Section Solution

Discussion Problem 1 Solution: Chain Reactions

This was one last problem to exercise your understanding of **Map**s and **Set**s, and in particular, the set union, intersection, and subtraction operations that come via **+**, **-**, and *****. This was a final exam question of mine some three years ago.

This first function was given to you in the 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.)

```
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 our **foreach** construct. 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.

```
static void computeAllScores(const Set<GPoint>& landMines,
                             Map<GPoint, int>& scores) {
    foreach (GPoint landMine in 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;
      foreach (GPoint explodingLocation in currentlyExploding) {
         foreach (GPoint potentialLocation in notYetExploded) {
            if (isInRange(explodingLocation, potentialLocation)) {
               soonExploding += potentialLocation;
            }
         }
      }

      notYetExploded -= soonExploding;
      currentlyExploding = soonExploding;
      second++;
   }

   return totalScore;
}
```

Discussion Problem 2 Solution: GCD

This algorithm—which was presented in iterative form in Section 2.2 of the reader—can be formulated recursively as well. Because the implementation is so short, you might trace through calls to **`gcd(12, 114)`**, **`gcd(114, 12)`**, and **`gcd(1001, 200)`** so you trust that it works. We won't always be able to trace through a recursive implementation, but when we can, it's a perfectly good way to confirm it's operational.

```
static int gcd(int x, int y) {
    if (x % y == 0) return y;
    return gcd(y, x % y); // argument order is important here
}
```

Discussion Problem 3 Solution: Twiddles

This problem is substantially more difficult that the preceding one, because the recursive substructure isn't nearly as obvious (and it isn't explained in the problem statement.) Key observation: finding twiddles is the same as fixing the first letter (one of up to five possibilities) and appending some twiddle of the remaining letters. A **`'c'`** at **`str`**'s position 0, for instance, encodes the fact that **`'a'`**, **`'b'`**, **`'c'`**, **`'d'`**, or **`'e'`** might contribute to a potential twiddle at position 0. And for each of those five possibilities at position 0, there are five contributions at position 1, and for each of those 25 possible possibilities between 0 and 1 combined, there are five independent contributions that might be made at position 2, and so on, and so on.

```
static void listTwiddles(const string& str, const Lexicon& lex) {
    listTwiddles("", str, 0, lex);
}
```

- The 0th argument is the empty string to clarify that no decisions made been made at the outset.
- The 2nd argument is **0** to be clear that **str[0]** is the character that tells us how me might extend the empty string into five different prefixes of length 1.

```
static void listTwiddles(const string& prefix, const string& str, int index,
                         const Lexicon& lex) {

    if (!lex.containsPrefix(prefix)) return; // not strictly necessary
    if (index >= str.size()) {
        if (lex.contains(prefix)) cout << prefix << endl;
        return;
    }

    for (char ch = str[index] - 2; ch <= str[index] + 2; ch++) {
        if (isalpha(ch)) {
            listTwiddles(prefix + ch, str, index + 1, lex);
        }
    }
}
```

Lab Problem 1 Solution: Letter Rectangles and Words

My implementation wraps the three-argument version around a call to a four-argument version. The overloaded version of **gatherWords**—that one that really does all of the work—keeps track of the running prefix built up by an ordered selection of (possibly rotated) rectangles leading up to the call. Initially, we haven't selected any rectangles, which is why my wrapper passes an empty string in as the 0th parameter.

```
static void gatherWords(const Vector<string>& rects,
                        const Lexicon& english, Lexicon& words) {
    Vector<string> copy = rects;
    gatherWords("", copy, english, words);
}

static void gatherWords(const string& prefix, Vector<string>& rects,
                        const Lexicon& english, Lexicon& words) {
    if (!english.containsPrefix(prefix)) // prefix is nonsense?
        return; // pretend we never made this call
    if (english.contains(prefix)) // prefix is a word?
        words.add(prefix); // incidentally print, but continue

    for (int i = 0; i < rects.size(); i++) {
        string rect = rects[i];
        rects.remove(i); // temporarily remove so it doesn't get used twice
        gatherWords(prefix + rect[0] + rect[1], rects, english, words);
        gatherWords(prefix + rect[1] + rect[0], rects, english, words);
        rects.insert(i, rect); // insert back so it can be used deeper down
    }
}
```