Review Questions:

1. What are the two different kinds of division that the / operator can do? Under what circumstances does it perform each?

Division: which discards any remainder

Floating-point division: which keep the remainder (fractional).

2. What are the definitions of the ``Boolean'' values true and false in C?

False: always represented by value zero.

True: Any non-zero value.

3. Name three uses for the semicolon in C.

- Terminating declarations
- Terminating statements
- Separating control expressions in a for loop.

4. What would the equivalent code, using a **while loop**, be for the example

```
for(i = 0; i < 10; i = i + 1)
printf("i is %d\n", i);
```

R =

```
i = 0;
while(i < 10){
        printf("i is %d\n", i);
        i = i + 1; }
```

5. What is the **numeric value** of the expression 3 < 4 ?

1

6. Under what **conditions** will this code print ``water''?

```
if(T < 32)
        printf("ice\n"); else if(T < 212)
        printf("water\n"); else printf("steam\n");
```

R = if T is less than 212 but greater than 32

7. What would this code print?

```
int x = 3;
if(x)
printf("yes\n"); else printf("no\n");
```

R = yes

8. (trick question) What would this code print?

```
int i;
for(i = 0; i < 3; i = i + 1)
printf("a\n");
 printf("b\n");
printf("c\n");
```

R =

```
a
b
c
```

Tutorial Section

(This section presents a few small but complete programs and asks you to work with them. If you're comfortable doing at least some of the exercises later in this assignment, there's no reason for you to work through this ``tutorial section.'' But if you're not ready to do the exercises yet, this section should give you some practice and get you started.)

1. Type in and run this program, and compare its output to that of the original ``Hello, world!'' program (exercise 1 in assignment 1).

```
#include <stdio.h>
int main() {
printf("Hello, "); printf("world!\n"); return 0; }
```

You should notice that the output is identical to that of the original ``Hello, world!'' program. This shows that you can build up output using multiple calls to printf, if you like. You mark the end of a line of output (that is, you arrange that further output will begin on a new line) by printing the ``newline'' character, \n.

2. Type in and run this program:

```
#include <stdio.h>
int main() {
int i;
printf("statement 1\n"); printf("statement 2\n"); for(i = 0; i < 10; i = i + 1) {
printf("statement 3\n"); printf("statement 4\n"); } printf("statement 5\n");
        return 0; }
```

This program doesn't do anything useful; it's just supposed to show you how control flow works--how statements are executed one after the other, except when a construction such as the for loop alters the flow by arranging that certain statements get executed over and over. In this program, each simple statement is just a call to the printf function.

Now delete the braces {} around statements 3 and 4, and re-run the program. How does the output change? (See also question 8 above.)

3. Type in and run this program:

```
#include <stdio.h>
int main() {
int i, j;
printf("start of program\n");
for(i = 0; i < 3; i = i + 1) {
printf("i is %d\n", i); for(j = 0; j < 5; j = j + 1)
printf("i is %d, j is %d\n", i, j); printf("end of i = %d loop\n", i); }
printf("end of program\n");
        return 0; }
```

This program doesn't do much useful, either; it's just supposed to show you how loops work, and how loops can be nested. The outer loop runs i through the values 0, 1, and 2, and for each of these three values of i(that is, during each of the three trips through the outer loop) the inner loop runs, stepping the variable j through 5 values, 0 to 4. Experiment with changing the limits (initially 3 and 5) on the two loops. Experiment with interchanging the two loops, that is, by having the outer loop manipulate the variable j and the inner loop manipulate the variable i. Finally, compare this program to the triangle-printing program of Assignment 1 (exercise 4) and its answer as handed out this week.

4. Type in and run this program:

```
#include <stdio.h>
int main() {
int day, i;
for(day = 1; day <= 3; day = day + 1) {
printf("On the %d day of Christmas, ", day);
printf("my true love gave to me\n"); for(i = day; i > 0; i = i - 1) {
if(i == 1) {
if(day == 1) printf("A "); else printf("And a "); printf("partridge in a pear tree.\n"); } else if(i == 2) {
        printf("Two turtledoves,\n"); } else if(i == 3) {
        printf("Three French hens,\n"); } printf("\n"); }
        return 0; } }
```

The result (as you might guess) should be an approximation of the first three verses of a popular (if occasional tiresome) song.

5. a. Add a few more verses (calling birds, golden rings, geese a-laying, etc.).

