C Exercises for CSE 1310 and CSE 1311

February 18, 2013

Contents

Chapter 1

Statements and `printf()`

1. `printf()` has a very large number of format specifiers. There are a number of options that can be used with the format specifiers for altering the way that information is displayed. Run the following program with different values for the format specifiers (sorry about using loops before discussed in class, but it makes this much simpler).

```
#include <stdio.h>

int main(void)
{
    int i;
    double x;

    for(i = 1; i < 10000; i *= 10)
        printf("%d\n", i);

    for(x = 1.234; x < 10000; x *= 10)
        printf("%lf\n", x);
}
```

   For the first `printf()`, some values to try are `%2d` and `%4d`. For the second `printf()`, some values to try are `%2lf`, `%4.2lf`, and `%8.5lf`.

2. Write a program that prompts a user for an integer and then evaluates the following mathematical function for that integer:

```
f(x) = 10 - 18x + 6x^2
```

Chapter 2

Conditionals

1. Compare the output from the following programs. Why is it different?

```
#include <stdio.h>

int main(void)
{
    int x = 8;

    if(x < 6)
        printf("%d < 6\n", x);
    else if(x < 8)
        printf("%d < 8\n", x);
    else if(x < 10)
        printf("%d < 10\n", x);
    else if(x < 12)
        printf("%d < 12\n", x);
    else if(x < 14)
        printf("%d < 14\n", x);
}
```

and

```
#include <stdio.h>

int main(void)
{
    int x = 8;

    if(x < 6)
        printf("%d < 6\n", x);
```

```
    if(x < 8)
        printf("%d < 8\n", x);
    if(x < 10)
        printf("%d < 10\n", x);
    if(x < 12)
        printf("%d < 12\n", x);
    if(x < 14)
        printf("%d < 14\n", x);
}
```

2. The following program doesn't print anything. Why? Using only a set of curly braces, change the program such that the second `printf()` statement will print.

```
#include <stdio.h>

int main(void)
{
    int age = 40;

    if(age > 50)
        if(age < 75)
            printf("first printf\n");
        else
            printf("second printf\n");
}
```

Chapter 3

Loops

1. Rewrite the following program, replacing the `for` loop with a `while` loop.

```
#include <stdio.h>

int main(void)
{
    int i;

    for(i = 8; i > 5; i--)
        printf("%d\n", i);
}
```

2. Rewrite the following program, replacing the `do-while` loop with a `for` loop. This may require more than just changing the loop.

```
#include <stdio.h>

int main(void)
{
    int x = 5;

    do
    {
        printf("%d\n", x);
        x++;
    }
    while(x < 5);
}
```

3. Write the code to sum the integers 7 through 11 inclusive (i.e., include 7 and 11). Use a loop to do it.

4. You have a program that prompts a user for a positive integer, which is stored in the variable `stop`. Write the code that will sum the integers from 1 to `stop` and then print out the answer. For example, if the user entered 3, the answer printed out would be 6 since $1 + 2 + 3 = 6$.

5. You have a program that prompts a user for two integers, which are stored in the variables `start` and `stop`. Write the code that will go from `start` to `stop`, printing "x is Even" if the number is an even number and "x is Odd" if the number is odd, replacing `x` with the integer value.

6. Write a program that prompts the user for an integer n and determines if n is prime.

7. Write a program that finds all prime numbers in the range of 17 to 91.

8. Write a program that prints the integers from 1 to 100 as a 10×10 table with the values right-aligned within their column. The results will be

```
  1   2   3   4   5   6   7   8   9  10
 11  12  13  14  15  16  17  18  19  20
 21  22  23  24  25  26  27  28  29  30
 31  32  33  34  35  36  37  38  39  40
 41  42  43  44  45  46  47  48  49  50
 51  52  53  54  55  56  57  58  59  60
 61  62  63  64  65  66  67  68  69  70
 71  72  73  74  75  76  77  78  79  80
 81  82  83  84  85  86  87  88  89  90
 91  92  93  94  95  96  97  98  99 100
```

9. Write a program that produces the integers in the range of 1 to 100, but only prints those that are prime. If the number produced is not prime, print spaces instead. Everything printed should result in a 10×10 table of either values printed or spaces. Basically this is problem 8 with only the prime numbers printed. The results will be

```
      2   3       5       7
 11      13              17      19
         23                      29
 31                      37
 41      43              47
         53                      59
 61                      67
 71      73                      79
         83                      89
                         97
```

Chapter 4

Functions

1. Write a function declaration for a function with the name `calcAge` that will receive a variable of type `int` (i.e., function parameter) and has a return type of `float`.

2. The following program makes a function call to a function called `multNums`. `multNums` takes two arguments, multiplies them together, and then returns the result. Write the function declaration and function definition for `multNums`. The return type and variable types that you write for `multNums` should match those of the variables involved in the function call of the sample program. Your function definition will include whatever code is necessary to perform the calculation.

```
#include <stdio.h>

/* function declaration goes here */

int main( void )
{
    int hours = 3;
    float rate = 5;
    float cost;

    cost = multNums(hours, rate);

    printf("cost is %2.f\n", cost);
}

/* function definition goes here */
```

3. What does the following program print?

```
#include <stdio.h>

int alpha(int a);
int beta(int b);

int main(void)
{
    int i;

    for(i = 12; i < 20; i = i + 3)
        if( alpha(i) && beta(i) )
            printf("%d\n", i);

}

int alpha(int a)
{
    if(a%3 == 0)
        return 1;
    else
        return 0;
}

int beta(int b)
{
    if(b%2 == 0)
        return 1;
    else
        return 0;
}
```

4. What does the following program print?

```
#include <stdio.h>

int delta(int a);
int epsilon(int b);

int main(void)
{
    int i;

    for(i = 1; i < 5; i++)
        printf("%d\n", delta(i) );

}

int delta(int a)
{
    int x;

    x = epsilon( 3 * a );

    return x;
}

int epsilon(int b)
{
    return 2 * b;
}
```

5. What does the following program print?

```
#include <stdio.h>

int alpha(int a);
int beta(int b);

int main(void)
{
    int i = 1;
    int x = 0;

    while( i < 5 )
    {
        x = x + alpha( beta(i) );
        printf("x is %d\n", x);
        i++;
    }
}

int alpha(int a)
{
    return 3 * a;
}

int beta(int b)
{
    if(b%2 == 0)
        return 1;
    else
        return 0;
}
```

6. Write a function that when given a positive integer x, determines how many of the divisors of x are evenly divisible by 3. Apply this function to the integers in the range 10 to 30.

7. Write a function that when given an integer x, returns the second smallest divisor of x (not counting 1). For example, if $x = 12$, then the function would return 3. If x does not have a second smallest divisor, then the function should return zero.

8. Write a function that receives an integer x and prints a pyramid x levels high using asterisks.

9. What does the following program print?

```
#include <stdio.h>

int beta(int b);

int main(void)
{
    int x;

    printf("%d\n", beta( 6 ) );
}

int beta(int b)
{
    switch( b )
    {
        case 4: return (5 * b);
        case 6: return (b + 10);
        case 8: return (b - 2);
        case 9: return (3 * b + 10);
    }

    return b;
}
```

10. The program below calls a function, `squared()`, that when given an integer x squares it n times. For example, if $x = 3$ and $n = 4$, then the function should calculate $((((3)^2)^2)^2)^2 = 43046721$. Write the function `squared()`.

```
#include <stdio.h>

int squared(int x, int n);

int main(void)
{
    int i, x, n;

    printf("Enter positive integer to be squared: ");
    scanf("%d", &x);

    printf("Enter number of times to square: ");
```

```
    scanf("%d", &n);

    printf("%d squared %d times is %d\n", x, n, squared(x, n) );
}
```

11. Write a function that when given two integers, x and y, returns the maximum divisor of x or y, excluding x or y themselves. For example, if the two integers are 8 and 9, the divisors of 8 (excluding 8) are 1, 2, and 4 while the divisors of 9 (excluding 9) are 1 and 3. The maximum is 4.

12. Modify problem 3.7 to use a function to find the prime numbers. The function will be given a single integer, n, and return 1 if n is prime and 0 otherwise.

13. Modify problem 3.9 to use your function for determining if n is prime.

Chapter 5

Arrays

1. The program below generates and stores the integers 0–4. Modify the program in two ways:

   (a) Change loop 2 so that it only prints the odd integers.

   (b) Undue the first changes, then modify loop 1 so that it only stores the odd integers.

```
#include <stdio.h>

int main(void)
{
    int data[5];
    int i;

    /* loop 1 */
    /* store the integers 0 through 4 */
    for(i = 0; i < 5; i++)
        data[i] = i;

    /* loop 2 */
    for(i = 0; i < 5; i++)
        printf("%d ", data[i]);
}
```

2. Write a program that will create an array to store the numbers 5, 6, 7 and then print out the contents of the array.

3. Write a program that will generate and print the integers 20–25, but in reverse order. That is, the program will print `25 24 23 22 21 20`. Then modify the program to store the integers in an array as you generate them, to be printed later.

