Software Development (CSCI2132)
Covers C

Lecture Dates: March 9th, 2012 to April 4th, 2012

- Pointers and Arrays
  - ***Array Name***: A pointer to the 0th element.
    - For example: int a[10];
      - int *p = &a[0]; is equivalent to
      - int *p = a;
      - Or: a[2] = 4 is equivalent to *(a+2) = 4;
      - Or: *(p+3) = 5 is equivalent to p[3] = 5;
    - *Note*: Indirection (*) is a unary operator. Unary operators have higher precidence than binary operators.
    - Example program:

```
int i;
for (i = 0; i<10; i++)
        a[i] = 0;

int *p;
for (p = a; p<a+10; p++) // Same as ^. Can also do p<&a[10]
        *p = 0;
```

  - You cannot make an array name point to a different element. Can treat it as a constant pointer. For example: a++. Is illegal.
  - The incremental (++), decremental (--) operators have higher precedence than *. Sometimes have to use brackets, if using both.
  - ***Array Parameters*** can also be declared as pointers.
    - For example:

```
int max_array(int *a, int len);
equivalent to
int max_array(int a[], int len);
```

    - Remember the gdb command: print *p@10
- Pointer Arithmetic vs. Array Subscripting
  - Compilers translate both to equivalent machine instructions.
  - Previously, pointer arithmetic was faster.
  - ***Compiler optimization***: Not just simply translating code line-by-line. May try to generate code equivalent in effect. Both will be the same speed.
  - To turn this on in gcc: -O3
  - Example program (projector – mergesort again, mergesort2.c):

```
#include <stdio.h>
```

```
...
int main(void) {
        ...
        for (p = array; p < array+len; i++)
                scanf("%d",p);
        ...
                printf("%d",*p);
        ...
}
...
                *r++ = *q++;
        ...
        while(q < right + len_right)
        ...
...
```

- Strings
    - In programming languages, a sequence of characters.
    - In C, the Strings are related to pointers.
        - A sequence of characters, terminated by a null character ('\0').
        - Very little overhead.
        - But we need to know where the sequence ends; the point of the null character. A character of ASCII value 0.
        - The length of a String: number of characters except for the null character.
        - Strings are stored as:
            - a char array, or
            - string literals (constant, cannot be modified)
    - String Literals
        - Example: “hello, world\n”
        - Using double quotes and content of string. Can contain escape sequences.
        - When stored in memory, will be a sequence of characters (stored as 1 byte each). The last byte will be \0. Using n+1 characters to store the literal of length n.

Reading: 12.2 – 12.3, 13.1
Next: rest of Chapter 13.
Programming Project: Proj 2, p. 275. Written exercises posted.
Midterm II: Average 71% (adjusted).
Second coding question was largest issue (linear much less efficient).

- Can be assigned to char pointers. Would act as an array in memory. For example:

```
char *p = "hello, world\n"; // points to starting pos of literal
char ch = "hello, world\n"[0]; // ch = 'h'
ch = *p; // ch = 'h'
```

- String Variables
  - Extra byte for null character.
    - For example: #define STRLEN 80

```
char str[STRLEN+1]; // Actual len of string: 0 ~ 80.
```

    - Storing “abc” in str (tedious):

```
str[0] = 'a';
str[1] = 'b';
str[2] = 'c';
str[3] = '\0';
```

  - Initializing a string variable.
    - For example:

```
char str1[6] = “abc”; // not string literal
```

    - Storage of str1: {'a','b','c','\0','\0','\0'}
    - Can omit length of string if provide initializer, like with other arrays.

```
char str2[] = “abc”; // len = 4
```

- String I/Os
  - printf can be used for output: conversion specification is %s
    - For example:

```
printf(“%s\n”,str2);
```

  - There's also puts.
    - puts(str): prints str followed by '\n'
    - For example:

```
puts(str2); // equivalent to above.
```

  - scanf can be used for reading: conversion specification is %s
    - For example:

```
scanf(“%s”,str); // str is already a pointer
```

    - Can only be used to read a word (no white space).
    - So, if the user input: ___hello,_world and then pressed enter (_ = space), the scanf would skip whitespace characters until it sees something that is not a whitespace. Will stop reading after hello,
    - What if we want to read one line of text?

