```cpp
// driver_app.cpp
// Michael E. Sparks, 11-16-16

// Program to enumerate all solutions to the N-Queens
// problem, for any sensible values of N.

#include <iostream>
#include <cstdlib>
#include "NQueens.h"

using namespace std;

// Driver application
int main(int argc,char **argv)
{
  NQueens my_nqueens;

  // Two calls to atoi seem wasteful, but the
  // N_t type used by NQueens objects is of
  // an unsigned type. If the shell passes in
  // a negative number, bad things will happen.
  if(argc > 1 && atoi(argv[1]) > 0)
    my_nqueens.set_N((N_t)atoi(argv[1]));
  // else we just use the NQueens class'
  // default value of N

  cout << endl << "Solutions to "
       << my_nqueens.get_side_length()
       << "-Queens:" << endl;

  my_nqueens.enumerate_NQueens_solutions();

  exit(EXIT_SUCCESS);
}
```

```cpp
// NQueens.cpp
// Michael E. Sparks, 11-16-16

#include <iostream>
#include "NQueens.h"

using namespace std;

// Because this predicate can score both complete and partial
// solutions, it enables a branch-and-bound approach to probing the
// solution space. (This is in contrast to enumerating and testing
// every potential (complete) puzzle solution.)
bool NQueens::is_it_feasible(const bool* soln, const N_t& N)
{
  bool q_seen{false};

  // Test for multiple queens on a given row
  for(auto i{0};i<N;++i,q_seen=false)
    for(auto j{0};j<N;++j)
      if(soln[IND(i,j,N)]) {
        if(q_seen)
          return false;
        else
          q_seen=true;
      }

  // Test for multiple queens on a given column
  for(auto j{0};j<N;++j,q_seen=false)
    for(auto i{0};i<N;++i)
      if(soln[IND(i,j,N)]) {
        if(q_seen)
          return false;
        else
          q_seen=true;
      }

  // Test for multiple queens on diagonals
  // parallel to the main diagonal.
  // Cases where diagonals include first column...
  for(auto i{N-2};i>=0;--i,q_seen=false)
    for(auto ii{i},j{0};ii<N;++ii,++j)
      if(soln[IND(ii,j,N)]) {
        if(q_seen)
          return false;
        else
          q_seen=true;
      }
  // ...and the rest.
  for(auto j{1};j<N-1;++j,q_seen=false)
    for(auto jj{j},i{0};jj<N;++jj,++i)
      if(soln[IND(i,jj,N)]) {
        if(q_seen)
          return false;
        else
          q_seen=true;
      }

  // Test for multiple queens on diagonals
  // parallel to the antidiagonal.
  // Cases where antidiagonals include last column...
  for(auto i{N-2};i>=0;--i,q_seen=false)
    for(auto ii{i},j{N-1};ii<N;++ii,--j)
      if(soln[IND(ii,j,N)]) {
        if(q_seen)
          return false;
        else
          q_seen=true;
      }
  // ...and the rest.
```



```cpp
  for(auto j{N-2};j>0;--j,q_seen=false)
    for(auto jj{j},i{0};jj>=0;--jj,++i)
      if(soln[IND(i,jj,N)]) {
        if(q_seen)
          return false;
        else
          q_seen=true;
      }

  return true;
}

// Verifies that a feasible solution has all queens placed.
// This could be used as a sanity check above and beyond using
// the is_it_feasible predicate on the last row. Doing so is
// really not necessary when using the recurse_over_solns member,
// though, as you would have already placed N-1 queens on the board
// in a satisfactory configuration before ever getting a chance to
// place the Nth queen.
bool NQueens::is_it_complete(const bool* soln, const N_t& N)
{
  N_t count{0};

  for(auto i{0};i<N;++i)
    for(auto j{0};j<N;++j)
      if(soln[IND(i,j,N)])
        ++count;

  return (count==N && is_it_feasible(soln,N));
}

// Recursion allows generalization of N-Queens to any "reasonable" N.
