The Winning Hand

New Mexico High School Supercomputing Challenge
Final Report
April 4, 2001

Team # 045
Lovington High School

Team Members:
Joel Lowry
Randy Brown
Tim Newell

Team Sponsor:
Pam Gray

Project Mentor:
Tom Laub

Table of Contents

Executive Summary

Everyone wants to win big. But lets get real! What are your chances? With this project, we have taken what seems to be a simple card game, and broken it down to its mathematical basis. Our team used the Monte Carlo methods, which simulates real world events. A program has been written that deals, identifies, and categorizes a hand. It counts the amount of each hand drawn, and then uses chosen because we wanted something that we were interested in, and at the same time, on a level to where we could gain applicable information that could be used and interpreted by everyone. Our project was started in C++, but eventually turned to Visual C++ for a quicker and more efficient means of programming**.** Our program acts in a series of several steps. First, through the header files that were written, we identified cards, decks, and hands. Then, through a program that linked all these together, cards were allowed to be drawn from the deck and into the hand. Next, the program identified each hand. At the same time, a patch was run to a file that would allow storage of the data, and this data was used to present a graph of distribution so that a margin of error could be calculated. After all hands have been identified, the program divides the total of each by the total trials and outputs the data. Our data from five-card stud was accurate to calculated values. This allowed us to perform the same procedures with seven-card stud with confidence that it would be correct. We hope to allow for more games and decks in the future, as well as adding a direct interface for amount of hands dealt. We also hope to move into parallelism at ome point. Future applications of this program go from current function, to, on a more general scale, more accurately predicting probability of any number of things. these statistics to calculate the probability of drawing this hand. This project was

The Winning Hand

INTRODUCTION:

Project Description:

The goal of this project was to present a valid set of information that depicts every different winning hand in a game of "5 Card Stud". To encompass a wider range of possibilities we developed our program so that we could change the number of players, although this isn't a property a user can change directly, which better ties our procedure and results to real world circumstances. First we began by researching the probability and mathematics of "5 Card Stud". After we gained enough knowledge to understand the equations we were ready to simulate the game of poker and compare the results. Since it is impossible to deal and decipher more then 100,000,000 hands physically we sought out the power of Visual C++.

The reason we chose this project was because this is a real world question and the results are attainable and the validity can be checked using advanced mathematics. We wanted something interesting, yet on a level that can be understood and appreciated by everyone.

The Program:

In working on our project, we went through many trials in terms of executable code. In our final code presented in Appendix A, we came to a program that defined the probabilities for drawing any given hand in both five and seven card stud. To achieve this goal, and to be sure that what we had were valid results, we first wrote a program that

sorted hands based on value. Any hand that did not come out to a valid winning hand was categorized as “nothing”, or a hand that would not win. Pairs, Two Pairs, Three of a Kind, Full Houses, Four of a Kind, Straights, Flushes, Straight Flushes, and Royal Flushes, were categorized accordingly. Next, we wrote a program that, through a nested loop, drew random hands, categorized the hands, and calculated the probability of drawing these hands based on solid observation. When this program was done, we compared it to the mathematically derived probabilities. When we were content with the accuracy of our program, we then ventured forward. From here, we wrote a program that would do the same things, only for a hand in seven-card stud. We did this in confidence that our program would be accurate, due to the previous testing with five-card stud. When these two programs were completed, we wrote a mother program that allowed for the combination of the two, and gave the user the option of which game to simulate, as well as how many trials of the chosen game to simulate. This is our final program.

The Math:

The math behind our program is based upon simply finding probability. Using Monte Carlo methods of simulation, we performed an unlimited number of checks and stored the numbers of each hand type found. Then, we divided the number of found hands by the number of overall trials. This gave us our probability. The error factor that we have in our code is based upon a graph of normal distribution drawn from previous trials. The value found was then placed on this graph as a medium, and the graph calculated distance from the true value.

RESULTS:

Knowing the derived mathematical probabilities was essential to the progression of our project. The derivatives allowed us to validate our program so that we could move on with the confidence that our work was accurate. The mathematically calculated values are as follows:

Hand	Number of Combinations	Probability
Nothing	1,302,540	.5011773940
One Pair	1,098,240	.4225690276
Two Pair	123,552	.0475390156
Three of a Kind	54,912	.0211284514
Straight	10,200	.0039246468
Flush	5,108	.0019654015
Full House	3,744	.0014405762
Four of a Kind	624	.0002400960
Other Straight Flush	36	.0000138517
Royal Flush	4	.0000015391
Total	**2,598,960**	**1.000000000**

The values derived from our program were as follows:

Hand	Probability
Nothing	.501176
One Pair	.422568
Two Pair	.047540
Three of a Kind	.021129
Straight	.003924
Flush	.001964
Full House	.001440
Four of a Kind	.000240
Other Straight Flush	.000014
Royal Flush	.000001

The values for seven-card stud were as follows:

Hand	Probability
Nothing	.193252
One Pair	.473037
Two Pair	.221653
Three of a Kind	.049169
Straight	.025396
Flush	.009
Full House	.025417
Four of a Kind	.001340
Other Straight Flush	.001355
Royal Flush	.0000014

CONCLUSIONS:

Explanation

In conclusion, our program displayed the probabilities of drawing represented hands in a game of five or seven card stud. These probabilities showed relative accuracy, and the marginal errors displayed the room for improvement.

Future Applications

In the future, we hope to perform these same procedures, and include different games, as well as adding more decks, and allowing for interface to decide the amount of players. This programs concepts would be useful in finding probabilities of unlimited nature.

ACKNOWLEDGMENTS

We would like to thank everyone involved in our project:

Mrs. Pam Gray- For giving us her time, her effort, her money, and the motivation to excel.

Mr. Jimmy Crawford- For making are trips possible and fun.

Mr. Tom Laub- For being so patient and helping us so much in every step of our learning and using C++, you pushed us to the finish and challenged us to do great.

Mrs. Tina Sikes- For helping us get started, and giving us the will to finish.

Ais Sponsers- For sponsoring an event that helps us gain so much knowledge in an area which we have never been challenged or exposed.

