Section Solutions 1

Problem One: Returning and Printing

```
void printLyrics_v1() {
    cout << "Havana ooh na na" << endl;
}
```

```
string printLyrics_v2() {
    return "Havana ooh na na";
}
```

```
string printLyrics_v3() {
    return "H";
}
```

```
char printLyrics_v4() {
    return 'H';
}
```

Of these four functions, only `printLyrics_v1` will print anything. Specifically, it prints out the string `"Havana ooh na na."` The name "`printLyrics`" is inappropriate for the other two functions, as those functions don't actually print anything.

The function `printLyrics_v1` doesn't return anything – it just sends information to the console. As a result, its return type should be `void`. The functions `printLyrics_v2` and `printLyrics_v3` each return strings, since C++ treats anything in double-quotes as a string. Finally, `printLyrics_v4` returns a `char`, since C++ treats anything in single-quotes as a character.

Problem Two: Recursion Tracing

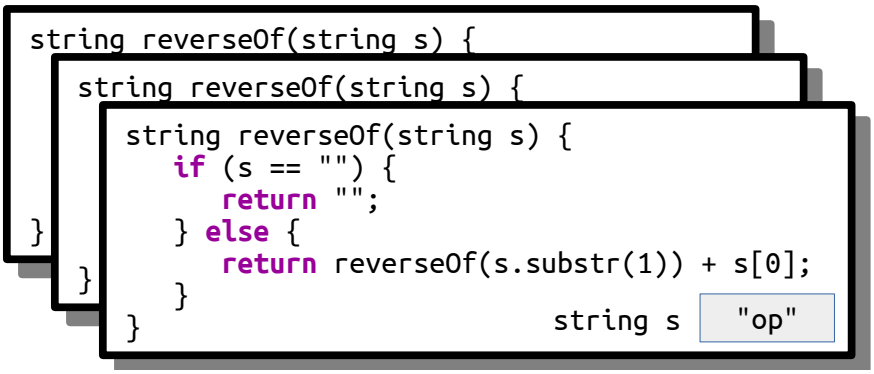
Our initial call to `reverseOf("stop")` looks like this:

```
string reverseOf(string s) {
   if (s == "") {
      return "";
   } else {
      return reverseOf(s.substr(1)) + s[0];
   }
}                              string s  "stop"
```

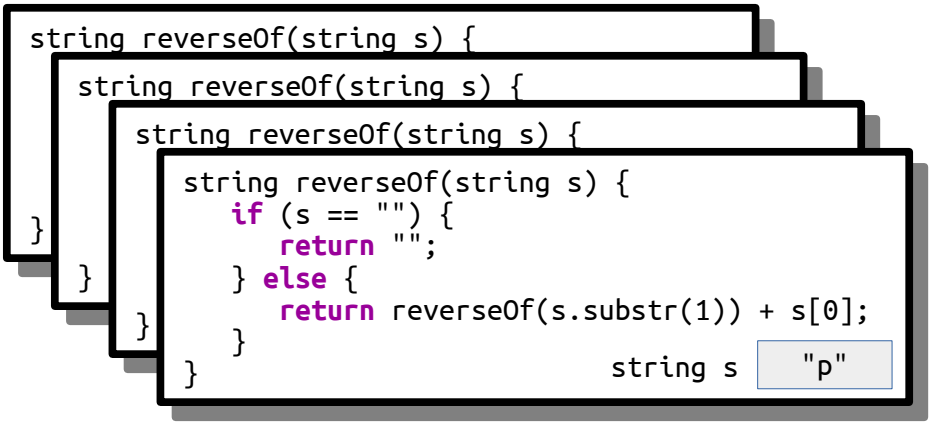
This call then fires off a call to `reverseOf("top")`, which looks like this:

```
string reverseOf(string s) {
}
   string reverseOf(string s) {
      if (s == "") {
         return "";
      } else {
         return reverseOf(s.substr(1)) + s[0];
      }
   }                              string s  "top"
```

