an integer variable, *argv* and *envp* are arrays regardless of what the declaration look likes.
- The name of the program is stored in *argv*[0], the rest of the arguments (if any) are stored in *argv*[1] — *argv*[*argc*-1].

(more . . .)

- DO NOT MODIFY ARGC OR ARGV VARIABLES!!!!!
- You are asking for trouble down the road.
- *envp* is like *argv* but other than knowing the first element is *envp*[0], there is no count.
- One either uses the standard library functions or iterates over it.

(more . . .)

```
char **ptr;


ptr=envp;
while(*ptr != NULL) {
   printf("%s\n",*ptr);
   ptr++;
}
```

What's in an arg

- Well, we can tell the user what we expect.
- We get what we get "and we don't throw a fit" (thank you 3rd grade teachers).
- Just assume that there is a reasonable user.
- But what about those programs that take arguments like "-d,"

Comparisons

- The standard function used is strcmp(),
- There is also strncmp(), and some others
- The idea is call the function with two arguments, an “unknown” and a “pattern”.
- If the strings are exactly the same the function returns 0. Otherwise it returns 1 or -1, depending on which argument is “larger”.

On to **wycat**

- The **wycat** utility, the one we will be writing, does some very simple things
  - If there are no arguments, copy standard input to standard output with NO modification/addition/exclusion.
  - If there are arguments, assume that they are files and copy them sequentially (one after the other) to standard output.
  - If an argument is "-", copy standard input to standard output.

Checking for errors

- The only way to create robust programs is to check everything for errors.
- You must save and understand the return values of functions.
- You must react correctly.
- To arbitrarily exit on errors is not “robust”.

Error messages

- Have talked briefly about *errno*.
- Include **errno.h** to access the *errno* variable
- Three basic items:
  1. the variable *errno*,
  2. the function *perror()*, and
  3. the function *strerror()*.

(more . . .)

- Only need *stdio.h* for the functions.
- *perror()* takes a constant string and prints the "human readable" error message based on *errno*.
- *strerror()* takes an error number and returns the string associated with that number.
- Allows more flexibility than *perror()*

That all means

- Check the return values of all the functions and do something smart (or reasonable) with them.
- Print meaningful error messages.
- Do **not** exit on the first error unless necessary.

The process

- Check the arguments
- If none, (argc == 1) then read from stdin and write to stdout.
- Else open the files one at a time
- Read each file in, write it to stdout.

(more . . .)

- If any of the arguments are “-” :
  - Then read from stdin and write to stdout,
  - Reading from stdin will result in an End of File indication IF the user types CTRL-D as the first character on a line a presses enter.
  - THIS DOES NOT CLOSE stdin. It just looks to the fread() function as if it were closed.
  - You do NOT HAVE TO TRY TO REOPEN stdin. It was never closed.
  - DO NOT READ 1 CHARACTER AT A TIME.

(more . . .)

- Close, using *fclose()*, every file you open
- **Do not** close stdin or stdout.

Some requirements

- You will only use the **fread**, **frwrite**, **fopen**, and **fclose** functions on files (including stdin and stdout).
- You will have a Makefile (that is in the instructions).

(more . . .)

- You will test it. That means compiling and running tests on department machines (NO HIVE).
- What you turn in will be what you tested. Figure that out.