BIBLIOGRAPHY

Davis, Stephen. C++ For Dummies 3rd Edition. Foster City: IDG Books Worldwide, Inc., 1998

Roberge, James. Engaged Learning For Programming in C++. Sudbury: Jones and Bartlett Publishers, Inc. 2001

Shammas, Namir. C++ For Dummies Quick Reference 2nd Edition. Foster City: IDG Books Worldwide, Inc., 1998

www.mathforum.org/dr.math/problems/ludwig.7.28.96.htm, Math Forum- Ask Dr. Math, Dr. Anthony, 7/28/96

www.geocities.com/durangobill/poker.html, Durango Bill's Poker Probabilities, Durango Bill, 2/2/02

http://mathforum.org/dr.math/problems/burtin2.11.99.htm, The Math Forum- Ask Dr. Math, Dr. Anthony, 2/6/02

APPENDIX A: The Final Code

Header Files:

card.h

```
#ifndef CARD_H
#define CARD_H

// Card class

//  Card objects have these properties, all private:
//    int FaceValue   = (1-13) indicating the face value of the card
//                      (1=Ace, 2-10, 11-13=Jack-King)
//    int Suit        = integer representation of suit (1,2,3,4)
//    string SuitName = the suit of the card ( "Diamond", "Heart", "Club", or "Spade" )
//    bool Drawn      = true or false indicating whether the card has been drawn

//  Card objects have these methods:
//   bool SetFaceValue(int i)      - sets FaceValue to i, returns true if successful
//   int GetFaceValue()            - returns the FaceValue of the card
//   bool SetSuitName(string suit) - sets the suit of the card by name, returns true if successful
//   bool SetSuit(int i)           - sets the suit of the card by number, returns true if successful
//   string GetSuitName()          - returns the SuitName of the card
//   bool IsDrawn()                - returns true if card has been drawn, false otherwise
//   bool Draw()                   - Draws the card and returns true if successful, false otherwise
//   void ReSet()                  - resets the Drawn property to false
//   void Print()                  - prints the card FaceValue and SuitName to the standard output
device
//   void PrintAll()               - same as Print() but prints the Drawn property too

#include <iostream>
#include <string>
using namespace std;

class Card
{
  private:
    int FaceValue;
    int Suit;
    string SuitName;
    bool Drawn;
```

```
  public:

// The SetFaceValue() method sets the FaceValue of the Card and
// returns true if successful or false if not
    bool SetFaceValue(int i)
    {
      if ( i < 1 || i > 14 )
      {
        return false;
      }
      else
      {
        FaceValue = i;
        if ( FaceValue == 1 ) FaceValue = 14;
        return true;
      }
    };

// The GetFaceValue() method get the FaceValue of the Card
    int GetFaceValue()
    {
      return FaceValue;
    };

// The SetSuitName() method sets the SuitName property the Card and
// returns true if successful or false if not
    bool SetSuitName(string suit)
    {
      if ( suit == "Club" )
      {
        SuitName = suit;
        Suit = 1;
        return true;
      }
      if ( suit == "Diamond" )
      {
        SuitName = suit;
        Suit = 2;
        return true;
      }
      if ( suit == "Heart" )
      {
        SuitName = suit;
        Suit = 3;
        return true;
      }
      if ( suit == "Spade" )
      {
        SuitName = suit;
```

```
        Suit = 4;
        return true;
      }

      return false;

    };

// The SetSuit() method sets the Suit property the Card and
// returns true if successful or false if not
    bool SetSuit(int i)
    {
      switch (i)
      {
        case 1 :
          SuitName = "Club";
          Suit = 1;
          return true;
        case 2 :
          SuitName = "Diamond";
          Suit = 2;
          return true;
        case 3 :
          SuitName = "Heart";
          Suit = 3;
          return true;
        case 4 :
          SuitName = "Spade";
          Suit = 4;
          return true;
        default :
          return false;
      }
    };

// The GetSuitName() method returns the SuitName property of the Card
    string GetSuitName()
    {
      return SuitName;
    };

// The GetSuit() method returns the Suit property of the Card
    int GetSuit()
    {
      return Suit;
    };

// The IsDrawn() method returns the Drawn property
   bool IsDrawn()
   {
```

```
    return Drawn;
  };

// The Draw() method simply sets the Drawn property to true
// and returns a true or false if successful or not
  bool Draw()
  {
    if ( !Drawn )
    {
      Drawn = true;
      return true;
    }
    else
    {
      return false;
    }
  };

// The Reset() method resets the Drawn property to false
  void Reset()
  {
    Drawn = false;
  };

// The Compare() method compares the argument card with the
// object card and returns true if they are the same
  bool IsSame(Card inCard)
  {
    if ( FaceValue == inCard.FaceValue  &&
         Suit == inCard.Suit )
    {
      return true;
    }
    else
    {
      return false;
    }
  };

// The Print() method of the card class prints out the
// card properties FaceValue and Suit
  void Print(void)
  {

// The switch statement choses which case to execute based
// on the value of FaceValue
    switch (FaceValue)
    {
      case 2 :
      case 3 :
```

```
        case 4 :
        case 5 :
        case 6 :
        case 7 :
        case 8 :
        case 9 :
        case 10 :
          cout << FaceValue << " of " << SuitName << "s ";
          break;
        case 11 :
          cout << "Jack of " << SuitName << "s ";
          break;
        case 12 :
          cout << "Queen of " << SuitName << "s ";
          break;
        case 13 :
          cout << "King of " << SuitName << "s ";
          break;
        case 14 :
          cout << "Ace of " << SuitName << "s ";
          break;
        default :
          cout << "OOPS, Don't recognize FaceValue!" << endl;
      }

    };

// The PrintAll() method of the card class prints out ALL of the
// card properties FaceValue, Suit, and Drawn
    void PrintAll(void)
    {

// The switch statement choses which case to execute based
// on the value of FaceValue
      switch (FaceValue)
      {
        case 2 :
        case 3 :
        case 4 :
        case 5 :
        case 6 :
        case 7 :
        case 8 :
        case 9 :
        case 10 :
          if ( Drawn )
          {
            cout << FaceValue << " of " << SuitName << "s " << "is drawn" << endl;
          }
          else
```

```
    {
     cout << FaceValue << " of "<< SuitName << "s " << "is not drawn" << endl;
    }
    break;
  case 11 :
    if ( Drawn )
    {
     cout << "Jack of " << SuitName << "s " << "is drawn" << endl;
    }
    else
    {
     cout << "Jack of " << SuitName << "s " << "is not drawn" << endl;
    }
    break;
  case 12 :
    if ( Drawn )
    {
     cout << "Queen of " << SuitName << "s " << "is drawn" << endl;
    }
    else
    {
     cout << "Queen of " << SuitName << "s " << "is not drawn" << endl;
    }
    break;
  case 13 :
    if ( Drawn )
    {
     cout << "King of " << SuitName << "s " << "is drawn" << endl;
    }
    else
    {
     cout << "King of " << SuitName << "s " << "is not drawn" << endl;
    }
    break;
  case 14 :
    if ( Drawn )
    {
     cout << "Ace of " << SuitName << "s " << "is drawn" << endl;
    }
    else
    {
     cout << "Ace of " << SuitName << "s " << "is not drawn" << endl;
    }
    break;
  default :
    cout << "OOPS, Don't recognize FaceValue!" << endl;
 }

};
```

```
};

#endif
```

