The Card Game WAR

The card game WAR is a simple card game for two players.[1] The 52 cards in a conventional deck are shuffled, and each player draws three cards. The remaining cards are placed in a pile facedown between the two players. Play then proceeds as a series of rounds. During each round both players select one of their three cards, and place it face up in front of them. If the ranks of both cards are the same, then both players retain their cards (setting them aside). Otherwise, the player with the highest-ranking card keeps both cards (again, setting them aside). After playing their card, players draw 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.

[1] This description is taken from Budd, it is not the usual version of the game, but is similar.

Object Oriented Programming

General approach:

1. Examine the problem statement and identify the *objects*.
2. Group the objects into *classes*.
3. Identify the *actions* (or *responsibilities*), which the classes perform.

Nouns identify objects and classes. The actions (or responsibilities) are identified by verbs.

Objects of the Game WAR

card

- Cards are ordered by the relationship rank – i.e., cards can be considered to have one higher than the other, or be equal.

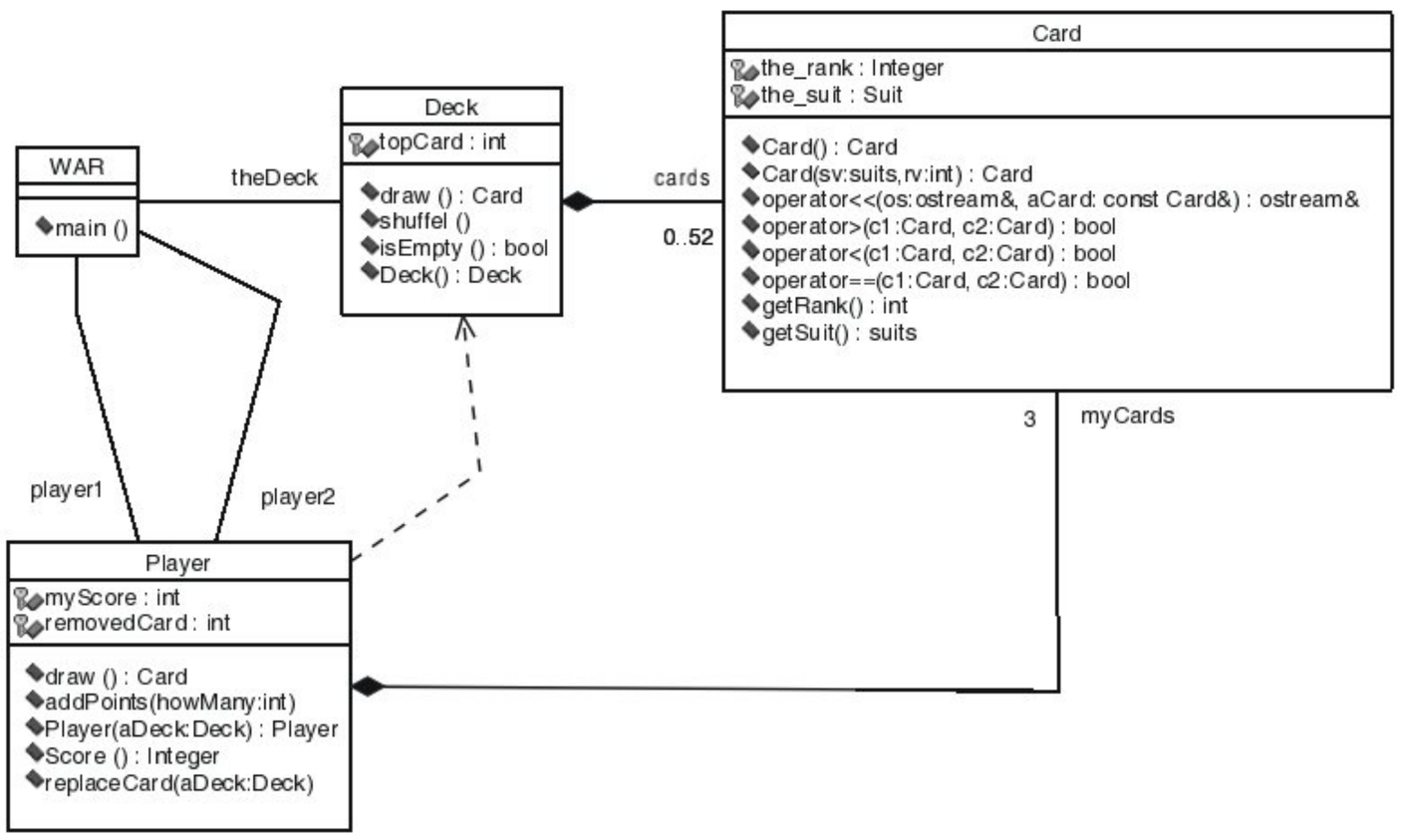
player

- Holds three cards.
- Draws cards.
- Plays cards.
- Keeps cards.

deck

- Contains 52 cards initially
- Is shuffled
- Cards are drawn from it.

UML Class Diagram of War



Why C++

Bjarne Stroustrup developed C++ with the following properties:

1. It allows the programmer to create new types that can be added to the language. (It allows ADTs to become concrete data types.)
2. It supports object oriented programming.
3. It improves on certain aspects of C; it is a "better C."

Templates

Templates allow us to define data structures capable of working on any other type.

Inheritance

New types can be created that share features of previously defined types. Many types can have a common parent type.

Polymorphism

Functions may be defined to operate on an abstract object, and when invoked with a specific object of a derived type, the action appropriate for the specific type will be performed.

The Class `Card`

Although not relevant to the game WAR, cards have two values: the suit and the rank.

The suit can be represented using an enumeration type:

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

The rank can be represented by an integer from 1 to 13.

The class Card is declared as follows:


```
class Card
{
public:
  // Constructors
  Card();            // initialize a card with default values
