“For every problem there is one solution which is simple, neat, and wrong.” – H. L. Mencken

**Instructions:** Read carefully through the entire exam first, and plan your time accordingly. Note the relative weights of each segment as a percentage of the total exam score.

This exam is **closed book, closed notes.** You may *not* refer to any books or other materials during the exam.

Write your answers on this exam. You may use both sides of the page.

When you are done, present your completed exam and your student identification to the instructor or proctors at the head table. If leaving before the exam period is concluded, please leave as quietly as possible as a courtesy to your neighbors.

**Name:**

**Student Number:**

**Signature:**
1. (Getline - 15%)

Write a function `getline(char s[], int limit)` which uses `getchar()` to read a string of text, “s”, of max length “limit”, from stdin and returns the length of the string. You may not use any other system calls, but you may assume that array `s` is large enough to hold the string. (Hint: Be prepared to deal with lines that end in '
', or the EOF character; newline characters should be included in your returned string, but end-of-files should not.)

```c
/**
 * getline() - read a line of input into the array 's', and return the
 * size of the resulting string.
 * Parameters:
 *      char s[] An array that will contain the string read.
 *      int limit Size of array 's'.
 * Returns: int number of characters in string.
 */
int getline(char s[], int limit)
{
    int c = 0, i = 0;

    /* Read until limit reached, end of file, or newline. */
    while ( (--lim > 0) && ((c=getchar()) != EOF) && (c != \n) )
    {
        s[i++] = c;
    }

    /* If last char was the newline, include it in the string. */
    if (\n == c)
    {
        s[i++] = c;
    }

    /* Don't forget null termination! */
    s[i] = \0;

    return i;
}
```
2. (A to F - 15%) Write an `atof()` function, which takes a character string as input and returns an equivalent floating point number. You may assume the string contains no leading white space, no plus or minus symbols, no scientific notation or exponents, and that the value will fit into a float. You may use any of the standard library functions prototyped in `ctype.h`, although this is not required.

```c
#include<ctype.h>

/**
 * atof - Given a character string, returns the equivalent floating point number.
 * Parameters: char s[] - A string of characters. We assume that the number spelled out in the array is positive, and that it will fit into a 'float' type variable. There may or may not be a decimal point.
 */
float atof(char s[])
{

    float value = 0.0, power = 0.0;
    int i = 0;

    for ( value = 0.0; isdigit(s[i]); i++)
        { value = value * 10.0 + (s[i] - '0'); }
    if ('.' == s[i])
        { i++; }
    for ( power = 1.0; isdigit(s[i]); i++)
        { value = value * 10.0 + (s[i] - '0'); power *= 10.0; }
    return value / power;
}
```
3. (Calculator - 15%)

Implement a rudimentary calculator using the getline() and atof() functions from the previous two problems. (You may assume that these functions work properly, even if you suspect yours do not.)

Your calculator’s main program should read in lines of text using getline() until a zero-length line is read. Assume no more than 30 characters of input per line, and assume each line contains valid input. Convert each line to a float using atof() and total them up into a double variable.

Finally, your program should print the total. Assume that your implementations of getline() and atof() are in other files that will be linked in with this program at compile time, but beware of implicit function declaration.

/* My calc.c files starts here: */

#include<stdio.h>
#define MAXLINE 31
int main()
{
    double sum = 0.0;
    char line[MAXLINE];

    float atof(char s[]);
    int getline(char line[], int max);

    while (getline(line, MAXLINE) > 0)
    {
        sum += atof(line);
    }
    printf("%g\n", sum);
}
4. (Palindrome - 15%) 

A palindrome is a string of letters or numbers that is the same backward as forward. Given the program below, which reads in a single string from the command line, write the palindrome function. You may NOT use any string.h functions.

