# ECE264: Advanced C Programming

Summer 2019

Week 4: Recursion (contd..), File handling

# Recursion – a real life example

"This is an increasingly common occurrence in our political **discourse**."
*Washington Post* Jun 25, 2019

**discourse:**
a formal discussion of a subject in speech or writing, as a dissertation, **treatise**, sermon, etc.

**treatise:**
a formal and systematic **exposition** in writing of the principles of a subject, generally longer andmore detailed than an essay.

**exposition:**
the act of **expounding**, setting forth, or explaining.

**expound:**
To set forth or state in detail

```
void LookUpDictionary(string n) {
      array<string> retVal = GetMeaning(n)
      foreach element in retVal:
            if meaning of element is known
                  continue;
            else
                  LookUpDictionary(element);
}
```

# Example - Factorial

- n! = n x (n-1) x (n-2) x . . . x 3 x 2 x 1

  (n–1)! = (n-1) x (n-2) x . . . x 3 x 2 x 1

  therefore,

  n! = n x (n-1)!

  *is this complete?*

- plug 0 to n and the equation breaks.

therefore,

$$n! = \begin{cases} n \times (n-1)! & \text{when } n \geq 1 \\ 1 & \text{when } n=0 \text{ // factorial of negative numbers not defined.} \end{cases}$$

# Example - Factorial

$$n! = \begin{cases} n \times (n-1)! & \text{when } n \geq 1 \\ 1 & \text{when } n = 0 \end{cases}$$

// factorial of negative numbers not defined.

```
int factorial(int n) {
      if(n >=1)
            return n * factorial(n-1);
      else
            return 1;
}
```

# Example - Factorial

```
int factorial(int n) {
      if(n == 0)
            return 1;
      else
            return n * factorial(n-1);
}
```

# Exercise

```
1 int ex1(char* str)
2 {
3     if(*str == ‘\0’)
4           return 0;
5     else
6           return 1 + ex1(str+1);
7 }
```

*what does the function ex1 do?*

# Using gdb to understand recursion

- Demo

```
#include<stdio.h>
int foo(int n)
{
	int retval = n;
	if (n == 0)
		return 1;
	retval = retval * foo(n-1);
	return retval;
}

int main()
{
	int x = foo(5);
	printf("foo(5)=%d\n",x);
}
```

# Exercise

- What happens in memory when recursion never terminates?

# Tail Recursion

```
void printStars(int n) {
      if(n ==1)
            return;
      printf(“*”);
      printStars(n-1);
}
```

- Recursive call is the last statement in the function

# Optimizing Tail Recursion

```
void printStars(int n) {
start: if(n ==1)
           return;
       printf(“*”);
       n=n-1;
       goto start;
}
```

- Recursive call replaced by goto statement

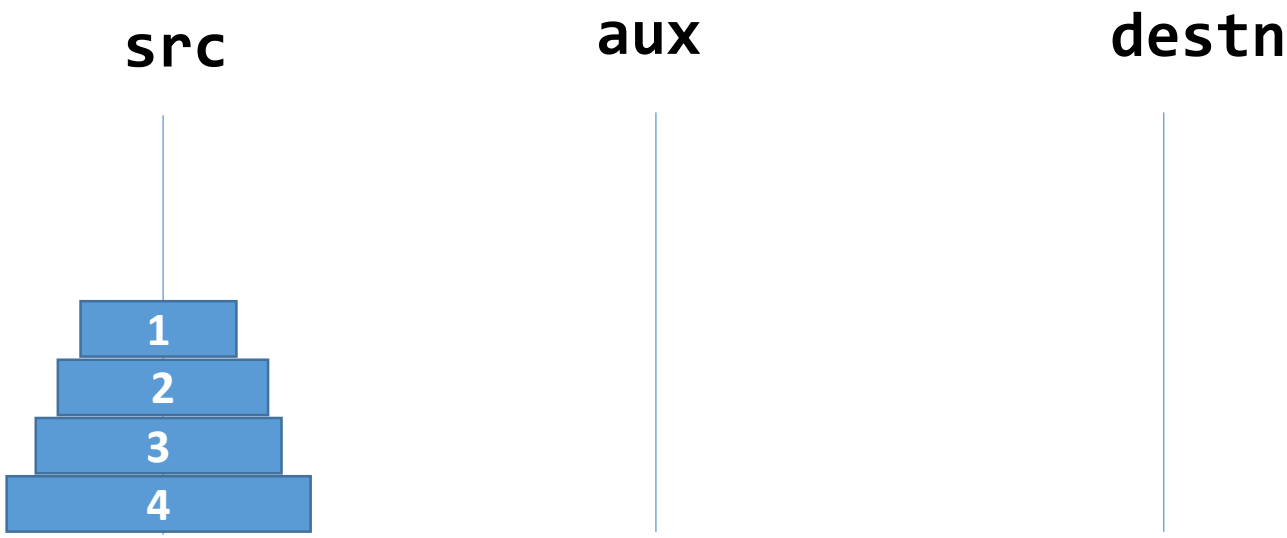
# Divide-and-conquer – a common recursive pattern