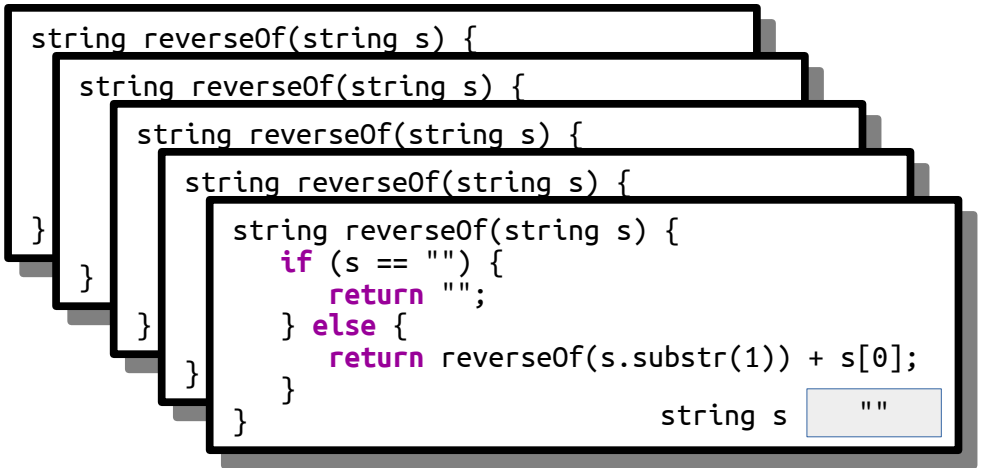
This call fires off a call to `reverseOf("op")`:



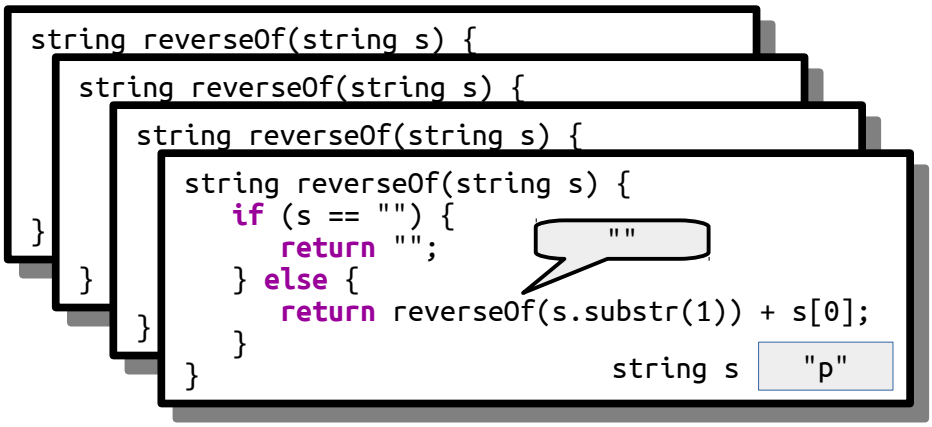
This in turn calls `reverseOf("p")`:



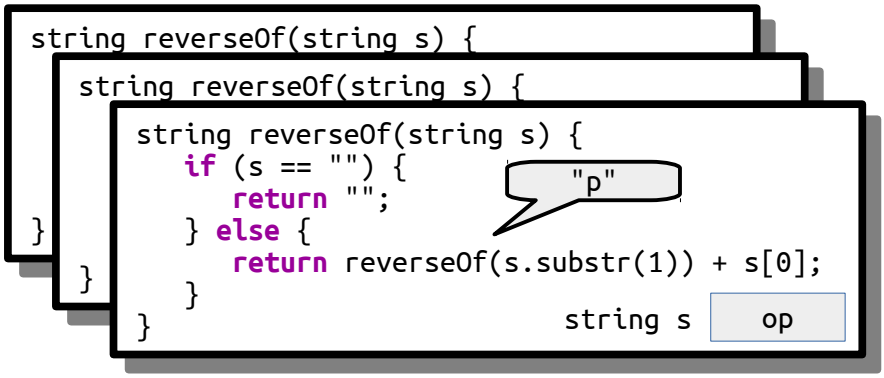
This in turn calls `reverseOf("")`:



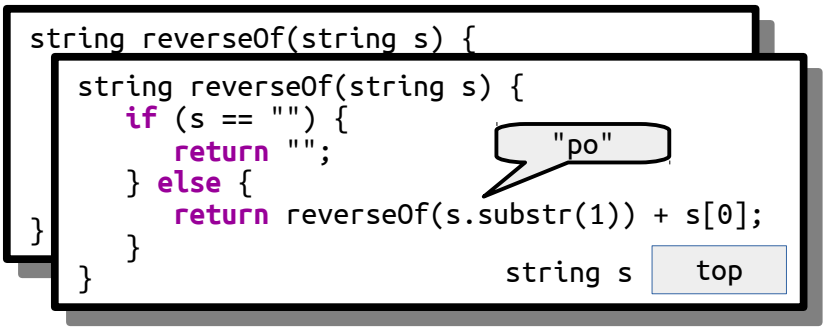
This triggers the base case and returns the empty string. (Notice that the reverse of the empty string "" is indeed the empty string ""):