    - gets is a function that can be used to read a line.
        - Unsafe.
        - Does not check whether or not variable has enough space. Behavior would be undefined. gcc will even issue a warning.
    - In most cases, usually design our own input function. For instance, defines behavior if input exceeds storage.

```
int read_line(char str[], int n) {
        // length of string will be n+1
        // returns number of characters read

        int ch, i = 0;
        while ((ch=getchar()) != '\n') if (i < n) str[i++] = ch; else break;
        str[i] = '\0';
        return i;
}
```

- Library Functions
    - #include <string.h>
    - Copying a string s2 into s1, and then return s1. Prototype:

```
char* strcpy(char *s1, const char *s2); // const means you cannot change value
within function
```

        - A possible implementation:

```
char* strcpy(char *s1, const char *s2) {
        char *p = s1; // used as iterator
        while ((*p++ = *s2++) != '\0') ;
        return s1;
}
```

        - Do not need to implement it like this though; can just call it.
        - For example:

```
char *s1[5] = “abc”;
strcpy(s1, “defg”);
```

    - Appending s2 to s1, and return s1. Prototype:

```
char* strcat(char *s1, char *s2);
```

    - Comparing s1 and s2 (lexographically). Prototype:

```
int strcmp(const char *s1, const char *s2);
```

        - Returns one of three values:
            - Negative (<0): s1 < s2.
            - Zero (0): s1 == s2.
            - Positive (>0): s1 > s2.
        - For example:

```
strcmp(“large”,”little”); // result: <0
```

        - This is case-sensitive.
    - Computing the length of a string. Returns length with only parameter is the string. Return type: size_t. Prototype:

```
size_t strlen(const char *s1);
```

        - The size_t type is used to record the size of objects in memory.
        - This type is implementation-defined. On different platforms, range of memory size is different. Do not know in advance what maximum range of size of object in memory is.

Reading: 13.2 – 13.6, Next: 13.7 – 15
Practice Programming Proj: Proj 1, p 311.
Solutions for MII posted online (Avg: 71, Highest: 96%).

- Command-Line Arguments
    - Passed as an array of Strings.
        - Specify header of main as: int main (int arg**c**, char* arg**v**[])
        - The variable argc **counts** the number of arguments. The variable argv means argument **values** (array of char pointers; equivalent: char** argv).
    - For example: Program, after compiling, is called sortwords.
        - ./sortwords orangle apple banana
        - Three arguments.

argv | ./sortwords\0, orangle\0, apple\0, banana\0, NULL (points to nothing)

        - argv[0] refers to the path of the program. Remaining pointers point to one argument, except for the last (null pointer). The value of argc is 4.
        - For instance, argv[2][4] refers to the character *e*.
    - Implementing **sortwords.c**: simply sorts arguments provided lexiographically.
        - See code online.

```
int main(int argc, char* argv[]) {
        ...
        // uses insertion sort
                while (j >= 1 && strcmp(argv[j], key) > 0) {
```

```
                argv[j+1] = argv[j];
                j--;
            }
            argv[j+1] = key;
        ...
        for (i = 1; i < argc; i++)
        ...
    }
```

- Writing Large Programs (PPT)
    - Header Files – files that allow different source files (*.c) to share:
        - Function Prototypes
        - Type Definitions
        - Macro Definitions
        - ...
    - Naming Convention: *.h
    - The #include directive: Tells preprocessor to open a specified file, insert its content into current file. Forms: #include <file name> or #include "file name"
        - <file name> searches /usr/include (directory, directories where system header files reside).
        - "file name" searches in current directory first and then system header file directories.
    - Example: Dividing Program into Multiple Files – decimal2binary (/public/make)
        - Step 1: Break program logically into source files (*.c).
            - *decimal2binary.c*
            - *stack.c*
        - Step 2: Sharing...
            - bit.h: typedef int Bit; // Type Definitions, can be used by other programs.
            - Not needed as STACK_SIZE is used by stack.c only // Macro Definitions
            - stack.h // Function Prototypes
        - Step 3: Protecting Header Files
            - Issue: Nested header files. For example: #include "bit.h" in stack.h.
            - Solution: Protect each header file using conditional compilation.
            - For example, in bit.h: #ifndef BIT_H (if not defined)

```
#ifndef BIT_H
#define BIT_H
typedef int Bit;
#endif
```

    - The Make Utility
        - Manages compilation nad linking of multi-file software.
        - Reads a *makefile* (makefile or Makefile) that specifies:
            - Target to be built.
            - Commands used to build them.
            - How modules of a software system depend on each other (**key** part).
        - Idea of dependencies.