  Card(suits, int);  // initialize a card with given values
  int getRank() const;
  suits getSuit() const;
protected:
  // Member variables (attributes)
  int rank;
  suits suit;
// Output a card to an output stream
friend ostream& operator<<(ostream&, const Card&);
// Define ordering relationships on cards
friend bool operator<(const Card&, const Card&);
friend bool operator>(const Card&, const Card&);
friend bool operator==(const Card&, const Card&);
};
```

Card Constructors

```
Card::Card()
// Initialize a card to default values
{
  rank = 1; // the ace
  suit = spade;
}

Card::Card(suits sv, int rv)
// Initialize a card to given values
{
  rank = rv;
  suit = sv;
}
```

Alternate version of Card Constructors

```
Card::Card() : rank(1), suit(spade) {}

Card::Card(suits sv, int rv) : rank(rv), suit(sv) {}
```

The alternate is more efficient. Generally it is coded inline as part of the class definition.

`public`, `protected`, `private` and `friend`.

The items declared `public` are visible to all functions.

The items declared `protected` are visible to functions that are members of the class, `friend`s of the class, members of friends of the class, or are members of classes derived from that class.

The items declared `private` are visible to functions that are members of the class, friends of the class, or members of friends of the class.

input/output of Cards

The insertion operator (<<) is defined for the built-in types. We can extend its usage to user defined types as follows:

```
ostream& operator<<(ostream& out, const Card& aCard)
{
  switch(aCard.rank)
  {
  case  1: out << "Ace"; break;
  case 11: out << "Jack"; break;
  case 12: out << "Queen"; break;
  case 13: out << "King"; break;
  default: out << aCard.rank; break;
  }
  switch(aCard.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;
}
```


Comparison of cards

The functions to compare card values are defined as follows:

```
bool operator<(const Card& c1, const Card& c2)
{ return c1.rank < c2.rank; }

bool operator>(const Card& c1, const Card& c2)
{ return c1.rank > c2.rank; }

bool operator==(const Card& c1, const Card& c2)
{ return c1.rank == c2.rank; }
```

The class Deck

```
class Deck
{
public:
  // constuctor
  Deck();             // creates an initial deck
  // operations on a deck (methods)
  void shuffle();     // randomly change the order of the cards
  bool isEmpty() const;  // return true of empty; false otherwise
  Card draw();        // return the next card
protected:
  // member variables (attributes)
  Card cards[52];     // hold collection of cards
  int topCard;        // index one greater than last card
};
```


