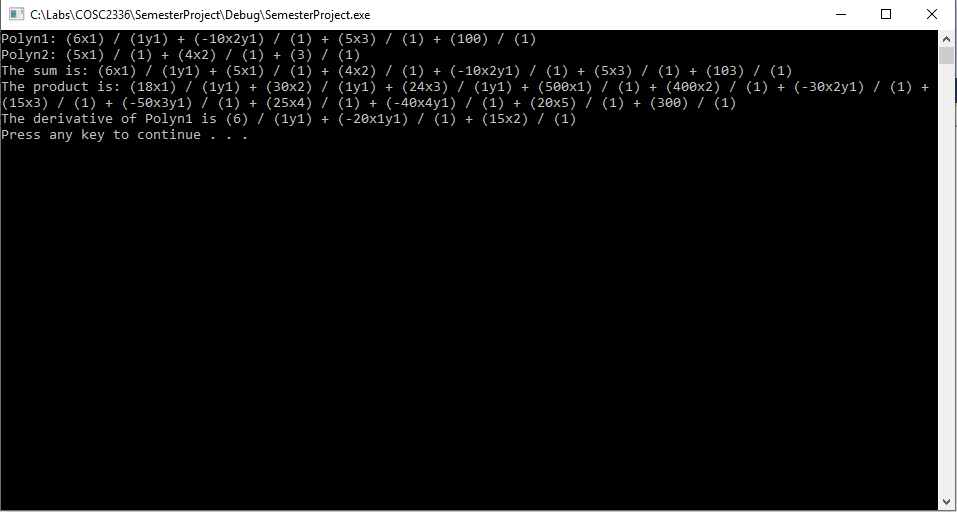
Name: William A. Brannon

Assignment: Lab 9

Date Due: Friday, May, 17, 2019

Class: Fundamentals of Programming III Section #1

**Program Output**



**SemesterProject.cpp**

// SemesterProject.cpp : Polynomial Math

// By William A. Brannon on 05/14/2019

//I started with the semester project, then tried to build on that code to do Lab 5.

//However, I broke the code in the process and this is being built upon what I had previously

//done for the semester project.

#include "stdafx.h"

#include <iostream>

#include <cctype>

#include <cstdlib>

#include <vector>

#include <list>

#include <algorithm>

#include <string>

using namespace std;

class Variable {

public:

char id;

int exp;

Variable() { // required by <vector>;

}

Variable(char c, int i) {

id = c; exp = i;

}

bool operator== (const Variable& v) const {

return id == v.id && exp == v.exp;

}

bool operator< (const Variable& v) const { // used by sort();

return id < v.id;

}

};

class Term {

public:

Term() {

coeff = 0;

}

int coeff;

vector<Variable> vars;

bool operator== (const Term&) const;

bool operator!= (const Term& term) const { // required by <list>

return !(\*this == term);

}

bool operator< (const Term&) const;

bool operator> (const Term& term) const { // required by <list>

return \*this != term && (\*this < term);

}

Term operator\* (const Term& other) const {

Term temp;

temp.coeff = coeff \* other.coeff;

temp.vars = vars;

for (Variable var : other.vars) {

bool found = false;

for (unsigned int i = 0; i < temp.vars.size(); i++) {

if (var.id == temp.vars.at(i).id) {

temp.vars.at(i).exp += var.exp;

found = true;

}

}

if (!found) {

temp.vars.push\_back(var);

}

}

return temp;

}

void input(string s) {

string temp = s;

if (s.at(0) == '-') {

coeff = -1;

temp = temp.substr(1);

}

else {

coeff = 1;

if (s.at(0) == '+') {

temp = temp.substr(1);

}

}

if (isdigit(temp.at(0))) {

string temp2 = "";

for (unsigned int i = 0; i < temp.length(); i++) {

temp2 += temp.at(i);

if (i == temp.length() - 1) {

coeff \*= stoi(temp2);

temp = "";

}

else if (isalpha(temp.at(i + 1))) {

coeff \*= stoi(temp2);

temp = temp.substr(temp2.length());

break;

}

}

if (temp == "") {

return;

}

char c = temp.at(0);

string j = "";

while (!temp.empty()) {

for (unsigned int i = 0; i < temp.length(); i++) {

if (isalpha(temp.at(i))) {

c = temp.at(i);

}

else {

j += temp.at(i);

}

if (i == temp.length() - 1) {

Variable tempVar(c, stoi(j));

vars.push\_back(tempVar);

temp = temp.substr(j.length() + 1);

}

else if (isalpha(temp.at(i + 1))) {

Variable tempVar(c, stoi(j));

vars.push\_back(tempVar);

temp = temp.substr(j.length() + 1);

j = "";

break;

}

}

}

}

else {

char c = temp.at(0);

string j = "";

while (!temp.empty()) {

for (unsigned int i = 0; i < temp.length(); i++) {

if (isalpha(temp.at(i))) {

char c = temp.at(i);

}

else {

j += temp.at(i);

}

if (i == temp.length() - 1) {

Variable tempVar(c, stoi(j));

vars.push\_back(tempVar);

temp = temp.substr(j.length() + 1);