deck.h

```
#ifndef DECK_H
#define DECK_H

// Deck class

// Deck objects have two properties, both private:
//   Card Cards[52] = an array of 52 Card objects
//   int CardsLeft  = the number of Card objects not already drawn

// Deck objects have these methods:
//  Deck()             - default constructor, constructs a deck upon instantiation
//  Card GetCard(int i) - returns Cards[i]
//  int GetCardsLeft()  - returns the number of cards left in the deck
//  Card DrawCard()     - draws a card from the deck and returns it
//  void Shuffle()      - Shuffles the deck and returns it to original condition

#include <iostream>
#include <string>
#include <cstdlib>
#include <cassert>
using namespace std;

#include "card.h"

class Deck
{

  private:
    Card Cards[52];
    int CardsLeft;

  public:
// Constructor for Deck class
// The constructor fills the Deck with cards and initializes
// the CardsLeft property to 52
    Deck()
    {
       string suit[4] = { "Diamond","Heart","Club","Spade" };

      int i,j;
      int counter=0;

      CardsLeft = 52;
```

```
    for ( j=0; j<4; j++)
    {
     for ( i=0; i<13; i++ )
     {
       Cards[counter].SetFaceValue(i+1);
       Cards[counter].SetSuit(j+1);
       Cards[counter].Reset();
       counter++;
     }
    }
  };

// GetCard returns the indicated card from the deck
  Card GetCard(int i)
  {
   return Cards[i];
  };

// GetCardsLeft returns the CardsLeft property
  int GetCardsLeft()
  {
   return CardsLeft;
  };

// DrawCard selects a Card at random from the remaining Cards
// in the Deck and sets the Drawn property of that Card to true
// and decrements the CardsLeft property.
  Card DrawCard(void)
  {

//   If the number of Cards left is zero, abort.
//   You should always check CardsLeft before using DrawCard
    assert( CardsLeft );

    int i;

//   If there is only one Card left, find it and return it
    if ( CardsLeft == 1 )
     {
      for ( i=0; i<52; i++ )
      {
       if ( !Cards[i].IsDrawn() )
       {
         Cards[i].Draw();
         CardsLeft--;
         return Cards[i];
       }
      }
     }
```

```
//    Take a random Card from the Deck
//    Get random Cards until you find one not drawn
      i = rand()%52;
      while ( Cards[i].IsDrawn() )
      {
        i = rand()%52;
      }

      Cards[i].Draw();
      CardsLeft--;
      return Cards[i];
//
// Alternate method of drawing cards, more efficient
// because never find card drawn, but must rebuild
// deck when shuffle is called. This is really more
// efficient when you are drawing multiple hands from
// a deck and the deck is getting low on cards.
//
//      Card TempCard;
//      i = rand()%CardsLeft;
//      TempCard = Cards[i];
//      TempCard.Draw();
//      for ( int j=i+1; j<CardsLeft; j++ )
//      {
//        Cards[j-1] = Cards[j];
//      }
//      CardsLeft--;
//      return TempCard;
    };


// The Shuffle method resets the Drawn property of all Cards
// in the Deck and resets the CardsLeft property to 52
    void Shuffle(void)
    {
      for ( int i=0; i<52; i++ )
      {
        Cards[i].Reset();
      }
      CardsLeft = 52;
      return;
// Alternate Shuffle() procedure, used with alternate
// Draw() procedure.
//      string suit[4] = { "Diamond","Heart","Club","Spade" };
//      int i,j;
//      int counter=0;
//      CardsLeft = 52;
//      for ( j=0; j<4; j++)
//      {
```

```
//       for ( i=0; i<13; i++ )
//       {
//         Cards[counter].SetFaceValue(i+1);
//         Cards[counter].SetSuit(j+1);
//         Cards[counter].Reset();
//         counter++;
//       }
//     }
   };

};

#endif
```

