CS 220: Introduction to Parallel Computing

Input/Output

Lecture 7

Input/Output

- Most useful programs will provide some type of input or output
- Thus far, we've prompted the user to enter their input directly
  - scanf
- There are more options:
  - Command line arguments
  - File I/O

Today's Agenda

- Command Line Arguments
- Reading and Processing Files
- Error Handling

Today's Agenda

- **Command Line Arguments**
- Reading and Processing Files
- Error Handling

Input From the Command Line

- Passing command line arguments is a common form of input:

```
./my_program testmode ./file.txt
```

- We see this often with Unix utilities:

```
ls -l /my/directory
```

- This makes providing input to a program easier, and allows for **scripting** as well:

```
./my_program ${MODE} ./file.txt
```

Command Line Arguments

- You may have noticed an alternative version of our main(void) function:

```
int main(int argc, char *argv[])
```

- This allows us to accept and process **command line arguments**
  - For example, when you run 'git status,' the string 'status' is passed to the main method
  - In fact, so is the name of the program, 'git'

Argument Attributes

- We receive two parameters:
  - **argc** – the number of command line arguments
  - **argv** – the arguments themselves
- Some notes:
  - argc will always be at least 1
  - argv will always start with the name of your program

    ‘a.out’

    ‘array’
- So if we want one argument, ‘status,’ we test whether

  `argc == 2`

Looking Closer: argv

- Another thing to notice: how argv is defined

```
char *argv[]
```

- A pointer to an array… Which we know is also represented by a pointer
  - Or in other words, a pointer to a pointer
- Here's another valid definition of argv:

```
char **argv
```

- So this is a 2D array: an array of character arrays

Processing Arguments

- Command line arguments are C strings
  - They are terminated by \0
- If we're looking for a status command, we can do a string comparison:

```
strcmp(argv[1], "status");
```

- If the string matches, we'll run the 'status functionality' in our hypothetical git program

strcmp

```
char stra[] = "Hello";
char strb[] = "Hello World!";

if (strcmp(stra, strb) == 0) {
    printf("They're the same!\n");
}
```

Why == 0?

Converting Arguments

- In many cases, you'll want to accept an integer on the command line
- Converting a string to integer is accomplished with the **atoi()** function
  - Available in the C standard library: #include <stdlib.h>
- There are some others: **atof()**, **atol()**…
- You may also wonder if there is an **itoa()** function
  - There is! But it is ***NOT*** part of the C standard

Demo: Command Line Args

Today's Agenda

- Command Line Arguments
- **Reading and Processing Files**
- Error Handling

Opening a File

```
/* This opens the file specified by the
   first command line argument: */
printf("Opening file: %s\n", argv[1]);
FILE *file = fopen(argv[1], "r");

/* Note the ”r”: open for reading */
```

Open Modes

- The basics:
  - r – read
  - w – write
  - a – append
- This isn't the full story, however: each mode can be followed by a '+'
  - r+ - open for read and write, file must exist
  - w+ - open for read and write, file is created if not present
- There are more details in the man page for **fopen()**

Reading Line by Line – **fgets**

- Once we have opened a file, we need to read it
- A common approach is reading line by line via the **fgets** function:

```
char line[500];
while (fgets(line, 500, file) != NULL) {
        /* Process the line */
}
```

- This uses a 500-character buffer to store the line
  - fgets will also stop once it finds a newline (\n) character

Rewinding a File

- When you reach the end of a file, you'll get a NULL or **EOF** return value
  - This tells you that you've reached the **E**nd **O**f **F**ile
- If you want to loop through the file again, go back to the start:
  - `fseek(file, 0L, SEEK_SET);`
  - `rewind(file); /* Note: old, deprecated */`
- You can also re-open the file

Cleaning Up

- It's good practice to also close your files when you're done with them:
  - `fclose(file)`
- Each file you open uses up a **file descriptor**
  - The operating system imposes limits on how many file descriptors can be open per program
- When you open several files, don't forget to close them when you're done!

String Tokenization

- A common use case for strings is **tokenization**
  - Or, splitting them based on characters
- Consider the following string:
  - `"Hello, how are you today?";`
- How can we retrieve each word individually?
  - [Hello,] [how] [are] [you] [today?]
  - Java/Python have nice split() methods for this
- In C, we can use **strtok**

Tokenizing a String

```
/* Tokenize based on space and newline
 * characters: */
char *token = strtok(line, " \n");
while (token != NULL) {
    /* do something with token */
    /* then grab the next token: */
    token = strtok(NULL, " \n");
}
```

Why include \n?

- Blank lines won't contain any tokens
- You'd expect strtok() to just return NULL immediately, but this is not the case
- If there are *no* tokens found, the entire string is returned
  - Makes more sense if we take a look at how strtok() is implemented

How **strtok** Works

- When it comes to C functions, strtok is one of the stranger ones
- First, we pass in the string we want to tokenize
- After that, we pass in NULL and it gives us the next token
- How does it even know what string to operate on?
  - strtok maintains a **global** pointer to the start of the most recent token

strtok – Global State

- In C, we have **global** and **local** variables
- Globals are defined outside of any function
  - For example, up above your main function
- Some C library functions even do this
  - When you #include them, they get added to your code
  - C provides the **static** keyword to restrict global variables' scope to their compilation unit
    - Generally compilation unit = file
    - This way we don't pollute the global namespace

strtok – Splitting Strings

- Beyond the strange pointer magic, we also need to know how strtok splits things up
- It scans through the string until it comes across one of the user-defined tokens
- The token is replaced with \0
- Now printing the string only prints up to the NUL byte
- To move to the next token, strtok simply changes the pointer to come **after** the NUL byte!

Today's Agenda

- Command Line Arguments
- Reading and Processing Files
- **Error Handling**

Error Handling

- Thus far we've pretended that errors don't happen
  - This can be okay: if a user doesn't enter a value, for instance, then maybe everything will just be zero
- Sometimes you do need to know when something went wrong
  - Example: trying to read a file that doesn't exist
- In C, errors are indicated with the **return code**
  - You need to check the function info to find out what the return codes mean

Printing Error Messages

- The perror() function is your friend when you want to get a user-friendly description of what went wrong

```
cannot access 'blah.txt': No such file or directory
```

- When an error occurs, the C library updates the **last error code**
- Calling perror() will look up this error code and print a friendly description
- You can add a prefix string to help you trace through your code

Opening a File: Error Handling

```
#include <stdlib.h>


FILE *file = fopen("data.txt", "r");
if (file == NULL) {
    perror("fopen");
    return EXIT_FAILURE;
}
```

Error handling – perror()

- One important note about using the perror() function: it only knows about C library errors
- If you prompt the user to enter a positive number and they enter a negative one instead, perror() won't help
- In those cases, you're on your own
  - But still make sure you report the error!

EXIT_FAILURE

- You may have noticed EXIT_FAILURE in the previous example
- This indicates that your program had to stop because of some type of error condition
- All programs provide an exit code – 0 generally means success

Quitting in Style – exit() function

```
#include <stdlib.h>

…

/* If you're not in main(), you can still quit
your program: */

void my_func(void)
{
        printf("Hello?\n");

        exit(EXIT_FAILURE);

}
```