4. For the program below, write the code that will store the values in `array1` in `array2` in reverse order. The output that should be produced is shown.

```
#include <stdio.h>

void printArray(int [], int);

int main(void)
{
    int array1[5] = {17, 23, 9, 4, 12};
    int array2[5];
    int length = 5;
    int i;

      /* code to store array1 in array2 in
         reverse order goes here            */

    printf("array1: ");
    printArray(array1, length);
    printf("array2: ");
    printArray(array2, length);
}

void printArray(int data[], int length)
{
    int i;

    for(i = 0; i < length; i++)
        printf("%2d ", data[i]);

    printf("\n");
}

/***********************
***      Output      ***

array1: 17 23  9  4 12
array2: 12  4  9 23 17

***********************/
```

5. Modify the program below to swap the elements in `array1` and `array2`.

```
#include <stdio.h>

void callPrintArray(int [], int [], int);
void printArray(int [], int);

int main(void)
{
    int array1[5] = {17, 23, 9, 4, 12};
    int array2[5] = {50, 75, 81, 13, 6};
    int temp;
    int length = 5;
    int i;

    printf("before swapping\n");
    callPrintArray(array1, array2, length);


      /* code for swapping goes here */


    printf("\nafter swapping\n");
    callPrintArray(array1, array2, length);
}

void callPrintArray(int array1[], int array2[], int length)
{
    printf("array1: ");
    printArray(array1, length);
    printf("array2: ");
    printArray(array2, length);
}

void printArray(int data[], int length)
{
    int i;

    for(i = 0; i < length; i++)
        printf("%2d ", data[i]);

    printf("\n");
}
```

```
/***********************
***      Output      ***

before swapping
array1: 17 23  9  4 12
array2: 50 75 81 13  6

after swapping
array1: 50 75 81 13  6
array2: 17 23  9  4 12

***********************/
```

6. Modify the looping structure of the program below so that it produces the output shown. Hint: this can be done by changing only two lines.

```
#include <stdio.h>

int main(void)
{
    int data[2][3] = { {15, 72,  9},
                       { 4,  8, 11} };
    int i, k;

    for(i = 0; i < 2; i++)
    {
        for(k = 0; k < 3; k++)
            printf("%2d ", data[i][k]);

        printf("\n");
    }
}

/*****************
***   Output   ***

15  4
72  8
 9 11

*****************/
```

7. Write the code for the function that will cause it to print the desired row of the array.

```
#include <stdio.h>

void printRow(int row, int data[][3]);

int main(void)
{
    int data[][3] = {{5, 6,  7},
                     {8, 9, 10}};
    int row;

    printf("Which row should I print (0 or 1)? ");
    scanf("%d", &row);

    printRow(row, data);
}

void printRow(int row, int data[][3])
{
    /* your code goes here */
}
```

8. The following program will print a specified range of the array. Write the code that will take the starting and stopping index values provided by the user and use them to print only the elements in this range.

```
#include <stdio.h>

int main(void)
{
    int data[] = {5, 6, 7, 8, 9, 10};
    int start, stop;
    int i;

    printf("Which element should I start with (0--4)? ");
    scanf("%d", &start);

    printf("Which element should I stop with (1--5)? ");
    scanf("%d", &stop);

    /* your coded goes here */
}
```

9. The program below will allow the user to provide lower and upper limits on the values to be printed. Values that are not printed are replaced with `XX`. Write the code to do this. Sample output is provided.

```
#include <stdio.h>

int main(void)
{
    int data[3][3] = {{50, 61, 17},
                      {35, 41, 59},
                      {43, 49, 60}};
    int low, high;
    int i, k;

    printf("The array is \n");
    for(i = 0; i < 3; i++)
    {
        for(k = 0; k < 3; k++)
            printf("%3d", data[i][k]);

        printf("\n");
    }

    printf("Provide a range of values to print.\n");
    printf("What is the lower limit? ");
    scanf("%d", &low);

    printf("What is the upper limit? ");
    scanf("%d", &high);
    printf("\n");

    /* your code goes here */
}

/************************************
***             Output              ***
The array is
 50 61 17
 35 41 59
 43 49 60
Provide a range of values to print.
What is the lower limit? 45
```

```
What is the upper limit? 65

 50 61 XX
 XX XX 59
 XX 49 60
*/
```

10. Write a program that declares a 2D array with three columns. The program passes the array to a function that replaces each element in the center column with the sum of the other two elements on the same row. For example, if the first row of the array is

$$5, 8, 10$$

then after the call to your function this row would be

$$5, 15, 10$$

Note that this can be done with a single loop that generates the subscript of the row. Create a second function that prints a 2D array with three columns.

11. Write a program that declares a 2D array of integers. The program calls a function, passing it the array and the number of rows in the array. The function sums and prints the odd elements of each row. For example, if the first row is

$$4, 3, 7, 8$$

then the sum that would be printed is 10 (since $3 + 7 = 10$). You can hard-code the number of columns in your function, but you should use the passed-in number of rows to make your function more general.

12. Write a program that declares a 2D array of doubles. The program then prompts the user for two integers, a and b, and sums the array elements whose values are between a and b. Assume that $a < b$. You can do all of this in `main()`.

13. Write a program that declares a 1D array with n elements. The program then prompts the user for an integer, a, such that $1 \leq a \leq n$. The program should perform error checking to make sure that a is in the required range. If it is, then the program should calculate the sum of the first a elements of the array.

14. The program below uses two functions, `printArray()` and `transpose()`. `printArray()` receives a 2D array of textttints as well as the number of rows in the array and prints the array. `transpose()` receives a 3×3 array of `int`s and changes it to its transpose. The transpose of a matrix reverses the rows and columns. That is, the rows become the columns and the columns become the rows. Since we are changing the array in place, our function requires that the matrix be square, but realize that the definition of the transpose doesn't impose this requirement. Write the two functions.

```
#include <stdio.h>

int main(void)
{
    int data[][3] = {{1, 2, 3},
                     {4, 5, 6},
                     {7, 8, 9}};

    printf("before\n");
    printArray(data, 3);
    transpose(data);
    printf("\nafter\n");
    printArray(data, 3);
}
```

Output:

```
before
  1  2  3
  4  5  6
  7  8  9

after
  1  4  7
  2  5  8
  3  6  9
```

15. The program below calls a function, `swap()`, that receives two 2D arrays with three columns each as well as an `int` that represents the number of rows in both arrays (both arrays should have the same number of rows). The function then compares the elements at the same location in both arrays and if both are even, then it swaps the elements between arrays. Write `swap()` and the function `printArray()`, which prints an array.

```
#include <stdio.h>

int main(void)
{
    int data[][3] = {{1, 2, 3},
                     {4, 5, 6},
                     {7, 8, 9}};
```

```
    int values[][3] = {{10, 10, 10},
                       {20, 20, 20},
                       {30, 30, 30}};
    int rows = sizeof( data ) / sizeof( data[0] );

    printf("before\n");
    printArray( data, rows );
    printf("\n");
    printArray( values, rows );

    swapEven(data, values, rows);

    printf("\nafter\n");
    printArray( data, rows );
    printf("\n");
    printArray( values, rows );
}
```

The output is:

```
before
  1  2  3
  4  5  6
  7  8  9

 10 10 10
 20 20 20
 30 30 30

after
  1 10  3
 20  5 20
  7 30  9

 10  2 10
  4 20  6
 30  8 30
```

16. The program below calls a function, `expand()`, that receives two arrays. One array is 3×3 and the other is 6×6. The function expands each element of the smaller array into a 2×2 block within the larger array. Write `expand()`.

Also write `printArray()`. Notice that this same function is used to print both arrays, even though each has a different number of columns. Write this function. Note that you will not be able to treat it as a 2D array, so you will need to do the pointer arithmetic yourself.

```
#include <stdio.h>

void expand(int small[][3], int large[][6], int size);
void printArray(int a[], int rows, int cols);

int main(void)
{
    int small[3][3] = {{1, 2, 3},
                       {4, 5, 6},
                       {7, 8, 9}};
    int large[6][6];
    int smallSize = 3;
    int largeSize = 6;

    printf("small array\n");
    printArray(&small[0][0], smallSize, smallSize);

    expand(small, large, smallSize);

    printf("large array\n");
    printArray(&large[0][0], largeSize, largeSize);
}
```

Output:

```
small array
  1  2  3
  4  5  6
  7  8  9
large array
  1  1  2  2  3  3
  1  1  2  2  3  3
  4  4  5  5  6  6
  4  4  5  5  6  6
  7  7  8  8  9  9
  7  7  8  8  9  9
```

17. Finish the program below by writing the code to change any number in the array to 99 if, and only if, the values immediately before and after the current number are both

even. The first and last elements of the array should not be changed. The array should be processed beginning with the element with index 1. Note that when processing a number, if the value before it has been changed to 99 that is what should be considered, not the value in the initial version of the array. Your program should still work if I changed the array values.

```
#include <stdio.h>

int main(void)
{
    int d[] = {9, 8, 2, 15, 6, 32, 10, 4};
```

Chapter 6

Memory

1. For the following program, which of the following is true? If you choose (a), (b), or (c), explain what the problem is and the solution to it.

   (a) The value produced by y = w + x will be incorrect.

   (b) The value produced by z = w - x will be incorrect.

   (c) The values produced by y = w + x and z = w - x will both be incorrect.