hand5.h

```
#ifndef HAND5_H
#define HAND5_H

// Class Hand5

// A Hand object has one private property:
//  Card Cards[5] = 5 cards objects

// A Hand object has these methods:
//  bool SetCard(int i,Card card) - sets the Card[i] to card and returns true or false for success
//  Card GetCard(int i)          - returns Card[i]
//  bool Draw(Deck& deck)        - draws five cards from deck and returns true or false for
success
//  void Sort()                  - sorts the cards by FaceValue and then by Suit
//  bool IsSame(Hand hand)       - returns true if hands are the same, false otherwise
//  void Print()                 - prints out the cards of the hand
//  bool IsStraight()            - returns true if hand is a straight, false otherwise
//  bool IsFlush()               - returns true if hand is a flush, false otherwise
//  bool IsStraightFlush()       - returns true if hand is a sraight-flush, false otherwise
//  bool IsRoyal()               - returns true if hand is royal, false otherwise
//  bool IsFullHouse()           - returns true if hand is a full-house, false otherwise
//  bool IsPair()                - returns true if hand is a pair, false otherwise
//  bool IsTwoPair()             - returns true if hand is two pair, false otherwise
//  bool IsThree()               - returns true if hand is three of a kind, false otherwise
//  bool IsFour()                - returns true if hand is four of a kind, false otherwise
//  bool IsValid()               - returns true if hand a valid hand, false otherwise

#include <iostream>
using namespace std;

#include "card.h"
```

```
#include "deck.h"
class Hand5
{

  private:
    Card Cards[5];
    bool Sorted;

  public:
    bool SetCard(int i, Card card)
    {
      if ( i < 1 || i > 5 ) return false;
      Cards[i-1] = card;
      Sorted = false;
      return true;
    };

    Card GetCard(int i)
    {
      if ( i < 1 ) i = 1;
      if ( i > 5 ) i = 5;
      return Cards[i-1];
    };

    bool Draw( Deck& deck )
    {
      if ( deck.GetCardsLeft() < 5 ) return false;
      int i;
      for ( i=0; i<5; i++ )
      {
        Cards[i] = deck.DrawCard();
      }
      Sorted=false;
      return true;
    };

    void Sort()
    {
      if ( Sorted ) return;
      Card TempCard;
      for ( int i=0; i<4; i++ )
      {
        for ( int j=i+1; j<5; j++ )
        {
          if ( Cards[i].GetFaceValue() > Cards[j].GetFaceValue() )
          {
            TempCard = Cards[i];
            Cards[i] = Cards[j];
            Cards[j] = TempCard;
          }
```

```
        else if (Cards[i].GetFaceValue() == Cards[j].GetFaceValue() )
        {
          if ( Cards[i].GetSuit() > Cards[j].GetSuit() )
          {
            TempCard = Cards[i];
            Cards[i] = Cards[j];
            Cards[j] = TempCard;
          }
        }
      }
    }
    Sorted=true;
  };

  bool IsSame(Hand5 hand)
  {
    hand.Sort();
    Sort();
    for ( int i=0; i<5; i++ )
    {
      if ( !Cards[i].IsSame(hand.Cards[i]) ) return false;
    }
    return true;
  };

  void Print()
  {
    for ( int i=0; i<5; i++ )
    {
      Cards[i].Print();
      cout << endl;
    }
  };

// Ace must be at high end
  bool IsHighStraight()
  {
    Sort();
    for ( int i=1; i<5; i++ )
    {
      if ( Cards[i].GetFaceValue() != (Cards[i-1].GetFaceValue() + 1) ) return false;
    }
    return true;
  };

// Ace must be at low end
  bool IsLowStraight()
  {
    Sort();
    if ( Cards[4].GetFaceValue() != 14 ) return false;
```

```
    if ( Cards[0].GetFaceValue() != 2 ) return false;
    for ( int i=1; i<4; i++ )
    {
      if ( Cards[i].GetFaceValue() != (Cards[i-1].GetFaceValue() + 1) ) return false;
    }
    return true;
  };

  bool IsStraight()
  {
    return ( IsHighStraight() || IsLowStraight() );
  };

  bool IsFlush()
  {
    for ( int i=0; i<4; i++ )
    {
      if ( Cards[i].GetSuit() != Cards[i+1].GetSuit() ) return false;
    }
    return true;
  };

  bool IsStraightFlush()
  {
    return ( IsStraight() && IsFlush() );
  };

  bool IsRoyal()
  {
    Sort();
    if ( IsStraight() && ( Cards[0].GetFaceValue() == 10 ) ) return true;
    return false;
  };

  bool IsFullHouse()
  {
    Sort();
// Two of a kind then three of a kind
    if ( Cards[0].GetFaceValue() == Cards[1].GetFaceValue() &&
         Cards[2].GetFaceValue() == Cards[3].GetFaceValue() &&
         Cards[3].GetFaceValue() == Cards[4].GetFaceValue() ) return true;
// Three of a kind then two of a kind
    if ( Cards[0].GetFaceValue() == Cards[1].GetFaceValue() &&
         Cards[1].GetFaceValue() == Cards[2].GetFaceValue() &&
         Cards[3].GetFaceValue() == Cards[4].GetFaceValue() ) return true;
// Must be false by now
    return false;
  };

  bool IsPair()
```

```
    {
      int nmatch = 0;
      for ( int i=0; i<4; i++ )
      {
        for ( int j=i+1; j<5; j++ )
        {
          if ( Cards[i].GetFaceValue() == Cards[j].GetFaceValue() ) nmatch++;
        }
      }
      if ( nmatch == 1 ) return true;
      return false;
    };

    bool IsTwoPair()
    {
      Sort();
// Assumes five of a kind not possible
// First two same and next two same, but differ, last differs
      if ( Cards[0].GetFaceValue() == Cards[1].GetFaceValue() &&
           Cards[2].GetFaceValue() == Cards[3].GetFaceValue() &&
           Cards[1].GetFaceValue() != Cards[2].GetFaceValue() &&
           Cards[3].GetFaceValue() != Cards[4].GetFaceValue() ) return true;
// First two same and last two same, but differ, middle differs
      if ( Cards[0].GetFaceValue() == Cards[1].GetFaceValue() &&
           Cards[3].GetFaceValue() == Cards[4].GetFaceValue() &&
           Cards[2].GetFaceValue() != Cards[0].GetFaceValue() &&
           Cards[2].GetFaceValue() != Cards[3].GetFaceValue() ) return true;
// First one differ, next two same and last two same, but differ
      if ( Cards[1].GetFaceValue() == Cards[2].GetFaceValue() &&
           Cards[3].GetFaceValue() == Cards[4].GetFaceValue() &&
           Cards[0].GetFaceValue() != Cards[1].GetFaceValue() &&
           Cards[2].GetFaceValue() != Cards[3].GetFaceValue() ) return true;
// Must be false by now
      return false;
    };

    bool IsThree()
    {
      Sort();
// First three are same and other differ
      if ( Cards[0].GetFaceValue() == Cards[1].GetFaceValue() &&
           Cards[1].GetFaceValue() == Cards[2].GetFaceValue() &&
           Cards[2].GetFaceValue() != Cards[3].GetFaceValue() &&
           Cards[3].GetFaceValue() != Cards[4].GetFaceValue() ) return true;
// Middle three are same and other differ
      if ( Cards[0].GetFaceValue() != Cards[1].GetFaceValue() &&
           Cards[1].GetFaceValue() == Cards[2].GetFaceValue() &&
           Cards[2].GetFaceValue() == Cards[3].GetFaceValue() &&
           Cards[3].GetFaceValue() != Cards[4].GetFaceValue() ) return true;
// Last three are same and other differ
```

```
    if ( Cards[0].GetFaceValue() != Cards[1].GetFaceValue() &&
         Cards[1].GetFaceValue() != Cards[2].GetFaceValue() &&
         Cards[2].GetFaceValue() == Cards[3].GetFaceValue() &&
         Cards[3].GetFaceValue() == Cards[4].GetFaceValue() ) return true;
// Must be false by now
    return false;
  };

  bool IsFour()
  {
    Sort();
// Assumes five of a kind not possible
// First four are same
    if ( Cards[0].GetFaceValue() == Cards[1].GetFaceValue() &&
         Cards[1].GetFaceValue() == Cards[2].GetFaceValue() &&
         Cards[2].GetFaceValue() == Cards[3].GetFaceValue() ) return true;
// Last four are same
    if ( Cards[1].GetFaceValue() == Cards[2].GetFaceValue() &&
         Cards[2].GetFaceValue() == Cards[3].GetFaceValue() &&
         Cards[3].GetFaceValue() == Cards[4].GetFaceValue() ) return true;
// Must be false by now
    return false;
  };

  bool IsValid()
  {
// Check for duplicate cards
    for ( int i=0; i<4; i++ )
    {
      for ( int j=i+1; j<5; j++ )
      {
        if ( Cards[i].IsSame(Cards[j]) ) return false;
      }
    }
// Check for five of a kind
    if ( Cards[0].GetFaceValue() == Cards[1].GetFaceValue() &&
         Cards[1].GetFaceValue() == Cards[2].GetFaceValue() &&
         Cards[2].GetFaceValue() == Cards[3].GetFaceValue() &&
         Cards[3].GetFaceValue() == Cards[4].GetFaceValue() ) return false;
// Must be true by now
    return true;
  };

};

#endif
```

