PROBLEM 1:  \((Output: (12 \text{ pts}))\)

Indicate the output of each of the \texttt{cout} statements below. Assume each statement is part of a program that compiles and runs. You can show your reasoning for partial credit.

```cpp
string word, name, phrase;
word = "rock";
name = "chess";
phrase = "the heart";
int number = 10;
int a = 6;

cout << word + name << endl;
cout << phrase.length() << endl;
cout << number % a << endl;
cout << number/3 << endl;
cout << number << "23" << 6 << endl;
cout << a + 3 * number << endl;
```

Output

PROBLEM 2:  \((Output \text{ Again}: (12 \text{ pts}))\)

Assume each statement below is part of a program that compiles and runs.

What is the output if the user types the number 35?

```cpp
int num;
cout << "Enter a number: ";  // user will type 35
cin >> num;

if (num > 10)
    cout << "one" << endl;
else if (num > 20)
    cout << "two" << endl;
else if (num > 40)
    cout << "try" << endl;
else
    cout << "fry" << endl;

cout << "sly" << endl;
if (num > 40)
    cout << "one" << endl;
else if (num > 20)
```

1
cout << "two" << endl;
else if (num > 10)
    cout << "try" << endl;
else
    cout << "fry" << endl;

cout << "sly" << endl;
if (num > 10)
    cout << "one" << endl;
if (num > 20)
    cout << "two" << endl;
if (num > 40)
    cout << "try" << endl;
else
    cout << "fry" << endl;

Output

PROBLEM 3: (Send Food: (12 pts))

You’ve been lazy about writing letters home and not all your relatives have email, so you’d like to write a program to print a form letter to relatives. The two function calls below should generate the output shown.

    LetterHome("Mom", "3rd", "wrote a neat program", "please");

    LetterHome("Aunt Jane and Uncle Fred", "6th", "ran with Nan", "you must");

Corresponding Output

Hi Mom,

In this 3rd week of school I wrote a neat program.
Mom, please, please,
please send food!

Hi Aunt Jane and Uncle Fred,

In this 6th week of school I ran with Nan.
Aunt Jane and Uncle Fred, you must, you must,
you must send food!

Write the function LetterHome.

PROBLEM 4: (The Lucky Seven: (12 pts))
Write the function `CountRolls2and5` whose header is given below. `CountRolls2and5` returns the number of times a pair of 6-sided dice are rolled until a 2 and 5 are rolled at the same time. The Dice class is below.

```cpp
class Dice
{
public:
    Dice(int sides);  // constructor
    int Roll();       // return the random roll
    int NumSides();   // how many sides this die has
    int NumRolls();   // # times this die rolled
private:
    // not shown
};
```

Here is an example where four rolls were necessary. The first roll gives 4 and 1, the second roll gives 3 and 5, the third roll gives 3 and 3, and the fourth roll gives 5 and 2. `CountRolls2and5` would return the count 4 in this case.

Complete `CountRolls2and5` below the following header.

```cpp
int CountRolls2and5()
// postcondition: returns the number of rolls for a pair of 6-sided dice
// until (and including) a 2 and 5 are rolled.
{
}
```

**PROBLEM 5: (PART A: No Unlucky Phone Numbers: 12 pts)**

Write the function `ValidNumber` whose header is given below. A valid phone number is a 7 digit number that does not contain the number 13 as any part of it. `ValidNumber` returns true if a phone number is valid, and it returns false if the number is not valid.

For example, `ValidNumber(3417345)` returns true, `ValidNumber(2761356)` and `ValidNumber(9138333)` both return false (13 appears in both of these numbers). You may assume that any numbers tested do not start with 0 or 1.

Hint: For the number 2761356, examine (or strip off) the last two digits (56), then the next to last two digits (35), then the next pair (13) and in this case you can stop early.

Complete `ValidNumber` below the following header.

```cpp
bool ValidNumber(int number)
// precondition: number is a 7 digit number that does not start with 0 or 1.
// postcondition: returns true if 13 is not a part of the number,
// otherwise, returns false
{
}
```
PART B: (Largest Lucky Phone Number: (12 pts))

Write the function MaxValidNumber whose header is given below. MaxValidNumber returns the largest valid phone number in a file, and returns 0 if none of the numbers in the file are valid. As in PART A, a valid phone number is a 7 digit number that does not contain the number 13 as any part of it.

For example, MaxValidNumber("numbers") returns 5873215 where the file numbers is shown below (note: 2130784, 8136709, and 9987613 are invalid and are not considered).

File: numbers

2130784
3487645
8136709
5873215
9987613

You may call the function ValidNumber that you wrote in PART A. Assume ValidNumber works correctly regardless of what you wrote for PART A.

Complete MaxValidNumber below the following header.

```cpp
int MaxValidNumber(string filename)
// precondition: filename is a file of 7 digit numbers that do not start with 0 or 1
// postcondition: returns the largest valid phone number in the file filename,
// returns 0 if the file has no valid phone numbers or is empty
{

    ifstream input;
    input.open(filename);

}
```

PROBLEM 6: (The Birds: (12 pts))

A class StateIterator is designed to facilitate processing information about states in the United States. A StateIterator variable can be used to process all 50 states, once the member function First is called. The order the states are processed is irrelevant.

```cpp
StateIterator iterator;

    iterator.First();
    cout << iterator.Current() << endl;
    cout << iterator.Bird() << endl;
    iterator.Next();
    cout << iterator.Current() << endl;
    cout << iterator.Bird() << endl;
```
The code segment shown above might produce the output below (assuming Alabama and North Carolina are the first two states processed).

Alabama
Yellowhammer
North Carolina
Cardinal

// StateIterator class
//
// This class will iterator over each of the 50 states in the United States.
//
// StateIterator() -- construct an iterator
//
// (Cannot use the iterator until First() is called)
//
// void First() -- Starts the iterator at the first state
//
// string Current() -- returns the current state, e.g., "Alabama", "North Carolina"
//
// bool IsDone() -- returns true if there are no more states to process
//
// (in this case, the iterator has no Current state),
//
// otherwise returns false
//
// void Next() -- advances to the next state
//
// string Bird() -- returns the state bird of the current state, e.g.,
//
// "Cardinal", "Robin"
//

class StateIterator
{

public:
    StateIterator(); // constructor
    void First(); // initialize iterator
    string Current(); // returns current U.S. state
    bool IsDone(); // true if iterator is done
    void Next(); // advance to next state
    string Bird(); // returns the name of the state bird
                    // for the current state

private:
    // not shown

};

Write the function PrintStatesForBird whose header is given below. PrintStatesForBird prints the names of states whose state bird is the variable bird.
For example, `PrintStatesForBird("Cardinal")` would print the states *North Carolina, Indiana* and all the other states with Cardinal as the state bird, one state per line.

Complete `PrintStatesForBird` below the following header.

```cpp
void PrintStatesForBird(string bird) {
    // postcondition: Prints the names of the states for which the parameter bird
    // is the state bird. Nothing is printed if the bird is not
    // a state bird for any state.
    {
    }
```