   (d) The values produced by y = w + x and z = w - x will both be correct.

```
/*
    on this machine, a short int is 2 bytes, an int is 4 bytes,
    and a long int is 8 bytes
*/
#include <stdio.h

int  main(void)
{
    short w = 20000;
    short x = 15000;
    short y;
    short z;

    y = w + x;
    z = w - x;

    printf("w + x = %hd\n", y);
    printf("w - x = %hd\n", z);
}
```

2. Run the following program:

```
#include <stdio.h>

int main(void)
{
    double x = 0.1, sum = 0;
    int i;

     for(i = 1; i <= 100; i++)
     {
        sum = sum + x;
        printf("%10.7f\n", sum);
     }
}
```

Now change the format specifier to `%20.17f`. What this demonstrates is that not all floating point numbers can be accurately represented. This problem is based on an example in [O'L09, pg. 15].

3. Run the following program:

```
#include <stdio.h>

int main(void)
{
    short int x = 32765;
    int i;

    for(i = 0; i < 5; i++)
    {
        printf("%d\n", x);
        x++;
    }
}
```

Was the output what you expected?

4. Why does the number of bytes reported by `sizeof` in `main()` differ from the number of bytes reported in `some_function`?

```
#include <stdio.h>

void some_function(int data[])
{
```

```
    printf("here data has a size of %d bytes\n", sizeof(data) );
}

int main(void)
{
    int data[] = {1, 2, 3};

    printf("data has a size of %d bytes\n", sizeof(data) );

    some_function(data);
}

/*************************************
***             Output              ***
data has a size of 12 bytes
here data has a size of 4 bytes
*/
```

5. Write a function that receives a string which represents a type of `int`. If the string is `short`, `int`, or `long`, the function should return the number of bits that are used to represent variables of this type. If the string is not one of the three mentioned above, the function should return `-1`. The function declaration is

   ```
   int bits(char type[]);
   ```

   Your function should work correctly regardless of whether it is executed on a 16-bit, 32-bit, or 64-bit operating system.

6. Write a function that receives an integer, `b`, that represents a number of bytes. If $1 \leq b \leq 8$, the function will return the largest positive value that an `unsigned int` of `b` bytes can represent. If `b` is not in the appropriate range, the function should return `-1`. The function declaration is

   ```
   long largest(int);
   ```

   Assume that this function will be executed on a 64-bit operating system. You can use the `pow()` function which returns a `double` with the value b^e when called with `pow(b, e)`.

Chapter 7

Strings

1. Create a character array and initialize it with some text. Use a loop to move through the characters, printing each individual character. When printing the characters, try it first using `%c` as the format specifier and then again with `%d` as the format specifier. This demonstrates that we can change the way the ASCII characters are displayed even though the way the text is stored in memory remains the same.

2. Change the program below to print `I love programming.` You should do this by using the values in `lovetext` to change `hatetext`. Hint: think about the relationship between the index values of the letters in `love` and the index values for the word `hate`. This can be done without creating any new variables.

```
#include <stdio.h>

int  main(void)
{
    int i;
    char hatetext[] = "I hate programming.";
    char lovetext[] = "love";

    /* Your code goes here. */

    printf("%s\n", hatetext);
}
```

3. What does the following program print?

```
#include <stdio.h>

int  main(void)
{
```

```
            /*          1         2         3         4 */
            /* 12345678901234567890123456789012345678901 */
    char text[] = "D   kfa i ifr s nfi   gfn   .";
    int i = 0;

    while(text[i] != '\0')
        if(text[i] == 'f')
            printf("\n", text[i++]);
        else
            printf("%c", text[i++]);
}
```

4. The following program has some bugs. What are they?

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

int main(void)
{
    char s[] = "Bob,53";
    char *token1, *token2, *del = ",";

    token1 = strtok(s, del);
    token2 = atoi( strtok(NULL, del) );

    printf("%s is %d years old.\n", token1, token2);
}
```

5. The following program has some bugs. What are they?

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

    int main(void)
    {
        char x1[] = "cat";
        char x2[] = "cadillac";

        if( strcmp(x1[0], x2[0]) == 0 )
            printf("the same");
        else
```

```
            printf("different");
    }
```

Chapter 8

Pointers

1. What format specifiers (in order) should be used in the `printf()` statement in the following program? Note that in the program the correct format specifiers have been replaced by Z.

```
#include <stdio.h>

int main(void)
{
    int x = 5;
    int *x_ptr = &x;

    printf("%Z, %Z, %Z, %Z\n", x, *x_ptr, &x, x_ptr);
}
```

   (a) %f, %p, %d, %p

   (b) %d, %d, %p, %p

   (c) %d, %p, %d, %p

   (d) %p, %d, %d, %p

   (e) %p, %p, %d, %d

2. We have the following code snippet

```
int main(void)
{
    int x = 5;
    int *ptrA = &x;
    int **ptrB = &ptrA;
}
```

What value does each of the following represent? If the answer is an address, state such and include which variable the address is for.

(a) x

(b) ptrA

(c) ptrB

(d) *ptrA

(e) *ptrB

3. On a machine in which addresses are 4 bytes, what is printed by the following program:

```
#include <stdio.h>

int  main(void)

                    /*          1         2         3         4 */
                    /* 12345678901234567890123456789012345678901 */
    char fun[]      = "Programming is fun.";
    char favorite[] = "My favorite class is programming.";
    char *x = fun;

    printf("%d\n", sizeof(fun));
    printf("%d\n", sizeof(favorite));
    printf("%d\n", sizeof(x));
}
```

4. Rewrite the following program, replacing the subscript notation with the corresponding pointer notation. Only change the parts of the program necessary for the replacement.

```
#include <stdio.h>

int main(void)
{
    int data[] = {1, 2, 3, 4, 5, 6};
    int i;

    for(i = 0; i < 6; i++)
        printf("%2d", data[i]);
}
```

5. What does the following program print?

```
#include <stdio.h>

int main(void)
{
    int data[] = {1, 2, 3, 4, 5, 6, 7};
    int *ptr = data;
    int i;

    printf(" i  *ptr\n--------\n");
    for(i = 2; i > -4; i--)
    {
        printf("%2d,  %2d\n", i, *ptr);
        ptr+=i;
    }
}
```

6. Rewrite the following program such that the function has a return type of `void` and the variable `y` gets its value using pointers.

```
#include <stdio.h>

int dbl(int num);

int main(void)
{
    int x = 13;

    x = dbl(x);

    printf("x doubled is %d\n", x);
}

int dbl(int num)
{
    return 2*num;
}
```

7. Write a program that uses pointers to pass a variable `x` by reference to a function where its value is doubled. The return type of the function should be `void`. The value of the variable should be printed from `main()` after the function call. The original value of `x` is 5.

8. Write a program that has a function that when passed a string will print every fourth character of the string. In `main()` you should create the string "This is an example string." and pass this to your function to be printed.

9. Write a program that creates an array of pointers to the strings "Programming" "is" and "fun." and then uses a loop to print out these strings such that the output is

   ```
   Programming is fun.
   ```

   Make sure that the output includes a space between each word, but do not include the space in your strings.

10. In this program is a function call to a function called `getRange()`. `getRange()` is passed the addresses of two variables, `low` and `high`, as well as a variable called `mid`. `getRange()` calculates a value for `low` of 75% of the value of `mid` and a value of 125% of the value of `mid` for the value of `high`. The values of all three variables are printed from main() after the function call. Using pointer notation, write the code for `getRange()` including the function definition.

    ```
    #include <stdio.h>

    void getRange(double *, double *, double);

    int main(void)
    {
        double low, high;
        double mid = 40;

        getRange(&low, &high, mid);

        printf("low = %5.2f, mid = %5.2f, high = %5.2f\n", low, mid, high
    }
    /*********************************************
    ***                 output                  ***

    low = 30.00, mid = 40.00, high = 50.00

    */
    ```

11. Using pointers, write a program that reverses the string

    ```
    ?krow siht diD
    ```

    and prints it out. Don't hard code the length.

12. Write a program that prints the diagonal (i.e., 10 50 90) of the following array.

```
int data[3][3] = {{10, 20, 30},
                  {40, 50, 60},
                  {70, 80, 90}};
```

    To do this you should create a function and pass the array to it to be processed and printed. Your function should use a loop and pointers to navigate the array. Don't hard-code the output.

13. Write a program in which you declare the following array in main()

```
int data[3][3] = {{11, 19, 12},
                  {10, 14, 15},
                  {18, 13, 12}};
```

    Pass the array to a function that prints the elements that are less than 15. Your function should use a loop (or multiple loops if you use a nested `for` structure) and pointers to navigate the array. Don't hard-code the output.