hand7.h

```
#ifndef HAND7_H
#define HAND7_H

// Hand class (7 card stud)

/*
Hands have the following private properties

  Cards Cards[7]         = 7 cards in a hand
  bool Sorted          = indicates whether the hand is sorted with true or false

Hands have the following private properties

  bool SetCard(int i, Card card)  = sets the Card[i] to card and returns true if successful
  Card GetCard(int i)            = returns Card[i]
  bool Draw(Deck& deck)          = draws cards from deck and returns true for success
  void Sort()                  = sorts cards by FaceValue and Suit
  bool IsSame(Hand hand)         = compares two hands and returns true if same
  void Print()                 = prints hand
  bool IsValid()                 = returns true if hand is valid hand

  bool IsPair()                = returns true if hand has only a single pair
  bool IsTwoPair()               = returns true if hand has two pairs
  bool IsThree()                 = returns true if hand is three of a kind
  bool IsFour()                = returns true if hand is four of a kind

  bool IsStraight()              = returns true if hand is straight
  bool IsFlush()               = returns true if hand is flush
  bool IsStraightFlush()         = returns true if hand is straight flush
  bool IsRoyal()               = returns true if hand is royal flush
  bool IsFullHouse()             = returns true if hand is full house
  int Match()                  = returns number of matches in hand
*/

#include <iostream>
using namespace std;
#include "hand5.h"

#include "card.h"
#include "deck.h"

class Hand7
{
private:
  Card Cards[7];
  bool Sorted;

public:

// Allows you to set individual cards of a hand
```

```
// if you need to. For example when you are testing
// methods of the Hand7 class, you need to be able
// to build specific hands.
  bool SetCard(int i, Card card)
  {
    if (i<1 || i>7)
    {
      return false;
    }
    Cards[i-1] = card;
    Sorted = false;
    return true;
  };

// This is how you can see what any particular
// card in a hand is.
  Card GetCard(int i)
  {
    if (i<1)
    {
      i=1;
    }
    if (i>7)
    {
      i=7;
    }
    return Cards[i-1];
  };

// draws cards from the deck

  bool Draw(Deck& deck)
  {
    if (deck.GetCardsLeft() < 7)
    {
      return false;
    }
    for (int i=0; i<7; i++)
    {
      Cards[i] = deck.DrawCard();
    }
    Sorted = false;
    return true;
  };

// sorts the hand

  void Sort()
  {
    if (Sorted)
```

```
    {
      return;
    }
    Card TempCard;
    for (int i=0; i<6; i++)
    {
      for (int j=i+1; j<7; j++)
      {
//        if (Cards[i].GetFaceValue() > Cards{j}.GetFaceValue())
        if (Cards[i].GetFaceValue() > Cards[j].GetFaceValue())
        {
          TempCard = Cards[i];
          Cards[i] = Cards[j];
          Cards[j] = TempCard;
        }
        else if (Cards[i].GetFaceValue() == Cards[j].GetFaceValue())
        {
          if (Cards[i].GetSuit() > Cards[j].GetSuit())
          {
            TempCard = Cards[i];
            Cards[i] = Cards[j];
            Cards[j] = TempCard;
          }
        }
      }
    }
    Sorted = true;
  };

// compares two hands

//  bool IsSame(Hand hand)
  bool IsSame(Hand7 hand)
  {
    hand.Sort();
    Sort();
    for (int i=0; i<7; i++)
    {
      if (!Cards[i].IsSame(hand.Cards[i]))
      {
        return false;
      }
    }
    return true;
  };

// Prints the hand

  void Print()
  {
```

```
    for (int i=0; i<7; i++)
    {
      Cards[i].Print();
      cout << endl;
    }
  };

// checks to see if hand is valid

  bool IsValid()
  {
    for (int i=0; i<6; i++)
    {
      for (int j=i+1; j<7; j++);
      {
        if ( Cards[i].IsSame(Cards[j]) )
        {
          return false;
        }
      }
    }
    Sort();
    if (Cards[0].GetFaceValue() == Cards[4].GetFaceValue() ||
      Cards[1].GetFaceValue() == Cards[5].GetFaceValue() ||
      Cards[2].GetFaceValue() == Cards[6].GetFaceValue())
    {
      return false;
    }
    else
    {
      return true;
    }
  };

// Checks for a single pair
  bool IsPair()
  {
    int nsame[3];
    int fv[3];
    int nmatch = Match(nsame,fv);

    if ( nmatch != 1 ) return false;
    if ( nsame[0] != 2 ) return false;

// If you got to here it is a pair.
    return true;
  };

// Checks for two pair
  bool IsTwoPair()
```

```
  {
    int nsame[3];
    int fv[3];
    int nmatch = Match(nsame,fv);

    if ( nmatch != 2 ) return false;
    if ( nsame[0] != 2 || nsame[1] != 2 ) return false;

// If you got to here it is two pair.
    return true;
  };

// Checks for a three of a kind
  bool IsThree()
  {
    int nsame[3];
    int fv[3];
    int nmatch = Match(nsame,fv);

    if ( nmatch != 1 ) return false;
    if ( nsame[0] != 3 ) return false;

// If you got to here it is three of a kind.
    return true;
  };


// Checks for four of a kind
  bool IsFour()
  {
    int nsame[3];
    int fv[3];
    int nmatch = Match(nsame,fv);

    if ( nmatch != 1 ) return false;
    if ( nsame[0] != 4 ) return false;

// If you got to here it is four of a kind.
    return true;
  };


// checks for any straight where ace is at hight end

  bool IsHighStraight()
  {
    Sort();

// Create a new hand with only the different cards
// no duplicate facevalues left in the new hand
```

```
    Hand7 hand;
    int ncards = 1;
    hand.SetCard(ncards,Cards[0]); // set the first cards equal
    for ( int i=1; i<7; i++)
    {
      if ( Cards[i].GetFaceValue() != Cards[i-1].GetFaceValue() )
      {
        ncards++;
        hand.SetCard(ncards,Cards[i]);
      }
    }

    if ( ncards < 5 ) return false; // not enough different cards

    bool flag;
    for ( int j=0; j<(ncards-5+1); j++ )
    {

      flag = true;
      for ( i=(j+1); i<(j+5); i++ )
      {
        if ( hand.GetCard(i+1).GetFaceValue() != (hand.GetCard(i).GetFaceValue() + 1) ) flag =
false;
      }
      if ( flag ) return true;

    }

    return false;

  };

// looks for any straight that starts with an ace

  bool IsLowStraight()
  {
    Sort();
// check for existence of aces and twos
    if ( Cards[0].GetFaceValue() != 2 ) return false;
    if ( Cards[6].GetFaceValue() != 14 ) return false;

// Create a new hand with only the different cards
// no pairs left in the new hand
    Hand7 hand;
    int ncards = 1;
    hand.SetCard(ncards,Cards[0]); // set the first cards equal
    for ( int i=1; i<7; i++)
    {
      if ( Cards[i].GetFaceValue() != Cards[i-1].GetFaceValue() )
      {
```

```
      ncards++;
      hand.SetCard(ncards,Cards[i]);
     }
    }

    if ( ncards < 5 ) return false; // not enough different cards

// Need only check that the lowest four cards are consecutive
    for ( i=1; i<4; i++ )
    {
     if ( hand.GetCard(i+1).GetFaceValue() != (hand.GetCard(i).GetFaceValue() + 1) ) return
false;
    }

// If you got to here, there must be a straight
    return true;

  };

// returns if hand is a straight

  bool IsStraight()
  {
    if ( IsHighStraight() ) return true;
    if ( IsLowStraight() ) return true;
    return false;
  };

// Returns true if hand is flush and
// all the cards in the flush.
  bool IsFlush(Hand7& hand)
  {
    int nhearts=0;
    int ndiamonds=0;
    int nclubs=0;
    int nspades=0;

    Hand7 hearts,diamonds,clubs,spades;

    for ( int i=0; i<7; i++ )
    {
     if ( Cards[i].GetSuitName() == "Club" )
     {
      nclubs++;
      clubs.SetCard(nclubs,Cards[i]);
     }
     if ( Cards[i].GetSuitName() == "Diamond" )
     {
      ndiamonds++;
      diamonds.SetCard(ndiamonds,Cards[i]);
```

```
      }
      if ( Cards[i].GetSuitName() == "Heart" )
      {
        nhearts++;
        hearts.SetCard(nhearts,Cards[i]);
      }
      if ( Cards[i].GetSuitName() == "Spade" )
      {
        nspades++;
        spades.SetCard(nspades,Cards[i]);
      }
    }

// Check for valid flush hands and return them
    if ( nclubs >= 5 )
    {
      hand = clubs;
      return true;
    }
    if ( ndiamonds >= 5 )
    {
      hand = diamonds;
      return true;
    }
    if ( nhearts >= 5 )
    {
      hand = hearts;
      return true;
    }
    if ( nspades >= 5 )
    {
      hand = spades;
      return true;
    }

// If you got to here there were no flushes
    return false;

  };

// returns if hand is a straight flush
  bool IsStraightFlush()
  {
    Hand7 hand;
    if ( IsFlush(hand) )
    {
      return ( hand.IsStraight() );
    }
    return false;
  };
```

```
// returns if hand is a royal flush
  bool IsRoyal()
  {
    if (IsStraightFlush())
    {
      if (Cards[0].GetFaceValue() == 10 ||
          Cards[1].GetFaceValue() == 10 ||
          Cards[2].GetFaceValue() == 10)
      {
        return true;
      }
      else
      {
        return false;
      }
    }
    else
    {
      return false;
    }
  };

// checks for full house

  bool IsFullHouse()
  {
    int nsame[3] = {0,0,0};
    int fv[3] = {0,0,0};
    int nmatch;

    nmatch = Match(nsame,fv);

/*
    if ( nmatch !=2 ) return false;
    if ( (nsame[0]+nsame[1]) != 5 ) return false;

//  If you got to here is a full house
    return true;

*/
    if ( nmatch < 2 )
    {
      return false;
    }
    else
    {
      int itwo=0;
      int ithree=0;
      for ( int i=0; i<3; i++ )
```

```
    {
      if ( nsame[i] == 2 )
      {
        itwo = i+1;
      }
      if ( nsame[i] == 3 )
      {
        ithree = i+1;
      }
    }
    if ( itwo != 0 && ithree != 0 )
    {
      return true;
    }
    return false;
  }

};

// Match returns the number of sets of matching cards (face value )
// Match needs a pointer to an int that has been allocated enough
// space to hold all possible matches. For example, for a 5 card
// hand nsame must be two int's long, for a 7 card hand it must be
// 3 int's long. fv holds the face values of the matches
  int Match(int* nsame, int* fv)
  {
    Sort();
    int nmatch = 0;
    int i = 0;

    while ( i < 7 )
    {
      if ( Cards[i].GetFaceValue() == Cards[i+1].GetFaceValue() )
      {
        nmatch++;
        fv[nmatch-1] = Cards[i].GetFaceValue();
        nsame[nmatch-1] = 1;
        while ( Cards[i].GetFaceValue() == Cards[i+1].GetFaceValue() )
        {
          nsame[nmatch-1]++;
          i++;
        }
      }
      i++;
    }

    return nmatch;
  };

};
```

```
#endif
```