}

else if (isalpha(temp.at(i + 1))) {

Variable tempVar(c, stoi(j));

vars.push\_back(tempVar);

temp = temp.substr(j.length() + 1);

}

}

}

}

}

string output() {

string temp = "";

temp += to\_string(coeff);

for (Variable var : vars) {

temp += var.id + to\_string(var.exp);

}

return temp;

}

int min(int n, int m) const {

return (n < m) ? n : m;

}

void sortVars() {

Variable temp;

//Sorts Variables Alphabetically

int mini;

for (unsigned int i = 0; i < vars.size(); i++) {

mini = i;

for (unsigned int j = i + 1; j < vars.size(); j++) {

if (vars[j].id < vars[mini].id) {

mini = j;

}

}

temp = vars[i];

vars[i] = vars[mini];

vars[mini] = temp;

}

}

};

class termFraction {

public:

Term numerator;

Term denominator;

void input(string s) {

size\_t found = s.find('/');

if (found != std::string::npos)

{

//substring based on where / is and then concatenate the right one with +

string s1 = s.substr(0,found);

string s2 = s.substr(found + 1);

numerator.input(s1);

denominator.input("+" + s2);

}

else {

numerator.input(s);

denominator.input("+1");

}

}

void cleanFraction() {

for (unsigned int i = 0; i < numerator.vars.size(); i++) {

for (unsigned int j = 0; j < denominator.vars.size(); j++) {

if (numerator.vars.at(i).id == denominator.vars.at(j).id) {

int difference = numerator.vars.at(i).exp - denominator.vars.at(j).exp;

if (difference > 0) {

numerator.vars.at(i).exp = difference;

denominator.vars.erase(denominator.vars.begin() + j);

}

else if (difference == 0) {

numerator.vars.erase(numerator.vars.begin() + i);

denominator.vars.erase(denominator.vars.begin() + j);

}

else if (difference < 0) {

difference \*= -1;

numerator.vars.erase(numerator.vars.begin() + i);

denominator.vars.at(i).exp = difference;

}

if (i != 0) {

i--;

}

if (j != 0) {

i--;

}

}

}

}

}

bool operator< (const termFraction& termFrac2) const { // used by sort();

if (denominator < termFrac2.denominator) {

return true;

}

else if (denominator == termFrac2.denominator) {

if (numerator < termFrac2.numerator) {

return true;

}

}

return false;

}

};

class Polynomial {

public:

Polynomial() {

}

Polynomial operator+ (Polynomial&);

Polynomial operator\* (Polynomial&);

Term derivativeOf(Term original, int n) {

//n is the index to the var id that you are taking the derivative of

Term temp = original;

if ((original.vars.size() == 1) && (original.coeff == 1) && (original.vars.at(0).exp == 1)) {

//if the term consists of just one unmodified variable, it is functionally a constant and thus the derivative is 0

temp.coeff = 0;

return temp;

}

if ((n > 0) && original.vars.at(n).exp == 1) {

//if the variable for the derivative is just one unmodified variable, it is functionally a constant and thus the derivative of the entire term is 0

temp.coeff = 0;

return temp;

}

temp.coeff \*= temp.vars.at(n).exp;

temp.vars.at(n).exp--;

if (temp.vars.at(n).exp == 0) {

temp.vars.erase(temp.vars.begin() + n);

}

return temp;

}

void sortVarsInTerms() {

for (termFraction x : terms) {

x.numerator.sortVars();

x.denominator.sortVars();

}

}

Polynomial derivative() {

Polynomial Ptemp;

for (termFraction x : terms) {

if (x.denominator.vars.empty()) {

if (x.numerator.vars.empty()) {

//this term is a constant that gets ignored/erased in the derivative.

}

else {

for (unsigned int i = 0; i < x.numerator.vars.size(); i++) {

termFraction tempFrac;

tempFrac.numerator = derivativeOf(x.numerator,i);

tempFrac.denominator = x.denominator;

Ptemp.terms.push\_back(tempFrac);

}

}

}

else {

for (unsigned int i = 0; i < x.numerator.vars.size(); i++) {

termFraction tempFrac;

tempFrac.numerator = derivativeOf(x.numerator,i) \* x.denominator;

tempFrac.denominator = x.denominator \* x.denominator;

Ptemp.terms.push\_back(tempFrac);

}

for (unsigned int i = 0; i < x.denominator.vars.size(); i++) {

termFraction tempFrac;

tempFrac.numerator = x.numerator \* derivativeOf(x.denominator,i);

tempFrac.numerator.coeff \*= -1;

tempFrac.denominator = x.denominator \* x.denominator;

Ptemp.terms.push\_back(tempFrac);

}

}

}

Ptemp.cleanup();

return Ptemp;

}

void error(char \*s) {

cerr << s << endl;

exit(1);

}

vector<termFraction> terms;

void input(string str) {

string temp = "";

for (unsigned int i = 0; i < str.length(); i++) {

temp += str.at(i);

//if the next character is an operator, create a term fraction and clear the string

if (i == str.length() - 1) {

termFraction tempFrac;

tempFrac.input(temp);

terms.push\_back(tempFrac);

temp = "";