void NQueens::recurse_over_solns(const N_t& row,const N_t& N) {
  for(auto i{0};i<N;board[IND(row,i++,N)]=false) {
    board[IND(row,i,N)]=true;

    // We could save time by caching results from testing the
    // feasibility of prior board configurations, but I don't
    // want to "clutter" this example with code for memoization,
    // use of coroutines, etc. We're more after demonstrating the
    // general "shape" of applying a filter to prune search efforts.
    if(is_it_feasible(board,N)) {
      if(row==N-1)
        print_board();
      else
        recurse_over_solns(row+1,N);
    }
  }

  return;
}

// interactively set N.
N_t NQueens::set_N(void)
{
  side_length=0;
  while(!side_length) {
    cout << "Please enter N (.GT. 0) for NQueens: ";
    cin >> side_length; // yes, this is dangerous
  }

  return (N_t)side_length;
}

// passively/ programmatically set N.
N_t NQueens::set_N(N_t proposed_N)
{
  try {
    // yes, this is also dangerous
```

```
    if((side_length=proposed_N)<=0)
      throw domain_error("N .LE. 0");
  }
  catch(domain_error& e) {
    cerr << "*** Warning: " << e.what()
         << " ; using " << default_size
         << " instead of " << side_length
         << " ***" << endl;
    side_length=default_size;
  }

  return (N_t)side_length;
}

// Manages the process of enumerating N-Queens solutions.
// Note that we're leaving plenty of optimization tricks on
// the table that would be evident upon studying the N-Queens
// problem in greater detail (e.g., symmetry of solutions).
// What we're after here is more to demonstrate the generality
// achievable with using a recursive solution.
void NQueens::enumerate_NQueens_solutions(void)
{
  N_t& N=side_length; // a reference to save typing!
  try {
    board=new bool[N*N];
  }
  catch(bad_alloc&) {
    cerr << "Insufficient memory to allocate "
         << N << " X " << N << " board\n";
    return;
  }

  clear_board();

/*
  // Here's a known solution for N .EQ. 8 :
  if(N==default_size) {
    board[IND(0,5,N)]=true;
    board[IND(1,0,N)]=true;
    board[IND(2,4,N)]=true;
    board[IND(3,1,N)]=true;
    board[IND(4,7,N)]=true;
    board[IND(5,2,N)]=true;
    board[IND(6,6,N)]=true;
    board[IND(7,3,N)]=true;

    print_board();
  }
*/

/*
  // Here's an approach to find all such solutions,
  // specifically all 92 for the 8-Queens problem:
  if(N==default_size) {

    for(auto i{0};i<N;
        board[IND(0,i++,N)]=false) { // first row
      board[IND(0,i,N)]=true;
      if(!is_it_feasible(board,N)) continue; // unnecessary here

      for(auto j{0};j<N;
          board[IND(1,j++,N)]=false) { // second
        board[IND(1,j,N)]=true;
        if(!is_it_feasible(board,N)) continue;

        for(auto k{0};k<N;
            board[IND(2,k++,N)]=false) { // third
          board[IND(2,k,N)]=true;
          if(!is_it_feasible(board,N)) continue;
```



```
          for(auto l{0};l<N;
              board[IND(3,l++,N)]=false) { // fourth
            board[IND(3,l,N)]=true;
            if(!is_it_feasible(board,N)) continue;

            for(auto m{0};m<N;
                board[IND(4,m++,N)]=false) { // fifth
              board[IND(4,m,N)]=true;
              if(!is_it_feasible(board,N)) continue;

              for(auto n{0};n<N;
                  board[IND(5,n++,N)]=false) { // sixth
                board[IND(5,n,N)]=true;
                if(!is_it_feasible(board,N)) continue;

                for(auto m{0};m<N;
                    board[IND(6,m++,N)]=false) { // seventh
                  board[IND(6,m,N)]=true;
                  if(!is_it_feasible(board,N)) continue;

                  for(auto n{0};n<N;
                      board[IND(7,n++,N)]=false) { // eighth (& last)
                    board[IND(7,n,N)]=true;
                    if(!is_it_complete(board,N)) continue;

                    print_board();
    }}}}}}}}
  }
*/

  // Now we need to generalize the pattern shown above
  // so that it can be applied to arbitrary values of N.