deck.cxx

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <iomanip>
using namespace std;
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#include "deck.h"
#include "hand5.h"
#include "hand7.h"

static const long DRAWS  = 100;
long int TRIALS;
static const int  HANDS = 1;
static const int BINS = 1000;

int FindBin(const double[], double, int);
bool DealHand5s(Deck&, Hand5[], int);
bool DealHand7s(Deck&, Hand7[], int);

int main()
{
  int nNum;

Start:

  cout << endl;

  cout << "1. Five Card Stud" << endl;
  cout << "2. Seven Card Stud" << endl;

  cout << endl;

  cout << "Which game would you like to simulate?" << endl;
  cin >> nNum;

  cout << endl;
  cout << "You want to simulate ";

  int iresult;
```

```
  long StarCount=0;
  char star[4];
  star[0]='|';
  star[1]='/';
  star[2]='-';
  star[3]='\\';

  Deck deck;

  long NumberFound[10];
//  0 - Nothing
//  1 - Pair
//  2 - Two Pair
//  3 - Three of a Kind
//  4 - Straight
//  5 - Flush
//  6 - Full House
//  7 - Four of a Kind
//  8 - Straight Flush
//  9 - Royal Flush (Straight)
  string HandType[10] = { "Nothing","Pair","Two Pair","Three of a Kind","Straight",
    "Flush","Full House","Four of a Kind","Straight Flush","Royal Flush" };

  double Probability[10]={0,0,0,0,0,0,0,0,0,0};
  double Probability2[10]={0,0,0,0,0,0,0,0,0,0};
  double Prob;

  ofstream out;
  out.open("deck.out");;

// Set up probaility bins for each hand type
// Set up the estimate bins for comparing to normal distribution
  double EstimateBounds[BINS+1];
  long NumberEstimates[BINS][10];
  EstimateBounds[0] = 0.0;
  int iBin;
  int iType;

  double BinWidth = 1.0/(double)BINS;
  for ( iBin=1; iBin<BINS; iBin++)
  {
    EstimateBounds[iBin] = EstimateBounds[iBin-1] + BinWidth;
  }
  EstimateBounds[BINS] = 1.0;

  for ( iType=0; iType<10; iType++ )
  {
    for ( iBin=0; iBin<BINS; iBin++ )
    {
      NumberEstimates[iBin][iType] = 0;
```