14. I wrote the following program to report which numbers mod 4 have a remainder of 1. Why does it give the output shown?

```
#include <stdio.h>

int main(void)
{
    int data[] = {4, 5, 6, 7, 8, 9, 10};
    int i;
    int div4;

    for(i = 0; i < 7; i++)
    {
        div4 = *(data + i)%4;
        if( div4 = 1 )
            printf("%2d %% 4 has a remainder of 1\n", *(data + i) );
    }
}

/************************************************
****                Output                   ****

 4 % 4 has a remainder of 1
```

```
 5 % 4 has a remainder of 1
 6 % 4 has a remainder of 1
 7 % 4 has a remainder of 1
 8 % 4 has a remainder of 1
 9 % 4 has a remainder of 1
10 % 4 has a remainder of 1

*/
```

15. The following program prints out the elements of the array. Why does this work since there is no row offset in the expression in the `printf()` statement?

```
#include <stdio.h>

int main(void)
{
    int data[3][4] = { {11, 12, 13, 14},
                       {15, 16, 17, 18},
                       {19, 20, 21, 22} };
    int i;

    for(i = 0; i < 12; i++)
        printf("%d ", *(*data + i) );
}
```

16. The following program is supposed to print out all of the array elements but instead gives the output shown below. Explain why the program works the way it does and how to fix it.

```
#include <stdio.h>

int main(void)
{
    int data[3][4] = { {11, 12, 13, 14},
                       {15, 16, 17, 18},
                       {19, 20, 21, 22} };
    int r, c;

    for(r = 0; r < 3; r++)
        for(c = 0; c < 4; c++)
            printf("%d ", (*(*data + r) + c) );
}
```

```
/*************************************************
****                Output                    ****

11 12 13 14 12 13 14 15 13 14 15 16

*/
```

17. Write a program that declares a 1D array of integers. The program calls a function, passing it the array, the number of elements in the array, and the addresses of two variables, `evensum` and `oddsum`, for storing integers The function produces two sums: the sum of the even integers in the array data and the sum of the odd integers in the array data. Pointers are used to update the values of `evensum` and `oddsum` declared in `main()`.

18. Write a program that creates a 1D array of integers. It should then prompt the user for an integer, n. Call a function, passing it the array, the size of the array, the value the user entered, and the addresses of two variables, `upper` and `lower`. The function will determine how many of the array elements are less than or equal to 25% of n and use pointers to assign this value to `lower`. The function will determine how many of the array elements are greater than or equal to 75% of n and use pointers to assign this value to `upper`.

19. Write a program that declares a 1D array of integers and also prompts the user for an integer n. The program then calls a function, passing it the array, the number of elements in the array, the integer n that the user entered, and the address of a variable called `count`. The function will determine the number of integers in the array greater than n and uses a pointer to store this count in the variable `count` declared in `main()`. After calling the function, the value of `count` should be printed from `main()`.

20. Store the values

    ```
    5, 8, 2, 3, 8, 9
    ```

    in an array in `main()`. In `main()` you will also prompt the user for an integer, x. Write a function that will receive the array, the number of elements in the array, and x. The function will determine how many values from the end of the array the value x occurs for the first time. Assume that the user always enters a value for x that occurs in the array.

    Write two versions of your function: one version that uses subscript notatation, and one version that only uses pointer notation. Example output is:

```
Number to look for: 5
5 is 5 values from the end
5 is 5 values from the end

Number to look for: 9
9 is 0 values from the end
9 is 0 values from the end

Number to look for: 8
8 is 1 values from the end
8 is 1 values from the end
```

21. In `main()`, store a set of values. Create a function that when given the array and the number of array elements, can reverse the order of the array elements. Also create a function for printing the array elements. For each of these two functions, create three versions:

    (a) subscript notation version

    (b) pointer notation version

    (c) version that uses one pointer that starts at the beginning of the array and another that starts at the end of the array. Use these two different pointers to swapping the array elements. For this version, the function to print will only have single pointer that moves down the array.

    Sample output:

```
original array
19 20 21 22 23 24
reversed
24 23 22 21 20 19
reversed again
19 20 21 22 23 24
reversed a third time
24 23 22 21 20 19
```

22. Write a program that can print, count, and sum the divisors of the integers in the range of 10 to 20. In `main()` you will create two variables, `count` and `sum`. In `main()` you will also generate each of the divisors in the range of 10 to 20 and pass them individually to a function while also passing the addresses of `sum` and `count`. The function will print the divisors of the integer passed as well as count and sum the divisors of this particular integer. After all of the divisors of this integer have been found, from `main()` you will print the count and the sum.

Sample output:

```
10:  1, 2, 5, 10 | count = 4, sum = 18
11:  1, 11 | count = 2, sum = 12
12:  1, 2, 3, 4, 6, 12 | count = 6, sum = 28
13:  1, 13 | count = 2, sum = 14
14:  1, 2, 7, 14 | count = 4, sum = 24
15:  1, 3, 5, 15 | count = 4, sum = 24
16:  1, 2, 4, 8, 16 | count = 5, sum = 31
17:  1, 17 | count = 2, sum = 18
18:  1, 2, 3, 6, 9, 18 | count = 6, sum = 39
19:  1, 19 | count = 2, sum = 20
20:  1, 2, 4, 5, 10, 20 | count = 6, sum = 42
```

23. Store the values

    ```
    1, 2, 3, 4, 5, 6
    ```

    in an array. Create two pointers: one that points to the first element of the array and anothr that points to the last element of the array. Use the pointers to print the array values in an alternating fashion:

    ```
    1 6 2 5 3 4
    ```

    Do this by creating an infinite loop that only stops when the pointer have reached each other. Be aware that whether there are an even or odd number of array elements and when you increment and decrement the pointers can make a difference as to whether the two pointers will ever have the same value.

24. added 6/29/2012 We have the following function declaration:

    ```
    int fx(int *p);
    ```

    This function receives an array of type `int`. All but the last element will be nonnegative; the last element will be negative, which can be used to determine when to stop processing array elements. Write `fx()`, which prints the array elements. Do this by using the pointer `p`. Do no create any additional variables, just increment the pointer to the next array element as needed.

25. The program below calls a function, `fx()`, that when given an array of strings will put the address of the shortest string in the first position (i.e., the one with index 0) and the address of the longest string in the last array position. Assume that the

array will not be used after printing the shortest and longest strings and therefore the other strings in the array are unimportant. Write `fx()` without hard-coding it to this particular set of strings. The types of the function parameters should match their use in the program. You do not have to perform any error-checking.

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

int main(void)
{
    char* t[] = {"horse", "elephant", "cat", "rabbit"};
    int n;

    n = sizeof( t )/ sizeof( t[0] );
    fx(t, n);

    printf("shortest is %s, longest is %s\n", t[0], t[n-1]);
}
```

Chapter 9

File I/O

1. You have a file, `payments.txt`, that contains a list of people and the amount of payments made by each person.

   ```
   Bob 250
   George 300
   Suzanne 250
   Tony 200
   ```

   Write a program that can read each line, printing the name and payment of each line. After all of the lines have been printed, print the sum of all payments.

2. You have a file, people.txt, with an unknown number of lines and four strings per line, with the strings separated by spaces. The size of the strings can differ from line to line, but none will ever contain more than 30 characters and all of them plus the spaces separating the strings will not total more than 100 characters.

   Write a program that can read each line of the file and, if the second string is `Smith`, write the line with the order of the strings reversed to the file smith.txt. Otherwise, do nothing with the line.

3. Write a program that can read a file, `names.txt`, that contains a single name on each line. If the name begins with a capital letter, print the name. If it begins with a lowercase letter, don't print it. Each name contains less than 20 letters. You don't need to do any error checking.

Chapter 10

Structures

1. Create a structure that could be used to store the following book information:

```
The C Programming Language, 2nd Edition
by Brian W. Kernighan and Dennis M. Ritchie
ISBN 0131103628
```

2. We have the following program that calls a function, `sortStruct()`, that sorts an array of structures by `name`. Write `sortStruct()`. Your function can use the constant `SIZE`, defined in the program, but no other values in the array should be hard-coded in your function.

```
#define SIZE 4    /* number of elements in the array */

#include <stdio.h>
#include <string.h>

struct names
{
    char name[20];
    int age;
};

void sortStruct(struct names *);

int main (void)
{
    struct names people[SIZE] = { {"Tom", 34}, {"Dick", 51},
                                  {"Jane", 29}, {"Harry", 48} };
    int i;
```

```
    printf("  Original\n\n");
    for(i = 0; i < SIZE; i++)
        printf("%s is %d years old\n", people[i].name,
                                       people[i].age);

    sortStruct(people);

    printf("\n  Sorted\n\n");
    for(i = 0; i < SIZE; i++)
        printf("%s is %d years old\n", people[i].name,
                                       people[i].age);
}

/*********************************************************
*****                      Output                    *****

  Original

Tom is 34 years old
Dick is 51 years old
Jane is 29 years old
Harry is 48 years old

  Sorted

Dick is 51 years old
Harry is 48 years old
Jane is 29 years old
Tom is 34 years old

*/
```

3. Write a program that stores the following information in an array of structures and then uses a pointer to the array to print the names of the mammals in the array:

```
zebra, mammal
gorilla, mammal
shark, fish
vulture, bird
rhino, mammal
```

   Your program should use the string library function `strcmp` to determine which animals are mammals.

4. added 6/29/2012 Create a structure definition for storing a person's first name and their age. Then write a program that creates an array of three structures and poplulates the values by prompting the user using a loop.

5. added 6/29/2012 Write a program that creates an array of structures that is initialized to 10 structures. The array will be populated by reading the following CSV file:

```
Ford,Mustang,1968
Honda,Civic,1988
Ford,Accord,1994
Mercury,Sable,2001
Mazda,Mazda3,2009
```

   The columns of the CSV file are `make`, `model`, and `year`. Since there are less than 10 lines, your program should keep track of how many entries are added to the array. After storing the file contents, prompt the user for starting and stopping years and print all of the cars within this range.

6. Write a program that reads `names.csv` and stores the values in an array of structures; assume that `names.csv` has no more than 100 lines. Each line of `names.csv` consists of a person's first name, last name, and age; these are separated by commas without any spaces. The program should print the last name and age of each person whose age is at least 50 and whose last name would come after "Lincoln" if sorted in ascending order. You do not have to perform any error-checking.

7. The program below calls a function, `addCar()`, that receives an array of structures representing the unique combinations of makes and models of cars as well as the make and model of a new car. The function also receives the address of an `int` that represents the count of how many array elements there are.

   Write `addCar()` without hard-coding your code to this particular data. Your function should determine if the combination of make and model it is given is already in the array. If so, do nothing. If not, add it to the next available spot in the array and update the count of array elements. The types of the function parameters should match their use in the program. You do not have to perform any error-checking.

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

struct car { char make[30];   char model[30]; };

int main(void)
{
    struct car unique[10] = {{"Ford", "Explorer"}, {"Honda", "Civic"},
```

```
                                {"Honda", "Accord"},  {"Chevy", "Malibu"}};
    int i, count = 4;

    addCar(unique, &count, "Ford", "Mustang");
}
```

Chapter 11

Bitwise Operations

1. Show the bit patterns resulting from the following operations:

   (a) ~ 0101

   (b) 0101 & 0011

   (c) 0101 ^ 0011

   (d) 0101 | 0011

2. What numbers will be produced by the following operations?

   (a) `8 << 1`

   (b) `1 << 3`

   (c) `1 << 15`

   (d) `400 >> 2`

   (e) `100 >> 1`

3. The program below prompts a user for an integer response (valid values are 1, 2, or 3) and then calls `checkResponse()`.

```
#include <stdio.h>

void checkResponse(short *, int);

int main(void)
{
    int i, response;
    short flags = 0;  /*  set all bits to zeros  */

    for(i = 0; i < 6; i++)
    {
```

```
            printf("give an integer value in the range 1 to 3:  ");
            scanf("%d", &response);
            if(response < 1 || response > 3)
                printf("That is an invalid response\n");
            else
                checkResponse(&flags, response);
        }

    }
```

`checkResponse()` prints the current response and then prints which responses have been given before. The responses are stored in the rightmost bits of `flags`. If we think of `flags` as having the following form:

```
0000000000000cba
```

then response 1 is stored in bit a, response 2 is stored in bit b, and response 3 is stored in bit c. For example, if the responses we have seen before are 1 and 3, then `flags` will have a bit pattern of:

```
0000000000000101
```

Write `checkResponse()`, which uses bitwise operations to check which bits are set to one and prints out that they have been seen before. After checking which responses have been seen before, the function should use bitwise operations to update `flags`.

Chapter 12

Miscellaneous

1. We have the following partial program, which calls `strikes()` three times.

```
#include <stdio.h>

void strikes(void);

int main(void)
{
    int i;

    for(i = 0; i < 3; i++)
        strikes();
}
```

   For the first time, it will print

```
Strike 1!
```

   For the second time, it will print

```
Strike 2!
```

   For the third time, it will print

```
Strike 3!  You're out!
```

   Write the function definition for `strikes()`. Your function definition should match the provided function declaration.

2. Write a program that prompts a user for his current direction (0 for north, 1 for south, 2 for east, and 3 for west) and responds with how he should turn to head north. Use enumerated data types to represent the directions and `if`/`else` statements instead of a `switch`.

   After checking which responses have been seen before, the function should use bitwise operations to update `flags`.

3. This problem has two parts, both of which use recursion.

   (a) Write a function that when given an integer n returns the sum of the integers from 1 to n. The function should use recursion to produce the sum. No printing will occur in the function.

   (b) The function `recursivePrint()` receives the address to the first character of a null-terminated string and uses recursion to print the string in reverse; write this function.

   The declarations are:

```
int recursiveSum(int n);
int recursivePrint(char *c);
```

Chapter 13

Dynamic Memory Allocation

1. Using `malloc()`, allocate the memory for a 1D array of type `int` that can store 5 integers. Then prompt the user for integers and store them in this array. The program should stop when either the user enters a negative integer or has provided 5 nonnegative integers. Print the array values that were stored.

2. Using `malloc()`, allocate the memory for a 1D array of type `int` that can store 5 integers. Then prompt the user for integers and store them in this array. The program should stop when the user enters a negative integer, however, if the array is full the memory should be grown using `realloc()`, doubling each time as necessary. Print the array values that were stored.

3. Using `malloc()`, allocate the memory for a 2D array of type `int` with 3 rows and 4 columns. The positions in the array should be assigned the sum of the array indices, for example, the [0][0] position will be assigned 0 + 0 = 0, the [2][3] position will be assigned 3 + 4 = 7. Print the array values that were stored.

4. We have a function, `fx()`, with the declaration of

   ```
   void fx(FILE *fp);
   ```

   where `fp` is a file pointer to a file that has been opened for reading. Each line of the file consists of a string with at most 20 characters. The number of lines is unknown, but is in the range of $1 \leq \text{lines} \leq 30$.

   Write `fx()` such that it reads the file once, storing the words. The function should print the words of the file in reverse order (i.e., last to first). All of the memory necessary to store the words should be allocated dynamically and freed before the function ends. Don't use structures and don't do something like this:

   ```
   char *words[30];
   ```

5. Write a function, `buildList()`, that reads a file of words, storing each word in a dynamically allocated array of type `char *`. The function will return the address of this array. The function declaration is:

```
char** buildList(FILE *fp, int *wordCount, int maxCount);
```

   where `fp` is a pointer to a file already open for reading, `wordCount` is the address of an `int` that stores the count of the number of words in the array, and `maxCount` is the maximum number of addresses that the array should be able to initially store. The function will create the array such that it can initially store `maxCount` addresses, but will double in size as necessary to store all of the words in the file. Each line consists of a single word and no word will require more than 50 bytes to store, including the terminating null.

6. Write `buildMatrix()`, which has the declaration

```
int** buildMatrix(FILE *fp, int n);
```

   where `fp` is a pointer to a file already open for reading. `buildMatrix()` will read the file pointed to by `fp` and from the information in the file construct an $n \times n$ adjacency matrix. The matrix will be dynamically allocated and it's address will be returned by the function. Each line of the file will consist of a comma-separated list of integers. The first integer on a line is the source node and each additional integer represents a node that the source node is connected to. For example, the first couple of lines of the file might look like this:

```
0,1,2
4,2,3
```

   Given this example, the second line says that node 4 is connected to nodes 2 and 3. Each row of your array should represent a source node and the columns will represent the destination nodes. In the matrix, a value of 0 indicates that the source and destination are not connected while a 1 indicates that they are connected.

7. Write `storeStudents()`, which has a declaration of

```
NODE * storeStudents(FILE *fp, int n);
```

   where `fp` points to a file that is already open. A sample line of the file is

```
Tom,Jones,1234
```

The function dynamically allocates space for an array of structures of size n; the structure definition is

```
typedef struct node {
    char first[50];
    char last[50];
    int ID;
} NODE;
```

Each line of the file consists of a first name, a last name, and an ID number, all separated by commas. IDs are unique. A student should only be added to the array if no one with that ID has already been added and the total number of students in the array is less than n. Assume that no line will consist of more than 125 characters. The function will return the address of the array of structures.

8. Write the function `update()`, which has a declaration of

```
void update(char *table[], char c)
```

where `table` is an array of addresses of strings for creating a hash table (strictly speaking, this isn't really a hash table). The hash table stores individual characters, with characters that hash to the same index being stored in the same string. For example, if the characters `'A'` and `'Q'` hash to the first position of the hash table and `'B'` hashes to the second position, the hash table would look like this:

```
[address] -> "AQ"
[address] -> "B"
```

`c` is the character passed to the function to be added to the hash table if it is not already in the table. The function `hash()` already exists and when given a character, returns an integer. This integer as well as the array size of `SIZE` (already created using `#define`) should be used to determine the index of where `c` should be stored. If the character is not already in the table, it should be placed in the location of the terminating null and a new terminating null added after this. Assume that the strings are large enough to hold all characters hashing to a particular location.

9. Write the function `uniqueCourses()`, which has a declaration of

```
char ** uniqueCourses(FILE *fp, int *wordCount)
```

where `fp` is a pointer to a file already open for reading. `uniqueCourses()` reads a file of courses and builds a list of the unique courses, that is, courses won't appear on the list more than once. An example line of the file is

```
CSE 1325,CSE 1320,CSE 1310
```

so it is necessary to extract the individual course names from this line. To store the course names, dynamically allocate space for an array of 50 variables of type `char *`; the function will return the address of this array. No course name will require more than 10 characters to store and no line of the file will consist of more than 100 characters. `wordCount` stores the address of a variable that keeps track of how many courses are added to the array.