- A problem can be broken into two or more smaller problems of similar or related type

  Array sum – a toy example

  Quicksort, Mergesort – realistic examples

# Tower of Hanoi




1. Move (n-1) disks from `src` to `aux` (using `destn`)
2. Move disk n from `src` to `destn`
3. Move (n-1) disks from `aux` to `destn` (using `src`)

# Tower of Hanoi – recursive code skeleton

```
void TOH(int n, Rod src, Rod destn, Rod aux) {


      TOH(n-1, src, aux, destn);

      print(“Move disk n from rod <src> to <aux>”);

      TOH(n-1, aux, destn, srcdestn);
}
```

# Tower of Hanoi – recursive code base case

```
void TOH(int n, Rod src, Rod destn, Rod aux) {


      TOH(n-1, src, aux, destn);

      print("Move disk n from rod <src> to <aux>");

      TOH(n-1, aux, destn, srcdestn);
}
```

- **if n = 0**

  ***no work to do!***

# Tower of Hanoi – recursive code base case

```
void TOH(int n, Rod src, Rod destn, Rod aux) {
      if(n == 0)
            return;
      TOH(n-1, src, aux, destn);

      print("Move disk n from rod <src> to <aux>");

      TOH(n-1, aux, destn, srcdestn);
}
```

# Tower of Hanoi – analysis

How many steps (`print` statements) do we need to move `n` disks from `src` to `destn` ?

```
void TOH(int n, Rod src, Rod destn, Rod aux) {
      if(n == 0)
            return;
      TOH(n-1, src, aux, destn);

      print("Move disk n from rod <src> to <aux>");

      TOH(n-1, aux, destn, srcdestn);
}
```

# Tower of Hanoi – analysis

```
void TOH(int n, Rod src, Rod destn, Rod aux) {
      if(n == 0)
            return;
      TOH(n-1, src, aux, destn);

      print("Move disk n from rod <src> to <aux>");

      TOH(n-1, aux, destn, srcdestn);
}
```

# Application Programming Interface (API)

- APIs are well defined methods with certain behavior
- Using APIs you can interact with the system in various ways
- Usually a prescription. Not an implementation
  - POSIX (portable operating system interface) for variants of Unix and other OS.
    - The ‘terminal’ program on MAC and Linux are compatible.
    - e.g. manipulating files, manage communication between programs (inter process communication / IPC)-sockets, signals, etc.

# File API for file manipulation

- Goal: read and write files
- Bulk of `stdio.h`
- Can be accessed and manipulated *only* through pointers of type `FILE*`
- `FILE` type is a structure containing members for indicating (among others):
  - position within the file, mode (text/binary)

  error, end-of-file etc.

