**C++ Part I (INFO1-CE9264) Fall 2014 – Final Examination**

Clement Chan

**Question 1 – Polynomial Functions**

Polynomial Class.h

#include<iostream>

#include<cmath>

#include<cstdlib>

using namespace std;

class Poly{

private:

int order; //order of the polynomial

int \*coeff;

public:

Poly();

Poly(int); //Create Nth order poly and initiates all coefficients

Poly(int,int\*);

~Poly(); //Destructor

Poly(const Poly &p); //Copy Constructor

//mutators and accessors

void set();

void set(int[],int);

int getOrder() const;

int \*get();

int get(int) const;

//Operators Overloading

Poly operator+(const Poly &rhs);

Poly operator-(const Poly &rhs);

Poly operator\*(const int scale);

Poly operator\*(const Poly &rhs);

friend const Poly operator\*(const int scale, const Poly &P);

friend const Poly operator\*(const Poly &P, const int scale);

bool operator==(const Poly &rhs);

const int &operator[](int) const;

int &operator[](int);

int operator()(int);

Poly &operator=(const Poly& rhs);

friend ostream &operator << (ostream &Out, const Poly &rhs);

};

Polynomial Method.h

#include "Polynomial.h"

//Default Constructor

Poly::Poly(){

order = 0;

coeff = new int[order];

coeff[0] = 1;

}

//Constructors with initiated values

Poly::Poly(int Order){

order = (Order+1);

coeff = new int[order];

//Define all the coefficients with 1

for(int i=0; i < order; i++){

coeff[i] = 1;

}

}

//Constructors with coeff included

Poly::Poly(int Order, int \*Coeff){

order = (Order+1);

coeff = new int[order];

for(int i=0; i < order; i++){

coeff[i] = Coeff[i];

}

}

//Copy Constructors

Poly::Poly(const Poly &p){

cout<<"This is Copy Constructor..."<<endl;

order = p.order;

coeff = p.coeff;

}

//Destructor

Poly::~Poly(){

cout<<"This is Destructor..."<<endl;

delete[] coeff;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Mutators and Accessors \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Poly::set(){

int Order;

int Coeff\_int;

cout << "Please Enter The Order Of The Polynomial. " << endl;

cin >> Order;

order = (Order+1);

coeff = new int[order];

cout<<"Please enter the coefficient starting from X^0, enter 0 if power does not exist." <<endl;

for(int i=0; i<order; i++){

cin >> Coeff\_int;

coeff[i] = Coeff\_int;

}

}

//void set coefficients

void Poly::set(int Coeff[], int size){

order = (size+1);

coeff = new int[order];

for(int i=0; i<order; i++){

coeff[i] = Coeff[i];

}

};

//Get coefficient value

int Poly::get(int a) const{

return coeff[a];

}

//Getting order

int Poly::getOrder() const{

return order;

}

//Getting the whole coeff

int\* Poly::get(){

return coeff;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Operator Overloading Methods \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//Operator + Overloading

Poly Poly::operator+(const Poly &rhs){

Poly P = \*this;

if(P.getOrder() > rhs.getOrder())

{

int \*coeff\_temp = new int [P.getOrder()];

for(int i=0; i < P.getOrder(); i++){

coeff\_temp[i] = 0;

}

for(int i=0; i < rhs.getOrder(); i++){

coeff\_temp[i] = rhs.coeff[i];

}

for(int i=0; i < P.getOrder(); i++){

P.coeff[i] = coeff[i] + coeff\_temp[i];

}

P.set(P.coeff, P.getOrder()-1);

return P;

}

else

{

int \*coeff\_temp = new int [rhs.getOrder()];

for(int i=0; i < P.getOrder(); i++){

coeff\_temp[i] = P.coeff[i];

}

P.coeff = coeff\_temp;

for(int i=0; i<rhs.getOrder(); i++){

rhs.coeff[i] = rhs.coeff[i] + P.coeff[i];

}

P.set(rhs.coeff, rhs.getOrder()-1);

return P;

};

};