10. The following program has some bugs. What are they?

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

int main(void)
{
    int rows = 2, cols = 3, r, c;
    int **d = malloc(rows *sizeof(int *));

    for(r = 0; r < rows; r++)
        for(c = 0; c < cols; c++)
            d[r][c] = r + c;

    for(r = 0; r < rows; r++)
    {
        for(c = 0; c < cols; c++)
            printf("%3d", d[r][c]);

        printf("\n");
    }
}
```

11. The following program has some bugs. What are they?

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

int main(void)
{
    int cols = 3, c;
    int d = malloc(cols *sizeof(int));

    for(c = 0; c < cols; c++)
        d[c] = 10*c;

    for(c = 0; c < cols; c++)
        printf("%3d", d[c]);
}
```

Chapter 14

Data Structures

1. In `main()` create a variable of type `int *`. Pass the address of this variable to a function that will create a dynamically allocated 1D array, whose address will be stored at the location passed in. The return type of the function will be `void`. Put some values in the array within the function, but then print them from `main()`. The point here is to see if you can successfully use a pointer in a function to update the value of a pointer variable created in `main()`. You may find it helpful to review the relationship between subscript notation and pointer notation for accessing array elements. For example, the following are equivalent:

   ```
   d[i] <==> *(d + i)
   ```

2. `stdlib.h` contains a function, `realloc()`, that will allow you to resize (in some sense) dynamically allocated memory. Dynamically create an array to store two `int`s. Repeatedly prompt the user for integers, only stopping when you get a negative integer. As long as you get nonnegative integers store them in your array, using `realloc()` to resize the array as necessary. Use the policy that you will double the amount of memory needed each time.

3. Create a doubly linked list, that is, a linked list in which each node not only keeps track of the address of the next node but also keeps track of the previous node. Use this to print the values in the linked list forward and backward.

4. Create a linked list of strings and then print the list values in sorted order by sorting the linked list after all strings have been stored.

5. Create a linked list of strings and then print the list values in sorted order by inserting new nodes such that the linked list is always sorted.

6. Create a circular linked list. In a circular linked list, the last node points to the first node, so the `next` node will never have an address of `NULL`. Note that there must still be a variable for keeping track of at least one of the nodes in the list.

7. Create a stack and use this to simulate the recursive approach to summing the integers from 1 to n. That is, instead of actually calling the function recursively, push the values onto the stack until you reach the stopping condition and then begin popping the values from the stack in order to produce the sum.

8. Create a directed graph and then after prompting the user for two nodes to use, determine if there is a path from the first node to the second.

9. Create a hash table that adds new key/value pairs to the tail end of the chains of nodes.

10. The function `build()` has the declaration

```
struct node * build(struct node *head, char *word);
```

where `head` is a pointer to the beginning of a linked-list and `word` is the address of a string. The structure definition for the nodes in the linked-list is

```
struct node {
    char name[21];
    struct node *next;
};
```

Write `build()`. It should only store the string pointed to by `word` if it is not already in the linked-list. It should also not allocate space for a new node unless it will be added to the linked-list. The function should return the address of the new node if it is created, otherwise it should return the address stored in `head`.

11. We have the following structure definition and function declaration:

```
struct node {
    int value[1000];   /*  each value[i] >= 0  */
    struct node *next;
};

double sumMax(struct node *head);
```

`sumMax()` receives the address to the beginning of a linked-list. It will find the arithmetic mean (i.e., the average) of the non-zero array values of each node and return the maximum of these averages. That is, the non-zero values in the array stored in each node have an average; `sumMax()` will find the maximum of these. Write `sumMax()`. Note that all array values are greater than or equal to zero.

12. Write `maxArcs()`, which has the declaration

```
int maxArcs(int **matrix, int n);
```

where `matrix` is a 2D array of an $n \times n$ adjacency matrix representing a **directed** graph. The function should return the index of the node that has the most incoming arcs (arcs are the connections between nodes). If there is a tie for the greatest number of incoming nodes, report the node with the smallest array index. For example, if nodes 3 and 8 each have six incoming arcs and this is the most for any nodes, report node 3 as having the most. Rows of the array represent source nodes, columns are the destination nodes.

13. Write `personList()`, which has a declaration of

```
NODE * personList(FILE *fp);
```

where `fp` points to a file that is already open. A sample line of the file is

```
Tom,Jones,1234
```

The function creates a linked list from the data using the structure definition:

```
typedef struct node {
    char first[50];
    char last[50];
    int ID;
    struct node *next;
} NODE;
```

Assume that no line will consist of more than 125 characters. The function will return the address of the head of the linked list.

14. Write the function `binarySearch()`, which has a declaration of

```
int binarySearch(int d[], int size, int q)
```

where `d` is a **sorted** array of type `int`, `size` is the number of elements in the list, and `q` is a number to be searched for in the list. If the number is in the array, return it's index; otherwise, return -1.

To search for the number, use a binary search (conceptually, the process is what you did in a homework, but it is applied to an array instead of a binary tree). In a binary search, `q` is compared to the middle array element. If `q` is equal to the middle array element, then return the index of this array element. If `q` is less than the middle element, move the search to the subset of the array to the left of the middle; if `q` is greater than the middle element, move the search to the subset of the array to the right of the middle. Each time, `q` is compared to the middle element of half of what was previously seached until `q` is found or the subset becomes too small.

For example, if we were searching the array below for the number 10, we would compare it to 17, then 8, and then 14. We would stop after this since the window of numbers is empty.

```
|2, 3, 8, 14, 17, 19, 23, 29, 31, 35|
|             17                    |
|      8     |
        | 14 |
          ||
```

15. We have an adjacency list that uses the following structure definitions:

```
struct sourceNode {
    char label;       /* character representing source node */
    struct node *destHead;
    struct node *destTail;
    struct sourceNode *next;
};
```

and

```
struct node {
    char label;       /* character representing destination node */
    struct node *next;
};
```

where `sourceNode` stores the source nodes and `node` is used to represent each destination node. The function `search()` has a declaration of

```
int search(struct sourceNode *head, char q)
```

where `head` points to the linked list of source nodes and `q` is a character to be searched for. Write `search()`, which returns the count of the number of occurrences of `q` as a destination node. For example, if the adjacency list consists of

src	dest
A	B, C
B	C
C	B

and `q` is `B`, then the function would return 2.

16. Write the function `update()`, which has a declaration of

```
void update(char *table[], char c)
```

where `table` is an array of addresses of strings for creating a hash table (this hash table only stores the keys). Each string is initially empty; that is, the terminating null is at position 0 of the array.

The hash table stores individual characters, with characters that hash to the same index being stored in the same string. For example, if the characters `'A'` and `'Q'` hash to the first position of the hash table and `'B'` hashes to the second position, the hash table would look like this:

```
[address] -> "AQ"
[address] -> "B"
[address] -> ""
```

`c` is the character passed to the function to be added to the hash table if it is not already in the table. The function `hash()` already exists and when given a character, returns an integer (which could be larger than the array size). This integer as well as the array size of `SIZE` (already created using `#define`) should be used to determine the index of where `c` should be stored. If the character is not already in the table, it should be placed in the location of the terminating null and a new terminating null added after this. Assume that the strings are large enough to hold all characters hashing to a particular location.

17. Write a program to create a linked list from the following data:

```
Ralph,30
Sally,99
Harry,20
Dick,99
Tom,10
```

where the first value of each name is stored as a string and the second is stored as an `int`. The program should have a function, `remove99()`, that removes nodes from the linked list whose `int` value is 99. That is, they should connect the nodes before and after the node to be removed and then free the memory of the node with a value of 99. You can assume that the first node in the linked list (i.e., the one pointed to by `head`) will never have a value of 99, but the last node might.

Appendix A

Answers

Answers to select problems.

A1.2

```
#include <stdio.h>

int main(void)
{
    int x, fx;

    printf("Enter an integer:  ");
    scanf("%d", &x);

    fx = 10 - 18*x + 6*x*x;

    printf("f(%d) is %d\n", x, fx);
}

/************************************************
***                 Output                    ***

Enter an integer:  3
f(3) is 10

*/
```

A3.3 An example program with its output is

```
#include <stdio.h>

int  main(void)
{
```

```
    int sum = 0;
    int i;

    for (i = 7; i <= 11; i++)
        sum += i;

    printf("the sum is %d\n", sum);
}
/*******************************
**********  Output   ***********
the sum is 45
*******************************/
```

A3.4 An example program with its output is

```
#include <stdio.h>

int  main(void)
{
    int stop;
    int total = 0;
    int i;

    printf("Please enter a positive integer: ");
    scanf("%d", &stop);

    /*****************************************************/
    /*  You would write from here                        */

    for (i = 1; i <= stop; i++)
        total += i;
        /* or you could have used total = total + i;*/

    printf("The sum from 1 to %d is %d\n", stop, total);

    /*  until here.                                      */
    /*****************************************************/
}

/*********************************************************
********                Output                 ***********
```

```
Please enter a positive integer: 10
The sum from 1 to 10 is 55
*/
```

A3.5 The program with results is

```
#include <stdio.h>

int  main(void)
{
    int start;
    int stop;
    int i;

    printf("Please enter a positive integer to start with: ");
    scanf("%d", &start);
    printf("Please enter a positive integer to stop with: ");
    scanf("%d", &stop);

    /*****************************************************/
    /*  You would write from here                        */
    for (i = start; i <= stop; i++)
    {
        if(i%2 == 0)
            printf("%d is Even\n", i);
        else
            printf("%d is Odd\n", i);
    }
    /*  until here.                                      */
    /*****************************************************/
}

/*****************************************************
*********************  Output   ***********************

Please enter a positive integer to start with: 3
Please enter a positive integer to stop with: 8
3 is Odd
4 is Even
5 is Odd
6 is Even
7 is Odd
```

```
8 is Even

*/
```