```
    }
  }

  long iTrial;
  long iDraw;

  time_t time0,time1,time2;

// Switch for the type of hand you are drawing
  switch( nNum )
  {
  case 1:
    {
      cout << "Five Card Stud" << endl;
      cout << endl;
      cout << "How Many Trials?" << endl;
      cin >> TRIALS;
      cout << endl;
      cout << "You chose to simulate Five Card Stud with " << HANDS << " other players
through " << TRIALS << " Trials." << endl;
      cout << endl;

  time1 = time( &time0 );

  Hand5 Hand5Drawn;
  Hand5 TestHand5;
  Hand5 Hands5Drawn[HANDS];

// Print work being done

  cout << endl;
  cout << "Working on " << TRIALS << " Trials of " << DRAWS << " draws each, " <<
HANDS << " hands at a time." << endl;
  out << endl;
  out << "Working on " << TRIALS << " Trials of " << DRAWS << " draws each, " << HANDS
<< " hands at a time." << endl;

// Begin loop to draw hands and count occurances

  for ( iTrial=0; iTrial<TRIALS; iTrial++)
  {
    for ( int j=0; j<10; j++ )
    {
      NumberFound[j] = 0;
    }

    for ( iDraw=0; iDraw<DRAWS; iDraw++ )
    {

      if ( !DealHand5s(deck,Hands5Drawn,HANDS) )
```

```
    {
     cout << " Hand failed to draw on draw number " << iDraw+1 << " of trial number " <<
iTrial+1 << "."<< endl;
     return 1;
    }
    else
    {
     Hand5Drawn = Hands5Drawn[0];

//   Determine what kind of hand was drawn.

     if ( Hand5Drawn.IsPair() )
     {
      NumberFound[1]++; // Pair
     }
     else
     {
      if ( Hand5Drawn.IsTwoPair() )
      {
       NumberFound[2]++; // Two Pair
      }
      else
      {
       if ( Hand5Drawn.IsThree() )
       {
        NumberFound[3]++; // Three of a Kind
       }
       else
       {
        if ( Hand5Drawn.IsStraight() )
        {
         if ( Hand5Drawn.IsFlush() )
         {
          if ( Hand5Drawn.IsRoyal() )
          {
           NumberFound[9]++; //Royal Flush
          }
          else
          {
           NumberFound[8]++; // Straight Flush
          }
         }
         else
         {
          NumberFound[4]++; // Straight
         }
        }
        else
        {
         if ( Hand5Drawn.IsFlush() )
```

```
            {
              NumberFound[5]++; // Flush
            }
            else
            {
              if ( Hand5Drawn.IsFullHouse() )
              {
                NumberFound[6]++; // Full House
              }
              else
              {
                if ( Hand5Drawn.IsFour() )
                {
                  NumberFound[7]++; // Four of a Kind
                }
                else
                {
                  NumberFound[0]++; // Nothing
                }
              }
            }
          }
        }
      }
    }

    deck.Shuffle();

// This is an "I'm still working" display.
    if ( iDraw%1000 == 0 )
    {
      iresult = putchar(13);
      iresult = putchar(star[StarCount%4]);
      StarCount++;
    }
  }

  for ( iType=0; iType<10; iType++ )
  {

    Prob = NumberFound[iType]/(double)DRAWS;

    iBin = FindBin(EstimateBounds,Prob,BINS);
    NumberEstimates[iBin][iType]++;

    Probability[iType] += Prob;
    Probability2[iType] += Prob*Prob;

  }
```

```
  }

  cout << endl;

// Print out estimate distributions to file
  out << "Number and probability distributions:" << endl;
  for ( iType=0; iType<10; iType++ )
  {
    for ( iBin=0; iBin<BINS; iBin++ )
    {
      out << setw(6) << EstimateBounds[iBin];
      for ( iType=0; iType<10; iType++ )
      {
        out << "   " << setw(6) << NumberEstimates[iBin][iType]
            << "   " << setw(6) << NumberEstimates[iBin][iType]/(double)DRAWS;
      }
      out << endl;
    }
    out << setw(6) << EstimateBounds[BINS] << endl;
  }

// Calculate final probabilities and statistics

  double Sigma[10];
  double SumProbability = 0.0;
  for ( iType=0; iType<10; iType++ )
  {

    Probability[iType]  = Probability[iType]/(double)TRIALS;
    Probability2[iType] = Probability2[iType]/(double)TRIALS;
    Sigma[iType] = (Probability2[iType] -
Probability[iType]*Probability[iType])/(double)(TRIALS-1);
    Sigma[iType] = sqrt( Sigma[iType] );

    if ( Sigma[iType] != 0.0 ) Sigma[iType] = Sigma[iType]/Probability[iType]*100.0;
    SumProbability += Probability[iType];
  }

// Print out final results

  cout << "                              Mean" << endl;
  cout << "          Hand   Probability       Variance" << endl;
   out << "                              Mean" << endl;
   out << "          Hand   Probability       Variance" << endl;

  for ( iType=0; iType<10; iType++ )
  {
    cout << setw(15) << HandType[iType] << setw(14) << Probability[iType] << setw(14) <<
Sigma[iType] << setw(1) << "%" << endl;
```

```
   out << setw(15) << HandType[iType] << setw(14) << Probability[iType] << setw(14) <<
Sigma[iType] << setw(1) << "%" << endl;
  }

  cout << setw(15) << "Total" << setw(14) << SumProbability << endl;
  out << setw(15) << "Total" << setw(14) << SumProbability << endl;

  time2 = time( &time0 );

  cout << endl;
  cout << "Time elapsed: " << time2-time1 << " seconds." << endl;
  out << endl;
  out << "Time elapsed: " << time2-time1 << " seconds." << endl;

     cout << endl;

          int nNum1;
          cout << "1.  Run Again." << endl;
          cout << "2.  Quit." << endl;
          cin >> nNum1;
          swich (nNum1)
          {
          case 1:
                  {
                    goto Start;
                    cout << endl;
                    break;
                  }
          case 2:
                  {
                          cout << endl;
                          break;
                  }
          default:
                  {
                          cout << endl;
                          break;
                  }
          }
          cout << endl;
     break;

    }

  case 2:
    {
     cout << "Seven Card Stud" << endl;
     cout << endl;
     cout << "How Many Trials?" << endl;
     cin >> TRIALS;
```

```
    cout << endl;
    cout << "You chose to simulate Seven Card Stud with " << HANDS << " other players
through " << TRIALS << " Trials." << endl;
    cout << endl;
    cout << endl;

  time1 = time( &time0 );

