**Here is a screenshot of the output of the cards program:**A black background with white text

AI-generated content may be incorrect.

**Here is the code:**

#include <iostream>

#include <string>

using namespace std;

enum CardSuit { CS\_CLUB, CS\_DIAMOND, CS\_HEART, CS\_SPADE };

enum CardRank { CR\_ACE, CR\_TWO, CR\_THREE, CR\_FOUR, CR\_FIVE, CR\_SIX, CR\_SEVEN, CR\_EIGHT, CR\_NINE, CR\_TEN, CR\_JACK, CR\_QUEEN, CR\_KING };

class Card {

public:

Card(CardSuit s = CS\_CLUB, CardRank r = CR\_ACE);

string getStr() const;

private:

static string toSuitStr(CardSuit s);

static string toRankStr(CardRank r);

CardSuit suit;

CardRank rank;

};

Card::Card(CardSuit s, CardRank r) {

suit = s;

rank = r;

}

string Card::getStr() const {

return toRankStr(rank) + " of " + toSuitStr(suit);

}

string Card::toSuitStr(CardSuit s) {

string ans;

switch (s) {

case CS\_CLUB:

ans = "Clubs";

break;

case CS\_DIAMOND:

ans = "Diamonds";

break;

case CS\_HEART:

ans = "Hearts";

break;

case CS\_SPADE:

ans = "Spades";

break;

}

return ans;

}

string Card::toRankStr(CardRank r) {

string ans;

switch (r) {

case CR\_ACE:

ans = "Ace";

break;

case CR\_TWO:

ans = "Two";

break;

case CR\_THREE:

ans = "Three";

break;

case CR\_FOUR:

ans = "Four";

break;

case CR\_FIVE:

ans = "Five";

break;

case CR\_SIX:

ans = "Six";

break;

case CR\_SEVEN:

ans = "Seven";

break;

case CR\_EIGHT:

ans = "Eight";

break;

case CR\_NINE:

ans = "Nine";

break;

case CR\_TEN:

ans = "Ten";

break;

case CR\_JACK:

ans = "Jack";

break;

case CR\_QUEEN:

ans = "Queen";

break;

case CR\_KING:

ans = "King";

break;

}

return ans;

}

//I changed the formatting on these case statements to make the code a little more compact and maybe easier to read. I also added that default return case that we talked about in class and that fixed my issue.

CardSuit itoCardSuit(int s) {

switch (s) {

case 0: return CS\_CLUB;

case 1: return CS\_DIAMOND;

case 2: return CS\_HEART;

case 3: return CS\_SPADE;

default: return CS\_CLUB;

}

}

CardRank itoCardRank(int r) {

switch (r) {

case 0: return CR\_ACE;

case 1: return CR\_TWO;

case 2: return CR\_THREE;

case 3: return CR\_FOUR;

case 4: return CR\_FIVE;

case 5: return CR\_SIX;

case 6: return CR\_SEVEN;

case 7: return CR\_EIGHT;

case 8: return CR\_NINE;

case 9: return CR\_TEN;

case 10: return CR\_JACK;

case 11: return CR\_QUEEN;

case 12: return CR\_KING;

default: return CR\_ACE;

}

}

int main() {

Card c1(CS\_SPADE, CR\_ACE);

cout << c1.getStr() << endl;

cout << endl << endl << endl;

Card c;

for (int suit = 0; suit < 4; suit++) {

for (int rank = 0; rank < 13; rank++) {

c = Card(itoCardSuit(suit), itoCardRank(rank));

cout << c.getStr() << " ";

}

cout << endl;

}

return 0;

}

**Here is a screenshot of the output of my fractions program:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Here is the code from my fractions.cpp:**

#include <iostream>

#include <cmath>

#include <string>

using namespace std;

class Fraction {

public:

Fraction(int num=0, int den=1);

string getString() const;

void Print() const;

int getNum() const;

int getDen() const;

double getVal() const;

Fraction operator\*(Fraction rhs);

Fraction operator/(Fraction rhs);

Fraction operator-(Fraction rhs);

Fraction operator+(Fraction rhs);

bool operator==(Fraction rhs);

bool operator!=(Fraction rhs);

bool operator<(Fraction rhs) const;

bool operator>(Fraction rhs) const;

bool operator<=(Fraction rhs) const;

bool operator>=(Fraction rhs) const;

friend ostream& operator<<(ostream& os, const Fraction& f);

private:

int gcd(int m, int n);

int numerator;

int denominator;

};

Fraction::Fraction (int num, int den) {

numerator = num;

denominator = den;

}

string Fraction::getString() const {

return to\_string(numerator) + " / " + to\_string(denominator);

}

void Fraction::Print() const {

cout << numerator << " / " << denominator << endl;

}

int Fraction::getNum() const {

return numerator;

}

int Fraction::getDen() const {

return denominator;

}

double Fraction::getVal() const {

return static\_cast<double>(numerator) / denominator; //I read up on static\_cast and I guess it's a "safer" way to cast variables

}

// I implemnted the fraction reduction in each operator overload. In hindsight, I should have made a separate function for it, or at least I think that would have been the better way.

// Guess I could always time it and see if there's a difference haha.