A4.1 float calcAge(int);

A4.2 `multNums` takes two arguments. The first should be of type `int` to match the variable `hours` and the second should be of type `float` to match the variable `rate`. The return type should be `float` to match the variable `cost`. The complete program is

```
#include <stdio.h>

float multNums(int, float);

int main( void )
{
    int hours = 3;
    float rate = 5;
    float cost;

    cost = multNums(hours, rate);

    printf("cost is %2.f\n", cost);
}

float multNums(int a, float b)
{
    float c;
    c = a * b;
    return c;
}
```

A4.6 We could generate all of the divisors of an integer x and for each ask if it is evenly divisible by 3. Instead, what I did was to only check if multiples of 3 are divisors of x.

```
#include <stdio.h>

int div3(int a);

int main(void)
{
    int i;
```

```
    for(i = 10; i <= 30; i++)
        printf("%d has %d divisors evenly divisble by 3.\n", i, div3(i) );
}

int div3(int a)
{
    int i, count = 0;

    /*  we only need to check the multiples of 3 */
    for(i = 3; i <= a; i = i + 3)
        if( a%i == 0 )
            count++;

    return count;
}
```

A4.7 We need to count the divisors (other than 1) of x. The count will never exceed two. When we find a divisor, we increment the count if this is the first divisor and return the divisor if it is the second divisor. If we never find a second divisor, then the function returns zero.

```
int div2nd(int x)
{
    int i, count = 0;

    for(i = 2; i <= x; i++)
        if(x%i == 0)
            if(count == 0)
                count++;
            else
                return i;

    return 0;
}
```

A4.8
```
void pyramid(int x)
{
    int i, k, diff = x;

    for(i = 0; i < x; i++)
    {
        /* print spaces */
```

```
            for(k = 1; k < diff; k++)
                printf(" ");

            /* print asterisks */
            for(k = 0; k <= x-diff; k++)
                printf("* ");

            diff--;
            printf("\n");
        }
    }
```

A4.10

```
int squared(int x, int n)
{
    int i;

    for(i = 0; i < n; i++)
        x = x*x;

    return x;
}
```

An alternative method, which uses recursion is

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

A5.2 Here is the program

```
#include <stdio.h>

int main(void)
{
    int i;
    double data[3] = {5, 6, 7};

    for(i =0; i < 3; i++)
        printf("%2.f\n", data[i]);
}
```

A7.2 The program with its results is

```
#include <stdio.h>

int  main(void)
{
    int i;
    char hatetext[] = "I hate programming.";
    char lovetext[] = "love";

    /***********************/
    /* changes start here  */

    /* the characters in hatetext have indices of
       2 through 5 while the characters in lovetext
       have indices of 0 through 3.  the key insight
       here is to realize that the index values differ
       by 2 between the corresponding characters in
       each array.  This allows you to use a loop to
       copy the value in lovetext to the appropriate
       place in hatetext.                          */

    for(i = 2; i < 2 + 4; i++)
        hatetext[i] = lovetext[i-2];

    /* changes stop here   */
    /***********************/

    printf("%s\n", hatetext);
}

/*******************************************
**************   Output   ******************

I love programming.

*/
```

A7.3

```
D   k
a i i
r s n
i   g
n   .
```

A8.1 `x` is an `int`, `*x_ptr` is the content of `x`, `&x` is the address of `x`, and `x_ptr` contains the address of `x`. Therefore, the correct answer is

```
(b) \%d, \%d, \%p, \%p
```

A8.2 The values are:

(a) `x` is the value 5

(b) `ptrA` is the address of `x`

(c) `ptrB` is the address of `ptrA`

(d) `*ptrA` is the value 5

(e) `*ptrB` is the content of `ptrA`, which is the address of `x`

A8.3 Each character of a string is one byte. Remember that strings include a terminating null, which also is one byte in size. A variable of type `char *` is a pointer to a `char`. Since it is a pointer, its content is an address and therefore `sizeof` gives us the size of the address.

```
20
34
4
```

A8.4 The pointer version is

```
#include <stdio.h>

int main(void)
{
    int data[] = {1, 2, 3, 4, 5, 6};
    int i;

    for(i = 0; i < 6; i++)
        printf("%2d", *(data +i) );
}
```

A8.5 There is no reason why `for` loops must begin at 0 or 1, with the counter increasing on each iteration of the loop.

```
 i  *ptr
--------
 2,   1
 1,   3
```

```
 0,   4
-1,   4
-2,   3
-3,   1
```

A8.6

```
#include <stdio.h>

void dbl(int* num);

int main(void)
{
    int x = 13;

    dbl(&x);

    printf("x doubled is %d\n", x);
}

void dbl(int* num)
{
    *num = 2 * *num;
}
```

A8.7 My version of the program is

```
#include <stdio.h>

void doublenum(int *);

int main(void)
{
    int x = 5;

    doublenum(&x);

    printf("x is %d\n", x);
}

void doublenum(int *a)
{
  *a = *a * 2;
}
```

A8.8 My version of the program is

```
#include <stdio.h>

void printString(char *);

int  main(void)
{
                /* 12341234123412341234123412341234 */
    char text[] = "This is an example string.";

    printString(text);
}

void printString(char *a)
{
    int counter = 0;

    while(a[counter] != '\0')
    {
        if((counter+1)%4 == 0)
            printf("%c", a[counter]);

        counter++;
    }
}
```

A8.9

```
#include <stdio.h>
int main(void)
{
    char *text[] = {"Programming",
                    "is",
                    "fun."};
    int i;

    for(i = 0; i < 3; i++)
    {
        if (i != 2)
            printf("%s ", text[i]);
        else
            printf("%s\n", text[i]);
    }
}
```

A8.10

```
void getRange(double *ptrA, double *ptrB, double c)
{
    *ptrA = 0.75 * c;
    *ptrB = 1.25 * c;
}
```

A8.11

```
#include <stdio.h>
int main(void)
{
    char text[] = "?krow siht diD";
    int size = sizeof(text);

    char *start = text;
    char *end = text + size - 2; /* subtract 2 to eliminate
                                    the terminating null  */

    while(end >= start)
    {
        printf("%c", *end);
        end--;
    }
}

/***********************************
***            output            ***

Did this work?

*/
```

A8.12

```
#include <stdio.h>

void diag(int (*a)[3] );

int main(void)
{
    int data[3][3] = {{10, 20, 30},
                      {40, 50, 60},
                      {70, 80, 90}};
    diag(data);
}

void diag(int (*a)[3] )
```

```
{
    int i;

    for(i = 0; i < 3; i++)
        printf("%d ", *(*(a + i) + i) );
}
```

A8.13

```
#include <stdio.h>

void lt15(int (*a)[3] );

int main(void)
{
    int data[3][3] = {{11, 19, 12},
                      {10, 14, 15},
                      {18, 13, 12}};
    lt15(data);
}

void lt15(int (*a)[3] )
{
    int i, j;

    for(i = 0; i < 3; i++)
        for(j = 0; j < 3; j++)
            if( *(*(a + i) + j) < 15 )
                printf("%d ", *(*(a + i) + j) );
}
```

A8.14 The condition in the `if` statement uses a single `=`, which assigns a value to `div4`, instead of `==`, which is used to test for equality.

A8.15 The row offset in `*(*data + i)` is zero. Each value of `i` that is then added to `*data` increments the address by the size of one `int`. Since array elements are stored in contiguous memory locations, the values that seem to go beyond the number of columns of the array simply reference the next row of values.

A8.16 The problem is in the pointer notation used to reference array elements from the `for` loop. The expression `(*(*data + r) + c)` is equivalent to `*(*(data + 0) + (r + c) )` where the row offset is zero and the column offset is `(r + c)`.