//Operator - Overloading

Poly Poly::operator-(const Poly &rhs){

Poly P = \*this;

if(P.getOrder() > rhs.getOrder())

{

int \*coeff\_temp = new int [P.getOrder()];

for(int i=0; i < P.getOrder(); i++){

coeff\_temp[i] = 0;

}

for(int i=0; i < rhs.getOrder(); i++){

coeff\_temp[i] = rhs.coeff[i];

}

for(int i=0; i < P.getOrder(); i++){

P.coeff[i] = coeff[i] - coeff\_temp[i];

}

P.set(P.coeff, P.getOrder()-1);

return P;

}

else

{

int \*coeff\_temp = new int [rhs.getOrder()];

for(int i=0; i < P.getOrder(); i++){

coeff\_temp[i] = P.coeff[i];

}

P.coeff = coeff\_temp;

for(int i=0; i<rhs.getOrder(); i++){

rhs.coeff[i] = P.coeff[i] - rhs.coeff[i];

}

P.set(rhs.coeff, rhs.getOrder()-1);

return P;

};

};

//Operator \* Scaler Overloading

Poly Poly::operator\*(const int scale){

Poly P = \*this;

for(int i=0; i < P.getOrder(); i++){

P.coeff[i] \*= scale;

}

P.set(P.coeff, P.getOrder()-1);

return P;

}

//For backward directions that take 2 arguments

const Poly operator\*(int scale, const Poly &P){

Poly P1 = P;

for(int i=0; i < P.getOrder(); i++){

P.coeff[i] \*= scale;

}

P1.set(P.coeff, P.getOrder()-1);

return P1;

}

const Poly operator \*(const Poly &P, int scale){

return scale \* P;

}

//Operator \* another polynomial

Poly Poly::operator\*(const Poly &rhs){

Poly P = \*this;

//Initializing the maximum order

int max\_order = (P.getOrder() + rhs.getOrder())-1;

//Initializing the new set for summation

int \*coeff\_final = new int[max\_order];

int \*coeff\_temp = new int[max\_order];

for(int i=0; i < max\_order; i++){

coeff\_final[i] = 0;

coeff\_temp[i] = 0;

}

for(int i=0; i < P.getOrder(); i++){

coeff\_final[i] = P.coeff[i];

}

for(int i=0; i<rhs.getOrder(); i++){

//Create a new summation array

if(i > 0){

delete[] coeff\_temp;

int \*coeff\_temp = new int[max\_order];

for(int i=0; i < max\_order; i++){

coeff\_temp[i] = 0;

}

}

for(int j=0; j<P.getOrder(); j++){

int total\_order = i + j;

if(i == 0){

coeff\_final[total\_order] = P.coeff[j]\*rhs.coeff[i];

}

else{

coeff\_temp[total\_order] = P.coeff[j]\*rhs.coeff[i];

coeff\_final[total\_order] += coeff\_temp[total\_order];

}

}

}

//Setting the coefficients are order into the class

P.set(coeff\_final, max\_order-1);

delete[]coeff\_final;

delete[]coeff\_temp;

return P;

};

//Operator boolean == for Poly Class

bool Poly::operator==(const Poly &rhs){

return true;

}

//const operator[]

int &Poly::operator[](int I){

if(I > order){

return coeff[I];

}

else{

return coeff[I];

}

}

const int &Poly::operator[](int I) const{

if(I > order){

return coeff[I];

}

else{

return coeff[I];

}

}

//Operator Honer's method()

int Poly::operator()(int a){

Poly P;

int max = order-1;

int init = coeff[max];

while(max > 0){

init = coeff[max-1] + a\*init;

max--;

}

return init;

}

//Operator overloading "="

Poly &Poly::operator=(const Poly &rhs){

if(this != &rhs){

delete[] coeff;

set(rhs.coeff,rhs.getOrder()-1);

}

return \*this;

}

//Outstream overloading

ostream& operator << (ostream &Out, const Poly &rhs){

int i = rhs.getOrder()-1;

int max;