Fraction Fraction::operator\*(Fraction rhs) {

Fraction result;

result.numerator = numerator \* rhs.numerator;

result.denominator = denominator \* rhs.denominator;

int common = gcd(abs(result.numerator), abs(result.denominator));

result.numerator /= common;

result.denominator /= common;

return result;

}

Fraction Fraction::operator/(Fraction rhs) {

Fraction result;

result.numerator = numerator \* rhs.denominator;

result.denominator = denominator \* rhs.numerator;

int common = gcd(abs(result.numerator), abs(result.denominator));

result.numerator /= common;

result.denominator /= common;

return result;

}

Fraction Fraction::operator-(Fraction rhs) {

Fraction result;

result.numerator = numerator \* rhs.denominator - denominator \* rhs.numerator;

result.denominator = denominator \* rhs.denominator;

int common = gcd(abs(result.numerator), abs(result.denominator));

result.numerator /= common;

result.denominator /= common;

return result;

}

Fraction Fraction::operator+(Fraction rhs) {

Fraction result;

result.numerator = numerator \* rhs.denominator + denominator \* rhs.numerator;

result.denominator = denominator \* rhs.denominator;

int cd = gcd(abs(result.numerator), abs(result.denominator));

result.numerator /= cd;

result.denominator /= cd;

return result;

}

// I implemented the equality and inequality operators by multiplying the numerators and denominators of the two fractions and comparing them.

bool Fraction::operator==(Fraction rhs) {

return numerator \* rhs.denominator == denominator \* rhs.numerator;

}

bool Fraction::operator!=(Fraction rhs) {

return numerator \* rhs.denominator != denominator \* rhs.numerator;

}

bool Fraction::operator<(Fraction rhs) const {

return numerator \* rhs.denominator < rhs.numerator \* denominator;

}

bool Fraction::operator>(Fraction rhs) const {

return numerator \* rhs.denominator > rhs.numerator \* denominator;

}

bool Fraction::operator<=(Fraction rhs) const {

return !(\*this > rhs);

}

bool Fraction::operator>=(Fraction rhs) const {

return !(\*this < rhs);

}

//This is the gcd function that was from the powerpoint. I don't think I changed anything in it.

int Fraction::gcd(int m, int n){

int oldm, oldn;

while (m%n != 0) {

oldm = m;

oldn = n;

m = oldn;

n = oldm%oldn;

}

return n;

}

//This is the ostream overload. We built this one in class.

ostream& operator<<(ostream& os, const Fraction& f) {

os << f.numerator << " / " << f.denominator;

return os;

}

int main() {

Fraction f1(55, 16); // Made 3 fractions to test the overloads

Fraction f2(6, 12);

Fraction f3(9, 5);

cout << "Fraction 1: " << f1 << endl;

cout << "Fraction 2: " << f2 << endl;

cout << "Fraction 3: " << f3 << endl;

cout << endl;

// Tests for all the different overloads. I just did random combinations of the 3 fractions I created to get various results. The outputs all seem to come out correctly, so let’s hope I implemented them all correctly!

// Multiplication

Fraction mult1 = f1 \* f2;

Fraction mult2 = f2 \* f3;

cout << f1 << " \* " << f2 << " = " << mult1 << endl;

cout << f2 << " \* " << f3 << " = " << mult2 << endl;

// Division

Fraction div1 = f1 / f3;

Fraction div2 = f2 / f3;

cout << f1 << " / " << f3 << " = " << div1 << endl;

cout << f2 << " / " << f3 << " = " << div2 << endl;

// Subtraction

Fraction sub1 = f1 - f2;

Fraction sub2 = f2 - f3;

cout << f1 << " - " << f2 << " = " << sub1 << endl;

cout << f2 << " - " << f3 << " = " << sub2 << endl;

// Addition

Fraction add1 = f1 + f2;

Fraction add2 = f2 + f3;

cout << f1 << " + " << f2 << " = " << add1 << endl;

cout << f2 << " + " << f3 << " = " << add2 << endl;

// Equality

cout << "(" << f1 << ") == (" << f1 << ") is " << (f1 == f1 ? "true" : "false") << endl;

cout << "(" << f1 << ") == (" << f2 << ") is " << (f1 == f2 ? "true" : "false") << endl;

cout << "(" << f1 << ") != (" << f2 << ") is " << (f1 != f2 ? "true" : "false") << endl;

cout << "(" << f1 << ") != (" << f1 << ") is " << (f1 != f1 ? "true" : "false") << endl;

cout << endl;

// I’m not sure if this was the most efficient way to do this. I feel like it could be done a little more efficiently.

cout << "(" << f1 << ") > (" << f2 << ") is " << (f1 > f2 ? "true" : "false") << endl;

cout << "(" << f1 << ") > (" << f3 << ") is " << (f1 > f3 ? "true" : "false") << endl;

cout << "(" << f2 << ") < (" << f1 << ") is " << (f2 < f1 ? "true" : "false") << endl;

cout << "(" << f3 << ") < (" << f1 << ") is " << (f3 < f1 ? "true" : "false") << endl;

cout << endl;

cout << "(" << f1 << ") >= (" << f1 << ") is " << (f1 >= f1 ? "true" : "false") << endl;

cout << "(" << f1 << ") >= (" << f2 << ") is " << (f1 >= f2 ? "true" : "false") << endl;

cout << "(" << f2 << ") <= (" << f1 << ") is " << (f2 <= f1 ? "true" : "false") << endl;

cout << "(" << f3 << ") <= (" << f1 << ") is " << (f3 <= f1 ? "true" : "false") << endl;

cout << endl;

return 0;

}