            - A directed, acyclic graph. Not a tree.
            - *Object file (*.o)*, a file containings machine instructions of one module (not executable). Typically generate one object file for each *.c file.
        - Using this dependency graph, can write down a makefile.
            - The -c flag orders gcc to *only* make the object file. Compiles without linking.
            - The -o flag orders gcc to *link* object files.
            - If we type make in a directory with a makefile, it will read this file and looks at the commands so it can execute them.
            - all, clean, etc. allows you to execute a command through make (i.e. make install, make all, make clean).
        - If a file is edited, only those that depend on it will be rebuilt. For example, if edit stack.c, only have to rebuild stack.o.
        - Debugging – gdb: -g for all gcc commands, break filename:line_number, break filename:function_name (filename can be omitted for second statement).

- Dynamic Storage Allocation and Linked Lists
    - Storage: Memory Storage
    - Structure: Aggregate C Type (recall: Array is an Aggregate Type).
        - Collection of data items. Call each data item a member; possibly of different types.
        - Each member is referred to by a name (rather than a subscript or index).

Reading: 13.7, 15, 16.1 (Structures). Next: 16.2, 17.1, 17.2, 17.4, 17.5
Optional Reading: UNIX Textbook – makefile (392-397).
Programming Practice: Project 3, p. 375.

    - Declaring a Structure
        - For example:

```
struct student {
int number;
char name[26]; // max size: 25 (null char)
char username[11];
} x,y;
```

        - The name after "struct" is known as the tag and is optional. Can declare, optionally, variables of this type afterwards.
        - Otherwise, declaring other variables is done by:

```
struct student z, *p, first_yr[200];
```

    - Accessing Members of a Structure
        - Use the dot operator (.)
        - For example:

```
z.number = 123456;
```

```
strcpy(z.name, “John King”);
```

- Arrow Operator
    - Shorthand for dereference + dot.
    - For example:

```
struct student *p = &z;
(*p).number = 222333;      // This is perfectly fine.
                                      // Dot has higher prec.
/* Equivalent to... */

p -> number = 222333;
```

- Structure Parameters
    - Nothing special about passing it as a parameter except for one thing.
    - If we want to pass a *large* structure, will be passed completely by value. Requires a lot of copying over. Inefficient, slow.
    - *Solution*: Pass pointers to structures.
- ***Dynamic Memory Allocation***
    - Utilizes stdlib.h, and `void * malloc(size_t size);`
        - The function will allocate a free memory block of given size; returns a pointer to an unused memory block of *size* bytes.
        - *Or*, returns the NULL pointer if the allocation *fails* (if out of memory).
    - The `void*` is a “generic pointer”. Just stores an address. Does not specify type of pointer.
    - To use the malloc function appropriately, need to pay special attention:

```
p = malloc(10000);
if (p == NULL) {
        ... // do something
}
```

-
    - Freeing memory locations uses `void free(void* ptr);`
        - Frees the memory block pointed to by ptr.
        - The space must be allocated using malloc (e.g. Done in the heap, dynamic memory).
        - After this function is called, the ptr will become a “dangling pointer”. The function does not change ptr.
    - Be aware of ***memory leaks***.
        - *Garbage*: Memory block that the program no longer has access to.
        - C does not do automatic garbage collecting (automatically deallocates). In Java, but not in C (because of efficiency).
        - A program that leaves behind garvage has a *memory leak*: this is something to **avoid**.
- ***Linked Lists***
    - Recall: [data | pointer to next] is a Node of a Linked List. Chained together to represent a sequence. First node: Head; Last node: Tail. The last pointer points to NULL.
    - Requires a *structure*. Each node can be represented as a structure.
    - Are useful when we need to add and delete from a sequence – very fast. An array

requires shuffling.

- Node Type

```
struct node {
        int value;
        struct node* next;
};
```

- Creating an Empty List

```
struct node* list = NULL; // point to 1st element of list
```

  - The above always points to the head of a list.
  - *Avoid* memory leaks! If we lose track of this, no longer access the list again.
- Program: Maintains student database (list.c).
  - Infinite loop to display a menu.
  - Different functions to handle list.
  - Inserting Node at Beginning of a Linked List: Allocate memory for node, store data in node, insert node into list.

```
        insert() {
                ...
                student->next = student_list;
                return student; // new head
        }

        search() {
                ...
                student_list = student_list->next;
        }

        delete() {
                // Locate node to be deleted.
                // Alter previous node so it points to node
// following one being deleted.
                // Call free to reclaim space occupied by deleted
                ...
                if (prev == NULL)
                        ... // delete the head
                else
                        prev->next = cur->next;
                ... // free; avoid memory leak
        }

        delete_list() {
                // Called before terminate program.
                // Removes entire list.
                // If we do not do this, when program terminates,
                // OS will reclaim all space anyway.
                // Why do we do this, then? Good for code reuse.
                while() {
                        ...
                        free(temp);
                }
                ...
        }
```