Deck constructor

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

Shuffle

```
void Deck::shuffle()
{
  random_shuffle(cards, cards+52);
}
```

`random_shuffle` is a member of the library algorithms. It takes the items of a collection starting with the one referenced by the first argument up to, but not including the one referenced by the last argument and randomly rearranges them.

is_empty

```
bool Deck::isEmpty() const
{
  return topCard <= 0;
}
```

draw

```
Card Deck::draw()
{
  if (isEmpty())
    throw "Empty Deck";
  return cards[--topCard];
}
```

The class player

```
class Player
{
public:
  // constructor
  Player(Deck&);
  // operators
  Card draw();
  void addPoints(int);
  int score() const;
  void replaceCard(Deck&);
protected:
  Card myCards[3];
  int myScore;
  int removedCard;
};
```


Implementation of player

```
Player::Player(Deck& aDeck)
{
  myScore = 0;
  for (int i = 0; i < 3; i++)
    myCards[i] = aDeck.draw();
  removedCard = 0;
}

Card Player::draw()
{
  removedCard = rand() % 3;
  return myCards[removedCard];
}

void Player::addPoints(int howMany)
{ myScore += howMany; }

int Player::score() const
{ return myScore; }

void Player::replaceCard(Deck& aDeck)
{ myCards[removedCard] = aDeck.draw(); }
```

The main program

```
int main()
{
  Deck theDeck;
  theDeck.shuffle();
  Player player1(theDeck);
  Player player2(theDeck);
  while (!theDeck.isEmpty())
  {
    Card card1 = player1.draw();
    cout << "Player 1 draws " << card1 << endl;
    Card card2 = player2.draw();
    cout << "Player 2 draws " << card2 << endl;
    if (card1 == card2)
    {
      player1.addPoints(1);
      player2.addPoints(1);
      cout << "Players tie" << endl;
    }
    else if (card1 > card2)
    {
      player1.addPoints(2);
      cout << "Player 1 wins round" << endl;
    }
    else
```



```
    {
      player2.addPoints(2);
      cout << "Player 2 wins round" << endl;
    }
    player1.replaceCard(theDeck);
    player2.replaceCard(theDeck);
  }
  cout << "Player 1 score " << player1.score() << endl;
  cout << "Player 2 score " << player2.score() << endl;

  return 0;
}
```

Adding a Human Player

Step 1: Modify the class Player to be an abstract base class by defining the function draw to be pure virtual.

```
class Player
{
public:
  // constructor
  Player(Deck&);
  // operators
  virtual Card draw() = 0;
  void addPoints(int);
  int score() const;
  void replaceCard(Deck&);
protected:
  Card myCards[3];
  int myScore;
  int removedCard;
};
```


Adding a human Player

Step2:

Declare the classes `humanPlayer` and `machinePlayer` as follows:

```
class humanPlayer : public Player
{
public:
  humanPlayer(Deck& aDeck) : Player(aDeck) {}
  Card draw();
};

class machinePlayer : public Player
{
public:
  machinePlayer(Deck& aDeck) : Player(aDeck) {}
  Card draw();
};
```

NOTE: Only the function draw is defined. All other functions come from the base class `player`.

Adding a human Player

Step 3:

Implement the draw function for humans:

