Magic(sanitized).cpp

```
#include "Vector.h"
#include "Square.h"


void PuzzleSolve(Square& S, Vector<int>& V) {
  int remaining = V.size();

  if (remaining==0) {
    if (S.valid())
      cout << S << endl;   // found a solution
  } else {
    /*
     * For each item of vector V, use that item to fill the chosen cell, and recurse
     */
    for (int rank = 0; rank<remaining; rank++) {

      if (S.add(V.elemAtRank(rank))) {  // if we can add the given value to the square,
        Vector<int> newV(V);            //    create a copy of vector V
        newV.removeAtRank(rank);        //      but without the used item
        PuzzleSolve(S,newV);            //    recurse
      }

      S.pop();                          // remove the previously added value
    }
  }
}


int main(int argc, const char* argv[]) {

  /*
   * The first command-line argument is used to specify n
   */

  /*
   * We create the initially empty square, and a list of values to
   * be used, from 1 to n^2.
   */
  Square S(n);
  Vector<int> L;
  for (int i=0; i<n*n; i++)
    L.insertAtRank(i,i+1);

  /*
   * Let the recursion begin...
   */
  PuzzleSolve(S,L);
}
```

Square(public sanitized).h

```
class Square {
 public:
  /*
   * Creates an nxn square.
   */
  Square(int width=3);

  /*
   * This is used to add a new value to an 'empty' cell of the square.
   * Which empty cell is left as an implementation detail of the Square.
   *
   * The boolean return value is 'false' if the newly added value is
   * known to cause a (partially) complete square which is guaranteed
   * to be invalid, no matter how the remaining squares are completed.
   */
  bool add(int value);

  /*
   * This removes the most recently added value from the square
   */
  void pop();

  /*
   * Return the width of the square
   */
  int width() const;

  /*
   * This accessor returns the (row,column) entry to value, where both
   * rows and columns are zero-indexed.
   *
   * Returns '-1' if the command fails (e.g., the indicies are invalid)
   */
  int get(int row, int column) const;

  /*
   * Checks validity of the current settings, ensuring that all rows,
   * columns and diagonals add up to the desired value.  Furthermore,
   * it verifies that each number from [1, n^2] has been used once,
   * and only once.
   */
  bool valid();

  /*
   * Destructor
   */
  ~Square();
};
```

Square(private sanitized).h

```
class Square {
 private:

  int n;          // We are representing an (n x n) square
  int max;        //   with desired values from 1 to n^2
  int target;     //   and desired sum for each row of n*(n^2+1)/2

  int **entry;    // two-dimensional array of entries
  int numFilled; // a count of the number of filled cells thus far
  bool *used;     // this is used for validation


  /*
   * The first of the following five functions is able to generically
   * check the validity of a particular cross-section (e.g., row,
   * column, diagonal).
   *
   * For legibility, we introduce the other four forms of the check,
   * though each of those is mapped back to the generic form.
   */
  bool checkGeneric(int startRow, int startCol, int deltaRow, int deltaCol);
  bool checkRow(int row) { return checkGeneric(row,0,0,1);  }
  bool checkCol(int col) { return checkGeneric(0,col,1,0);  }
  bool checkDiag()       { return checkGeneric(0,0,1,1);    }
  bool checkRevDiag()    { return checkGeneric(n-1,0,-1,1); }

  /*
   * Presuming that (row,col) was the most recently set entry, this
   * method attempts to determine whether that entry invalidates the
   * partial solution.
   *
   * If it becomes clear that this solution cannot be extended to a
   * valid solution, this method returns false.  Otherwise it returns
   * true (Note that it still may be impossible to complete the
   * solution).
   */
  bool partialValidate(int row, int col);

  /*
   * Checks whether the current (partial) settings is in canonical form.
   * That is with top-left corner as the smallest of the corners, and
   * top-right corner as the smaller of its two adjacent corners.
   */
  bool canonical();

  /*
   * A representative of a cell, for convenience
   */
  struct Cell {
    int r;
    int c;
  };

  /*
   * In an nxn square, there are n^2 spots to fill in eventually.
   * Assuming that 'prevCount' cells have already been filled, this
   * routine identifies where in the square the next insertion should be
   * placed.
   */
  Cell whichCell(int prevCount, int n);
};
```