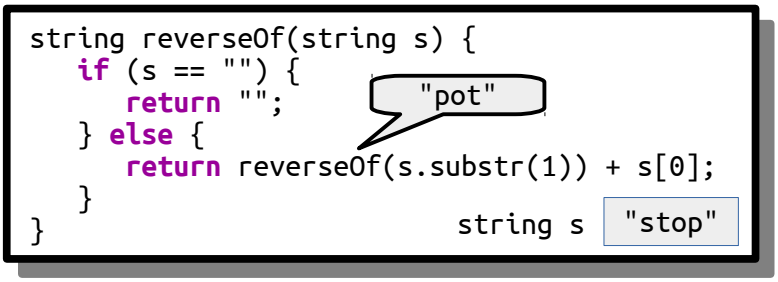
We now append p to return "p":



We now append o to return "po":



We append t to return "pot":



And finally we append s to return "pots" back to whoever called us. Yay!

Problem Three: Testing and Debugging

Here's a list of the errors in the code:

1. It uses the `string` type instead of the `char` type when representing individual characters in the string. This will cause the code to not compile.
2. On the first iteration of the loop, we will try to look at the -1st character of the string, which will probably cause a crash and definitely is wrong.
3. The `return` statement inside the `for` loop means that we'll never look at more than one pair of characters; the function will exit as soon as the `return` statement is executed, so we can't progress from one iteration to the next.
4. If the string doesn't contain any doubled characters, the function never returns a value.

We can fix all of these errors by rewriting the code like this:

```
bool hasDoubledCharacter(string text) {
    for (int i = 1; i < text.size(); i++) {
        char current  = text[i];
        char previous = text[i - 1];
        if (current == previous) {
            return true;
        }
    }
    return false;
}
```

To make things cleaner, we could remove the `current` and `previous` variables:

```
bool hasDoubledCharacter(string text) {
    for (int i = 1; i < text.size(); i++) {
        if (text[i] == text[i - 1]) {
            return true;
        }
    }
    return false;
}
```

Although we hadn't talked about pass-by-`const`-reference in the first week of class, we really should use pass-by-`const`-reference here because we are reading but not modifying the `text` parameter:

```
bool hasDoubledCharacter(const string& text) {
    for (int i = 1; i < text.size(); i++) {
        if (text[i] == text[i - 1]) {
            return true;
        }
    }
    return false;
}
```

(Continued on the next page…)

Now, let's talk testing. Notice that the test cases we have are purely for

- strings that have doubled characters,
- where the doubled letters are at the beginning,
- where there are no undoubled characters,
- where the doubled characters are lower-case letters, and
- that have length exactly two.

To find some classes of strings that don't have these properties, we can simply break all of the above rules and see what we find! So let's write tests for each of the following types of strings:

- strings that don't have doubled letters;
- strings that have doubled letters, but not at the beginning;
- strings that have doubled letters, but also some non-doubled letters;
- strings that have doubled non-letter characters; and
- strings whose lengths aren't two (maybe shorter strings or longer strings).

Here's some sample tests we could write:

```
STUDENT_TEST("Strings without doubled characters") {
    EXPECT(!hasDoubledCharacter("abcd")); // Nothing doubled
    EXPECT(!hasDoubledCharacter("aba"));  // a appears twice, but not consecutively
    EXPECT(!hasDoubledCharacter("Aa"));   // Not technically the same character
}

STUDENT_TEST("Strings with doubled characters not at the front") {
    EXPECT(hasDoubledCharacter("abb"));      // Back
    EXPECT(hasDoubledCharacter("abcddabc")); // Middle
}

STUDENT_TEST("Strings with doubled non-letter characters") {
    EXPECT(hasDoubledCharacter("**")); // Symbols
    EXPECT(hasDoubledCharacter("  ")); // Spaces
    EXPECT(hasDoubledCharacter("00")); // Numbers
    EXPECT(hasDoubledCharacter("!!")); // Punctuation
}

STUDENT_TEST("Short strings") {
    EXPECT(!hasDoubledCharacter(""));  // Too short
    EXPECT(!hasDoubledCharacter("a")); // Too short
}
```