//Search for the maximum polynomial output

for(int j=0; j<=i; j++){

int test = rhs.get(j);

if(abs(test)>0){

max = j;

}

}

while (i >= 0){

if(rhs.get(i)!=0){

if(rhs.get(i) >= 1){

if(i == max){

Out << "";

}else{

Out << "+";}

};

if(i > 1){

Out << rhs.get(i);

Out << "X^";

Out << i;

}else if (i == 1){

if(abs(rhs.get(i))==1 && rhs.get(i) < 0){

Out << "-";

Out << "X";

}else if(abs(rhs.get(i))==1 && rhs.get(i) > 0){

Out << "X";

}else{

Out << rhs.get(i);

Out << "X";}

}else if (i == 0)

{Out << rhs.get(i);}

}

i--;

}

return Out;

}

Polynomial Main.cpp

#include "Polynomial\_Method.h"

int main(){

Poly P1, P2;

int Order\_1 = 4;

int Order\_2 = 7;

int Coeff\_1[] = {-19,1,-12,3,2};

int Coeff\_2[] = {-19,1,-6,0,0,7,0,2};

//Initialize the coefficients in the class

P1.set(Coeff\_1,Order\_1);

P2.set(Coeff\_2,Order\_2);

Poly P3 = P1 + P2;

cout << "The Polynomial of P1 + P2 is: " << P3 << endl;

P1.set(Coeff\_1,Order\_1);

P2.set(Coeff\_2,Order\_2);

P3 = (P1 - P2);

cout <<"The Polynomial of P1 - P2 is: " << P3 << endl;

P1.set(Coeff\_1,Order\_1);

P2.set(Coeff\_2,Order\_2);

P3 = P1 \* 10;

cout << "The Polynomial of P1 \* 10 is: " << P3 << endl;

P1.set(Coeff\_1,Order\_1);

P2.set(Coeff\_2,Order\_2);

P3 = 10 \* P1;

cout << "The Polynomial of 10 \* P1 is: " << P3 << endl;

P1.set(Coeff\_1,Order\_1);

P2.set(Coeff\_2,Order\_2);

P3 = P2 \* P1;

cout << "The Polynomial of P1 \* P2 is : " << P3 << endl;

//Flag that shows yes or no

bool flag = (P1 == P2);

cout << "The flag of P1 == P2 is: " << flag << endl;

//Assigning new value to P2

P1.set(Coeff\_1,Order\_1);

P2.set(Coeff\_2,Order\_2);

cout << "P1[3] is: " << P1[3] << endl;

cout << "P2[5] is: " << P2[5] << endl;

P1[3] = P2[5];

cout << "After operatration, P1[3] is: " << P1[3] << endl;

//Horners Method

P1.set(Coeff\_1,Order\_1);

int Z = P1(5);

cout << "The value given X=5 for Polynomial P1 is: " << Z << endl;

//Other Testing of the function

Poly P4, P5;

P4.set();

P5 = P4;

P1.set(Coeff\_1,Order\_1);

Poly P6 = P1 + P5;

cout<<"The result of the sum of entered coefficient and P1 is: " << P6 << endl;

P5 = P4;

P1.set(Coeff\_1,Order\_1);