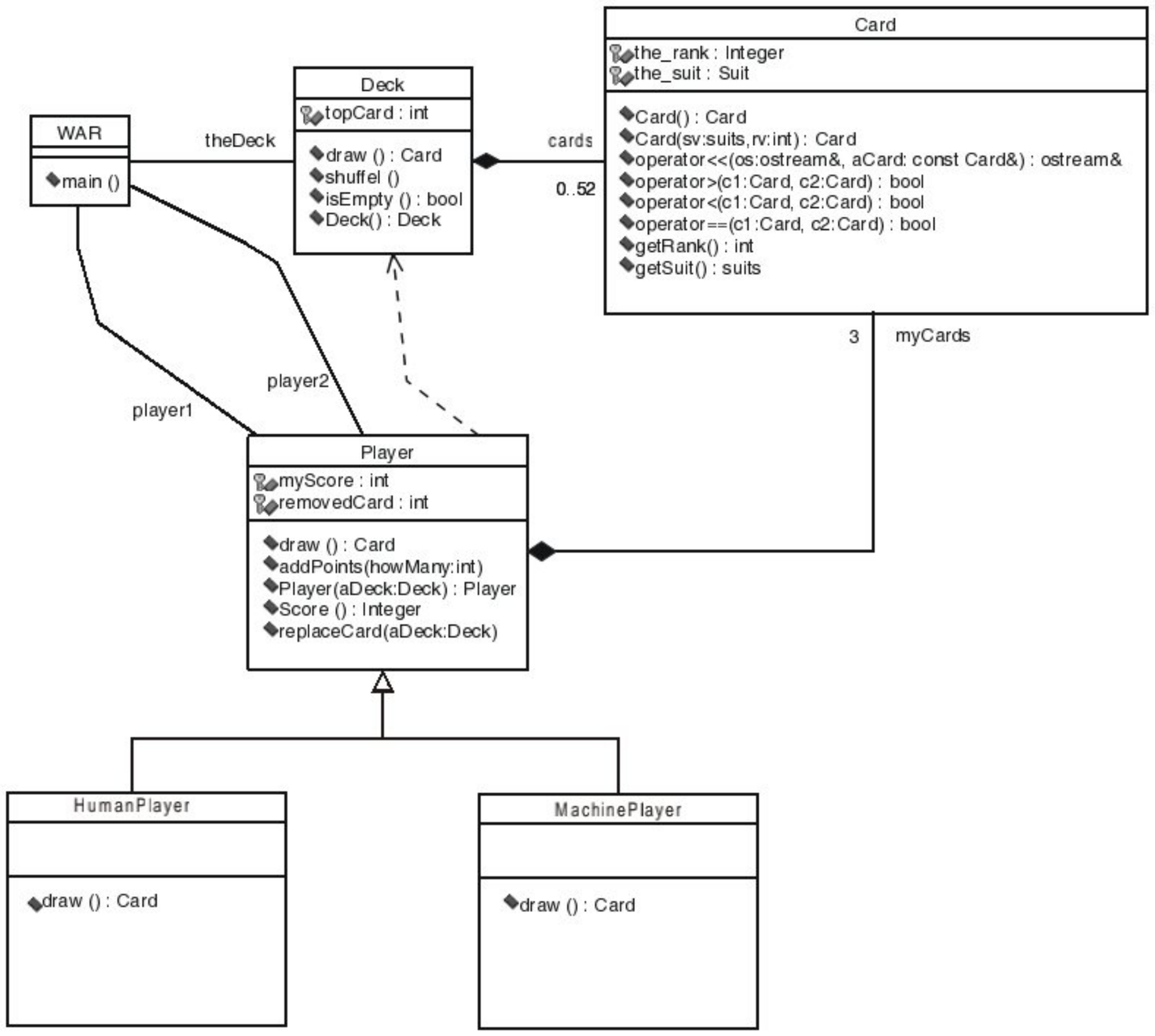
```
Card humanPlayer::draw(){
  cout << "you currently hold in your hand:" << endl;
  cout << "a) " << myCards[0] << endl;
  cout << "b) " << myCards[1] << endl;
  cout << "c) " << myCards[2] << endl;
  cout << "Which one do you want to play? ";
  char answer;
  removedCard = -1;
  while (removedCard == -1){
    cin >> answer;
    switch (answer){
    case 'a' : removedCard = 0; break;
    case 'b' : removedCard = 1; break;
    case 'c' : removedCard = 2; break;
    default:
      cout << "Please specify a, b, or c" << endl; break;
    }
  }
  return myCards[removedCard];
}
```


Adding a human Player

Step 4:

Implement the draw function for machings

```
Card machinePlayer::draw()
{
  removedCard = rand() % 3;
  cout << "draws " << myCards[removedCard] << endl;
  return myCards[removedCard];
}
```