  Hand7 Hand7Drawn;
  Hand7 TestHand7;
  Hand7 Hands7Drawn[HANDS];

// Print work being done

  cout << endl;
  cout << "Working on " << TRIALS << " Trials of " << DRAWS << " draws each, " <<
HANDS << " hands at a time." << endl;
  out << endl;
  out << "Working on " << TRIALS << " Trials of " << DRAWS << " draws each, " << HANDS
<< " hands at a time." << endl;

// Begin loop to draw hands and count occurances

  for ( iTrial=0; iTrial<TRIALS; iTrial++)
  {
   for ( int j=0; j<10; j++ )
   {
    NumberFound[j] = 0;
   }

   for ( iDraw=0; iDraw<DRAWS; iDraw++ )
   {

    if ( !DealHand7s(deck,Hands7Drawn,HANDS) )
    {
     cout << " Hand failed to draw on draw number " << iDraw+1 << " of trial number " <<
iTrial+1 << "."<< endl;
     return 1;
    }
    else
    {
     Hand7Drawn = Hands7Drawn[0];

//   Determine what kind of hand was drawn.

     if ( Hand7Drawn.IsPair() )
     {
      NumberFound[1]++; // Pair
     }
     else
```

```
{
 if ( Hand7Drawn.IsTwoPair() )
 {
  NumberFound[2]++; // Two Pair
 }
 else
 {
  if ( Hand7Drawn.IsThree() )
  {
   NumberFound[3]++; // Three of a Kind
  }
  else
  {
   if ( Hand7Drawn.IsStraight() )
   {
    if ( Hand7Drawn.IsFlush(TestHand7) )
    {
     if ( Hand7Drawn.IsRoyal() )
     {
      NumberFound[9]++; //Royal Flush
     }
     else
     {
      NumberFound[8]++; // Straight Flush
     }
    }
    else
    {
     NumberFound[4]++; // Straight
    }
   }
   else
   {
    if ( Hand7Drawn.IsFlush(TestHand7) )
    {
     NumberFound[5]++; // Flush
    }
    else
    {
     if ( Hand7Drawn.IsFullHouse() )
     {
      NumberFound[6]++; // Full House
     }
     else
     {
      if ( Hand7Drawn.IsFour() )
      {
       NumberFound[7]++; // Four of a Kind
      }
      else
```

```
                {
                  NumberFound[0]++; // Nothing
                }
              }
            }
          }
        }
      }
    }

    deck.Shuffle();

// This is an "I'm still working" display.
    if ( iDraw%1000 == 0 )
    {
      iresult = putchar(13);
      iresult = putchar(star[StarCount%4]);
      StarCount++;
    }
  }

  for ( iType=0; iType<10; iType++ )
  {

    Prob = NumberFound[iType]/(double)DRAWS;

    iBin = FindBin(EstimateBounds,Prob,BINS);
    NumberEstimates[iBin][iType]++;

    Probability[iType] += Prob;
    Probability2[iType] += Prob*Prob;

  }
 }

 cout << endl;

// Print out estimate distributions to file
 out << "Number and probability distributions:" << endl;
 for ( iType=0; iType<10; iType++ )
 {
  for ( iBin=0; iBin<BINS; iBin++ )
  {
    out << setw(6) << EstimateBounds[iBin];
    for ( iType=0; iType<10; iType++ )
    {
      out << "   " << setw(6) << NumberEstimates[iBin][iType]
          << "   " << setw(6) << NumberEstimates[iBin][iType]/(double)DRAWS;
    }
```

```
      out << endl;
    }
    out << setw(6) << EstimateBounds[BINS] << endl;
  }

// Calculate final probabilities and statistics

  double Sigma[10];
  double SumProbability = 0.0;
  for ( iType=0; iType<10; iType++ )
  {

    Probability[iType]  = Probability[iType]/(double)TRIALS;
    Probability2[iType] = Probability2[iType]/(double)TRIALS;
    Sigma[iType] = (Probability2[iType] -
Probability[iType]*Probability[iType])/(double)(TRIALS-1);
    Sigma[iType] = sqrt( Sigma[iType] );

    if ( Sigma[iType] != 0.0 ) Sigma[iType] = Sigma[iType]/Probability[iType]*100.0;
    SumProbability += Probability[iType];
  }

// Print out final results

  cout << "                                Mean" << endl;
  cout << "          Hand   Probability       Variance" << endl;
   out << "                                Mean" << endl;
   out << "          Hand   Probability       Variance" << endl;

  for ( iType=0; iType<10; iType++ )
  {
    cout << setw(15) << HandType[iType] << setw(14) << Probability[iType] << setw(14) <<
Sigma[iType] << setw(1) << "%" << endl;
    out << setw(15) << HandType[iType] << setw(14) << Probability[iType] << setw(14) <<
Sigma[iType] << setw(1) << "%" << endl;
  }

  cout << setw(15) << "Total" << setw(14) << SumProbability << endl;
  out << setw(15) << "Total" << setw(14) << SumProbability << endl;

  time2 = time( &time0 );

  cout << endl;
  cout << "Time elapsed: " << time2-time1 << " seconds." << endl;
  out << endl;
  out << "Time elapsed: " << time2-time1 << " seconds." << endl;

     cout << endl;
     break;
```

```
    }

  default:
    {
      cout << "???" << endl;
      cout << endl;
      cout << "I don't know what game you want to simulate." << endl;
      goto Start;
      cout << endl;
      cout << endl; break;
    }
  }

  return 0;
}


int FindBin(const double Bounds[], double Value, int NBins)
{
  int iBin;
  for ( iBin=1; iBin<NBins+1; iBin++ )
  {
    if ( Bounds[iBin] > Value ) break;
  }
  if ( iBin > NBins ) iBin = NBins;
  return iBin;
}

bool DealHand5s(Deck &deck, Hand5 hands[], int nhands)
{
  int iHand,iCard;
  Card card;
  for ( iCard=1; iCard<6; iCard++ )
  {
    for ( iHand=0; iHand<nhands; iHand++ )
    {
      if ( deck.GetCardsLeft() == 0 ) return false;
      card = deck.DrawCard();
      hands[iHand].SetCard(iCard,card);
    }
  }
  return true;
}

bool DealHand7s(Deck &deck, Hand7 hands[], int nhands)
{
  int iHand,iCard;
  Card card;
  for ( iCard=1; iCard<8; iCard++ )
  {
```

```
   for ( iHand=0; iHand<nhands; iHand++ )
   {
     if ( deck.GetCardsLeft() == 0 ) return false;
     card = deck.DrawCard();
     hands[iHand].SetCard(iCard,card);
   }
  }
  return true;
}
```

This is the final code that we used to execute our program.