Software Development (CSCI2132)
Covers C

Lecture Dates: February 10th to March 7th, 2012

- **Exiting from a Loop**
  - *break*, *continue* like in Java, but there are no break labels.
  - These are often known as "Jump Statements".
  - There is also the *goto* statement. It can jump to any statement (provided it has a label) in the same function. This is why we do not need the break labels. Can use this instead.
    - How do we declare a label for a statement?
      - Syntax: Label is a C identifier.
      - identifier: statement;
    - How do we use the goto statement to jump to it?
      - goto identifier;
- **Example**: Writing a program to print numbers from 1-10, each one its own line. Just for sake of demonstrating goto, will use it here even though it is not necessary.

```
#include <stdio.h>

int main(void) {

        int i = 1;
  loop: printf("%d\n",i);
        i++;
        if (i <= 10) goto loop;
        return 0;

}
```

- Transfering the control structure backwards using the goto statement.
- *goto* is very old. Even though Java does not have it, we could jump around with a label since assembly. BASIC used it extensively (how people originally created loops).
- Use of *goto* has become much less popular; people have had the notion of doing *structured programming*. People have proved for any computable function: **sequencing** (one part after another), **repetition** (for, while, etc.), and **selection** (if, branching).
  - Can do anything without it.
  - Also makes it harder to read and edit your code.
- The question is whether or not we should use *goto*.
  - Sometimes, *goto* can improve efficiency.
  - For example:

```
while (...)  {

                switch(...) {
                        goto loop_done;
```

```
            }

        }

    loop_done: ...
```

- Not doing as much testing if you were to have another variable declared for testing at each while iteration.
- There are other cases as well. Posted some optional reading.

- For this course: only use *goto* to break out of nested loops (e.g. Switch).
- **The null statement**
  - Essentially, an empty statement: ;
  - Why would we need this? Sometimes we do not need anything in a loop.
  - For example:

```
for(d = 2; d <= n && n % d != 0; d++)
        ;
if (d < n) printf("%d is not a prime number.\n", n);
```

Reading: 6.4-6.5

- **Software Development Life Cycle, Testing, Debugging**
  - Something we should always have in mind when working on our programming assignments.
  - These slides will be put online.
  - *Software Development Life Cycle* (SDLC)
    - General term describing structure imposed on development of a software product.
    - Major steps done in the development of the software.
    - Purpose
      - To reduce risk of missing a deadline.
      - To ensure product quality (more robust).
    - Many models have been proposed to describe SDLC.
    - **Waterfall Model**
      - Sequential design process
        - Requirements Analysis
        - (Software) Design
        - Implementation (Coding)
        - Verification (Testing)
        - Maintenance (Patches...)
      - Easy to understand; use this for assignments.
      - Widely used; reinforces notion of "design before coding".
      - Clear milestones.
      - Disadvantage: Often not practical – clients may change requirements. Designers

may not be aware of implementation difficulties.

      - **Rapid Prototyping Model**
        - Addresses problem where user may change specifications.
          - Preliminary Requirements
          - Fast prototyping. (Not complete product).
          - User evaluation of prototype.
          - Repeat above steps if necessary.
          - Discard prototype, development of software using a formal process (e.g. Waterfall).
        - Advantages:
          - Ensures software product meets requirements.
          - Reduce time/cost if clients request changes during process.
        - Disadvanteges:
          - Adequate, appropriate user involvement may not be possible, always.
          - Cost of prototype development (time and money).
      - Other models.
        - Studied in Software Engineering course.
        - Choose an appropriate model depending on context, software being developed.
    - *Software Testing*
      - Motivation
        - Robust.
        - Maintain reputation.
        - Lower cost: fixing a bug before release.
      - Job positions on testing.
      - What do we test?
        - Whether it works.
        - Where it meets specification(s), which contains:
          - Description of input.
          - Description of output.
          - Set of conditions.
          - Specifying what output should be, given input and conditions.
      - Mindset
        - How to make the program fail?
        - Typical test cases:
          - Regular
          - Boundary (not just values but also states of input)
          - Error
      - Types of Testing
        - White Box
          - Know the code.
          - Use internal knowledge of implementation to guide selection of test cases.
          - Achieving maximum code coverage.
        - Black Box
          - Do not know code. Way our assignments are tested.
          - Use specification to guide selection of test cases.