Revised main program

```
int main()
{
  srand(unsigned(time(0)));
  Deck theDeck;
  theDeck.shuffle();
  Player* pp1 = 0;
  Player* pp2 = 0;
  while (!pp1)
  {
    cout << "Is player 1 a human or a machine? ";
    string answer;
    cin >> answer;
    if (answer == "human")
      pp1 = new humanPlayer(theDeck);
    else if (answer == "machine")
      pp1 = new machinePlayer(theDeck);
  }
  while (!pp2)
  {
// the same as for pp1.
  }
```

```
  Player& player1 = *pp1;
  Player& player2 = *pp2;
  while (!theDeck.isEmpty())
  {
// Essentially the same as before.
}
  cout << "Player 1 score " << player1.score() << endl;
  cout << "Player 2 score " << player2.score() << endl;

  return 0;
}
```


structs

- A `struct` is a collection of related data items.
- The individual components of a `struct` can contain data of different types. We can use a `struct` to store a variety of information about a person, such as name, marital status, age, and data of birth.
- Each data item is stored in a separate member; we can reference each data item stored in a `struct` through its member name.

```
ID: 1234
Name:  Caryn Jackson
Gender:  Female
Number of Dependents:   2
Hourly Wage:  6.00
Total wages:  240.00
```

```
struct employee
{
  int ID;
  string name;
  char gender;
  int num_depend;
  float  rate, tot_wages;
};

    employee clerk, janitor;
```


Variable `clerk`

`id`	1234
`name`	Caryn Jackson
`gender`	F
`num_depend`	2
`rate`	6.00
`tot_wages`	240.00

`struct` Type Declaration

[This is a restricted form and does not show the full generality of the construct.]

Form: `struct` *struct-type* {
type$_1$ *id-list*$_1$;
type$_2$ *id-list*$_2$;

.
.
.

type$_n$ *id-list*$_n$;
};

Interpretation: The identifier *struct-type* is the name of the `struct` being described. Each *id-list*$_i$ is a list of one or more member names separated by commas; the data type of each member in *id-list*$_i$ is specified by *type*$_i$.

Note: *type*$_i$ can be any fundamental or user-defined data type, including a structured type such as an array or `struct`. If *type*$_i$ is a user-defined data type, it must be defined before the `struct` is defined. The name of the `struct` itself is defined as soon as `struct` is reached. Thus, a `struct` may refer to itself.

Member Access

‹postfix expression› ::= ‹postfix expression› . ‹name›

‹postfix expression› must be an object of a class (`struct`) type, and ‹name› must be a member of that class (`struct`).

```
clerk.ID = 1234;
clerk.name = "Caryn Jackson";
clerk.gender = 'F';
clerk.num_depend = 2;
clerk.rate = 6.00;
clerk.tot_wages = clerk.tot_wages +clerk.rate * 40.0;

cout << "The clerk is ";
switch (clerk.gender){
  case 'F': case 'f': cout << "Ms. "; break;
  case 'M': case 'm': cout << "Mr. "; break;
}
cout << clerk.name;
```

classes

`class` ‹class name›

```
{
    public:
```

list of class variables, types, constants, etc. that may be accessed by name from outside the class

list of prototypes for each function that may be accessed by name from outside the class

```
    private:
```

list of class variables, types, constants, etc. that are intended to be hidden for reference from outside the class

list of prototypes for each function intended to be hidden from outside the class.

```
};
```

Note: In C++ `structs` and `classes` are identical constructs with the exception that the default for `structs` is `public` while the default for `classes` is `private`. We will use `structs` only for those classes which only contain data and the data is public.

Constructors and Member Initialization

The purpose of the constructor is to convert raw memory into an object. This is accomplished by initializing the data members.

Two methods to initialize a data member:

1. Use an assignment statement
2. Member initialization expression

The latter is more efficient.

Constructor Syntax

‹class name›(‹argument list›) ‹member initialization list› { ‹body›}

‹class name› is the name of the class. A constructor is a special member function whose name is the same as the class and has no return type.

‹argument list› is a list of zero or more formal arguments

‹member initialization list› ::= ‹empty› | : ‹member initialization expression› ‹member initialization tail›

‹member initialization tail› ::= ‹empty› | , ‹member initialization expression› ‹member initialization tail›

‹member initialization expression› ::= ‹member name›(‹expression›)

Using a Class, and the `this` object.

The statement:

```
Deck theDeck;
```

defines an object (or variable) of type Deck.

The statement

```
theDeck.shuffle();
```

will result in the deck being shuffled.