Problem Four: Human Pyramids

The key recursive insight here is that a human pyramid of height 0 is a pyramid of no people, and that a human pyramid of height *n* is a group of *n* people supporting a human pyramid of *n*-1 people. Using that idea, we can write this function:

```
int peopleInPyramidOfHeight(int n) {
    if (n == 0) {
        return 0;
    } else {
        return n + peopleInPyramidOfHeight(n - 1);
    }
}
```

As a note, you can directly evaluate `peopleInPyramidOfHeight(n)` by computing $n(n + 1) / 2$. We'll see a really cool intuition for this later in the quarter!

Problem Five: Random Shuffling

Here is one possible solution:

```
string randomShuffle(string input) {
   /* Base case: There is only one possible permutation of a string
    * with no characters in it.
    */
   if (input == "") {
      return input;
   } else {
      /* Choose a random index in the string. */
      int i = randomInteger(0, input.length() - 1);

      /* Pull that character to the front, then permute the rest of
       * the string.
       */
      return input[i] + randomShuffle(input.substr(0, i) + input.substr(i + 1));
   }
}
```

This function is based on the recursive observation that there is only one possible random shuffle of the empty string (namely, itself), and then using the algorithm specified in the handout for the recursive step.

Problem Six: Computing Averages

There are many ways to write this function. Here's one.

```
Statistics documentStatisticsFor(istream& input) {
    /* Read an initial value out of the file. We're going to use this to seed
     * the minimum and maximum values.
     */
    double val;
    input >> val; // Should do error-checking, but we'll skip that here.

    /* Create a Statistics object and initialize its min and max values. */
    Statistics info;
    info.min = val;
    info.max = val;

    /* Keep track of the sum of the values and the number of values so that we
     * can report back the average value. Initially, the sum is whatever value
     * we just read, and the number of values read is 1.
     */
    double total = val;
    int numValues = 1;

    /* Read the rest of the file, updating the min/max as we go along with the
     * sum and number of values.
     */
    while (input >> val) {
       numValues++;
       total += val;

       if (val > info.max) info.max = val;
       if (val < info.min) info.min = val;
    }

    /* Compute the average. */
    info.average = total / numValues;
    return info;
}
```

Problem Seven: Haiku Detection

```
/* We will assume that
 * this function works. It's fun to
 * implement. Try it!
 */
int syllablesIn(string word);

/* These are prototypes.
 * They let us call these functions
 * Before they're defined.
 */
bool isHaiku(string line1, string line2, string line3);
int syllablesInLine(string line);

int main() {
      string line1 = getLine("Enter the first line: ");
      string line2 = getLine("Now, enter the second line: ");
      string line3 = getLine("Enter the third line: ");

      /* Given these three lines,
       * check whether they're a haiku,
       * then show the result.
       */
      if (isHaiku(line1, line2, line3)) {
            cout << "The text you entered" << endl;
            cout << "Goes 5 - 7 - 5, so it" << endl;
            cout << "is a haiku. Yay!" << endl;
      } else {
            cout << "Though you have tried hard," << endl;
            cout << "The three lines you entered are" << endl;
            cout << "Not a haiku. Awww." << endl;
      }

      return 0;
}

/* Given a poem
 * of three lines, returns whether
 * it is a haiku.
 */
bool isHaiku(string line1, string line2, string line3) {
      return syllablesInLine(line1) == 5 &&
             syllablesInLine(line2) == 7 &&
             syllablesInLine(line3) == 5;
}
```

```
/* Counts the number of
 * syllables in a line of
 * text, then returns it.
 */
int syllablesInLine(string text) {
      /* To split apart the
       * text, make a TokenScanner
       * and configure it.
       */
      TokenScanner scanner(text);
      scanner.ignoreWhitespace();

      int numSyllables = 0;
      while (scanner.hasMoreTokens()) {
            /* If this token is
             * a word, count its syllables
             * and update total.
             */
            string token = scanner.nextToken();
            if (scanner.getTokenType(token) == WORD) {
                  numSyllables += syllablesIn(token);
            }
      }

      return numSyllables;
}

/* Did you notice that
 * all the comments are haikus?
 * Same with the output!
 */
```