# Data Structures in C++

# Chapter 2

Tim Budd

Oregon State University
Corvallis, Oregon
USA

# Classes and Object-Oriented Programming

Purpose of this chapter

- Introduce classes
- Introduce idea of object-oriented programming
- Programming using software components
- Characterising software by behavior

We will introduce these by means of an extended example program.

# The Card game War


During each round both players select one of their three cards, and places it face up. If ranks are the same, then both players retain their cards (setting them aside). Otherwise, player with largest ranking card keeps both cards. Each player draws one card from the deck to replace the card just played. The game ends when the deck is exhausted, and the player with the most cards wins.

# Nouns and Verbs

The nouns in the problem description identify the components of the game, the verbs identify what they do.

Components:

- Card – know its rank and suit
- Deck – shuffle, draw, know when empty
- Player – draw card from hand, keep own score

Will make software component for each.

# The Class Card

```
enum suits {diamond, club, heart, spade};

class Card {
public:
        // constructors
    Card    ( );
    Card    (suits, int);

        // data fields
    int     rank;
    suits   suit;
};
```

# Constructors

A special type of function, used to combine *creation* and *initialization*, thereby ensuring that every object is properly initialized. Invoked automatically when object is declared.

```
Card::Card    ( )
   // initialize a new Card
   // default value is the ace of spades
{
   rank = 1;
   suit = spades;
}

Card::Card    (suits sv, int rv)
   // initialize a new Card using the argument values
{
   rank = rv;
   suit = sv;
}
```

# Overloaded Names

A function with more than one name is said to be *overloaded*.

Note how constructor is overloaded.

Each version of an overloaded function must have unique argument types.

Other overloaded function we have seen – operator <<.

# Test our Abstraction

```
void main() {
    Card cardOne;
    Card cardTwo(diamond, 7);

    cout ≪ "Card one\n";
    cout ≪ cardOne.rank ≪ "\n";
    cout ≪ "Card two\n";
    cout ≪ cardTwo.rank ≪ "\n";
}
```

Not very nice, aces and face cards are numbers, not text, and can't print suits. Should be able to do better.

# Output operator for Card

```
ostream & operator ≪ (ostream & out, Card & aCard)
   // output a textual representation of a Card
{
   switch (aCard.rank) {    // first output rank
      case 1:
         out ≪ "Ace";
         break;
      case 11:
         out ≪ "Jack";
         break;
      case 12:
         out ≪ "Queen";
         break;
      case 13:
         out ≪ "King";
         break;
      default:    // output number
         out ≪ aCard.rank;
         break;
      }
```

```
    switch (aCard.suit) {    // then output suit
        case diamond:
            out << " of Diamonds";
            break;
        case spade:
            out << " of Spades";
            break;
        case heart:
            out << " of Hearts";
            break;
        case club:
            out << " of Clubs";
            break;
        }
    return out;
}
```

We have overloaded the >> operator by providing a new meaning for printing when using Cards.

# New Test Program

```
void main() {
   Card cardOne;
   Card cardTwo(diamond, 7);

   cout << "Card One:" << cardOne << "\n";
   cout << "Card Two:" << cardTwo << "\n";
}
```

Much nicer output.

# Pass-by-Reference

Note use of ampersand in declaration – signifies pass-by-reference.

Used when passing large structures (such as streams) that are modifed in the function.

Alternative, if nothing specified, is pass-by-value, which creates a *copy* of the argument.

Use pass-by-reference when you don't want to make a copy of the argument. (unless argument is integer, float, pointer, or something else trivial to copy).

# The Class Deck

- A deck must maintain a collection of cards
- The deck must be able to shuffle the cards it holds
- The deck must be able to tell the user whether or not it is empty
- The user of the deck must be able to draw a card (assuming the deck is nonempty)

```
class Deck {
public:
        // constructor
    Deck();

        // operations on a deck
    void    shuffle    ();
    bool    isEmpty    ();
    Card    draw       ();

protected:
    Card    cards[52];
    int     topCard;
};
```

# Protected and Public

Protected are things that can be accessed only by functions associated with the class.