The object to the left of the dot is assigned to the implicit parameter `this` when a member function is called. (Actually a pointer to the object is passed to the pointer parameter `this`.) Within the execution of the member function, members (both public and private) are accessed by their names and the compiler implicitly interprets this as accessing the member through the this pointer.

Operator functions

The form

‹operand 1›‹operator›‹operand 2›

is interpreted by C++ as either

`operator`‹operator›`(`‹operand 1›`,`‹operand 2›`)`

or

‹operand1›`.operator`‹operator›`(`‹operand 2›`)`

For example

```
a + b
```

is

```
operator+(a, b)
```

or

```
a.operator+(b)
```


Member vs Non-member Operator Functions

The form

‹class›`::operator`‹opr›`(`‹right hand operand›`)`

is generally used for operators which modify the left hand operand. Their return value is the modified operand.

The form

`operator`‹opr›`(`‹left hand operand›, ‹right hand operand›`)`

is generally used

1. when the result is a new object
2. when the left hand operand is an external class which we wish to extend to operate on the right hand class – e.g. the I/O stream classes.

Example, the class complex

Complex numbers contain both a real part and an imaginary part. Mathematically they are generally represented by the form

$$a + bi$$

where a is the real part,

b is the imaginary part,

and i is the square root of –1.

Addition and subtraction.

If $x = a + bi$ and $y = c + di$ are complex numbers then

$x \pm y = (a + bi) \pm (c + di) = (a \pm c) + (b \pm d)i$

Multiplication

If $x = a + bi$ and $y = c + di$ are complex numbers then

$x \times y = (a + bi) \times (c + di) = a \times c + a \times di + bi \times c + bi \times di$

$= a \times c + (a \times d + b \times c)i + b \times d \times i \times i$

$i \times i = -1$

Therefore $x \times y = (a \times c - b \times d) + (a \times d + b \times c)i$

Similarly division can be determined to be

$$\frac{a + bi}{c + di} = \frac{ac + bd + (bc - ad)i}{c^2 + d^2}$$

Input and output:

output a complex number in the form (a, b)