Reading: 16.2, 17.1, 17.2, 17.4, 17.5 (will have deletion which we did not talk about)
Next: 17.3
No Programming Project; A7Q1 we can work on – A7Q2 is a bonus question.
Bonus Exam: Covers all lectures on C up to and incl. Lecture on March 21.
Two programming questions; use functions (one will use dyn. Allo.)
See aids allowed.
Familiarize yourself with C syntax, functions, etc.
Practice! Work on more difficult programming projects in textbook.
Challenging problems posted.

- Sort a linked list.
  - *Remember*: Quicksort is often faster on arrays (random access), but mergesort is faster on linked lists.
  - How do we divide the list into two?
    - Divide – *Solution 1*: Use one loop to counter number n of elements in linked list. Start from head and follow to n/2 to reach head of second sub-list.
    - Divide – *Solution 2*: Use two pointers. Move second pointer towards tail twic while moving first pointer once. When second pointer reaches the tail, first pointer is at middle.
    - Second is more efficient (save arithmetic calculation; counting how many elements, etc.).
  - Program is in public folder (sortlist.c).

```
        // Add one feature to previous program.

        // Mergesort: Divide, conquer, combine.
        // nlog(n) – more efficient than bubble sort, etc.

        mergesort() {
                ...
                list1->next = NULL;
                ...
        }

        merge() {
                ...
                        prev = curr;
                ...
                return list;
        }
```

- What happens when we free, allocate memory, exactly?
- ***Heap*** (Free Store) – used in dynamic memory allocation.
    - How do we know how much space a process needs in advance? We have no idea.
    - Makes sense to have a *big pool of memory* available. Can allocate from that pool.
    - So, in essence, the heap is *a large pool of memory for dynamic storage allocate.*
    - Therefore: `malloc` - allocates memory from the heap.
    - And: `free` - "returns" a memory block to the heap.
    - Advantages and Disadvantages
        - Large data (large arrays, structures) can (should) be allocated in the heap. Otherwise, may run out of memory.
        - Dynamically allocated memory stays allocated until deallocated explicitly or by process termination (both an advantage and disadvantage). *Avoid memory leaks*.
        - Heap allocation is slower than stack allocation (which is automatic because of popping and pushing of function calls). There are algorithms from the OS to handle the heap; have runtime.

Reading: n/a. None of this is in the textbook.
Next: Dynamically allocated arrays, resizable arrays (used in Java – ArrayList). Some information in 17.3.

- Dynamic Arrays
    - Dynamically Allocated Strings
        - Set size to string length + 1
        - Can be used in string functions (i.e. Standard library functions).
        - For example, concatenating two strings without modifying either string.

```
char* concat(const char* s1, const char* s2) {
        char* result;
        result = malloc(strlen(s1) + str2len(s2) +1);
        if (result == NULL) {
                printf("Error: malloc failed\n");
                exit(EXIT_FAILURE);
        }
        strcpy(result,s1);
        strcat(result,s2);
        return result;
}

char *p;
p = concat("abc","def");
```

```
...
free(p);
```

- For example, reverse words in a String (reversewords.c). Such as: "Do or do not, there is no try."
  - Idea: Scan backwards; identify words – copy words into a temporary ***buffer*** – copy buffer back to original string.
  - Buffer: Dynamic Memory Allocation

```
buffer = malloc(len+1);
...
buffer[write_pos++] = line[word_start++];
...
free(buffer);
```

  - Solution above is solid; natural. But we can improve space efficiency – not using a buffer? Simply reverse the entire String and then reverse each word.

```
void reverse_string() {
        while (start < end) {
                ...
        }
}

void reverse_words() {
        ...
        while (...)
                end++;
        end--;
        ...
}
```

- Dynamically Allocated Arrays
  - When the size of the array is not known at compile time.
  - For example, allocating an array with *n* integers.

```
int *array, i;
array = malloc(sizeof(int)*n);
if (array == NULL) { ... }
for (i = 0; i < n; i++) array[i] = 0;
...
free(array);
```

  - Dynamically Allocated Arrays vs. Variable-Length Arrays
    - When should we use either?
    - Dynamically Allocated Arrays are put into heap (heap allocation).
    - Variable Length Arrays are allocated in the stack (stack allocation).