# File pointers (`FILE *`)

- Necessary to interact with File APIs
- A `FILE` object's (also called stream) address cannot be used with APIs:

```
#include<stdio.h>
void foo() {
      FILE* fp1 = fopen(…);
      FILE fp2 = *fp1;
      fprintf(&fp2, …);
}
```

- `FILE` pointers are not used for accessing and manipulating just files.
- Each stream (`FILE` object) associated with external physical device (file, keyboard, display, printer, serial port, etc.)

# File access API (`fopen`)

- `FILE* fopen(const char* filename, const char* mode)`
  - `filename` can contain a path to a file (relative and absolute pathname)
  - `mode` can be “r”, “w”, “a”, or the previous with an extension “+”: “r+”, ”w+”, ”a+”; `mode` can also be “b” (binary)
  - Returns valid `FILE` pointer on success and `NULL` on failure. Also, error code is set to 0 on success and a negative number on failure.

- Example

```
#include<stdio.h>
int main() {
  char* fileName1 = “tmp1.txt”;
  FILE* fp1 = fopen(fileName1,”r”);
  FILE* fp2 = fopen(“tmp2.txt”,”w”);
  FILE* fp3 = fopen(“tmp3.txt”,”w+”);
  ...
}
```

- Checking return value

```
#include<stdio.h>
int main() {
  FILE* fp = fopen(“tmp1.txt”,”r”);
  if(fp == NULL) {
      perror(“There was an error”);
      return EXIT_FAILURE;
  }
  ...
}
```

# File access API (`fclose`)

- `int fclose(FILE* fp)`
  - Returns 0 on success, EOF on failure
  - EOF is a special integer with value -1

# Detour - Error Handling

*Important to check for errors*

- errno, perror, strerror, ferror, feof
  1. errno: variable of int type. Defined in errno.h. Set by system calls when any error occurs. Never set to zero
  2. perror("Oops");
  3. strerror(errno) //string meaning of errno
     - printf("Oops %s",strerror(errno));
  4. ferror(fp) //checks the error indicator in the FILE object
     - if(ferror(fp)) { printf("Oops") };
  5. feof(fp) //checks the end of file indicator in the FILE object

# Formatted input and output

- `fprintf, fscanf`

testinput1 (a text file)

```
ID      FirstName LastName
179004  Zara      KRAUSE
373672  Bradley   MARKS
399365  Kannon    HOOD
```

# Formatted input and output

- `fprintf, fscanf`
  - `int fscanf(FILE* fp,`
    `const char* format, ...);`

//On success, returns the number of values assigned

  - `int fprintf(FILE* fp,`
    `const char* format, ...);`

//On success, returns the number of bytes written to fp

# Special streams

- stdin, stdout, stderr
  - stdin: input (keyboard)
  - stdout: output (terminal / display)
  - stderr: error

# Unformatted input and output

- `fputc, fgetc, fgets, fputs`
  - `int fgetc(FILE* fp)` //reads the next char from input stream fp
  - `int fputc(int c, FILE* fp)` //writes the char c into the current position in output stream fp
  - `char* fgets(char* str, int count, FILE* fp)` //reads a string of length count-1 bytes from stream fp, writes into the array str
  - `int fputs(const char* str, FILE* fp)` //writes every char in array str (except the '\0' char) to fp.

# Direct input and output

- `fwrite and fread`
  - `int fread(void* buffer, size_t size, size_t count, FILE* fp);`

  //reads up to count objects of size and puts them in the array buffer.

  - `int fwrite(void* buffer, size_t size, size_t count, FILE* fp);`

  //writes up to count objects of size and puts them in the array buffer.

  On success, both return the number of objects read/written

# File positioning

- `fseek, ftell`
  - `int fseek(FILE* fp, long offset, int origin)` `origin` indicates position indicators:
    - `SEEK_SET, SEEK_END, SEEK_CUR`
  - `long ftell(FILE* fp)` //on success, returns the current position indicator in stream fp

# To know more about `FILE` API

- type on the command prompt (‘terminal’):
  `man <API>`
- Type ‘q’ to quit once done seeing the manual (man) pages