        - Achieving maximum coverage of cases given in specification.
    - *Debugging*
        - A methodical process of finding, reducing bugs or defects in a computer program.
        - Key step: Identifying where things go wrong.
            - Track program state (current location, current values of variables, number of iterations through a loop).
            - Find when expected program state does not match actual program state.
        - Easiest way to do this is using **printf debugging**.
            - Using printf statements to print information while executing.
            - For example, to track location, values of variables, iterations.
            - Linear approach: Start from beginning until you reach bug.
            - Binary search: Select a half-way point, and narrow it down.
            - However, this is *time consuming* for larger programs: modifying program, recompiling, rerunning.
        - We can also use gdb
            - Symbolic, source-level debugger.
            - Allows programmers to
                - Access another program's state as it is running.
                - Map state to source code (variable names, line numbers, etc. - why we compile with -g option).
                - View variable values.
                - Set breakpoints.

Software Development – February 13, 2012
C, Continued

- Basic Types (of Variables)
    - ***Integer*** Types: Different types.
        - Use a **specifier** before the int.
            - Signed and unsigned (at least zero). If we ignore this, default is signed.
            - Can have different ranges: occupy different number of bits.
            - Short, long. Specify size of int type.
        - **Combinations** of specifiers.
            - Fewest number of bits to longest number (6 types).
            - short int (unsigned short int)
            - int (unsigned int)
            - long int (unsigned long int)
        - **Order** of specifiers in combinations does not matter.
            - For example: unsigned long int is equivalent to long unsigned int.
        - When we write down these types, we can usually ***drop*** *the* int.
            - For example: unsigned long is equivalent to unsigned long int
            - For example: long int = long
        - The **ranges** of values represented by each integer type. Vary between machines.
            - Use one bit to indicate sign when signed.
            - Typically, int uses one machine word for storage. For example, in a 16-bit machine

we use 16 bits for one machine word. Maximum value: 2^15 – 1 = 32,767 (remove 1 because of storing 0). Its minimum value is: -2^15 = -32,768.

- In the same machine, an unsigned int would have a maximum: 2^16 - 1 = 65,535.
- Reason for limitation: Efficiency.
  - A machine word is a fixed group of bits labeled as a unit by the instruction set. So if an int variable is stored in one machine word, it is just one machine instruction when adding two integers.
  - In Java, int range is fixed – Java is maximized for portability (can run on different machines efficiently).
- Determining the ranges (on our machine – system dependent): including the header file `<limits.h>` and using some macros defined in the file.
  - INT_MIN: Minimum value of an int.
  - INT_MAX: Maximum value of an int.

- Integer literals (**constants**) are by default int (if it is not outside of the range). A larger type would be used in that latter case.
  - But sometimes we want to use a larger type.
  - For the compiler to treat a constant as long, we can write an "L" after the constant.

- ***Floating*** Types:
  - Different levels of precision.
    - `float` is a single-precision floating-point value.
    - `double` is a double-precision floating-point value (more precision than float).
    - `long double` is for extended precision (more precision than double).
  - How much precision do each of these provide? Implementation-defined.
  - Most modern compilers follow IEEE standard 754 (same as Java). More details in textbook. Enough to know that:
    - float: 6 significant digits (ignore leading, trailing zeros).
    - double: 15 significant digits.
  - Floating literals (**constants**) are by default the double type. To explicitly specify it as a float type, we suffix with an "f".
- ***Character*** Types:
  - Big difference between `char` in C and in Java.
    - Uses 8 bits in C. Java uses 16.
    - When C was proposed, 8 bits was enough for ASCII.
    - 16 is necessary for Unicode.
  - Example: `char ch = 'a';`
  - C treats these as *small integers*.
    - Each of them has an integer value.
    - On most machines, these correspond directly to the characters' ASCII values.
    - This includes bluenose.
    - Can hold negative values.
  - Example:

```
if('a'<=ch && ch<='z')
        ch = ch – 'a' + 'A'; // Converts to uppercase.
```