        - So it's essentially a question of whether or not we want to use the heap or stack.
        - Heap is for big arrays – takes time to allocate (slow). If it is small, can leave it in the stack – fast allocation. From a pure efficiency point-of-view: big vs. small arrays.
        - However, there are portability considerations. Dynamically Allocated Arrays work for any version of C. Variable-Length Arrays are a C99 feature (most compilers support, but there are some exceptions, like Visual C++ where a library function is provided instead).
    - Recall: Mergesort
        - Merge function was implemented using Variable Length Arrays as temporary arrays.
        - Now that we learned Dynamically Allocated arrays, we recognize it would be better for portability and because we are unsure of what the size could be for the array.
        - See: mergesort3.c

```
void merge() {
        ...
        left = malloc(sizeof(int) * len_left);
        ...
        right = malloc(sizeof(int) * len_right);
        ...
        free(left);
        free(right);
}
```

- Dynamic Arrays (Resizable Arrays)
    - An array structure where we can add or remove elements (at the end). Similar to ArrayList from Java.
    - Have to give an initial size; if we keep inserting elements, have to increase size.
    - Inserting an element at the end of the array.
        - Pseudocode:

```
If array is full:
        resize the array to twice its capacity
        copy contents to new array (new memory location)
        free the old memory block
store the new element
```

        - Implementation (dynamicarray.c):

```
int main(void) {
        ...
}

struct vector* create() {
        ...
        vec-> array = malloc(sizeof(int)*capacity);
        ...
}

void destroy() {
        // Reclaim space.
        free(vec->array);
        free(vec);
}

void push_back() {
        ...
        vec->array[i] = temp[i];
        ...
        free(temp);
        ...
}
```

- Following C++ name of this data structure, will define these as:

```
struct vector {
        int *array;
        int capacity;
        int size; // number of elements currently stored
};
```

- Why double the space? Why not increase it by a given number (fixed value)?
  - Efficiency
  - Insight: Say an array has *n* elements, increasing its size means its new size would be 2*n*.
    - This means *n* push_back operations are necessary in order to increase an array's size from *n* to 2*n*.
    - The copying over of 2*n* elements can be amortized to *n* push_back operations.
    - In the amortized sense, 2 copies per push_back operation.
  - But do not necessarily have to double; if we multiply by any constant other than 2, this is also true but a different constant will result in the amortization. For example, in Java, increased by 3/2. In Cython, 8/7.
- Deleting the last element.
  - If the array is less than a quarter full, we halve the capacity. Can do a similar analysis to show that the pop_back operations are amortized.

```
void pop_back() {
        ...
        free(temp);
        ...
}
```

        - When to use Dynamic Arrays.
            - Fast random access – growable and shrinkable.
            - One thing to pay attention to: sometimes we have to copy large arrays over.
            - Should know if we want to use this for real-time programming.
            - In most other applications, this is acceptable because of the sort of analysis given above.

Reading: None
Next: 19.3 – 19.5

- Abstract Data Types (ADT)
    - Often a good idea to put data structures into modules so we can use them from other programs – good for code reuse.
    - Also learned how to organize our code into multiple files.
    - Now, let's go a step further: hide implementation details so that a client module does not know how the data structures are implemented.
    - And this way, if actual implementation changes, do not have to change client module.
    - *Example*: Stack data type.
    - An ***abstract data type*** is a type in which the representation is hidden.
        - Client modules can use the type to declare variables.
        - Client works without the knowledge of the structure of these variables.
        - Can call functions provided by the module to operate on them.
        - *This is the notion of* ***data abstraction****.*
    - ***Incomplete Types***
        - Formally a type that describes objects but lacks information to determine their sizes.
        - Can declare something like: `struct t;`
            - This is allowed.
            - If we only share this with client module, only sharing tag. Client has no idea how implemented.
            - This is not enough. Cannot determine the size.
        - So we then make use of the pointer(s).
            - Using: `typedef struct t* T;`
            - Declaring a different name for a pointer of type struct t.
            - The size of the pointer is fixed. Do not need to know about struct t.
            - If we share this in the header file, client module can declare pointers of this type.
        - See stack.h.
            - Declaring a stack ADT.
            - In the adt folder of the public folder.
            - Use incomplete Stack type as parameters to call functions.

```
// stack.c
```

```
...
s->contents = malloc(sizeof(Item)*initial_size);
...
```