Public features can be accessed outside of the class.

We have "hidden" the direct access to the data fields – this is a good idea.

# Arrays and Initialization

```
class Deck {
public:
      // constructor
   Deck();

      // operations on a deck
   void    shuffle    ();
   bool    isEmpty    ();
   Card    draw       ();

protected:
   Card    cards[52];
   int     topCard;
};
```

Elements of the array are initialized using no-argument constructor (termed the *default* constructor).

Will be given other values by the constructor for `Deck`.

# Deck Constructor

```
Deck::Deck ( )
   // initialize a deck by creating all 52 cards
{
   topCard = 0;
   for (int i = 1; i <= 13; i++) {
      Card c1(diamond, i), c2(spade, i), c3(heart, i), c4(club, i);
      cards[topCard++] = c1;
      cards[topCard++] = c2;
      cards[topCard++] = c3;
      cards[topCard++] = c4;
      }
}
```

Note use of local variables.

increment/subscript idiom – very common in C and C++ programs.

# To Shuffle, Need Random Numbers

To shuffle, need a source of random numbers. C++ run-time library provides one function, but not exactly what we need. But we can build what we need. First problem, rand() returns an arbitrary integer. How would we convert this into a value between larger than or equal to zero and smaller than max?

```
    // rand() returns a random integer
    // take remainder when divided by max
    // to produce value in desired range
unsigned int rval = rand();
return rval % max;
```

# Function Objects

Now the next part is a bit tricky. The library function we want to call requires an OBJECT that ACTS like a function. But function calling is an operator in this language, so we can write this as follows:

```
class randomInteger {
    public:
        unsigned int operator () (unsigned int);
};
```

An object that can be used like a function is called a function object.

# Body of the randomInteger class

Note carefully the various parts of the random integer object.

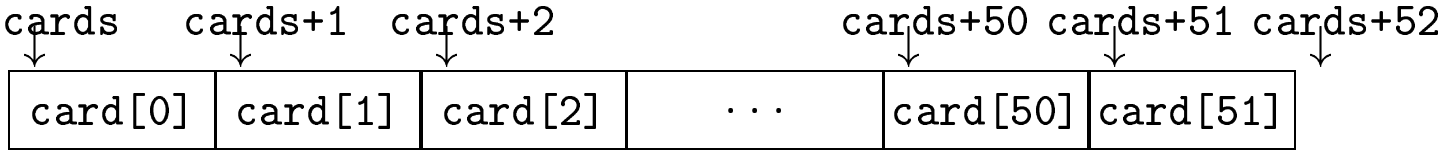
```
unsigned int randomInteger::operator () (unsigned int max)
{
        // rand return random integer
        // convert to unsigned to make positive
        // take remainder to put in range
    unsigned int rval = rand();
    return rval % max;
}
```

# The old Shuffle routine

```
randomInteger randomizer; // global variable randomizer object

void Deck::shuffle ( )
    // randomly shuffle the cards array,
    // using the generic algorithm random_shuffle
{
    random_shuffle (cards, cards+52, randomizer);
}
```

Uses a *generic algorithm* provided by the standard C++ library. This algorithm needs a pointer to the front and back of the array, as well as the function object that can be used as the random number generator.



# Qualified Names

```
void Deck::shuffle ( )
    // randomly shuffle the cards array,
    ...
```

Note the name of the function being defined.

A qualified name describes both the class name and the member function name – neither by itself is sufficient to fully identify the function.

Similar to the way we use first names and last names to identify people.

# Draw a Card

```
Card Deck::draw ( )
   // return one card from the end of the deck
{
   if (! isEmpty())
      return cards[−−topCard];
   else {      // otherwise return ace of spades
      Card spadeAce(spade, 1);
      return spadeAce;
      }
}
```

Note defensive programming – always assume if somebody can use your software component incorrectly, they will.

Note call on *isEmpty* – asking "am I empty?"