Accept the output form as the input, but accept a single real without parenthesis as a real value with zero imaginary part.

```
// FILE: complex.h
// Declaration of complex number class
#ifndef COMPLEX_H_       // prevent multiple includes
#define COMPLEX_H_
#include <iostream>

class complex
{
public:
  complex (float rp = 0, float ip = 0)
    : re(rp), im(ip) {}
  complex& operator+=(const complex&);
  complex& operator-=(const complex&);
  complex& operator*=(const complex&);
  complex& operator/=(const complex&);
  friend complex operator+(const complex&, const complex&);
  friend complex operator-(const complex&, const complex&);
  friend complex operator*(const complex&, const complex&);
  friend complex operator/(const complex&, const complex&);
  friend std::ostream& operator<<(std::ostream&, const complex&);
  friend std::istream& operator>>(std::istream&, complex&);
  float real() const
  {return re;}
  float imig() const
  {return im;}
```



```
  bool operator==(const complex& rhs) const
  {
    return re == rhs.re && im == rhs.im;
  }
private:
  float re;
  float im;
};
#endif
```

```
// FILE: complex.cpp
// Implementation of the complex number class

#include "complex.h"

complex& complex::operator+=(const complex& rhs)
{
  re += rhs.re;
  im += rhs.im;
  return *this;
}

complex operator+(const complex& lhs, const complex& rhs)
{
  complex result = lhs;
  return result += rhs;
}

complex& complex::operator-=(const complex& rhs)
{
  re -= rhs.re;
  im -= rhs.im;
  return *this;
}
```



```
complex operator-(const complex& lhs, const complex& rhs)
{
  complex result = lhs;
  return result -= rhs;
}

complex operator*(const complex& lhs, const complex& rhs)
{
  float real_part = lhs.re*rhs.re - lhs.im*rhs.im;
  float imag_part = lhs.re*rhs.im + lhs.im*rhs.re;
  return complex(real_part, imag_part);
}

complex& complex::operator*=(const complex& rhs)
{
  *this = *this * rhs;
  return *this;
}
```

```
complex operator/(const complex& lhs, const complex& rhs)
{
  float denominator = rhs.re*rhs.re + rhs.im*rhs.im;
  float real_part = (lhs.re*rhs.re + lhs.im*rhs.im) /
                                                denominator;
  float imag_part = (lhs.im*rhs.re - lhs.re*rhs.im) /
                                                denominator;
  return complex(real_part, imag_part);
}

complex& complex::operator/=(const complex& rhs)
{
  *this = *this / rhs;
  return *this;
}

std::ostream& operator<<(std::ostream& os, const complex& rhs)
{
  os << '(';
  os << rhs.re;
  os << ',';
  os << rhs.im;
  os << ')';
  return os;
}
```



```
std::istream& operator>>(std::istream& is, complex& rhs)
{
  char c;
  is >> c;
  if (!is) return is;        // return if error or eof
  if (c == '(')
  {
    is >> rhs.re;
    if (!is) return is;
    is >> c;
  is >> c;
  if (!is) return is;
  if (c != ',')
  {
    is.putback(c);                  // put it back
    is.clear(std::ios::failbit);  // indicate incorrect input
    return is;                      // return immediatly
  }
  is >> rhs.im;
  if (!is) return is;
  is >> c;
  if (!is) return is;
  if (c != ')')
  {
    is.putback(c);
    is.clear(std::ios::failbit);
    return is;
```

```
    }
    return is;
  }
  is.putback(c);
  is >> rhs.re;
  rhs.im = 0;
  return is;
}
```



```
// FILE: test.cpp
// Program to test complex number class
#include "complex.h"
#include <iostream>
using std::cin;
using std::cout;
using std::cerr;
using std::endl;
using std::istream;

complex read_number(istream is)
{
  complex value;
  while (!(is >> value))
  {
    cerr << "Bad input format -- try again" << endl;
    is.clear(0);
    is.ignore(255, '\n');
    is.clear(0);
  }
  return value;
}
```

```
int main()
{
  cout << "Enter a pair of complex numbers" << endl;
  cout << "Enter all zeros to quit" << endl;
  complex c1, c2, c3;
  for (;;)
  {
    c1 = read_number(cin);
    c2 = read_number(cin);
    if (c1 == complex(0,0) && c2 == complex(0,0)) return 0;
    c3 = c1 + c2;
    cout << c1 << " + " << c2 << " == " << c3 << endl;
    c3 = c1 - c2;
    cout << c1 << " - " << c2 << " == " << c3 << endl;
    c3 = c1 * c2;
    cout << c1 << " * " << c2 << " == " << c3 << endl;
    c3 = c1 / c2;
    cout << c1 << " / " << c2 << " == " << c3 << endl;
    c3 = c1;
    c3 += c2;
    cout << c1 << " += " << c2 << " == " << c3 << endl;
    c3 = c1;
    c3 -= c2;
    cout << c1 << " -= " << c2 << " == " << c3 << endl;
    c3 = c1;
    c3 *= c2;
```



```
    cout << c1 << " *= " << c2 << " == " << c3 << endl;
    c3 = c1;
    c3 /= c2;
    cout << c1 << " /= " << c2 << " == " << c3 << endl;
  }
}
```

Assignment 1

Due 25 February 2002

The simplified game of WAR is still not very interesting. The HumanPlayer gets to look at his cards and can apply some strategy, but the MachinePlayer merely randomly picks a card to play. Furtheremore, if the HumanPlayer plays second, he can pick a card that will beat the other player's card, if he has one. Also, in traditional card games the Ace is considered the "highest" card, even though its rank value is one.

Make the following three changes to the program:

1) Whoever wins the current hand, will play second in the next hand.

2) Change the comparison operators so that Ace is greater than the other ranks. Note: you cannot change the value of the Ace, it must remain at one.

3) Add the following strategy to the MachinePlayer.

   a. If it does not know the card played against it, i.e., it is playing first, pick the largest card.

   b. If it knows the card played against it, pick the smallest card that is greater than the card played against it.

   c. If it knows the card played against it, and it cannot play a card that will win, play the smallest card held.

Note: The logic of the main program should only be changed to implement change (1) above. It should still be structured so that it does not treat player1 and player2 differently.