Week 10 - Wednesday

COMP 2400

Last time

- What did we talk about last time?
- Exam 2
- Before that:
  - Review
- Before that:
  - Binary trees
  - Unions
  - Time

Questions?

Project 5

Quotes

Measuring programming progress by lines of code is like measuring aircraft building progress by weight.

Bill Gates

Back to Time

Time structures

- Many time functions need different structs that can hold things
- One such struct is defined as follows:

```
struct timeval
{
      time_t tv_sec;          // Seconds since Epoch
      suseconds_t tv_usec;    // Extra microseconds
};
```

gettimeofday()

- The **gettimeofday()** function offers a way to get higher precision timing data
- Its signature is:

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

- From the previous slide, **timeval** has a **tv_secs** member which is the same as the return value from **time()**
- It also has a **tv_usec** member which gives microseconds (millionths of a second)
- The **timezone** pointer **tz** is obsolete and should have **NULL** passed into it
- Include **sys/time.h** (**not** the same as **time.h**) to use this function

`ctime()`

- What about printing out a human-readable version of the time?
- **`ctime()`** takes a **`time_t`** value and returns a string giving the day and time

```
printf(ctime(time(NULL)));
// Prints Wed Mar 20 11:42:34 2024
```

- Alternatively, **`strftime()`** has a set of specifiers (similar to **`printf()`**) that allow for complex ways to format the date and time

Broken down time structure

```
struct tm
{
   int tm_sec; // Seconds (0-60)
   int tm_min; // Minutes (0-59)
   int tm_hour; // Hours (0-23)
   int tm_mday; // Day of the month (1-31)
   int tm_mon; // Month (0-11)
   int tm_year; // Year since 1900
   int tm_wday; // Day of the week (Sunday = 0)
   int tm_yday; // Day in the year (0-365; 1 Jan = 0)
   int tm_isdst; /* Daylight saving time flag
   > 0: DST is in effect;
   = 0: DST is not effect;
   < 0: DST information not available */
};
```

gmtime(), localtime(), and mktime()

- **gmtime()** and **localtime()** convert a **time_t** value to a struct that contains "broken down" time
  - **gmtime()** gives UTC time (used to be called Greenwich Mean Time)
  - **localtime()** gives the local time, assuming it is set up correctly

```
time_t seconds = time(NULL);
struct tm* brokenDownTime = NULL;
brokenDownTime = localtime(&seconds);
if( brokenDownTime->tm_wday == 1 )
     printf("It's just another manic Monday.\n");
```

- **mktime()** can convert from a broken down time back into **time_t**

Jiffies

- How accurate is the microsecond part of **`gettimeofday()`**?
- It depends on the accuracy of the software clock in your system
- This clock measures time in units called **jiffies**
- A jiffy used to be 10 milliseconds (100 Hz)
- They raised the accuracy to 1 millisecond (1000 Hz)
- Now, it can be configured for your system to 10, 4 (the default), 3.3333, and 1 milliseconds

Process time

- For optimization purposes, it can be useful to know how much time a process spends running on the CPU
- This time is often broken down into
  - **User time:** the amount of time your program spends executing its own code
  - **System time:** the amount of time spent in kernel mode executing code for your program (memory allocation, page faults, file opening)

The `time` command

- You can time a program's complete execution by running it with the **`time`** command
  - It will give the real time taken, user time, and system time
- Let's say you've got a program called **`timewaster`**
  - Run it like this:

```
time ./timewaster
```

  - Output might be:

```
real 0m4.84s
user 0m1.030s
sys 0m3.43s
```

File I/O

Files

- Think of a file as a stream of bytes
- It is possible to read from the stream
- It is possible to write to the stream
- It is even possible to do both
- Central to the idea of a stream is also a file stream pointer, which keeps track of where in the stream you are
- We have been redirecting **`stdin`** from and **`stdout`** to files, but we can access them directly as well

fopen()

- To open a file, call the **fopen()** function
- It returns a pointer to a **FILE** object
- Its first argument is the path to the file as a null-terminated string
- Its second argument is another string that says how it's being opened (for reading, writing, etc.)

```
FILE* file = fopen("data.txt", "r");
```

`fopen()` arguments

- The following are legal arguments for the second string

Argument	Meaning
`"r"`	Open for reading. The file must exist.
`"w"`	Open for writing. If the file exists, all its contents will be erased.
`"a"`	Open for appending. Write all data to the end of the file, preserving anything that is already there.
`"r+"`	Open a file for reading and writing, but it must exist.
`"w+"`	Open a file for reading and writing, but if it exists, its contents will be erased.
`"a+"`	Open a file for reading and writing, but all writing is done to the end of the file.

fprintf()

- Once you've got a file open, write to it using **fprintf()** the same way you write to the screen with **printf()**
- The first argument is the file pointer
- The second is the format string
- The third and subsequent arguments are the values

```
FILE* file = fopen("output.dat", "w");
fprintf(file, "Yo! I got %d on it!\n", 5);
```

fscanf()

- Once you've got a file open, read from it using **fscanf()** the same way you read from keyboard with **scanf()**
- The first argument is the file pointer
- The second is the format string
- The third and subsequent arguments are pointers to the values you want to read into

```
FILE* file = fopen("input.dat", "r");
int value = 0;
fscanf(file, "%d", &value);
```

Closing files

- When you're doing using a file, close the file pointer using the **`fclose()`** function
- It's a good idea to close them as soon as you don't need them anymore
  - It takes up system resources
  - You can only have a limited number of files open at once
  - You can't always open a file in one program when it's open in another
  - Data might not be written to a file unless you explicitly close it

```
FILE* file = fopen("input.dat", "r");
int value = 0;
fscanf(file, "%d", &value);
fclose(file);
```

Example 1

- Write a program that prompts the user for an integer ***n*** and a file name
- Open the file for writing
- Write the value ***n*** on the first line of the file
- Then, print ***n*** random numbers, each on its own line
- Close the file

Example 2

- Write a program that reads the file generated in the previous example and finds the average of the numbers
- Open the file for reading
- Read the value ***n*** so you know how many numbers to read
- Read the ***n*** random numbers
- Compute the average and print it out
- Close the file

fputc() and putc()

- If you need to do character by character output, you can use **fputc()**
- The first argument is the file pointer
- The second is the character to output
- **putc()** is an equivalent function

```
FILE* file = fopen("output.dat", "w");
for(int i = 0; i < 100; ++i)
    fputc(file, '$');
```

fgetc() and getc()

- If you need to do character by character input, you can use **fgetc()**
- The argument is the file pointer
- It returns the character value or **EOF** if there's nothing left in the file
- **getc()** is an equivalent function

```c
FILE* file = fopen("input.dat", "r");
int count = 0;

while( fgetc(file) != EOF )
    ++count;

printf("There are %d characters in the file\n", count);
```

Ticket Out the Door

Upcoming

Next time…

- Users and groups
- Binary files
- Low-level file I/O

Reminders

- Keep working on Project 5
- Read LPI Chapters 4 and 5