# Testing to see if Deck is Empty

```
bool Deck::isEmpty ( )
    // return true if the deck has no cards
{
    return topCard <= 0;
}
```

# In-line Function Definitions

```
class Deck {
public:
      // constructor
   Deck ( );

      // operations
   void    shuffle ( )
      { random_shuffle (cards, cards+52, randomizer); }
   bool    isEmpty ( )
      { return topCard < 0; }
   Card    draw ( );

protected:
   Card    cards[52];
   int     topCard;
};
```

Should be used only for very short function bodies.

# The class Player

```
class Player {
public:
        // constructor
    Player (Deck &);

        // operations
    Card    draw ( );
    void    addPoints (int);
    int     score ();
    void    replaceCard (Deck &);

protected:
    Card    cards[3];
    int     myScore;
    int     removedCard;
};
```

# Constructor for Player

```
Player::Player (Deck & aDeck)
    // initialize the data fields for a player
{
    myScore = 0;
    for (int i = 0; i < 3; i++)
        cards[i] = aDeck.draw();
    removedCard = 0;
}
```

# Draw a Card from Hand

```
Card Player::draw ( )
   // return a random card from our hand
{
   removedCard = randomizer(3);
   return cards[removedCard];
}
```

Note this function is same name as one in `Deck`, but no confusion can arise (at least to the computer, won't talk about the programmer).

```
void    Player::replaceCard (Deck & aDeck)
   // replace last card played with new card
{
   cards[removedCard] = aDeck.draw();
}
```

# Keeping Score

```
void    Player::addPoints (int howMany)
    // add the given number of points to the current score
{
    myScore += howMany;
}

int    Player::score ( )
    // return the current score
{
    return myScore;
}
```

Functions to access or update a data field are called *accessor functions* and *mutators*.

# The Big Game

```
void main() {
    Deck theDeck; // create and shuffle the deck
    theDeck.shuffle();

    Player player1(theDeck); // create the two
    Player player2(theDeck); // players

    while (! theDeck.isEmpty() ) {
        Card card1  = player1.draw();
        cout << "Player 1 plays " << card1 << "\n";
        Card card2 = player2.draw();
        cout << "Player 2 plays " << card2 << "\n";

        if (card1.rank == card2.rank) { // tie
            player1.addPoints(1);
            player2.addPoints(1);
            cout << "Players tie\n";
            }
        else if (card1.rank > card2.rank) {
            player1.addPoints(2);
            cout << "Player 1 wins round\n";
            }
```

```
        else {
            player2.addPoints(2);
            cout << "Player 2 wins round\n";
            }
    }
    cout << "Player 1 score " << player1.score() << "\n";
    cout << "Player 2 score " << player2.score() << "\n";
}
```

# An Interactive Game

Wouldn't it be better if the game were interactive?

```
. . .
You current hold in your hand:
a) Ace of spades
b) 3 of clubs
c) seven of diamonds
which one do you want to play? b
Human plays 3 of clubs
Computer plays 7 of spades
Computer wins round
You currently hold in your hand:
a) Ace of spades
b) 4 of diamonds
c) seven of diamonds
which one do you want to play?
. . .
```

# The Human Player class

Make a class called `HumanPlayer`, that is exactly the same as the class `Player`, only the `draw` method does something different.
Simply replace the first declaration of `Player` in our game with `HumanPlayer`, and keep everything else the same.
Illustrates the great advantage of programming using components, called *encapsulation*.

# The Human Player Draw routine

```
Card   HumanPlayer::draw ()
   // draw one card from the current hand
{
   cout << "You currently hold in your hand:\n";
   cout << "a) " << cards[0] << "\n";
   cout << "b) " << cards[1] << "\n";
   cout << "c) " << cards[2] << "\n";
   cout << "Which one do you want to play? ";
   char answer[80];
   removedCard = -1;
   while (removedCard == -1) {
      cin >> answer;    // read response
      if (answer[0] == 'a')
         removedCard = 0;
      else if (answer[0] == 'b')
         removedCard = 1;
      else if (answer[0] == 'c')
         removedCard = 2;
      if (removedCard != -1)
         return cards[removedCard];
      cout << "please specify a, b or c\n";
      }
}
```