```
// This program demonstrates solves the knapsack problem

// By: Chris Gregg
// Date: April 23, 2018

#include <iostream>
#include "console.h"
#include "vector.h"
#include "simpio.h"

using namespace std;

// objectT struct
struct objectT {
    int weight;
    int value;
};

// function prototypes
int fillKnapsack(Vector<objectT> &objects, int targetWeight);
int fillKnapsack(Vector<objectT> &objects, int weight, int score);

int main() {
    // solution: 44
    int values[] = {12,10,8,11,14,7,9};
    int weights[] = {4,6,5,7,3,1,6};
    int targetWeight = 18;

    // solution: 67
    //int values[] = {5,20,3,50,5,4,15,12,6,7};
    //int weights[] = {6,15,11,12,6,11,13,7,17,13};
    //int targetWeight = 25;

    // solution: 7
    //int values[] = {3,4,5,6};
    //int weights[] = {2,3,4,5};
    //int targetWeight = 5;

    int numItems = sizeof(values) / sizeof(int);

    Vector<objectT> testObjects;

    for (int i=0; i < numItems; i++) {
        objectT object;
        object.value = values[i];
        object.weight = weights[i];
        testObjects.add(object);
    }

    cout << "Best solution has a best score of: "
         << fillKnapsack(testObjects, targetWeight) << endl;
    return 0;
}

int fillKnapsack(Vector<objectT> &objects, int targetWeight) {
    return fillKnapsack(objects,targetWeight,0);
}

int fillKnapsack(Vector<objectT> &objects, int weight, int bestScore) {
    if (weight < 0) return 0; // we tried too much weight!
        int localBestScore = bestScore;
        int obSize = objects.size();
        for (int i = 0; i < obSize; i++) {
            objectT originalObject = objects[i];
            int currValue = bestScore + originalObject.value;
            int currWeight = weight - originalObject.weight;
            // remove object for recursion
            objects.remove(i);
            currValue = fillKnapsack(objects,currWeight,currValue);
            if (localBestScore < currValue) {
                localBestScore = currValue;
            }
            // replace
            objects.insert(i,originalObject);
        }
        return localBestScore;
}
```

```
// solveMaze should be passed a human-readable maze
// with a start and finish, and a start position of 1,1
// dead-end paths that have been tested should be marked
// with lowercase x
// The correct path should be marked with periods, .
void solveMazeRecursive(Grid<int> &maze) {
    solveMazeRecursive(1,1,maze);
    maze[1][1] = 'S'; // replace start, which is removed during solving stage
}

bool solveMazeRecursive(int row, int col, Grid<int> &maze) {
    // be careful! 2d arrays seem backwards and upside down
    // compared to cartesian coordinates:
    // maze[0][0] == top left
    // maze[1][0] == one down, zero to the right
    // maze[0][1] == zero down, one to the right
    // maze[row][col] == row down and col to the right

    if (maze[row][col] == 'X') {
        return false;
    }

    if (maze[row][col] == '.') {
        return false;
    }

    if (maze[row][col] == 'F') {
        return true;
    }

    maze[row][col] = '.';

    // Recursively call solveMazeRecursive(row,col)
    // for north, east, south, and west
    // If one of the positions returns true, then return true

    // north
    if (solveMazeRecursive(row-1,col,maze)) {
        return true;
    }

    // east
    if (solveMazeRecursive(row,col+1,maze)) {
        return true;
    }

    // south
    if (solveMazeRecursive(row+1,col,maze)) {
        return true;
    }

    // west
    if (solveMazeRecursive(row,col-1,maze)) {
        return true;
    }

    maze[row][col] = 'b';
    return false;
}
```