- printf can output characters. Just need a new placeholder: %c
- scanf can input characters. Using the same placeholder: %c
  - However, scanf does not skip whitespace characters when taking in a character.
  - For example, can read spaces, newlines, etc.
  - Example: scanf("%c",&ch); vs. scanf(" %c",&ch);
    - These are not equivalent.
    - First statement reads character-by-character.
    - The second statement is used to skip white-space characters.
- Also in the stdio.h header is the getchar function.
  - For example: ch = getchar();
  - It is equivalent to scanf("%c",&ch);
  - Fast and simple – scanf required a great deal of rules (pattern matching). This function, however, merely reads characters and does not have to do pattern matching.
- Example:

```
        // A user enters a message. We want to convert it                //
completely to uppercase characters. Can read
        // characters one-by-one and use above.

        #include <stdio.h>

        int main(void) {

                int ch; // Not using char here. See below.
                // Comparison higher prec than assignment.
                while ((ch=getchar()) != '\n') {
                        if (ch >= 'a' && ch <= 'z') ch -= 'a' + 'A';
                        printf("%c",ch);
                }
                printf("\n"); // Did not output a newline above.
                return 0;
        }
```

- Why did we use int?
  - We learned in Unix before that we can specify the end-of-file metacharacter. In C, we can specify EOF (typically negative) and it is implementation-defined whether char is unsigned or not so there will be problems if it is not signed.
  - Always safe to use int when using getchar.

- ***Type Conversions***
  - More flexible than Java.
  - For example: declaring a float f = 3.4 which is double by default.
  - There are implicit conversions in C which is handled by compilers. Need not ask compiler to do any conversions.
    - When the operands in an arithmetic or logical expression do not have the same type:

    - Promoting: promotes operands to “narrowest” type that will safely accommodate both values.
    - For example: float f; double d; int i;
        - d = d + f // float promoted to double and then add
        - f = f + i // int promoted to float (can store more)
  - When the type of the expression (on the right side) of an assignment does not match the type of the variable (declaration on the left side):
    - Converting the right side to left side.
    - For example: int i = 8.92; // i will only store 8

Reading: C 7.1 – 3
Next Lecture: C 7.4 – 6, 8.1
Programming Exercise: Project 8 (p158)
Written Exercises: On website.
Assignment due this week.

- Casting
  - C-type casting tells the compiler to treat a variable as of a different type.
  - Syntax: (type) expression;
  - For example: Computing the fractional part of a float value:

```
float f, frac_part;
frac_part = f – (int) f;
```

  - When we need to override the compiler's default behavior and force it to do a conversion that we want (explicitly). This is different than above which was implicit (type conversions).
  - Another example:

```
float quotient;
int dividend = 5;
int divisor = 4;
quotient = dividend/divisor; // assigning 1 to a float = 1
quotient = (float)dividend/divisor; // 1.25 assigned
quotient = (float)(dividend/divisor); // 1 assigned
quotient = 1.0f*dividend/divisor; // 1.25 assigned
```

- Type Definitions using typedef
  - Assigning alternative names to existing types.
  - Syntax: typedef typename alternative;
  - For example: typedef int Bool; Bool flag;
    - Works in C89.
    - Makes program *more understandable*.
    - Improves portability.
  - Another example:

```
typedef int Quantity; // fast
typedef long Quantity; // bigger range
// Just change one line of code when changing platforms.
```

- The sizeof operator
  - C language appears to not specify size; implementation-defined.
  - This operator can be used to determine how many bytes are required to store values of a particularly type.
  - Syntax: `sizeof(type);`
  - Example:

```
sizeof(char); // 1 byte.
          sizeof(int); // 2, 4, 8 (16,32,64 bit)
```

- Arrays
  - Types
    - Scalar type: a data type composed of a single element.
    - An aggregate type: A data type composed of multiple elements.
    - Array structures are *aggregate types*.
  - One-Dimensional Arrays
    - Sequence of values of same type. Consecutive sequence of bytes in memory.
    - Declaration
      - Syntax: `type name[size]` where size is an integer constant expression
      - For example: `int n[10];`
      - When you declare an array, memory storage for the array is allocated immediately (why size must be assigned).
      - Arrays defined in this way in a function are stored in *stack memory* (used to store local variables).
      - Advantage of this is that memory stored in stack (stack allocation) is fast.
        - Variable sizes are fixed.
        - Values are there until function terminates, returns. Cannot free memory.
        - Recall: *heap memory* was for dynamically allocated variables (variable sizes are not fixed – determined by user input, for instance) and so want to free memory not used – requires management (slow).
          - Java used heap for arrays at inception (very slow in comparison).
          - Java 6 brought about *escape analysis* – did not change syntax (merely a change in compiler); does an analysis to see if an array can be allocated in stack or not.
          - This explains why difference in performance between C and Java is not so large nowadays.
    - Using macros for array lengths makes it easier to modify your program, prevents errors.
      - Example:

```
#define LEN 10
...
int a[LEN];
```

