Solution 1: Publishing Stories

There’s the one-pass approach that just appends characters from the template to the running story, unless that character is '{', in which case we extract everything up through the matching '}' and replace it with a string from the data map.

```cpp
const string generateStory(const string& storyTemplate,
                           const Map<string, string>& data) {
  string story;
  for (int i = 0; i < storyTemplate.size(); i++) {
    if (storyTemplate[i] != '{') {
      story += storyTemplate[i];
    } else {
      int end = storyTemplate.find('}', i + 1);
      string token = storyTemplate.substr(i + 1, end - i - 1);
      story += data[token];
      i = end;
    }
  }
  return story;
}
```

Another approach is to iterate over the data map using a modern for loop dialect and drive the substitution that way. It’s less efficient, but it’s more straightforward, and a perfectly acceptable answer for the purposes of a discussion section.

```cpp
static string substituteToken(string story,
                              const string& token, const string& value) {
  int start = 0;
  while (true) {
    int found = story.find(token, start);
    if (found == string::npos) return story;
    story.replace(found, token.size(), value);
    start = found + value.size() + 1;
  }
}

string generateStory(const string& storyTemplate,
                      const Map<string, string>& data) {
  string story = storyTemplate;
  for (const string& token : data) {
    story = substituteToken(story, '{' + token + '}', data[token]);
  }
  return story;
}
```
Solution 2: Topswopping and Topswop Numbers

```java
static int getTopswopNumber(Stack<Integer> s) {
    int count = 0;
    while (true) {
        int top = s.peek();
        if (top == 1) return count;
        Queue<Integer> q;
        for (int i = 0; i < top; i++) q.enqueue(s.pop());
        while (!q.isEmpty()) s.push(q.dequeue());
        count++;
    }
    return count;
}
```

Solution 3: Chain Reactions

This first function was given to you in the section handout, as it’s algorithmically straightforward. It answers the question as to whether or not an exploding mine—the one referenced by `base`—can detonate a second mine—referenced by `target`. (By the way, we use `GPoint`s from "gtypes.h" to model the locations of land mines, because a full implementation would need to render land mines in a graphics window anyway, and those renderings would need to be framed in terms of `GPoint`s.)

```java
const static double kThresholdDistance = 50;
static bool isInRange(const GPoint& base, const GPoint& target) {
    double dx = base.getX() - target.getX();
    double dy = base.getY() - target.getY();
    return dx * dx + dy * dy <= kThresholdDistance * kThresholdDistance;
}
```

The function you’re responsible for needs to add a `GPoint` to the `scores` map for every `GPoint` that appears in the `landMines` set. The only way to reach all entries in a set is to use iteration via a `for` loop. On behalf of each `GPoint` in `landMines`, we compute the score achieved by manually detonating the mine at that location and simulating the chain reaction of detonations that stem from it.

```java
static void computeAllScores(const Set<GPoint>& landMines, Map<GPoint, int>& scores) {
    for (const GPoint& landMine: landMines) {
        scores[landMine] = computeScore(landMine, landMines);
    }
}
```

Of course, the interesting work comes with `computeScore`, which identifies all land mines that detonate at the one-second mark, and then all land mines that detonate at the two-second mark, and so on, until no more mines explode, all the while keeping tabs on how many points the chain reaction of detonations gets you.
static int computeScore(const GPoint& init, const Set<GPoint>& landMines) {
    int second = 0;
    int totalScore = 0;
    Set<GPoint> notYetExploded = landMines;
    notYetExploded -= init; // not exploding now, but they may eventually
    Set<GPoint> currentlyExploding;
    currentlyExploding += init; // at the outset, only init detonates

    while (!currentlyExploding.isEmpty()) {
        int score = 100 * second * second;
        totalScore += currentlyExploding.size() * score;
        Set<GPoint> soonExploding;
        for (const GPoint& explodingLocation: currentlyExploding) {
            for (const GPoint& potentialLocation: notYetExploded) {
                if (isInRange(explodingLocation, potentialLocation)) {
                    soonExploding += potentialLocation;
                }
            }
        }
        notYetExploded -= soonExploding;
        currentlyExploding = soonExploding;
        second++;
    }

    return totalScore;
}

Solution 4: Stepping Stones

static void generatePath(const Grid<string>& stones, const stone& start, const stone& finish, Vector<stone>& shortest) {
    Vector<stone> startPath;
    startPath += start;
    Queue<Vector<stone> > partialPaths;
    partialPaths.enqueue(startPath);
    while (true) {
        shortest = partialPaths.dequeue();
        const stone& last = shortest[shortest.size() - 1];
        if (last == finish && shortest.size() % 3 == 2) return;
        for (Direction dir = NORTH; dir <= WEST; dir++) {
            stone neighbor = getNeighboringStone(last, dir);
            if (hopIsAllowed(stones, shortest, neighbor)) {
                Vector<stone> clone = shortest;
                clone += neighbor;
                partialPaths.enqueue(clone);
            }
        }
    }
}

Because statements like last == finish & shortes

Sizes like last == finish & shortes.size() % 3 == 2 are pretty

Because statements like last == finish & shortest.size() % 3 == 2 are pretty crunchy, I decided that the stone hopping details could be abstracted away into a helper function so that the top-level algorithm is clearly breadth-first search. A core value of mine (and this is Jerry typing) is that you should always, always, always work to improve the narrative of an algorithm, and decomposing as I have here is a surefire way to do that.
static bool hopIsAllowed(const Grid<string>& stones, Vector<stone>& shortest, const stone& neighbor) {
    if (!stones.inBounds(neighbor.row, neighbor.col) ||
        (shortest.size() > 1 && neighbor == shortest[shortest.size() - 2])) {
        return false;
    } // out of bounds or a U-turn? reject

    const stone& end = shortest[shortest.size() - 1];
    const string& beforeColor = stones[end.row][end.col];
    const string& afterColor = stones[neighbor.row][neighbor.col];
    return (beforeColor == afterColor) == (shortest.size() % 3 != 1);
}