STANDARD LIBRARY FUNCTIONS

CS 23200

Big Picture

- ☑ Developing programs on *nix computers
- ☐ C Language
  - ☑ Familiar aspects of C (variables, operators, basic I/O, control flow, functions)
  - ☑ Pointers
  - ☑ Structures and related constructs
  - ☑ File operations
  - ☑ Multi-file programs
  - ☐ Standard library functions
- ☐ *nix tools

Standard Library

- Functions that are available and consistent across systems
- Useful for a variety of tasks
  - File operations
  - String manipulation
  - Date/time functions
  - Math functions
  - ...
- Seen several of these functions before
  - fprintf, strlen, strncpy
- File operations are a big part of the standard library that we already covered

The Goal

- A good knowledge of the standard library comes mainly from (lots of) practice
  - More than can be done in a semester
- Our goal: give you an idea of what types of functions are available
  - Not to cover the whole standard library
  - Can look up available functions in Appendix B of K&R
  - Online reference: http://www.cplusplus.com/reference/clibrary/
    - The C standard library is a subset of what is available in C++

Outline

- Error handling
- Standard library categories
  - String manipulation
  - Date/time functions
  - Math functions

Error Handling

- Most (if not all) standard library functions have some way of indicating an error occurred
  - Look at documentation for details
  - Return value (e.g., NULL)
  - Setting `errno`
    - A global variable for error number
    - Use `perror(const char *s)` to print `s: error message\n` to stderr

Error Handling

- Should always check for errors
- If error, then what?
  - No exception mechanism in C
  - Can return an error code from your own function
  - Might print a message to stderr before returning
  - Just crashing is NEVER acceptable
- Error-checking makes your program robust to …
  - Security attacks
  - Hardware failures
  - Uninformed users (which is typically all of them :-))
  - Programming errors

Outline

- Error handling
- Standard library categories
  - **String manipulation**
  - Date/time functions
  - Math functions

String Manipulation

- sscanf
- snprintf
- Use examples to illustrate some other string functions and manipulation tricks

sscanf Example

- A file contains
  - Number of rows
  - The actual data (3 ints in each row)
- Goal: sum the columns
- Could use **f**scanf
  - Loop over lines

```
res = fscanf(fp, "%d %d %d", ...);
```

  - Cannot tell if a line is too short
    - fscanf will just grab ints from the next line

```
#define NUMCOLS 3
...

int sums[NUMCOLS] = {0,0,0};
const int bufSize = 200;
char line[bufSize];
int i, j, res, n;
res = fscanf(fp, "%d\n", &n);
if(res != 1){ ... }
for(i=0; i<n; i++){
   int vals[NUMCOLS];
   fgets(line, bufSize, fp);
   res = sscanf(line, "%d %d %d\n", vals, vals+1, vals+2);
   if(res != 3)
      fprintf(stderr, "Line %d was too short.\n", i+1);
   else{
      for(j=0; j<NUMCOLS; j++)
         sums[j] += vals[j];
   }
}
printf("%d %d %d\n", sums[0], sums[1], sums[2]);
```

snprintf

- snprintf: convert numerical types into strings
  - Like printf
- To store in a buffer, use snprintf

```
int snprintf(char* buffer, size_t bufSize,
             const char* format, ...);
```

- Works just like fprintf except the output goes to buffer instead of a file

Returning to Complete a Prior Example

- The file "data.txt" contains the following information:
  - First line: the number of rows that follow
  - Each subsequent line: a list of whitespace-delimited integers
    - Each list can be a different length
- Read in the data and store each row in its own array of ints (for later processing)

The Steps

- Open the file and verify the file pointer
- Read the number of lines
- Create an array with one element for each line
- Loop over the lines
  - Read in the line
  - Determine how many ints there are
  - Allocate space for those ints
  - Read the ints into that space

  } These were postponed until now
- Close the file

Completing the Function

- The steps:
  - Determine how many ints there are
  - Allocate space for those ints
  - Read the ints into that space
- What might our function declaration be?

```
int* createIntArray(char* line, int* pNumInts);
```

  - Creates an array of ints from the string
  - Returns a pointer to the array
  - Also sets *pNumInts to be the length of the array

Completing the Function

- The steps:
  - Determine how many ints there are
  - Allocate space for those ints
  - Read the ints into that space

```
int* createIntArray(const char* line, int* pNumInts);
```

Completing the Function

```
int* createIntArray(char* line, int* pNumInts);
```

- When processing data from the user, think about all the ways they could break things!
  - Initial delimiters
  - Consecutive delimiters
  - Negative numbers
  - Characters that are not numbers
- Now we write pseudocode for the steps...

