ECE 220

Lecture x000E - 03/05

Slides based on material originally by: Yuting Chen & Thomas Moon

Recap

- Last week we discussed:
  - Recursion
  - Recursion & runtime stack
  - C to LC3
  - Recursion with backtracking
  - Good vs. bad recursion
- Some problems we discussed:
  - Recursive binary search
  - Towers of Hanoi
  - Exiting a maze
  - N-queens problem
  - Etc.

Today

- Discussion on *memoization*
- Deeper discussion on I/O in C
  - I/O with peripherals (keyboard & console)
  - I/O with files
- Exercises

Good recursion vs. bad recursion

- Consider the recursive Fibonacci function from last time.

```
long long fib(long long n){
     long long sum;

     if (n == 0 || n == 1)
          return 1;
     else {
          sum = (fib(n-1) + fib(n-2));
          return sum;
          }
}
```

- Let's do an activity
- Convert this function to an iterative version.
- Compare run times.

Memoization

- Can we keep the recursive formulation but somehow not repeat calculations/recursive calls?
- Key idea: Once we calculate a value, let us *cache* it for future use in a lookup “table” (actually array).

Some concepts

- Concept of a *stream*
  - A sequence of bytes made available *over time*
  - An *abstraction* made to deal with objects/data whose size cannot be known beforehand & contents may not be *all* available
- Different from arrays:
  - Arrays are finite in size, elements can be accessed in any order
  - Streams are potentially infinite; we only have access to the data seen till current time.

Ever thought where does the word *streaming* come from?

Streams for I/O

- A *text stream* is for example:
  - the sequence of ASCII characters printed to the monitor by a single program
  - the sequence of ASCII characters entered by the user during a single program
  - the sequence of ASCII characters in a single file
- We can only access the the characters in the order they are provided

Streams for I/O: standard streams

- C has three standard streams available: `stdin`, `stdout`, `stderr`.
    - `stdin` maps from the keyboard to the program via the input *buffer*.
    - `stdout` and `stderr` maps from the program to the console via the output *buffer*.
- **Buffer:** an implementation of the **queue** abstract datatype to decouple the *producer* from the *consumer* - FIFO data structure.

Buffers

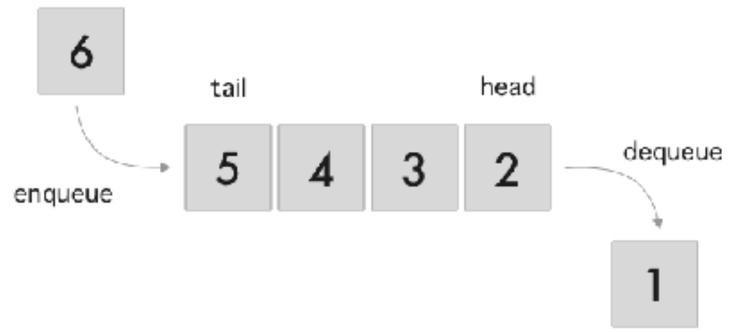
Yes *buffering* too!

- Why queue/buffer?
  - Correcting input
  - Collecting output
  - Streaming videos

- *Flushing* or *releasing* a buffer causes its contents to be released into its respective stream.

- Input buffer is released when the user presses the enter or return key (↵).

- Output buffer is released when the program submits a newline character “`\n`”.



Example

- What is the input for?
  - ABCD↵

```
char in1, in2, in3;
in1 = getchar();
in2 = getchar();
in3 = getchar();
printf("result:\n");
printf("%c", in1);
printf("%c", in2);
printf("%c", in3);
```

- `getchar()` reads one ASCII character from the keyboard.
- Equivalent to the IN TRAP routine in LC3.

What if you type in: A↵, B↵, C↵?

Example

- What is the expected output for the following snippet of code?

```
int main(){
  putchar('a');
  sleep(5);
  putchar('b');
  putchar('\n');
}
```

- `putchar()` displays one ASCII character the console.
- Equivalent to the OUT TRAP routine in LC3.

- What about?

```
int main(){
  putchar('a');
  sleep(5);
  putchar('b');
  putchar('\n');
}
```

`stdout` vs. `stderr` ?

- Normal program output is conventionally directed to `stdout` while warnings and errors are directed to `stderr`

- On *nix systems we can separate the output of the program using redirection.

Program text

```
fprintf(stdout, "Normal output1\n");
fprintf(stderr, "Error1 \n");
fprintf(stdout, "Normal output3\n");
fprintf(stderr, "Warning1\n");
```

Invocation

```
./a.out >a.log 2>err.log
```

Typical I/O functions

- `getchar`: Reads an ASCII character from the **keyboard**
- `putchar`: Writes an ASCII character to the **monitor**
- `fgetc`: Reads an ASCII character from ***stream***
- `fputc`: Writes an ASCII character to ***stream***
- `fgets`: Reads a string (line) from ***stream***
- `fputs`: Writes a string (line) to ***stream***
- `fscanf`: Read formatted string (line) from ***stream***
- `fprintf`: Write formatted string (line) to ***stream***

File based I/O

- To read or write to files in C we open and close *file streams* using the functions `fopen` and `fclose`.