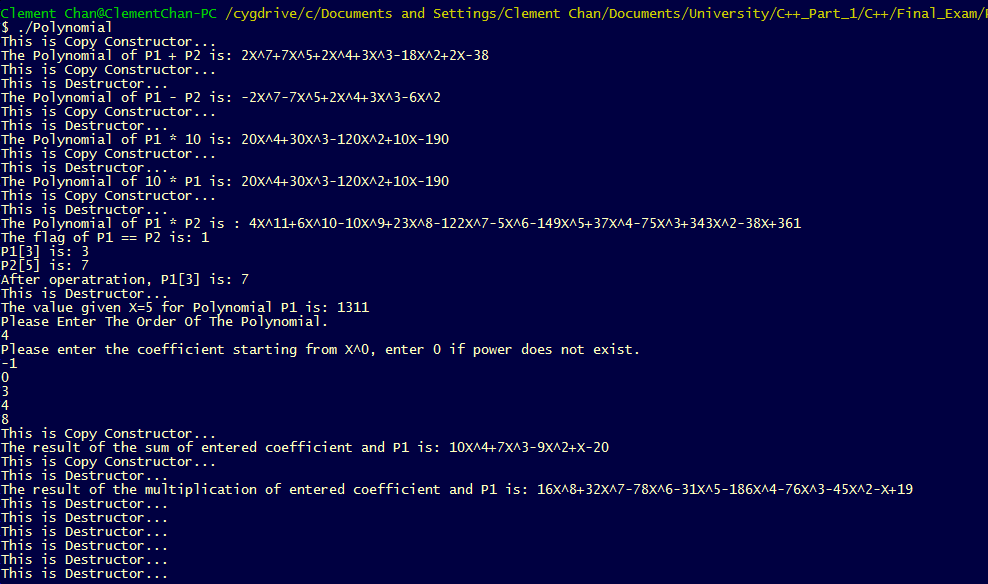
P6 = P1 \* P5;

cout<<"The result of the multiplication of entered coefficient and P1 is: " << P6 <<endl;

return 0;

}

**Output**

****

**Question 2 – Points**

Points.h

#include <iostream>

#include <cmath>

using namespace std;

class Point{

private:

int \*x;

int \*y;

int \*distance;

public:

//Constructors and Destructors

Point(); //Default Constructor

Point(int, int); //Parametric Constructor

Point(const Point &p); //Copy Constructor

~Point(); //Destructor

//Mutators and Accessors

void reset\_location(int, int);

void set\_distance(int);

int getX() const;

int getY() const;

int getdistance() const;

//Operator Overloading

Point &operator = (const Point &p);

Point operator - (Point &p);

bool operator == (const Point &p);

Point operator \* (const int scale);

friend const Point operator \*(const int scale, const Point &P);

friend ostream &operator << (ostream &out, const Point &rhs);

};

Points\_Method.cpp

#include "Points.h"

//Method definitions in Header files

Point::Point(){

cout<<"This is Default Constructor"<<endl;

x = new int;

y = new int;

distance = new int;

\*x=0;

\*y=0;

\*distance=0;

}

Point::Point(int a, int b){

cout<<"This is Parametarized Constructor"<<endl;

x = new int;

y = new int;

distance = new int;

\*x=a;

\*y=b;

\*distance=0;

}

Point::Point(const Point &p){

cout<<"This is the Copy Constructor"<<endl;

x = new int;

y = new int;

distance = new int;

\*x= \*p.x;

\*y= \*p.y;

\*distance = \*p.distance;

}

void Point::reset\_location(int value\_x, int value\_y){

\*x=value\_x;

\*y=value\_y;

}

void Point::set\_distance(int dist){

\*distance = dist;

}

int Point::getX() const{

return \*x;

}

int Point::getY() const{

return \*y;

}

int Point::getdistance()const{

return \*distance;

}

Point::~Point(){

cout<<"Here is the destructor." << endl;

delete[] x;

delete[] y;

delete[] distance;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Operator Overload \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//Assignment Operator

Point &Point::operator=(const Point &p){

if(this != &p){

reset\_location(p.getX(), p.getY());

set\_distance(p.getdistance());

}

return \*this;

}

//Boolean Operator

bool Point::operator==(const Point &p){

return true;

}

Point Point::operator-(Point &p){

Point P = \*this;

\*P.x = P.getX()\*p.getX();

\*P.y = P.getY()\*p.getY();

\*P.distance = int(sqrt(\*P.x + \*P.y));

return P;

}

Point Point::operator\*(const int scale){

Point P = \*this;

\*P.x \*= scale;

\*P.y \*= scale;

P.reset\_location(\*P.x, \*P.y);

return P;

}

const Point operator\*(const int scale, const Point &P){

\*P.x \*= scale;

\*P.y \*= scale;

return P;

