

Section Solution 1

Problem 1: Compiler, compiler, what do you want?

The line that will not compile in this example is the line that performs the assignment `dir = num;`. Why not? Enumerated types are represented as integers at runtime, but at compile time, the C++ compiler provides additional type checking information to make your code safer. In this case, since `num` is in integer type, the compiler will not let you assign an integer to an enumeration. Could this be fixed? Of course! All we need to do is add in a typecast so that the compiler's convinced that you know what you are doing. The following line does the same thing, but will actually compile:

```
dir = directionT(num);
```

Problem 2: Removing all occurrences of a character

If we want to remove the occurrences of the letter one at a time, we can write the following function:

```
/**
 * Function: RemoveAllOccurrences
 * Usage: s = RemoveAllOccurrences(input, ch);
 * -----
 * This function takes a string and returns a corresponding string with all
 * the occurrences of a given letter removed. It uses a for loop to iterate
 * through the string, testing each character to see if it matches
 * the letter to remove, and building the string character by character.
 */

string RemoveAllOccurrences(string text, char ch)
{
    string result;
    for (int i = 0; i < text.length(); i++) {
        if (text[i] != ch) {
            result += text[i];
        }
    }

    return result;
}
```

We can also do the same thing by using the `find` and `substr` methods from the `string` class:

```

string RemoveAllOccurrences(string text, char ch)
{
    int pos;
    string result = text;

    while (true) {
        pos = result.find(ch);
        if (pos == string::npos) break;
        result = result.substr(0, pos) +
            result.substr(pos + 1, result.length() - pos - 1);
    }

    return result;
}

```

Problem 3: Rot-N

There are many different ways of writing the Rot13 function itself. The one listed below works reasonably well, but most anything with the same result is equally good or better.

```

static const int kAlphaLength = 26;
void RotN(string& message, int shiftAmount)
{
    for (int i = 0; i < message.length(); i++) {
        char ch = message[i];
        if (isalpha(ch)) {
            if (isupper(ch)) {
                ch = (((ch - 'A') + shiftAmount) % kAlphaLength) + 'A';
            } else {
                ch = (((ch - 'a') + shiftAmount) % kAlphaLength) + 'a';
            }
            message[i] = ch;
        }
    }
}

```

Problem 4: Find and Replace

Both prototypes of the **FindAndReplace** function have their benefits:

```
string FindAndReplace(string text, string find, string replace);
```

will create a new copy of the string you are working on, so that both strings will be available for later manipulation. However, most normal use only requires the modified version of the string. If we use the second prototype

```
void FindAndReplace(string& text, string find, string replace);,
```

we don't have to worry about confusing the modified and unmodified strings. All we get is the modified string, which might make for more understandable code. So, we will prefer the second prototype of **FindAndReplace**. Now, we have to implement the function, as shown below:

```
void FindAndReplace(string& text, string find, string replace)
```

```

{
    int startPos = 0;
    while (true) {
        int findPos = text.find(find, startPos);
        if (findPos == string::npos) break;
        text = text.replace(findPos, find.length(), replace);

        // this line serves to ensure that we don't get stuck in
        // an infinite loop when we replace the strings
        startPos = findPos + replace.length();
    }
}

```

Problem 5: Simple file reading

```

#include <iostream>
#include <fstream>
#include "strutils.h"

int SumFromFile(string filename)
{
    ifstream in;

    in.open(filename.c_str());

    // check to make sure the ifstream is valid
    if (in.fail()) Error("File could not be opened.");

    int num, sum = 0;
    while (true) {
        in >> num;
        if (in.fail()) break;
        sum += num;
    }

    in.close();
    return sum;
}

```

Problem 6: Legit Sudoku

```

#include "genlib.h"

static const int kTotalSize = 9;
static const int kSubSquareSize = 3;

/**
 * Predicate Function: isValidSudokuSolution
 * -----
 * This function determines whether the region
 * of the board with its top left corner at
 * (row, col) and with the given width and height
 * contains one instance of each number between 1 and 9.
 */

bool isValidRegion(Grid<int>& board, int row, int col, int height, int width)
{
    if (height * width != kTotalSize)
        Error("Bogus region passed to isValidRegion");

    for (int n = 1; n <= 9; n++) {
        int numOccurrences = 0;
        for (int dRow = 0; dRow < height; dRow++) {
            for (int dCol = 0; dCol < width; dCol++) {
                if (board(row + dRow, col + dCol) == n) {
                    numOccurrences++;
                }
            }
        }

        if (numOccurrences != 1) return false;
    }

    return true;
}

/**
 * Function: isValidSudokuSolution
 * Usage: valid = isValidSudokuSolution(board);
 * -----
 * This function takes a 9-by-9 Grid of integers and determines
 * whether or not this board corresponds to a valid Sudoku solution.
 */

bool isValidSudokuSolution(Grid<int>& board)
{
    for (int row = 0; row < kTotalSize; row++) {
        if (!isValidRegion(board, row, 0, 1, kTotalSize)) {
            return false;
        }
    }

    for (int col = 0; col < kTotalSize; col++) {
        if (!isValidRegion(board, 0, col, kTotalSize, 1)) {
            return false;
        }
    }
}

```

```
for (int row = 0; row < kTotalSize; row += kSubSquareSize) {
    for (int col = 0; col < kTotalSize; col += kSubSquareSize) {
        if (!isValidRegion(board, row, col, kSubSquareSize, kSubSquareSize)) {
            return false;
        }
    }
}

return true;
}
```