Fall 2009
Introduction to C - Programming Assignment #3

Due Date: October 14, 2009, 11:59pm

For this assignment, you are going to need to be able to read data from an input text file. This is relatively straightforward assuming you know the appropriate commands. You will need to know how to open a file for reading, reading from a file using fscanf, and closing the file. The following is extracted from Arup Guha's notes on file I/O.

How to Create a File Pointer

In order to read from a file, or write to a file, you MUST use a pointer to that file. Here is a declaration of a file pointer:

```
FILE *ifp;
```

In order to properly "initialize" a file pointer, it must be set to point to a file. In order to do this, we must specify the following:

1) Name of the file
2) Mode ("r" for read or "w" for write)

There is a function call that uses this information to open the appropriate file and return a pointer to it. It's name is fopen. Here is an example of its use:

```
ifp = fopen("input.txt", "r");
```

You'll notice that the first parameter to the `fopen` function is a string storing the name of the file to be opened. The second parameter is also a string. For our purposes, this string will be "r". When we open a file in "r" (reading) mode, the file should already exist and the `fopen` function returns a pointer to the beginning of that file. If the file doesn't exist, `fopen` returns NULL.

How to Read from an Input File

This works nearly the same as reading from the keyboard. In fact, imagine pre-typing all of your responses you would type in the keyboard into a file and then running a program that read from that file instead of the keyboard. In that situation, the new program would work identically.

Every time you use the `fscanf` function to read in a piece of information from a file, the file pointer reads in the next token, returns it and advances to the following token. Here is an example of how to read in an integer from the file we previously opened:

```
fscanf(ifp, "%d", &num);
```

The first parameter tells the computer to look to where ifp is pointing instead of the keyboard. The rest works the same as `scanf`.

Imagine that the file we just opened, "input.txt" contains several integers separated by white space, with the end of the data being signified by a 0. Here is how we would read in and sum all the numbers from the file:

```
FILE *ifp;
int num = -1, sum = 0;

if ((ifp = fopen("input.txt", "r")) == NULL) {
     printf(“File Not Found!\n”);
     exit(1);
}

while (num != 0) {
  fscanf(ifp, "%d", &num);
  sum += num;
}
fclose(ifp);
```

Closing a file

You'll notice that the last line above uses a new function call, `fclose`. All files that are opened must also be closed. In order to close a file, you must simply pass the file pointer to the file you want to close into the `fclose` function. Forgetting to close a file may corrupt the contents of that file.

Here is a complete program that reads in a file containing numbers and outputs their sum to the screen:

```
#include stdio.h

int main() {

  FILE *ifp;
  int num = -1, sum = 0;

  if ((ifp = fopen("input.txt", "r")) == NULL) {
     printf(“File Not Found!\n”);
     exit(1);
  }

  while (num != 0) {
    fscanf(ifp, "%d", &num);
```

```
    sum += num;
  }

  fclose(ifp);

  printf("The sum is %d\n", sum);
  return 0;
}
```

Objective

1. Practice using loops.
2. Learn how to read data from an input text file.
3. Learn how to use an array to store data and solve problems that require the storage and manipulation of that data.

Problem A: Abundant Numbers

Arup really cares about divisibility. He especially likes numbers that have lots of divisors. In particular, he likes numbers with proper divisors that sum to greater than the number itself. These numbers are called, "abundant numbers." A proper divisor of a number is any number strictly less than it that divides evenly into it. For example, 6 is a proper divisor of 12, but 12 is not. 12 is an example of an abundant number because 1+2+3+4+6 = 16 and 16 is greater than 12. As this example shows, one way to find out all the proper divisors of a number is to try each integer (starting at 1) out until you reach the number divided by 2. (It's impossible for any number in between 7 and 11 to divide evenly into 12 because it is guaranteed that the result of the division is less than 2 since we are dividing by a greater number, and greater than 1, since we are not dividing by the number itself.)

Write a program that reads in a list of numbers from a file, and for each number, determines and prints out whether or not that number is abundant.