	3	2		-	4	9	3			\t	4	\0

Completing the Function

- Count the number of ints
  - Point to beginning of line
  - Set count to 0
  - While there is more string to process:
    - Check for an int
    - Move the pointer past the int
- Allocate space
  - Call malloc
  - Check for NULL
- Fill in the ints
  - Point to beginning of line
  - For 1 to number of ints:
    - Read the int
    - Move the pointer past the int

An Aside: Using %n with sscanf

- Using sscanf with %n
  - A way to tell how many characters have been processed (including leading whitespace)
  - Does not read anything from the string
  - Does not count toward the return value (i.e., number of parsed fields)
- Example:

```
char myString[] = "-389 4 186854";

char* pNext = myString;
int theNumber, charsRead, res;
while(1){
   res = sscanf(pNext, "%d%n", &theNumber, &charsRead);
   if(res != 1){
      break;
   }
   pNext = pNext + charsRead; /* advance the pointer */
}
```

```
int* createIntArray(const char* line, int* pNumInts){
  const char *pChar;
  int res, val, charsRead;
  /** count the ints **/
  *pNumInts = 0;
  pChar = line;
  /* While there is more string to process */
  while(1){
    /* Check for an int */
    res = sscanf(pChar, "%d%n", &val, &charsRead);
    if(res == 1){
      (*pNumInts)++;
      /* Move the pointer past the int */
      pChar = pChar + charsRead;
    }
    else{
      break;
    }
  }
```

```
int* createIntArray(const char* line, int* pNumInts){
   ...

   int* arr = NULL;

   /** allocate the space **/
   arr = malloc(sizeof(int) * (*pNumInts));
   if(arr == NULL)
      return NULL;
```

```
int* createIntArray(const char* line, int* pNumInts){
  ...
  /** convert the tokens to ints, filling arr **/
  int i;
  pChar = line;
  /* for each integer */
  for(i=0; i<*pNumInts; i++){
    /* read the int */
    sscanf(pChar, "%d%n", &(arr[i]), &charsRead);
    /* move the pointer past the int */
    pChar = pChar + charsRead;
  }
  return arr;
}
```

- Using sscanf is more robust (and more concise) for this example than parsing character-by-character
- Other cases are better without sscanf (example upcoming)

Complete the following function

- str : string with integers separated by single spaces
- Should return a pointer to the start of the first number larger than maxVal, or NULL if no such number exists

```
const char* findLarge(const char* str, const int maxVal){
   int charsRead, val;
   const char *next = str;

   while( sscanf(next, "%d%n", &val, &charsRead) == 1 ){
      if(val > maxVal){
         /* return a pointer to this number */
         if(*next == ' '){ next++; }
         return next;
      }
      next = next + charsRead;
   }
   return NULL;
}
```

A String Example

- Goal: copy source file to destination file, replacing search string with replacement string
- Program takes 4 command-line arguments
  - source file name
  - destination file name
  - search string
  - replacement string
- First step: Break it down into pieces

A String Example

- Verify the command-line arguments
  - Error checking is CRITICAL any time you are dealing with information coming from outside the program
    - Command-line arguments, standard input, files, etc.
- Open the files
- Loop over the lines
  - Read a line from the source
  - Write the line to destination
    - Includes the search and replace functionality
- Close the files