}

ostream &operator << (ostream &out, const Point &rhs){

out << "The x coordinate of X is: ";

out << rhs.getX() << endl;

out << "The y coordinate of X is: ";

out << rhs.getY() << endl;

out << "The distance is: ";

out << rhs.getdistance() << endl;

return out;

}

void foo(Point &p, Point p1){

int x = p.getX();

int y = p1.getY();

Point temp(x,y);

cout << "x coordinate from Point X is:" << temp.getX()<< endl;

cout << "y coordinate from Point Y is:" << temp.getY()<< endl;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Main class to display \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(){

Point X(3,4), Y(10,40), Z, K;

int distance = (X-Y).getdistance();

cout<<"The distance between X and Y is: " <<distance<<endl;

X.reset\_location(3,4);

Z = 10 \* X;

cout << "X coordinate of 10\*X is: " << Z.getX() << endl;

cout << "Y coordinate of 10\*X is: " << Z.getY() << endl;

X.reset\_location(3,4);

Z = X\*10;

cout << "X coordinate of X\*10 is: " << Z.getX() << endl;

cout << "Y coordinate of X\*10 is: " << Z.getY() << endl;

X.reset\_location(3,4);

int x = X.getX();

int y = X.getY();

cout << "x coordinate in X is: " << x << endl;

cout << "y coordinate in Y is: " << y << endl;

cout << X;

Z = X-Y;

cout << Z;

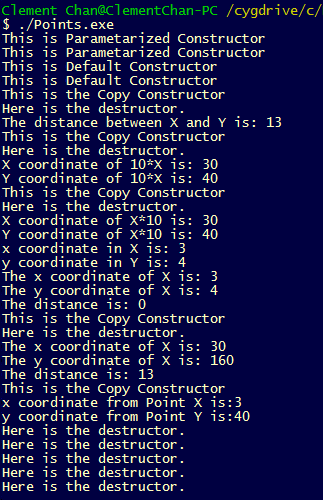
//Use foo to display the combinations of coordinates

foo(X,Y);

return 0;

}

**Output**



**Question 3 – Dates and Bonds**

Date.h

#include <iostream>

#include <cmath>

using namespace std;

//Global Variables;

static const int daysMonth[]={31,28,31,30,31,30,31,31,30,31,30,31};

static const int monthsYear = 12;

class Date{

private:

int day;

int month;

int year;

int \*date;

public:

//Constructors and Destructors

Date(); //Default Constructors

Date(int, int, int); //Parametric Constructors

Date(const Date &D); //Copy Constructor

~Date(); //Destructors

//mutators & accessors

int getDays() const;

int getMonth() const;

int getYear() const;

void DisplayDate() const; //Show Days, Months and Years

int calcDiff(const Date &Today);

//Operator overload

Date& operator = (const Date &D);

};

Date\_Method.h

#include "Date.h"

//Constructors and Destructors

//Default Constructors

Date::Date(){

day = 0;

month = 0;

year = 0;

date = new int[3];

for(int i=0; i<3; i++){

date[i] = 0;

}

}

//Parameterized Constructors

Date::Date(int a, int b, int c){

cout<<"This is Parameterized Constructors..."<<endl;

date = new int[3];

for(int i=0; i<3; i++){

date[i] = 0;

}

if(b > daysMonth[a-1] || a > monthsYear || a < 0){

cout<<"The values entered were not in the calendar.."<<endl;

}

month = a;

day = b;

year = c;

}

//Destructors

Date::~Date(){

cout<<"This is Destructor.."<<endl;

delete[] date;

}

//Copy Constructors

Date::Date(const Date &D){

day = D.day;

month = D.month;

year = D.year;

for(int i=0; i<3; i++){

date[i] = D.date[i];

}

}

//Mutators and Accessors

int Date::getDays()const{

return day;

}

int Date::getMonth()const{

return month;

}

int Date::getYear()const{

return year;

}

void Date::DisplayDate()const{

cout << getMonth() << "/" << getDays() << "/" << getYear() << endl;

}

//Operator overload

Date& Date::operator=(const Date &D){

day = D.getDays();

month = D.getMonth();

year = D.getYear();

return \*this;