Read your data from a file named, "numbers.txt". The format for this file is specified in the section below titled, "Input Specification."

Input Specification

1. The first integer line of the file will be a positive integer, *n* (less than 1000), indicating the number of test cases in the file.
2. The next *n* lines will contain a single positive integer each (less than 10000), with the number you are to determine is abundant or not for that test case.

Output Specification

Output a line with one of the two the following formats for each test case:

```
Test case #t: X is abundant.

Test case #t: X is NOT abundant.
```

where t represents the number of the test case (starting with 1), and X represents the number for that case.

Output Sample

Here is one sample output of running the program. Note that this test is NOT a comprehensive test. You should test your program with different data than is shown here based on the specifications given. The user input is given in *italics* while the program output is in bold.

Sample Input (numbers.txt)

```
3
6
12
17
```

Sample Output (Corresponding to Sample Input File)

```
Test case #1: 6 is NOT abundant.
Test case #2: 12 is abundant.
Test case #3: 17 is NOT abundant.
```

Problem B: Group Dinner at Lazy Moon

You have finally gotten sick of always ordering the same exact pizza at Lazy Moon. You are now looking to diversify your orders. Also, you and your friends have determined it's faster for one person to order for everyone than for everyone to order separately. Thus, each time you go, either you or one of your friends will put in several orders.

To simplify this problem (as many restaurants do), each different pizza order will be given a number (starting from 0) and each of these orders will have a price. Your program will read in this information from a file named "pizza.txt" and then use it to determine the prices of different orders of pizza. The information for these orders will ALSO be in the file. The specifications for the file format are given below.

Input Specification

1. The first line of the input file will contain a single positive integer, n $(n < 100)$, specifying the number of different pizza orders for Lazy Moon.
2. The next n lines will each contain one positive real number specifying the price for that corresponding pizza. (The first line will have the price for pizza 0, the next line for pizza 1, etc.)
3. The following line will contain a single positive integer k ($k < 50$) representing the number of orders you have to evaluate.
4. The next k lines will contain information about each of the k orders with one order per line.
5. Each of these lines will contain n non-negative integers, representing how many of those pizzas, respectively are in the order. (For example, if n=5 and the line contains the values 0 0 6 0 9, then the order contains 6 slices of pizza #2 and 9 slices of pizza #4.)

Output Specification

For each of the *k* test cases, output a line with the following format:

```
On day X, you will spend $YY.YY at Lazy Moon.
```

where X is the test case number (starting with 1), and YY.YY is the price of the pizza displayed with exactly 2 digits after the decimal. (Note: The price may exceed 100 dollars, so YY simply represents a dollar amount and not the exact number of digits in that dollar amount.

Sample Input File (pizza.txt)

```
5
3.00
3.50
4.50
5.00
6.00
3
1 1 1 1 1
0 0 2 1 6
1 0 3 2 3
```

Sample Output (Corresponding to Sample Input File)

```
On day 1, you will spend $22.00 at Lazy Moon.
On day 2, you will spend $50.00 at Lazy Moon.
On day 3, you will spend $44.50 at Lazy Moon.
```

Deliverables

You must submit following two .c files:

1) *abundant.c*, for your solution to problem A
2) *pizza.c* for your solution to problem B

over WebCourses.

Restrictions

Although you may use other compilers, your program must compile and run using DevC++. Please use DevC++ to develop your program. Your program should include a header comment with the following information: your name, course number, section number, assignment title, and date. Also, make sure you include ample comments throughout your code describing the major steps in solving the problem.

Grading Details

Your program will be graded upon the following criteria:

1) Your correctness
2) Your programming style and use of white space. (Even if you have a plan and your program works perfectly, if your programming style is poor or your use of white space is poor you could get 10% or 15% deducted from your grade.)
3) Compatibility to DevC++. (If your program does not compile in either of these environments, you will get a **sizable** deduction from your grade, likely to be over 50%)