Verify the Arguments

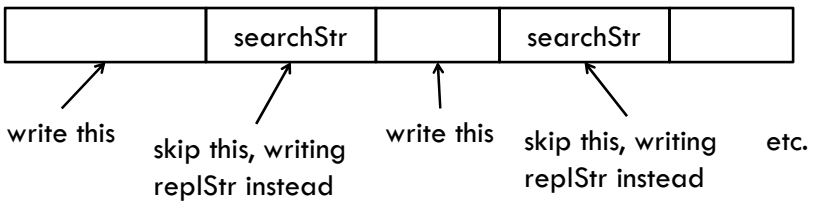
```
enum errorCodes {ERR_INVALID_ARGS=1};
int main(int argc, char* argv[]){
   /* provide names for readability */
   const char *sourceFile, *destFile, *searchStr, *replStr;

   if(argc != 5){
      fprintf(stderr, "USAGE: sourceFile destFile "
               "searchStr replacementStr\n");
      return ERR_INVALID_ARGS;
   }

   sourceFile = argv[1];
   destFile = argv[2];
   searchStr = argv[3];
   replStr = argv[4];
```

Open the Files

```
FILE *fpIn, *fpOut;
fpIn = fopen(sourceFile, "r");
if(fpIn == NULL){
   fprintf(stderr, "Could not open source file (%s).\n",
            sourceFile);
   return ERR_FILE_OPEN;
}
fpOut = fopen(destFile, "w");
if(fpOut == NULL){
   fprintf(stderr, "Could not open destination (%s).\n",
            destFile);
   return ERR_FILE_OPEN;
}
```

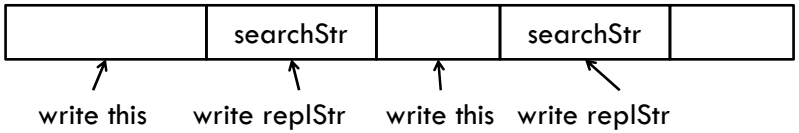
Loop Over the Lines

- What will the flow look like?
  - Read in a line
  - Then what?
    - Might suggest replacing search string in the buffer
      - Messy, computationally expensive
    - Instead, just do replacement when we write the file



Read in a Line

```
const int maxLength = 10000;
char *line = malloc(sizeof(char) * maxLength);
if(line == NULL){
   fprintf(stderr, "Could not allocate memory.\n");
   return ERR_MALLOC;
}

while(fgets(line, maxLength, fpIn) != NULL){
   int lineLen;
   lineLen = strlen(line);
   if(lineLen > 0){
      if(line[lineLen-1] != '\n'){
         fprintf(stderr, "WARNING: line is too long or the"
               " file does not end with a blank line.\n");
         line[lineLen-1] = '\n';
      }
```




```
...
char *pNextSpot, *pNextSearchStr;
pNextSpot = line;
do{
   pNextSearchStr = strstr(pNextSpot, searchStr);
   if(pNextSearchStr == NULL)
      fputs(pNextSpot, fpOut);
   else{
      /* we can overwrite the first character of
         pNextSearchStr because we don't need it */
      *pNextSearchStr = '\0';
      fputs(pNextSpot, fpOut);
      fputs(replStr, fpOut);

      pNextSpot = pNextSearchStr + strlen(searchStr);
   }
} while(pNextSearchStr != NULL);
```

An Exercise

- Write a function

```
int countOccurrences(const char* toSearch,
                     const char* target);
```

that returns the number of (non-overlapping) occurrences of target in toSearch

```
const int len = strlen(target);
const char *pNext = toSearch;
int count = 0;
do{
   pNext = strstr(pNext, target);
   if(pNext != NULL){
      count++;
      pNext += len;
   }
} while(pNext != NULL);
return count;
```

Other Notable String Functions

- is???(c) indicates if c is in a given category
  - isdigit(c), isspace(c), isalpha(c)
- strncpy: copy strings (seen before)
- int strcmp(const char* str1, const char* str2);
  - Compares strings, returning
    - <0 if str1 < str2
    - 0 if str1 equals str2
    - >0 if str1 > str2

An Exercise

- Write a function that indicates if an array of strings is sorted or not

```
int isSorted(const char** strings, const int n){
   int i;
   for(i=0; i<n-1; i++){
      if(strcmp(strings[i], strings[i+1]) > 0){
         return 0;
      }
   }
   return 1;
}
```

Outline

- Error handling
- Standard library categories
  - String manipulation
  - **Date/time functions**
  - Math functions