A8.20

```
#include <stdio.h>

int fxPointers(int* a, int n, int x);
```

```
int fxSubscript(int a[], int n, int x);

int main(void)
{
    int data[] = {5, 8, 2, 3, 8, 9};
    int n = 6, x;

    printf("Number to look for: ");
    scanf("%d", &x);

    printf("%d is %d values from the end\n", x, fxPointers(data, n, x) );
    printf("%d is %d values from the end\n", x, fxSubscript(data, n, x) );
}


int fxPointers(int* a, int n, int x)
{
    int i, count = 0;

    for(i = n-1; i >= 0; i--)
        if( *(a + i) == x )
            break;
        else
            count++;

    return count;
}

int fxSubscript(int a[], int n, int x)
{
    int i, count = 0;

    for(i = n-1; i >= 0; i--)
        if( a[i] == x )
            break;
        else
            count++;

    return count;
}
```

A8.21
```
#include <stdio.h>
```

```
void fxPointers(int* a, int n);
void fxPointers2(int* a, int n);
void fxSubscripts(int a[], int n);

void printPointers(int* a, int n);
void printPointers2(int* a, int n);
void printSubscripts(int a[], int n);

int main(void)
{
    int data[] = {19, 20, 21, 22, 23, 24};
    int n = 6;

    /* reverse array */
    fxPointers(data, n);
    printPointers(data, n);

    /* reverse back to original order */
    fxSubscripts(data, n);
    printSubscripts(data, n);

    /* reverse again */
    fxPointers2(data, n);
    printPointers2(data, n);
}

void fxPointers(int* a, int n)
{
    int i, temp;

    for(i = 0; i < n/2; i++)
    {
        temp = *(a + i);
        *(a + i) = *(a + n - 1 - i);
        *(a + n - 1 - i) = temp;
    }
}

void fxPointers2(int* a, int n)
{
    int i, temp;
```

```
    int* startptr = a;
    int* endptr = a + n - 1;

    for(i = 0; i < n/2; i++)
    {
        temp = *startptr;
        *startptr = *endptr;
        *endptr = temp;
        startptr++;
        endptr--;
    }
}

void fxSubscripts(int a[], int n)
{
    int i, temp;

    for(i = 0; i < n/2; i++)
    {
        temp = a[i];
        a[i] = a[n - 1 - i];
        a[n - 1 - i] = temp;
    }
}

void printPointers(int* a, int n)
{
    int i;

    for(i = 0; i < n; i++)
        printf("%d ", *(a + i) );

    printf("\n");
}

void printPointers2(int* a, int n)
{
    int i;

    for(i = 0; i < n; i++)
    {
        printf("%d ", *a);
```

```
        a++;
    }

    printf("\n");
}

void printSubscripts(int a[], int n)
{
    int i;

    for(i = 0; i < n; i++)
        printf("%d ", a[i] );

    printf("\n");
}
```

A8.22

```
#include <stdio.h>

void fx(int num, int* sum, int* count);

int main(void)
{
    int i, sum, count;

    for(i = 10; i <= 20; i++)
    {
        printf("%d:  ", i);
        fx(i, &sum, &count);
        printf(" | count = %d, sum = %d\n", count, sum);
    }

}

void fx(int num, int* sum, int* count)
{
    int i;
    *sum = 0;
    *count = 0;

    for(i = 1; i <= num; i++)
        if(num%i == 0)
        {
```

```
                if(i == num)
                    printf("%d", i);
                else
                    printf("%d, ", i);

                *sum += i;
                *count += 1;
            }
    }
```

A8.23
```
    #include <stdio.h>

    int main(void)
    {
        int data[] = {1, 2, 3, 4, 5, 6};
        int* begin = data;
        int* end = data + 5;

        while( 1 )
        {
            printf("%2d", *begin);
            printf("%2d", *end);

            if( begin == end || begin == end - 1 )
                break;
            else
            {
                begin++;
                end--;
            }
        }
    }
```

A9.1
```
    int main(void)
    {
        FILE *inptr;
        char name[20];
        int amt;
        int sum = 0;

        inptr = fopen("payments.txt", "r");

        while ( fscanf(inptr, "%s %d", name, &amt) != EOF )
```

```
    {
        printf("%s made a payment of $%d\n", name, amt);
        sum += amt;
    }
    printf("The total payments are $%d\n", sum);

    fclose( inptr );
}
```

A9.2

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

int main(void)
{
    FILE *inptr, *outptr;
    char s1[31];
    char s2[31];
    char s3[31];
    char s4[31];

    inptr = fopen("people.txt", "r");
    outptr = fopen("smith.txt", "w");

    while ( fscanf(inptr, "%s %s %s %s", s1, s2, s3, s4) != EOF )
    {
        if(strcmp(s2, "Smith") == 0)
            fprintf(outptr, "%s %s %s %s\n", s4, s3, s2, s1);
    }

    fclose( inptr );
    fclose( outptr );
}

/***********************************************
***                people.txt                ***

Bob Jones male engineer
Susan Smith female manager
Tom Smith male singer
Mary Wilson female teacher

*/
```

```
/***********************************************
***                  smith.txt                  ***

manager female Smith Susan
singer male Smith Tom

*/
```

A9.3

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

int main(void)
{
    FILE* inptr;
    char name[20];

    inptr = fopen("names.txt", "r");

    while( fscanf(inptr, "%s", name) != EOF )
    {
        if(strcmp(name, "a") < 0)
            printf("%s\n", name);
    }
}
```

A10.1

```
struct book
{
    char title[50];
    char author1[30];
    char author2[30];
    short edition;
    char isbn[11];
};
```

A10.2

```
void sortStruct(struct names *people)
{
    int swaps = 1;
    char tempname[20];
    int tempage;
    int i;

    while(swaps)
    {
```

```
        swaps = 0;
        for(i = 0; i < SIZE-1; i++)
        {
            if(strcmp(people[i].name, people[i+1].name) > 0)
            {
                strcpy(tempname, people[i].name);
                strcpy(people[i].name, people[i+1].name);
                strcpy(people[i+1].name, tempname);
                tempage = people[i].age;
                people[i].age = people[i+1].age;
                people[i+1].age = tempage;
                swaps = 1;
            }
        }
    }
}
```

A10.3

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

typedef struct
{
    char name[10];
    char type[10];
} ANIMAL;

int main(void)
{
    ANIMAL zoo[] = { {"zebra", "mammal"},
                     {"gorilla", "mammal"},
                     {"shark", "fish"},
                     {"vulture", "bird"},
                     {"rhino", "mammal"} };
    ANIMAL *ptr = zoo;
    int i;

    printf("the mammals are:\n\n");
    for(i = 0; i < 5; i++)
    {
        if(!strcmp("mammal", ptr->type))
            printf("%s\n", ptr->name);
        ptr++;
```

```
    }
}
```

A11.1 i. just flip the bits to get 1010

ii. for the `logical and`, it takes two 1's in a column to produce a 1 in the output column

$$\begin{array}{r} 0101 \\ \& \; 0011 \\ \hline 0001 \end{array}$$

iii. for the `logical or`, a single 1 in a column of the input will produce a 1 in the same column of the output

$$\begin{array}{r} 0101 \\ | \; 0011 \\ \hline 0111 \end{array}$$

iv. for the `logical xor`, a 1 must occur only once in a column of the input in order for there to be a 1 in the same column of the output

$$\begin{array}{r} 0101 \\ \hat{} \; 0011 \\ \hline 0110 \end{array}$$

A11.2 To solve these, you must keep in mind that each bit shift doubles (if `<<`) or halves (if `>>`) the value.

i. `8 << 1 = 16`

ii. `1 << 3 = 8`

iii. `1 << 15 = 32768`

iv. `400 >> 2 = 100`

v. `100 >> 1 = 50`

A11.3 We use `&` (bitwise and) to check if a bit has been set (i.e., given a value of one). Since we are checking specific bits, we need to compare the bit pattern of `flags` to a number whose bit pattern has a one (or multiple ones) in the desired location (or locations). For the numbers used below, it is helpful to see the binary representation (only the four rightmost bits are shown) of the base-10 numbers used:

When we compare `flags` to 7, we are actually checking if any of the three bits we are interested in have been set. After printing which responses have been seen before, we update `flags` so that the current response is stored.

decimal	binary
1	0001
2	0010
4	0100
7	0111

```
void checkResponse(short* flags, int response)
{
    int flip;

    printf("You gave response %d\n", response);

    if(*flags & 1)
        printf("  we have seen response 1 before\n");
    if(*flags & 2)
        printf("  we have seen response 2 before\n");
    if(*flags & 4)
        printf("  we have seen response 3 before\n");
    if( (*flags & 7) == 0 )
        printf("  we haven't seen any responses before\n");

    flip = 1 << (response - 1);
    *flags = *flags | flip;
}

/*************************************************
***                  Output                     ***
    give an integer value in the range 1 to 3:  3
    You gave response 3
      we haven't seen any responses before
    give an integer value in the range 1 to 3:  9
    That is an invalid response
    give an integer value in the range 1 to 3:  2
    You gave response 2
      we have seen response 3 before
    give an integer value in the range 1 to 3:  2
    You gave response 2
      we have seen response 2 before
      we have seen response 3 before
    give an integer value in the range 1 to 3:  2
    You gave response 2
      we have seen response 2 before
```

```
          we have seen response 3 before
        give an integer value in the range 1 to 3:  1
        You gave response 1
          we have seen response 2 before
          we have seen response 3 before
    */
```

A12.1 In order for our function to remember that it has been called before, we need to use a static variable.

```
void strikes(void)
{
    static int t = 1;

    if(t == 3)
        printf("Strike %d!  You're out!\n", t);
    else
        printf("Strike %d!\n", t);

    t++;
}
```

A12.2

```
#include <stdio.h>

enum direction {north, south, east, west};

int main(void)
{
    enum direction dir;

    printf("Which way are you heading?  ");
    scanf("%d", &dir);

    if(dir == north)
        printf("Keep going; you are on the right track.\n");
    else if(dir == south)
        printf("Turn 180 degrees.\n");
    else if(dir == east)
        printf("Turn 90 degrees left.\n");
    else if(dir == west)
        printf("Turn 90 degrees right.\n");
    else
        printf("That isn't a valid direction.\n");
```

```
}
```

Bibliography

[O'L09] Dianne P. O'Leary. *Scientific Computing with Case Studies*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2009.