        - Array Subscripting; accessing elements in an array.
            - From 0 to n-1. a[0], a[1], ..., a[n-1]
            - Recall: C does no checks to ensure you are within bounds of an array when subscripting. Does not require subscript bounds to be checked. This is for efficiency – requires multiple machine instructions.
            - So what happens when you a subscript goes out of range? *Undefined behavior*.
                - Keep track of array sizes.
                - When debugging, this is something to check.
        - Assigning Initial Values to an Array (Array Initialization)
            - For example: int a[4] = {1,2,3,4};
            - Do not even require size in this case (compiler can tell): int a[] = {1,2,3,4};
            - These two statements are equivalent.
            - But sometimes we need the size because we just want to initialize part of the array: int a[10] = {1,2,3}; The remaining values will be given value 0.
            - So by declaring an array like this: int a[10] = {0}; All values will be given 0.
            - When an initializer is not present, unlike Java, the array is not given any initial values. This is for the sake of efficiency. Java chooses to initialize arrays, variables, etc. to ensure everything is safe.
- Application: **Binary Search Algorithm**
    - Sorted array, search for a value in this array. *Recursive*.
    - Look at the middle, compare to that value, and then determine which half of the array it must be in.

```
#include <stdio.h>
#define LEN 10

int main(void) {
	int array[LEN];
	int lower, upper, middle, key;
	int i;
	printf(“Enter %d numbers in ascending order:\n”,LEN);
	for (i = 0; i<LEN;i++) scanf(“%d”,&array[i]);
	lower = 0; upper = LEN – 1;
	middle = (lower+upper)/2;
	printf(“Enter number to search for: “);
	scanf(“%d”,&key);
	while (array[middle] != key && lower <= upper) {
		if (key < array[middle])
			upper = middle-1;
		else
			lower = middle+1;
		middle = (lower+upper)/2;
	}
	if (lower <= upper)
		printf(“%d is the %d-th number you entered.\n”,key,middle+1);
```

```
        else printf(“Not found.\n”);
        return 0;
}
```

Reading: 7.4-5, 8.1
Example (above) is on course website (Lecture summary page).
Next Lecture: 8.2-3, 9.1-4

- Multi-Dimensional Arrays
  - In C, arrays can have any number of dimensions.
  - For example, a 2-dimensional array: `int m[5][9];`
    - Can be used to represent a matrix, table.
    - See below:

int m[5][9]	0	1	2	3	4	5	6	7	8
0	m[0][0]	m[0][1]	m[0][2]	m[0][3]	m[0][4]	m[0][5]	m[0][6]	m[0][7]	m[0][8]
1	...	m[1][1]	...	m[1][3]	...	...	...	...	...
2									
3									
4									

2D array m

  - Array Subscripting: `m[i][j]` is an element in row i, column j.
  - Actual Memory Storage: Stored an array of arrays.

row[0]			row[1]			row[2]			row[3]			row[4]		
m[0][0]	...	m[0][8]	m[1][0]	...	m[1][8]	m[2][0]	...	m[2][8]	m[3][0]	...	m[3][8]	m[4][0]	...	m[4][8]

row-major order

  - Ex: `int t[3][3] = {{1,0,0},{0,1,0},{0,0,1}}; // Identity matrix.`
    - Inner bracers can be omitted (but is not as readable).
    - But, can do something like: `int z[3][3] = {0}; // Initializes entire array to 0.`
- Variable-Length Arrays (C99)
  - Before, arrays needed to be specifically declared using a constant to declare its size.
  - The length of a variable-length array is a non-constant expression.
  - For example (note that we're not declaring all variables at the beginning; C99):

```
int len, i;
```

```
        printf(“Enter the number of integers: “);
        scanf(“%d”,&len);
        // Make sure len has right value when declaring a VLA.                //
Memory is allocated when an array is declared.
        int array[len];
        printf(“Enter %d integers: “, len);
        for (i = 0; i<n; i++)
            scanf(“%d”,&array[i])
```

- Example Program: Testing whether a matrix is a latin square (an n-by-n matrix is a latin square if each row is a permutation of 1,2,...,n, and each column is also a permutation of 1,2,...,n). **Sudoku** (9x9 latin square with latin square sub-regions).