#include<stdio.h>  
/* Prototype for palindrome. You will write this function. */  
int palindrome(char s[]);

int main(int argc, char **argv)  
{ /* This program expects a string on the command line. If it is not */  
  /* there, (as in, there is only the program name), or if there are */  
  /* too many arguments, print an error and exit. */  
  if (argc != 2)  
  { fprintf(stderr, "Usage: %s <STRING>
", argv[0]); return(-1); }  

  printf("String \\
  is%s a palindrome.\n", argv[1], palindrome(argv[1]) ? "" : " not");
}

/**  
* palindrome - returns true if string is a palindrome.  
* Parameter: char s[] A string of characters.  
* Returns: True if s is a palindrome, false otherwise.  
*/  
int palindrome(char s[])  
{

  int low = 0, high = 0;  
  /* Send 'high' out to end of the string, to null terminator. */  
  while (s[high]) high++;

  /* Low and high close in from both ends of string. */  
  /* If low becomes >= high, we’ve traversed entire string without */  
  /* a mismatch. */  
  for (low = 0, high--; low < high; low++, high--)  
  { if (s[low] != s[high]) return 0;  
    /* If we got this far, the string is a palindrome. */  
    return 1;
  }
}
5. (Int to Bit String - 15%) Consider the code below:

```c
#define MAXBITS 32
int main()
{
    char s[MAXBITS + 1];
    unsigned int n = 0;
    int low = 0, high = 0;

    /* Input three integers, repeat until input is good. */
    do { printf("Enter integer: "); fflush(NULL); scanf("%u", &n);
         printf("Enter low bit: "); fflush(NULL); scanf("%d", &low);
         printf("Enter high bit: "); fflush(NULL); scanf("%d", &high);
    } while( (low < 0) || (high < low) || ((MAXBITS) <= high) );

    /* Call itobs() function to have bits in range high:low of n
     * put in string s as characters. */
    itobs(s, n, low, high);

    /* Print results. */
    printf("Bits %d through %d of %u == %s\n", high, low, n, s);
}
```

Example runs:

- Given "4294967295", (the largest unsigned int,) "0", and "31" as the bit range, this program outputs:

  Bits 31 through 0 of 4294967295 == 11111111111111111111111111111111

  which is all 32 one-bits.
- Given "1 0 0" as input, this program prints:

  Bits 0 through 0 of 1 == 1

  Notice that this means the low/high bit range is inclusive; we print from the lowest, zeroth bit, up through and including the zeroth bit.
- Given "1025", "10", and "12", this program prints,

  Bits 12 through 10 of 1025 == 001

  Note that the most significant bit is printed furthest left.
Implement the `itobs()` function below. Note that the string is returned through array parameter `char s[]`.

/**
 * `itobs()` - Converts an integer into a character string showing the
 * bits from 'high' down to 'low' of the integer.
 * Parameters:
 * char s[] An array that will contain the '0' and '1' characters
 * in a string that represents the range of bits requested. We
 * make no assumptions about the prior content of s, but assume
 * that the array is large enough to hold the answer.
 * int n The integer we are converting into a printable string
 * of bits.
 * int low The lowest bit in the range. Assume that this integer
 * is non-negative, and that 0 means the least-significant bit.
 * int high The highest bit in the range. Assume that this integer
 * is greater than or equal to 'low', but smaller than the number
 * of bits in an unsigned integer on this machine. If
 * high == low, we will still put one printable character into s.
 * Returns: Void. But the parameter 's' is an array reference, so
 * the real result of this function can be stored there.
 */
void itobs(char s[], unsigned int n, int low, int high)
{

    int i = 0;