- A file is a sequence of ASCII characters (or binary) stored in some storage device.

- To read or write a file, we declare a `FILE` pointer

  - `FILE` is a standard type defined in the `stdio.h`

```
FILE *infile;
infile = fopen("myfile.txt", "w")
```

Opening files

```
FILE* fopen(char* filename, char* mode)
```

`mode` is one of "`r`" (read), "`w`" (write) or "`a`" (append).

`fopen` returns a NULL pointer (failed to open file) or a pointer to the file stream.

`filename` is a string that is a valid filename on the operating system.

Reading & writing files

- To read/write to files one must:
  - Open the file in the correct mode - `fopen`
  - Do writing/reading (e.g: `fputs`, `fgets`, etc.)
  - Close the file - `fclose`

```
int fclose(FILE *stream);
```

Returns 0 (success) or EOF (failure)

EOF is a macro standing for End-Of-File… commonly represented as -1.

```
int feof(FILE *stream)
```

Will return nonzero value if reached end of a file stream.

Exercise

- Here is the syntax for `fputc` and `fgetc`. Using these write a program that takes a file `lower.txt` and converts its contents to uppercase in `upper.txt`.

```
int fgetc(FILE* stream)
int fputc(int character, FILE* stream)
```

Note*:* Both indicate *success* (character read/written) or *failure* (EOF) in their return values.

I/O one line at a time

```
char* fgets(char* string, int num, FILE* stream)
```

- Parameters
  - `string`: Pointer to a destination array
  - `num`: Max # of char to be copied into `string`
  - `stream`: Input stream
- Return value: NULL (failure) or pointer to `string` (success).

I/O one line at a time

```
int fputs(const char* string, FILE* stream)
```

- Parameters
  - `string`: Pointer to a source array
  - `stream`: Output stream
- Return value: Success (non-negative value) or failure (EOF).

Exercise

- Write a function that will prompt the user for a name and a description N number of times.
  - The name will be a maximum of 20 `chars` long
  - The description will be a maximum of 100 `chars` long
- Write out each name and description to a file (one after the other).

Formatted I/O

```
int fprintf(FILE* stream, const char* format, …)
```

- Parameters:
  - `stream`: Output stream
  - `format`: String that specifies the formatting details
  - *Additional arguments*: variables to replace a format specifiers
- Return value: Success (number of characters written), Failure (negative number)

Formatted I/O

```
int fscanf(FILE* stream, const char* format, …)
```

- Parameters:
  - `stream`: Input stream
  - `format`: String that specifies the formatting details
  - *Additional arguments*: pointers to store data that is read in
- Return value: Success (number of items read), Failure (EOF).

Variable argument lists

- Note that `fprintf` and `fscanf` accepted a ***variable*** number of arguments (depending on format specifier).

- How does this work on the run time stack?

- Recall arguments are pushed *right-to-left*.
  - Last argument pushed will always be format specifier
  - Sufficient to examine format specifier to know number of parameters.

If you wondered why … well now you know!

Dynamic memory allocation

- In the exercise prompting the user for a name and description we had to set the size of the array at compile time.

- Can we make the decision on the size of the data (i.e. memory it is going to occupy) *dynamically* at **run-time?**

- **This lead to two important functions: `malloc` and `free`**

Dynamic memory allocation

```
void *malloc(size_t size)
```

- Parameters
  - `size`: Number of bytes to allocate
  - `size_t`: A *type* defined in the user library ~ unsigned integer
- Return value: NULL (failure) or pointer to beginning of allocated block (success).

https://en.cppreference.com/w/c/memory/malloc

Dynamic memory allocation

```
void free(void *ptr)
```

- Parameters
  - `*ptr`: Pointer to beginning of block to be *deallocated*.
- Return value: `void`
- Memory allocated via `malloc` ***must*** be deallocated via `free` or reallocated via `realloc` to prevent memory leaks!

https://en.cppreference.com/w/c/memory/free **https://en.cppreference.com/w/c/memory/realloc**

Can we do this using dynamic memory allocation?

Exercise

- Write a function that will prompt the user for a name and a description N number of times.
  - The name will be a maximum of 20 `chars` long
  - The description will be a maximum of 100 `chars` long
- Write out each name and description to a file (one after the other).

Next time

- More on dynamically allocating memory
  - `malloc()`
  - `free()`
- Structures (combining data types a.k.a `structs`)