Date and Time

- Dates and times are surprisingly complex
  - Time zones
  - Daylight savings time
  - Leap years
  - Leap seconds
  - Many, many text formats for dates and times
  - See http://dx.doi.org/10.1145/1941487.1941505 for an interesting article about leap seconds

Representing Date and Time

- Resolution is in seconds
  - Use clock(void) function for finer resolution
- Two formats:
  - `time_t` : number of seconds
  - `struct tm{`

```
int tm_sec;  /* in [0, 61] */
int tm_min;  /* in [0, 59] */
int tm_hour; /* in [0, 23] */
int tm_mday; /* day of month; in [1, 31] */
int tm_mon;  /* in [0, 11] */
int tm_year; /* years since 1900 */
int tm_wday; /* days since Sunday; in [0,6] */
int tm_yday; /* days since Jan 1; in [0,365] */
int tm_isdst;  /* Daylight Savings Time flag */
};
```

Date/Time Example

```
void printTimeUntil(const struct tm* pEvent){
   time_t currTimeT, eventTimeT;
   long seconds;
   int days, hours, minutes;

   eventTimeT = mktime(pEvent);
   currTimeT = time(NULL);
   seconds = (long)difftime(eventTimeT, currTimeT);

   days = seconds / (60 * 60 * 24);
   seconds %= 60 * 60 * 24;
   hours = seconds / (60 * 60);
   seconds %= 60 * 60;
   minutes = seconds / 60;
   seconds %= 60;
   printf("%d days, %d:%02d:%02d to go.\n",
            days, hours, minutes, (int)seconds);
}
```

Another Example

```
void printTime(){
   const int bufSize = 300;
   char buf[bufSize];
   struct tm utcTime, localTime;
   time_t currentTime = time(NULL);

   utcTime = *(gmtime(&currentTime));
   localTime = *(localtime(&currentTime));

   strftime(buf, bufSize, "%B %d %Y  %I:%M %p (%Z) ",
         &localTime);
   printf(buf);
   strftime(buf, bufSize, " equals %I:%M %p (%Z)\n",
         &utcTime);
   printf(buf);
}
```

Date/Time Notes

- gmtime and localtime return pointers to a static structure
  - Will be overwritten with each call
  - Not thread-safe!
- strftime has lots of format specifiers

An Exercise

- Write a function that takes an array of struct tm pointers and returns the index of the earliest event
- What are the steps?
  - Loop over all events
    - If the event is earlier than the earliest one seen so far, mark it as the earliest
- How do we figure out which of two events is earlier?
  - Can use < on time_t values (but not on struct tm's)

```
int findEarliest(const struct tm **events,
                 const int numEvents){

   time_t earliestSoFar, tempTime;
   int earliestIndex, i;

   if(numEvents < 1)
      return -1;

   earliestIndex = 0;
   earliestSoFar = mktime(events[0]);
   for(i=1; i<numEvents; i++){
      tempTime = mktime(events[i]);
      if(tempTime < earliestSoFar){
         earliestIndex = i;
         earliestSoFar = tempTime;
      }
   }
   return earliestIndex;
|
```

Outline

- Error handling
- Standard library categories
  - String manipulation
  - Date/time functions
  - **Math functions**

Math Functions

- sin, cos, log, log10, exp, sqrt, ceil, floor, fabs
- Need to #include <math.h> and link with the math library (add a `-lm` at the end of the command line)
- Some math-like functions are in stdlib.h
  - abs (for integers)
  - int rand(void)
    - Returns a pseudo-random integer in the range 0 to RAND_MAX
    - First seed the random-number generator once in main: srand(time(NULL));

Miscellanea

- int system(const char* s)
  - Executes the command s in the shell
  - This will make your code system-dependent!
  - <stdlib.h>
- limits.h
  - Contains constants for limits of integral types
  - INT_MAX, INT_MIN, LONG_MAX, LONG_MIN, etc.
- float.h
  - Contains constants for limits of floating point types (among other things)
  - Note: FLT_MIN is the smallest MAGNITUDE float (e.g., 1E-37)
    - The smallest value float is -1.0 * FLT_MAX