6. b. Here is a scrap of code to print words for the days instead of digits:

```
if(day == 1) {
        printf("first"); } else if(day == 2) {
        printf("second"); } else if(day == 3) {
        printf("third"); }
```

7. Incorporate this code into the program.

a. Here is another way of writing the inner part of the program:

```
printf("On the %d day of Christmas, ", day); printf("my true love gave to me\n"); if(day >= 3)
printf("Three French hens,\n"); if(day >= 2)
printf("Two turtledoves,\n"); if(day >= 1) {
```

if(day == 1) printf("A "); else printf("And a "); printf("partridge in a pear tree.\n"); } Study this alternate method and figure out how it works. Notice that there are no else's between the if's.

Exercises:

(As before, these range from easy to harder. Do as many as you like, but at least two or three.)

1. Write a program to find out (the hard way, by counting them) how many of the numbers from 1 to 10 are greater than 3. (The answer, of course, should be 7.) Your program should have a loop which steps a variable (probably named i) over the 10 numbers. Inside the loop, if i is greater than 3, add one to a second variable which keeps track of the count. At the end of the program, after the loop has finished, print out the count.

2. Write a program to compute the average of the ten numbers 1, 4, 9, ..., 81, 100, that is,

the average of the squares of the numbers from 1 to 10. (This will be a simple modification of Exercise 3 from last week: instead of printing each square as it is computed, add it into a variable sum which keeps track of the sum of all the squares, and then at the end, divide the sum variable by the number of numbers summed.)

If you keep track of the sum in a variable of type int, and divide by the integer 10, you'll get an

integer-only approximation of the average (the answer should be 38). If you keep track of the sum in a variable of type float or double, on the other hand, you'll get the answer as a floating-point number, which you should print out using %f in the printf format string, not %d. (In a printf format string, %d prints only integers, and %f is one way to print floating-point numbers. In this case, the answer should be 38.5.)

3. Write a program to print the numbers between 1 and 10, along with an indication of

whether each is even or odd, like this: 1 is odd 2 is even 3 is odd ...

(Hint: use the % operator.)

4. Write a program to print the first 7 positive integers and their factorials. (The factorial of 1

is 1, the factorial of 2 is 1 * 2 = 2, the factorial of 3 is 1 * 2 * 3 = 6, the factorial of 4 is 1 * 2 * 3 * 4 = 24, etc.) [Extra credit: why did I only ask for the first 7?]

5. Write a program to print the first 10 Fibonacci numbers. Each Fibonacci number is the

sum of the two preceding ones. The sequence starts out 0, 1, 1, 2, 3, 5, 8, ...

Recursion

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Usually recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

(1. Using loops) vs (2. Without using loops)

1. Factorial

Given n of 1 or more, return the factorial of n, which is n * (n-1) * (n-2) ... 1. Compute the result recursively (without loops).

factorial(1) → 1 factorial(2) → 2 factorial(3) → 6

2. Fibonacci

The fibonacci sequence is a famous bit of mathematics, and it happens to have a recursive definition. The first two values in the sequence are 0 and 1 (essentially 2 base cases). Each subsequent value is the sum of the previous two values, so the whole sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21 and so on. Define a recursive fibonacci(n) method that returns the nth fibonacci number, with n=0 representing the start of the sequence.

fibonacci(0) → 0 fibonacci(1) → 1 fibonacci(2) → 1

3. BunnyEars

We have bunnies standing in a line, numbered 1, 2, ... The odd bunnies (1, 3, ..) have the normal 2 ears. The even bunnies (2, 4, ..) we'll say have 3 ears, because they each have a raised foot. Recursively return the number of "ears" in the bunny line 1, 2, ... n (without loops or multiplication).

bunnyEars2(0) → 0 bunnyEars2(1) → 2 bunnyEars2(2) → 5

4. Power N

Given base and n that are both 1 or more, compute recursively (no loops) the value of base to the n power, so powerN(3, 2) is 9 (3 squared).

powerN(3, 1) → 3 powerN(3, 2) → 9 powerN(3, 3) → 27

5. Sum Digits

Given a non-negative int n, return the sum of its digits recursively (no loops). Note that mod (%) by 10 yields the rightmost digit (126 % 10 is 6), while divide (/) by 10 removes the rightmost digit (126 / 10 is 12).

sumDigits(126) → 9 sumDigits(49) → 13 sumDigits(12) → 3