- File Manipulation
  - Remember: In Unix everything is either a file or a process.
  - Streams and Files
    - In C, a stream is *any source* of input **or** *any destination* of output.
    - Streams may be associated with various (I/O) devices:
      - Standard streams: stdin, stdout, stderr.
    - C abstracts all file operations into operations on streams (of bytes).
      - Input stream.
      - Output stream.
    - Accessing files means accessing streams. Streams are accessed through file pointers.
      - Variables of type FILE*
      - stdio.h
    - Text Files vs. Binary Files
      - Treated differently in C. Different functions to access.
      - The notion of lines applies to text files only.
      - Storing a value, for example 12345:
        - Text File: five characters '1','2','3','4','5' (5 bytes).
        - Binary File: store its binary representation – sizeof(int) bytes if stored as an int.
  - Opening a Text File
    - `FILE* fopen(const char* filename, const char* mode);`
    - Filename: In fact, a pathname (can be absolute starting from root or relative).
    - Mode: Can take three values for text files "r" for reading, "w" for writing – file need not exist – overwrites, "a" appending – file need not exist – appends.
    - If it cannot open the file, `fopen` returns a NULL pointer.
  - Closing a File
    - `int fclose(FILE* fp);`
    - 0 succeeds, EOF fails.
  - Formatted I/O
    - printf, scanf are special cases of:
    - `int fprintf(FILE* stream, const char* format, ...);`
    - `int fscanf(FILE* stream, const char* format, ...);`

Reading: 19.3 – 19.5, 22.1 – 22.2; Next: 22.3 – 22.6
Project 1, page 506

  - Comparing I/O Methods
    - `printf(...) is equivalent to fprintf(stdout,...)`
    - `scanf(...)is equivalent to fscanf(stdin,...)`
  - Error Messages (Printing to stderr)
    - We've only learned printf. If we want to follow the standard of Unix utilities, we can use `fprintf(stderr,...)`.

- Let's use what we've learned to write a simple program...

```
// Write a line of text to a file.

FILE* fp; // file pointer
fp = fopen("hello.txt","w");

if (fp == NULL) {
        fprintf(stderr,"Cannot create the file hello.txt\n");
        exit(EXIT_FAILURE);
}

fprintf(fp, "hello, world");
fclose(fp);
```

- Character I/O Functions
  - Output Function: int putc(int c, FILE* stream);
    - Returns EOF if there is a write error; otherwise, will return c.
    - Recall: putchar(c). Writes a single character to stdout. So, this is equivalent to putc(c, stdout).
  - Input Function: int getc(FILE* stream);
    - Returns EOF when end of file is reached, otherwise returns a character.
    - Analogous to getchar(). Equivalent to getc(stdin);
  - Example: Let's say we want to count the number of characters in a text file.

```
#include <stdio.h>
#include <stdlib.h>

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

        FILE* fp; // file pointer
        int count = 0; // counter

        // Getting name of file from command line.
        if (argc != 2) {
                fprintf(stderr, "Usage: cntchar filename\n");
                exit(EXIT_FAILURE);
        }
        if ((fp = fopen(argv[1], "r")) == NULL) {
                fprintf(stderr, "Cannot open %s\n", argv[1]);
                exit(EXIT_FAILURE);
        }

        // Count characters.
        while (getc(fp) != EOF)
                count++;
        // Print result.
        printf(“There are %d characters.\n”, count);
        fclose(fp);
        return 0;

    }
```

- Binary Files
  - It's important to specify what type of file you are opening. Values are encoded differently.
  - We should use a different mode argument for binary files: still using fopen function.
    - However, the mode parameter, instead of just “r”, “w”, and “a”, is also suffixed with a “b” at the end (example: “rb”).
    - The sort of functions we use to write (and read from) to a file will be different, though.
  - Block I/O Functions – allow you to read/write from/to binary files.
  - Block Input
    - Reading blocks of memory (bytes) and store into a sufficiently large block of memory.
    - In other words, reading the elements of an array from a stream.
    - The prototype is: size_t fread(void* ptr, size_t size, size_t count, FILE* stream);
      - The ptr stores the address of an array. Using void* because we just want the address; does not matter what type of array it is.
      - The size specifies the size of each element in bytes.
      - The count specifies the size of the total array (number of elements).
      - Result is stored into a memory block that ptr points to.
    - The return value is the number of elements read. We can check whether or not the read is successful by comparing to count.
  - Block Output
    - The prototype is: size_t fwrite(const void* ptr, size_t size, size_t count, FILE* stream);
    - The result is the number of elements that is written.
    - Writes content of array pointed to by ptr.
  - Example: Instead of a linked list (where data is lost after program ends), can store data to a binary file. Student Database. Source code is at studentdb.c.

```
    void save() {
        ...
        fp = fopen(filename, “wb“);
        ...
        fclose(fp);
    }

    student node* load() {
        ...
        if (fread(student, sizeof(struct node),1,fp) != 1)
break;
        ...
        fclose(fp);
}
```

More functions are in the textbook. Library functions. Do not need to know for exam.
Reading: 22.3, 22.4, 22.6 (Just functions covered in class).
Next: UNIX Shell Programming (Chapter 8)
Project 4, page 585 (C).