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

        int main(void) {
            int size;
            printf(“Enter the size of the latin square: “);
            scanf(“%d”,&size);
            int square[size][size];
            int i, j;
            printf(“Enter a matrix (%d-by-%d) of integers in the range [1-%d]: \n”.
size,size,size);
            for (i = 0; i<size; i++)
                ...
                scanf(...,&square[i][j])

            bool visited[size];

            ...

                    if (visited[square[i][j]-1]) {

            ...

                    if (visited[square[j][i]-1]) {
```

Reading: Chapter 8
Next: Chapter 9 (Functions)
Practice: Project 16 (pp. 181)
Writing Exercises online.
Assignment 5 posted (2 programming questions).

- Challenging examples will be used in class; on problems for practice – not required for midterm/final. See slides online for February 27th.

- **Midterm II**: Exam 2. 47 minutes. Same room as last time (1014 Management Building). Cheat sheet – letter size, both sides. Incl: Labs 5, 6. Last year's midterm. Written exercises. Assignments 4, 5. Will have to predict output of C programs. More time for coding questions.
- Functions in C
    - Set of statements grouped together and given a name.
    - Similar to Java (methods).
    - One difference: C functions are not the member of any class. Was developed before OOP.
    - Like Java, using it to avoid duplication of code, easier to reuse code, etc.
    - Recursive functions are natural solutions to many problems.
    - Function Definitions
        - A function that returns the larger value of two parameters. First part of syntax is return type (can use void), then the function name (max), and then parameters (in brackets; can be void or empty – void is more explicit).

```
int max(int a, int b) {
        int c; // Local variable; defined inside function.
        c = (a>b) ? a : b;
        return c;
}
```

        - Calling the function is as we have done.

```
int main(void) {
        int a = 5, b = 4;
        printf("%d\n",max(a,b));
        return 0;
}
```

        - Return Values
            - One restriction is that functions cannot return arrays, because of efficiency. (Copying elements of one array to another – using a loop).
            - Two library functions we've used: printf, scanf. They both return values.
                - printf – number of characters printed. Sometimes we want to explicitly state we are discarding value: `(void)printf("hello,world\n");` Like having more documentation.
                - scanf – more useful. For a robust program, we want to detect errors in user input.
                    - Number of data items successfully read and assigned.
                    - Returns a EOF (end of file; negative macro) if an input failure occurs before any data item can be read. Not 0.
                    - Example of this:

```
if (scanf("%d%d",&i,&j) != 2) {
        printf("invalid input.\n");
        return 1;
}
```

- Declarations (Prototypes)
  - return-type name(parameters);
  - Discarding the block.
  - For example:

```
#include <stdio.h>

int max(int, int);
int main(void) {
        ...
        return 0;
}
int max(int a, int b) {
        ...
}
```

  - Parameter names can be omitted.
  - Prototypes allow compiler to check (only has to scan once).
  - In C99, either a declaration or definition must be present prior to function call.
    - Function should be before main function, however, if no prototype is given. In C89, not required – follows rules and makes assumptions.
    - In this course, follow C99 rules and provide prototype at beginning, place them before the main.
- Arguments
  - When we call a function, the variables provided are arguments, but parameters are the actual specified placeholders.
  - Arguments are passed by value.
    - When a value of an argument is passed, copying the value over.
    - If make changes to that parameter during function execution, will not be reflected on arguments.
    - For example, the following will print 4, 5 – the function does not actually swap the values in the calling function:

```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}

int main(void) {
        ...
        int a = 4, b = 5;
        swap(a,b);
        printf("%d, %d\n",a,b);
        ...
}
```

        - How do we actually do this? Using pointers – elegant workaround. Java typically has workarounds.
      - Array Arguments
        - Must provide the length of the array as one parameter.
        - Do not have a `.length` value like in Java.
        - For example: `int max_array(int a[], int len) { ... }`
        - For its prototype: `int max_array(int [], int);`
        - If a function changes the element of an array parameter, change is reflected in corresponding argument (like in Java). Pointer is passed (see later).

Reading: 9.1 – 9.4 (C)
Next: 9.5 – 9.6, 10 (C)
Practice: Project 1, p. 216.
Written: Posted online.

- What happens when a call is made to a function?
- **Call Stack** – a stack data structure (see before in the course).
  - Specificially stores information about the active functions of a program.
  - Related to function calls – also used in other programming languages.
  - A *stack* is a data structure where data is stored in a last-in, first-out fashion.
    - Think: a stack of books, magazines.
    - Can put additional books in, but must be added to the top (push operator).
    - Retrieving a book in the middle means you must remove books one-by-one (pop the elements out).
  - When a function is called, the compiler creates what is known as a *stack frame*.
    - Contains information about the function.
    - Storage for its parameters, local variables, return value, etc.
  - This stack frame is pushed onto the call stack.
    - Top of the stack is the function which was most recently put onto the stack (most recently called).
    - For example: calling the function *max* while in the *main* function.
      - Main function stack frame is pushed into the stack (on the top/bottom of the stack).
      - Main function calls the max function; a new max stack frame is pushed onto the stack (is now on the top; main is on the bottom).
    - Information of the stack frame at the top is available (for access).
  - When a function at the top returns (is done executing), its stack frame is popped (out of the stack). Return value is given to function that called it.
  - The stack frame of the calling function (the latter) is now at the top of the call stack. Can now access it, etc. Can continue execution.
  - *Remember*: The stack part of process memory storage contains all of this information.
- **Recursion** – a function is recursive if it calls itself.
  - Very useful for tasks that can be defined in terms of similar subtasks that does not depend on the whole.
  - For example: Computing the power of a number.

```
int power(int x, int n) {
        if (n == 0) return 1; // base case
        else return x*power(x,n-1); // recursive case
}
```

- For example: 3^4.
  - Stack (Top): 3,0 – returns 1
  - 3,1 – returns 3*1
  - 3,2 – returns 3*3*1
  - 3,3 – returns 3*3*3*1
  - 3,4 – returns 3*3*3*3*1
  - main – gets the value 81 (bottom of stack)
- Could do this using a for loop (iterative). Would actually be more efficient. Recursion has overhead associated with stack pushing/popping.
  - But there are problems where recursion is more natural.
  - For example: Mergesort. Quicksort is faster for arrays in practice.
    - Mergesort is faster for linked lists (in practice). Easy to be made I/O efficient. Easy to be made a parallel algorithm.
    - Divide-and-Conquer
    - Dividing n-element array into two subarrays of n/2 elements. Sort each recursively and combine.

```
#include <stdio.h>



void mergesort(int array[], int lower, int upper);

void merge(int array[], int lower, int middle, int upper);



int main(void) {

  int len;



  printf("Enter the length of the array: ");

  scanf("%d", &len);
  int array[len];


  printf("Enter %d integers:\n", len);


  for (int i = 0; i < len; i++)

    scanf("%d", &array[i]);


  mergesort(array, 0, len-1);


  printf("The sorted array is: \n");

  for (int i = 0; i < len; i++) {

    printf("%d", array[i]);

    if (i < len - 1)

      printf(" ");

  }

  printf("\n");


  return 0;

}


void mergesort(int array[], int lower, int upper) {

  if (lower < upper) {
 // Recursive case.
    int middle = (lower + upper) / 2;

    mergesort(array, lower, middle);
    mergesort(array, middle+1, upper);

    merge(array, lower, middle, upper);

  }

// If one element, array already sorted.
}


void merge(int array[], int lower, int middle, int upper) {

  int len_left = middle - lower + 1;

  int len_right = upper - middle;

  int left[len_left], right[len_right];

  int i, j, k;


  for (i = 0; i < len_left; i++)

    left[i] = array[lower + i];


  for (j = 0; j < len_right; j++)

    right[j] = array[middle + j];


  i = 0;

  j = 0;

  k = lower;


  while (i < len_left && j < len_right) {

    if (left[i] <= right[j]) {

      array[k] = left[i];
      i++;
    }
    else {
      array[k] = right[j];
      j++;
    }

    k++;
  }

  while (i < len_left) {
    array[k] = left[i];
    i++;
    k++;
  }

  while (j < len_right) {
    array[k] = right[j];
    j++;
    k++;
  }
}
```

- Another example: Generate permutations of a set of elements. n! Permutations.