  // Given the "similarity" of the nested for-loops,
  // a recursive solution to the problem becomes apparent.
  recurse_over_solns(0,N);

  delete [] board;

  return;
}
```

```
// NQueens.h
// Michael E. Sparks, 11-16-16

#include "Square_board.h"

// Objects of type NQueens exhaustively enumerate
// all solutions to the N-Queens puzzle.
class NQueens : public Square_board
{
  private:

    // Predicate tests whether a solution (partial
    // up through complete) is potentially correct.
    bool is_it_feasible(const bool* soln, const N_t& N);

    // Test if a solution to the puzzle both has N queens
    // placed and is feasible.
    bool is_it_complete(const bool* soln, const N_t& N);

    // Auxiliary function used by enumerate_NQueens_solutions
    void recurse_over_solns(const N_t& row,const N_t& N);

  public:

    // set_N uses run-time polymorphism / overloading to
    // give user code the option of setting N (to something
    // other than default_size) interactively or passively.
    // Neither of these member functions is written in an
    // especially robust manner, as this app is just a toy.
    N_t set_N(void); // interactive
    N_t set_N(N_t proposed_N); // passive/ programmatic

    // This method builds a set of all possible solutions
    // satisfying the underlying constraints of this puzzle.
    void enumerate_NQueens_solutions(void);
};
```

```
// Square_board.cpp
// Michael E. Sparks, 11-16-16

#include <iostream>
#include "Square_board.h"

using namespace std;

// Allow users of the class to learn side_length
N_t Square_board::get_side_length(void)
{
  return side_length;
}

// Initialize the board, setting all positions to false
void Square_board::clear_board(void)
{
  for(auto i{0};i<side_length;++i)
    for(auto j{0};j<side_length;++j)
      board[IND(i,j,side_length)]=false;

  return;
}

// Report configuration to stdout
void Square_board::print_board(void)
{
  cout << endl;
  for(auto i{0};i<side_length;++i)
    for(auto j{0};j<side_length;++j) {
      cout << (board[IND(i,j,side_length)] ? 'Q' : '-') << " ";
      if(j==side_length-1)
        cout << endl;
    }
  cout << endl;

  return;
}
```

```
// Square_board.h
// Michael E. Sparks, 11-16-16

// Accommodates square boards having side lengths from 1 up to 0xFFFF units.
// Whether the host machine can acommodate boards of that size is another
// matter.
typedef unsigned short N_t;

// The Square_board class implements a square board suitable
// for representing a chess or checkers game state, for example.
// It is intended for use as a base class only.
class Square_board
{
  public:

    N_t get_side_length(void); // allow users to learn side_length

  protected:

    static const N_t default_size {8}; // 8-Queens is the typical form

    N_t side_length {default_size};

    bool* board {nullptr}; // num(positions) = side_length ** 2

    void clear_board(void); // set all positions to false

    void print_board(void);
};

// The following macro allows us to index a one-dimensional
// dynamically allocated array as if it were a two-dimensional,
// fixed-size array. With C's malloc, it's possible to create
// a dynamic array of pointers, each of which can be used to
// allocate a dynamic array of unit type elements. That's great
// for jagged-edge arrays, for example. (Here, we wish to focus
// on C++'s new/delete, though.) This indexing approach is also
// more performant, as data are stored contiguously in memory.
// Extension of this concept for indexing one-dimensional
// memory as yet higher-order arrays (e.g., 3D, 4D) is left
// as an exercise.
#define IND(R, C, LEN) R * LEN + C
```