- Basic UNIX Shell Programming
  - Why shell scripting?
    - System adminstration. When a shell starts, a few scripts are run automatically to configure the system.
    - Fast prototyping. Doing a lot of testing.
    - UNIX Philosophy of breaking projects into sub-tasks. Sometimes, a pipeline is not enough to do the tasks we want to do.
  - Shell Programs, Scripts
    - Are text files that stores any series of shell commands.
    - After written, all we have to do is grant it an executible permission (*chmod*) and then we can run it. "Interpreted"; no need for compilation.
    - For example, let's say we want to use a few commands to print the current system status.

```
// current.sh – extension is convention

#! /bin/bash                    // which shell to run in
# print current status          // comment; starts with #
whoami                                  // username
pwd                                     // working directory
ls                                      // list directory
```

    - To run the above script, must first change its permissions: `chmod u+x current.sh`.
    - And then simply enter: `./current.sh`
    - First line: What shell.
      - Only a #: run in current shell program, from which executed.
      - `#!pathname`: explicitly specifies which shell program to use.
      - Neither of these two lines: uses Bourne shell (older than any other shell).
  - Interpretation
    - Do not require compilation. Easy to modify.
    - Disadvantage: Slower. Series of text commands, not binary instructions.
  - Variables
    - Names: Same as C.
    - Using = and $. For example:

```
i=1
echo $i
```

- Predefined Local Variables
  - The pathname of the script: $0
  - To get the n-th command line argument: $n
    - For example: $1,...,$9,${10},...
    - The $# variable reports the number of command-line arguments. Excludes $0.
- Operators
  - ((expression))
  - =, +, -, ++, --, *, /, %, ** (exponentiation)
  - Change orders of operations using ( ).
- For example:

```
// add.sh (./add.sh 45 54)

#!/bin/bash

((sum = $1 + $2))
echo The sum of $1 and $2 is $sum.
```

- Conditional Expressions
  - Arithmetic Test
    - ((expression))
    - Operators: <= >= < > == != ! && ||
  - String Comparisons
    - [expression]
    - Be aware of two space characters in above.
    - Operators: == !=
- Control Structures
  - The if statement; different syntax from C.

```
if condition1; then
        commands
[elif condition2; then
        commands]
[else
        commands]
fi
```

    - For example:

```
// add.sh (./add.sh 45 54)

#!/bin/bash
if (($# != 2)); then
        echo usage: ./add.sh num1 num2
        exit
```

```
fi
...
```

- The for statement.

```
for var in word {word}*
do
        commands
done
```

  - Using a set of words seperated by spaces.
  - Do not declare types for variables in shell; all treated as strings. When used in arithmetic operations, automatically treat them as numbers. (Floating numbers require external commands).
  - For example: Say we want to sort the contents of all .txt files in a current directory, and store them in *.txt.sorted files.

```
#!/bin/bash
for file in *.txt
do
        sort $file > $file.sorted
done
```

Reading: UNIX Chapter 8 (Only what we covered in class) – up to for loop.
Next: UNIX Chapter 8 (Rest of chapter).

- Command Substitution
  - Using the ` accents to surround a command. This command will be executed and its output is inserted in the command line.
  - For example,

```
echo There are `ls l wc -l` files in the current directory
```

  - And, extending a previous example,

```
#!/bin/bash

for file in `ls *.txt`
do
        sort $file > $file.sorted
done
```

  - **Note**: Semi-colons can be placed at the end of statements. Should be used if commands are all on the same line. For now, just use different lines.
- The case statement (similar to switch)

        - The syntax:

```
case var in
        word{|word}*)
                commands
                ;;
        ...
esac
```

        - For example, printing whether there is a (2132) lecture today...

```
#!/bin/bash

day=`date | cut -f 1 -d “ “`

case day in
        Mon|Wed|Fri)
                echo 2132 lectures
                ;;
        Tue|Thur)
                echo no 2132 lectures
                ;;
        Sat|Sun)
                echo no lectures
                ;;
esac
```

- Advanced Shell Scripting
    - Conditional Expression for Files
        - The syntax:

```
[ option ]
```