}

int Date::calcDiff(const Date &Today){

Date temp;

int month;

int day;

int year;

temp.month = (getMonth() + 9) % 12;

temp.year = (getYear() - getMonth()/10);

temp.day = 365\*temp.getYear() + temp.getYear()/4 - temp.getYear()/400 +(temp.getMonth()\*306 +5)/10 +(temp.getDays()-1);

month = (Today.getMonth() + 9) % 12;

year = (Today.getYear() - Today.getMonth()/10);

day = 365\*year+year/4-year/400+(month\*306+5)/10+(Today.getDays()-1);

temp.day -= day;

return temp.getDays();

}

Bond.h

class Bond{

private:

char \*name;

float value;

Date purchaseDate;

Date maturityDate;

Date Today;

public:

Bond();

Bond(char\*, float, const Date &D1, const Date &D2);

//mutators and accessors

char\* get\_name() const;

float get\_price() const;

int getDays(const Date &D)const;

int getMonth(const Date &D)const;

int getYear(const Date &D)const;

//Calculate the summaries

void Display(const Date&D)const;

void setToday(Date &hoy);

int getCalcDiff(const Date &D, const Date &D1);

int daysToMaturity(Date &D);

};

Bond\_Method.h

#include "Bond.h"

//Constructors and Destructors

//Default Constructors

Bond::Bond(){

cout<<"This is Default Constructor..."<<endl;

cout<<"Not Enough Information..."<<endl;

exit(0);

}

//Parameterized Constructors

Bond::Bond(char\* a, float b, const Date &D1, const Date &D2){

cout<<"This is Parameterized Constructor..."<<endl;

name = a;

value = b;

purchaseDate = D1;

maturityDate = D2;

}

char\* Bond::get\_name()const{

return name;

}

float Bond::get\_price()const{

return value;

}

int Bond::getDays(const Date &D)const{

return D.getDays();

}

int Bond::getMonth(const Date &D)const{

return D.getMonth();

}

int Bond::getYear(const Date &D)const{

return D.getYear();

}

void Bond::Display(const Date&D)const{

D.DisplayDate();

}

void Bond::setToday(Date &hoy){

Today = hoy;

}

//Calculate days to maturity through value

int Bond::daysToMaturity(Date &D){

int day = getDays(maturityDate);

int month = getMonth(maturityDate);

int year = getYear(maturityDate);

Date mature(month,day,year);

//Implement calcDiff from Date

int diff = mature.calcDiff(D);

return diff;

}

Date\_Bond\_Main.cpp

#include "Date\_Method.h"

#include "Bond\_Method.h"

int main(){

Date Today(4,18,2012);

Date Maturity(12,31,2025);

Date Purchase(2,28,2012);

Bond Y("NYC\_Obligation", 885.0, Purchase, Maturity);

Bond \*bond\_ptr = new Bond("GW\_Bridge\_Obligation", 895.0, Purchase, Maturity);

int Y\_days = Y.daysToMaturity(Today);

float Y\_years = Y\_days / 365;

cout<<"The name of the First Bond is : "<< Y.get\_name() <<endl;

cout<<"The price of the First Bond is : "<< Y.get\_price() << endl;

cout<<"The difference between Maturity Date and Today's Date is: "<< Y\_days << " Days."<<endl;

cout<<"The difference between Maturity Date and Today's Date is: "<< Y\_years << " Years."<<endl;

int ptr\_days = bond\_ptr -> daysToMaturity(Today);

float ptr\_years = ptr\_days / 365;

cout<<"The name of the Second Bond is : "<< bond\_ptr -> get\_name() <<endl;

cout<<"The price of the Second Bond is : "<< bond\_ptr -> get\_price() << endl;

cout<<"The difference between Maturity Date and Today's Date is: "<<ptr\_days<< " Days."<<endl;

cout<<"The difference between Maturity Date and Today's Date is: "<<ptr\_years<< " Years."<<endl;

return 0;

}

**Output**