```
#include <stdio.h>

#define LEN 4

void swap(int array[], int i, int j);
void permute(int array[], int low, int len);

int main(void) {
  int array[LEN];
  int i;

  for (i = 0; i < LEN; i++)
    array[i] = i + 1;

  permute(array, 0, LEN);
}

void permute(int array[], int low, int len) {
  int i;

  if (low == len) {
    for (i = 0; i < len; i++)
      printf("%d ", array[i]);
    printf("\n");
  }
  else {
    for (i = low; i < len; i++) {
      swap(array, low, i);
      permute(array, low+1, len);
      swap(array, low, i);
    }
  }
}


void swap(int array[], int i, int j) {
  int temp = array[i];
  array[i] = array[j];
  array[j] = temp;
}
```

        - Idea of using recursion to solve this.
            - Breaking task into sub-tasks.
            - Print one permuted array if we already fixed first i elements.
            - Permute(A,k,n) where generate permutation of array from element k to element n-1 (subscript starting at 0).
                - If k == n, print permutation and return.
                - For i in k, k+1, ..., n-1.
                    - Swap A[i] with A[k].
                    - Permute(A,k+1,n)
                    - Swap A[i] with A[k].
- Program Organization
    - Local Variables
        - Defined inside the body of a function.
        - ***Automatic storage duration***: memory storage is automatically allocated when the

enclosing function is called, and deallocated when it returns. Think: Stack frame.
    - ***Block Scope***: Visible (can use) from the point of declaration to the end of the enclosing function (body).
    - Parameters: Automatic, block.

      Reading: 9.5 – 9.6, 10.1
      Next: Rest of Chapter 10, 11.
      Lab this afternoon. Useful: helps practice debugging, C.

- External Variables
    - Declared outside any function. Includes main function.
    - These variables have ***static storage duration***: permanent memory storage location. Always stored in the same place so values can be retained throughout the program execution. Only deallocated when program finishes execution.
    - Remember that process memory is comprised of data, heap, stack, etc. External variables are contained in the data part of process memory (permanent data).
    - ***File Scope***: Visible (can use) from declaration to the end of the enclosing file.
- Suggested Layout for organizing a C program found in a single file.
    - #include for imports.
    - #define for macros.
    - Type definitions (defining a type).
    - External variables.
    - Function prototypes (except main).
    - Main function.
    - Function definitions (does not include main).
- Example:

```
// Converts a decimal number for a binary number.
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

#define STACK_SIZE 100

typedef int Bit;

Bit contents[STACK_SIZE];
int top = 0;

void make_empty(void);
bool is_empty(void);
bool is_full(void);
void push(Bit i);
Bit pop(void);
void stack_overflow(void);
void stack_underflow(void);
```

```
int main(void) {

        int decimal;
        Bit bit;
        printf("Enter a decimal: ");
        scanf("%d",&decimal);
        while(decimal>0) {
                bit = decimal%2;
                push(bit);
                decimal /= 2;

        }

        ...

        Bit pop(void) {
                if (is_empty())
                        stack_underflow();
                else
                        return contents[--top];
        }

        void stack_overflow(void) {
                printf("Error: stack overflow!\n");
                exit(EXIT_FAILURE);
        }

        ...

}
```

- The exit function is a function in stdlib.h.
    - There are two popular return codes: EXIT_SUCCESS and EXIT_FAILURE. Macros. Values are implementation-defined.
    - In most compilers, they are 0 and 1 respectively. Effect is the same as using the return function in the main function but can be called anywhere.
- *Avoid using external varibles unnecessarily*.
    - Hard to maintain/debug.
    - Makes it difficult to reuse code.
- Blocks/Compound Statements
    - Not something new. For example, in the if statement – grouped using curly braces.
    - Inside these curly braces we can also have declarations as well as statements.
    - For variables declared in blocks:
        - Storage is ***automatic***.
            - Allocated when enter the block.

        - Deallocated when exit the block.
    - Are ***block scope***. Scope to end of block.
- Scope
    - *Rule*: When blocks are nested (block used loosely here – can be a function or file), declarations in inner blocks hide those in outer blocks.
    - In other words, two variables with same name in two nested blocks – the more inner-nested variable overwrites the other.
    - For example:

```
1     int i;
2     void f(int i) {
3           i = 1;
4
5           if(i>0){
6                 int i;
7                 i = 4;
8           }
9           i = 14;
10    }
11
12    void h(void) {
13          i = 5;
14    }
```

        - The variable call at 3 uses the declaration at 2.
        - The variable call at 7 refers to 6.
        - The variable call at 9 uses 2.
        - The variable call at 13 uses the declaration at 1.

- ***Pointers***
    - Very important feature in C.
    - Essential to memory management.
    - Allows you to refer to a variable by its memory location. Can treat memory as a sequence of bytes (refered to by unique memory addresses).
    - Pointer Variables
        - A pointer is a variable. Special because it stores a memory address.
        - Declarations: `type *pointer_name;`
        - The variable called pointer_name is now a pointer that can only store memory addresses for those variables of the declared type; data of said type.
        - For example: `int *p; char *p; int **p; // The last is a pointer to pointer to ints.`
        - Every pointer can point only to data of a particular type (its reference type).
        - To store the address of a variable, we use the address operator: &. Putting it before any variable obtains that variable's memory address.
    - For example:

```
int i, *p;
```

```
p = &i;        // Memory address of i stored in p.
               // In other words, p points to i.

int j;

int *p = &j;
```

- We can use the value of a pointer to refer to the data stored at the address it points to.
  - Use the Indirection Operator *.
  - Allows you to access the variable that a pointer points to.
  - For example:

    ```
    int i = 5;
    int *p = &i;
    (*p)++;
    printf("%d %d\n",i,*p);
    ```

  - This will show: `6 6`.
- Warning: This is the source of many bugs in programs.
  - If a pointer p is not initialized (not pointing to anything), accessing `(*p)` will access some random memory address.
  - Results in all sorts of different problems. Behavior is undefined.

End of content for midterm 2.
Reading: 10.3 – 10.5, 11.1 – 11.3
Next: rest of chapter 11, 12.
Programming Exercise: Proj 1, p. 238.

- Pointer Assignment
  - The = operator copies pointers of *the same type*.
  - This copies the memory address associated with that pointer, stored in that pointer variable. Still consistent with before; assigning values.
  - For example:

```
int i = 5, j = 10;
int *p = &i;
int *q;
int *r = &j;
*r = *p; // j = i. Not pointer assignment.
q = p; // Pointer assignment.
(*q)++; // i++
printf("%d %d %d %d %d", i, j, *p, *q, *r);
```

  - This will output: `6 5 6 6 5`
- Pointer Arguments
  - Pointers allow us to do many tasks in C related to memory management.

- For example, we can modify multiple variables in the *caller*.
- Pointers tell us the location of variables. So, passing the address of variables to a function means the function knows the location of variables assigned outside of its scope.
- For example: The Swap Function – actually working!

```
void swap(int *a, int *b) {

        int temp;
        temp = *a;
        *a = *b;
        *b = temp;

}
```

- So, say, somewhere in the main function:

```
int a = 4, b = 5;
swap(&a,&b);
printf("%d %d\n", a, b);
```

- The output is: 5 4
- Projector Example: Getting statistics from an array. Computing min, max, arg, stddev – statistics.c.

```
int main() {
        ...
        statistics(array, LEN, &min, &max, &average, &stddev);
        // Using a backslash at the end of a string literal line is the same as
putting it on one line.
        ...
}

void statistics(double array[], int len, double* min, double* max, double*
average, double* stddev) {
        *min = array[0];
        *max = array[0];
        *average = array[0];
        ...
}
```

- The reason for the & operator in the scanf functions.
  - Parameters are actually declared as pointers.
  - Address of variable is passed to it; stores value according to memory location.

- Pointer Arithmetic
  - **For pointers that point to array elements only**. Each element of an array has an address.

  - For example, adding an integer to a pointer.
    - If p points to a[i].
    - The expression p + j points to a[i+j].
    - For example, the below outputs 0 1:

```
int a[10] = {9}
int *p = &a[1];
(*(p+3))++;
printf("%d %d\n",a[1],a[4]);
```

  - Subtracting an integer to a pointer.
    - Similar.
    - If p points to a[i], then p – j points to a[i-j]
  - Subtracting one pointer from another.
    - They point to elements of the **same** array.
    - If p points to a[i], and q to a[j], then p – q is equivalent to i – j.
    - Example – result is -5:

```
int a[10];
int *p = &a[0];
int *q = &a[5];
printf("%d\n", p-q);
```

- Pointer Comparisons
  - When they point to elements of **the same array**.
  - Result of comparison: comparing the indices of elements.
  - For example (continued from above) with output 1:

```
printf("%d\n", p < q);
```

- Pointer arithmetic on pointers that do not point to array elements is *undefined*.

Midterm #2: Questions will be asked on gdb, pointers.
Reading: 11.3 – 11.5, 12.1
Next: 12.2 – 12.3, start 13.
Programming Practice: Proj 2, Page 256.