        - Note the space after and before the first and last paranthesis.
        - The following are examples...

```
# Checking types of files.

[ -e file ] # true if file exists
[ -f file ] # true if file exists and a regular file
[ -d file ] # true if file exists and is a directory

# Checking permissions of files.

[ -r file ] # true if exists, readable by current user
[ -w file ] # true if exists, writable by current user
[ -x file ] # true if exists, executable by current user
```

- Functions
  - Syntax:

    ```
    name() {
            ...
    }
    ```

  - Looks like a C function without return type, parameters.
  - Can still use pre-defined local variables (command-line arguments) to take in *parameters*. In other words, done by a standard positional mechanism.
  - Returning from a function.
    - Using the `return` command.

    ```
    return [value] # brackets here meaning optional
    ```

    - This value is returned to the caller. Note, though, that this value must be an integer and be between 0 and 255.
    - Accessing this value from the caller requires the use of another predefined variable (return value is also known as an *exit code*). Use `$?`
    - Will immediately have the return value of the last returned function.
    - For example,

    ```
    #!/bin/bash

    add() {
            (( sum = $1 + $2 ))
            #echo $sum – can put this to work around 255 limit
            # by using command substitution
            return $sum
    }

    if (( $# != 2)); then
            # print error message
    fi

    add $1 $2
    result=$?
    echo The sum of $1 and $2 is $result
    ```

  - Exit codes.
    - The return value of a function is treated as an exit code. After we run a UNIX command, an exit code is retrieved.
    - To access it, typically use the `$?` command.
    - Can also use the exit command (`exit number`) to terminate the program.

        - If number is omitted, returns the exit code of the previous command.
        - Returning without an argument, the exit code of the last command is executed in the function.
        - The different between return and exit is the same as C.
    - Example: Backup script (backup.sh)
        - For each file in the source directory, copy it to the destination directory iff
            - The file is a regular file
            - It does not exist in the destination directory
        - Prints out the file name each time a file is copied.

```
...
for filename in `ls $1`
        ...
        cp $1/$filename $2
...
```

Reading: Chapter 8

- C Appendum – Topics Students Sometimes Have Trouble With
    - Pointers as Function Arguments
        - Can be used to change the value of the variable it points to.
        - **Example**: Find the largest, second largest elements in an array.

```
void find_two_largest(int a[], int len, int *largest, int *second_largest) {

        int i;
        if (a[0] > a[1]) {
                *largest = a[0];
                *second_largest = a[1];
        }
        else {
                *largest = a[1];
                *second_largest = a[0];
        }
        for (i = 2; i < len; i++) {
                if (a[i] > *largest) {
                        *second_largest = *largest;
                        *largest = a[i];
                }
                else if (a[i] > *second_largest)
                        *second_largest = a[i];
        }
}
```

- Pointer Arithmetic
    - **Example**: Use a pointer to *sum* the elements in an array.

```
int sum_array(int a[], int n) {

        int *p, sum = 0;
        for (p = a; p < a+n; p++) sum += *p;
        return sum;

}
```

- String Variables
    - Essentially are character arrays that terminate with NULL character (\0).
    - Character pointers can point to strings.
    - Fix the following function (that supposedly creates an identical copy of a string):

```
char* duplicate(char *p) {

        char *q;
        q = malloc(strlen(p)+1); // extra byte for null char
        if (q == NULL) return NULL;
        strcpy(q,p);
        return q;

}
```

-
    - Not allocating the memory will result in a segmentation fault. Pointer pointing to insufficient amount of memory.
- Linked Lists
    - Avoiding the dangling pointer problem.
    - Happens when you free the space a pointer points to and not account for it.
    - Will the following code work (deletes all nodes and reclaims memory)?

```
for (p = first; p != NULL; p = p->next) {
        free(p);
}
```

-
    - Will have a dangling pointer. Instead:

```
struct node* temp;
p = first;
while (p != NULL) {
        temp = p;
        p = p->next;
        free(temp);
}
```

- Final Exam and Beyond
    - A question that will ask to break a program into source file, header files (without implementing them) – write a makefile for them. Study example in Lecture 25, Assignment 7 sample solution.
    - UNIX scripting problem (fill-in-the-blank).
    - Implement a few functions on linked lists.
    - Software development tools, incl. those learned in labs, like git.
    - If liked the experience of Bonus Programming Exam, try out the ACM Programming Contest. (Joined in teams; 5 hours for each team – about 10 questions).
    - Further Reading: Pointers to Functions (which are loaded in memory), etc.
    -