} else if (str.at(i + 1) == '+' || str.at(i + 1) == '-') {

termFraction tempFrac;

tempFrac.input(temp);

terms.push\_back(tempFrac);

temp = "";

}

}

cleanup();

}

string output() {

string temp = "";

for (unsigned int i = 0; i < terms.size(); i++) {

temp += '(' + terms.at(i).numerator.output() + ") / (" + terms.at(i).denominator.output() + ')';

if (i < terms.size() - 1) {

temp += " + ";

}

}

return temp;

}

void cleanup() {

//sort all the components of the polynomial

sort(terms.begin(), terms.end());

sortVarsInTerms();

//remove terms with a coefficient of 0

vector<termFraction> tempTerms;

for (termFraction term : terms) {

termFraction tempTerm = term;

bool erase = false;

if (term.numerator.coeff == 0) {

erase = true;

}

else if (term.denominator.coeff == 0) {

erase = true;

}

if (!erase) {

tempTerms.push\_back(tempTerm);

}

}

terms = tempTerms;

//combine like terms

tempTerms.clear();

for (termFraction term : terms) {

bool found = false;

for (unsigned int i = 0; i < tempTerms.size(); i++) {

if ((term.denominator == tempTerms.at(i).denominator) && (term.denominator.coeff == tempTerms.at(i).denominator.coeff) && (term.numerator == tempTerms.at(i).numerator)) {

found = true;

tempTerms.at(i).numerator.coeff += term.numerator.coeff;

break;

}

}

if (!found) {

tempTerms.push\_back(term);

}

}

terms = tempTerms;

//eliminate variables based on difference in exponents if they exist in the numerator and denominator

tempTerms.clear();

for (termFraction term : terms) {

termFraction tempTerm = term;

for (unsigned int i = 0; i < tempTerm.numerator.vars.size(); i++) {

for (unsigned int j = 0; j < tempTerm.denominator.vars.size(); j++) {

if (tempTerm.numerator.vars.at(i).id == tempTerm.denominator.vars.at(j).id) {

tempTerm.cleanFraction();

}

}

}

tempTerms.push\_back(tempTerm);

}

terms = tempTerms;

//sort all the components of the polynomial

sort(terms.begin(), terms.end());

sortVarsInTerms();

}

};

// two terms are equal if all varibles are the same and

// corresponding variables are raised to the same powers;

// the first cell of the node containing a term is excluded

// from comparison, since it stores coefficient of the term;

bool Term::operator== (const Term& term) const {

int i;

for (i = 0; i < min(vars.size(), term.vars.size()) &&

vars[i] == term.vars[i]; i++);

return i == vars.size() && vars.size() == term.vars.size();

}

bool Term::operator< (const Term& term2) const { // used by sort();

if (vars.size() == 0)

return false; // \*this is just a coefficient;

else if (term2.vars.size() == 0)

return true; // term2 is just a coefficient;

for (int i = 0; i < min(vars.size(), term2.vars.size()); i++)

if (vars[i].id < term2.vars[i].id)

return true; // \*this precedes term2;

else if (term2.vars[i].id < vars[i].id)

return false; // term2 precedes \*this;

else if (vars[i].exp < term2.vars[i].exp)

return true; // \*this precedes term2;

else if (term2.vars[i].exp < vars[i].exp)

return false; // term2 precedes \*this;

return ((int)vars.size() - (int)term2.vars.size() < 0) ? true : false;

}

Polynomial Polynomial::operator+ (Polynomial& polyn2) {

Polynomial result;

vector<termFraction> tempTerms;

//combine the polynomials into one by merging their vectors

tempTerms.reserve(terms.size() + polyn2.terms.size());

tempTerms.insert(tempTerms.end(), terms.begin(), terms.end());

tempTerms.insert(tempTerms.end(), polyn2.terms.begin(), polyn2.terms.end());

result.terms = tempTerms;

//clean up the new polynomial

result.cleanup();

return result;

}

Polynomial Polynomial::operator\* (Polynomial& polyn2) {

Polynomial result;

for (termFraction tempOne : terms) {

for (termFraction tempTwo : polyn2.terms) {

termFraction tempFracResult;

tempFracResult.numerator = tempOne.numerator \* tempTwo.numerator;

tempFracResult.denominator = tempOne.denominator \* tempTwo.denominator;

result.terms.push\_back(tempFracResult);

}

}

result.cleanup();

return result;

}

int main() {

Polynomial polyn1, polyn2;

polyn1.input("5x3+6x1/y1-10x2y1+100");

polyn2.input("4x2+5x1+3");

cout << "Polyn1: " << polyn1.output() << endl;

cout << "Polyn2: " << polyn2.output() << endl;

Polynomial polynTester = polyn1 + polyn2;

cout << "The sum is: " << polynTester.output() << endl;

polynTester = polyn1 \* polyn2;

cout << "The product is: " << polynTester.output() << endl;

polynTester = polyn1.derivative();

cout << "The derivative of Polyn1 is " << polynTester.output() << endl;

system("pause");

return 0;

}