    /* For each bit in the range [high..low] inclusive, we shift a single
     * '1' bit up to the correct bit position, and binary AND it with the
     * target integer. Wherever a bit is found, place a '1' character
     * into the array, otherwise put a '0' character.
     */
    for (i = 0; i <= high - low; i++)
    {
        s[i] = ((1 << (high - i)) & n) ? '1' : '0';
    }
    /* Don’t forget the string must be null-terminated. */
    s[high - low + 1] = 0;
}
6. (KNIGHTS A VISITING: 25%) Consider the code below:

```c
#define MAX 8
/* Yours! All Yours!!! */
void solve(int board[MAX][MAX], int i, int j, int piece);

/* printboard - Print out the board, showing each knight move. */
/* Assume this function is properly implemented in another file. */
void printboard(int board[MAX][MAX]);

/** knightmove - Given an (i,j) coordinate and a move number, update *
* i and j to new values for each of the eight possible *
* knight moves. This function does NOT check the new *
* coordinates to see if they are valid, that is, *
* on the board. */
void knightmove(int *i, int *j, int move)
{
    switch(move)
    { /* Calculate each of the 8 possible moves. */
        case 0: *i += 2; *j += 1; break;
        case 1: *i += 1; *j += 2; break;
        case 2: *i += -1; *j += 2; break;
        case 3: *i += -2; *j += 1; break;
        case 4: *i += -2; *j += -1; break;
        case 5: *i += -1; *j += -2; break;
        case 6: *i += 1; *j += -2; break;
        case 7: *i += 2; *j += -1; break;
    }
}

int main()
{
    int board[MAX][MAX];
    int i = 0, j = 0;

    /* Initialize board. */
    for (i = 0; i < MAX; i++) for (j = 0; j < MAX; j++) board[i][j] = 0;
    /* This loop will try solving puzzle with each possible first move. */
    for (i = 0; i < MAX; i++) for (j = 0; j < MAX; j++)
    { board[i][j] = 1; /* Place first knight. */
        solve(board, i, j, 2); /* Try to solve rest. */
        board[i][j] = 0; /* Unplace first knight. */
    }
}
```
This program generates and prints all solutions to the “Knights-A-Visiting” problem, a classic chess riddle that asks how a chess knight can move to every square on the chessboard exactly once. A solution is printed as a chessboard with an integer in each square showing which move of the knight visited that square (moves 1 through 64).

The chess knight has eight potential moves from a given square, which are best described as an “L-shape”. Note that the previous page already includes a function (knightmove()) that when called with (i,j) coordinates and a move number from 0 to 7 returns the resulting (i,j) coordinates after each of the eight possible moves. The printboard() function (not shown here) does the obvious thing and prints out the 2-D chessboard.

Finish this program by implementing the recursive solve() function. (Hint: We’re looking for a brute-force, recursive search of the possible combinations here – there are no known “clever” algorithms for finding solutions to this problem.)

```c
/**
 * solve() - recursive solver for the Knights-A-Visiting problem.
 * Parameters:
 * int board[MAX][MAX] A two-dimensional array representing
 * the chessboard. Locations contain a 0 if a square has not
 * been visited, or a positive integer indicating the n-th move
 * of the knight has visited the square.
 * int i,j Coordinates of last knight placed.
 * int piece Number of next knight move to place.
 */
void solve(int board[MAX][MAX], int i, int j, int piece)
{
    int nexti = 0, nextj = 0, move = 0;
    /* Base case: If we’re on piece 65, we’ve placed all 64. */
    if (piece > MAX * MAX) { printboard(board); return; }
    /* Recursive case: Try placing another knight, and recurse. */
    for (move = 0; move < 8; move++)
    { /* Get possible coordinates of next knight. */
        nexti = i; nextj = j; knightmove(&nexti, &nextj, move);

        if ( (nexti >= 0) && (nexti < MAX) && /* Check if i in bounds. */
             (nextj >= 0) && (nextj < MAX) && /* Check if j in bounds. */
             (board[nexti][nextj] == 0) ) /* Check if square empty.*/
        {
            board[nexti][nextj] = piece; /* Place n-th knight. */
            solve(board, nexti, nextj, piece+1); /* Recurse next knight. */
            board[nexti][nextj] = 0; /* Unplace n-th knight. */
        }
    }
}
```