FINM32200 Finals Practice

Table of Contents

Swap two integers 3

Swap using pointers 3

Factorial\_Creator 3

Prime\_Number 5

Fibonacci\_Number\_Finder 6

Square\_Root 7

Currency\_Converter 8

Circle 9

Customer 11

Currency\_Converter\_Classes 14

Options 18

Bubble Sort 24

Currency Converter Currency Factory Singleton 25

Option with Option Factory Singleton 30

Bank Accounts – Inheritance 37

Bank Accounts – Inheritance – 2 Levels 42

Option with Inheritance 48

Animal Farm Inheritance 60

Polygon 64

Adder Template Class 67

Vector STL 68

Iterator STL 68

List STL 69

Set STL 69

Map STL 70

Map STL 2 70

Bank Accounts Sequence Container 71

Bank Accounts Multiple Accounts per Customer 71

Currency Converter with Map 73

Currency Converter with Maps 75

Pointers Demo 79

Bank Account with Pointers 81

Currency Converter Pointers 83

Jarrow Rudd Binomial Tree 87

Assertions 97

Chrono 97

Exceptions 97

FileOps 98

IOManipulators 100

IOStreams 100

StringStream 101

Deque 102

Singly Linked List 102

Node 103

Slicing 105

Vector 106

Vector vs. List 109

Throw Exception 111

Circle Monte Carlo 111

Vector Midterm 113

# Swap two integers

Swap\_Integers.h

#pragma once

void Swap\_Integers(int integer1, int integer2);

Swap\_Integers.cpp

#include "Swap\_Integers.h"

void Swap\_Integers(int &x, int &y)

{

int temp;

temp = x; //save value at address x

x = y;

y = temp;

return;

}

Main.cpp

#include "Swap\_Integers.h"

#include <iostream>

using namespace std;

int main() {

int a = 100;

int b = 200;

swap(a, b);

cout << "a and b are now swapped so that a = " << a << " and b = " << b << endl;

return 0;

}

# Swap using pointers

void swap(int \*r, int \*s)

{

int temp = \*r;

\*r = \*s;

\*s = temp;

return;

}

# Factorial\_Creator

Factorial.h

#pragma once

int factorial(int x);

Factorial.cpp

#include "Factorial.h"

// for loop

int factorial(int x)

{

int result = 1;

for (int i = x; i > 0; --i)

{

result = result \* i;

}

return result;

}

/\*

// while loop

int factorial(int x)

{

int i = x;

int result = 1;

while (i > 0)

{

result = result\*i;

--i;

}

return result;

}

// do/while loop

int factorial(int x)

{

int i = x;

int result = 1;

do

{

result = result\*i;

--i;

} while (i > 0);

return result;

}

\*/

Main.cpp

#include <iostream>

#include "Factorial.h"

using namespace std;

int main()

{

int a = 5;

cout << "The factorial of " << a << " is " << factorial(a) << endl;

return 0;

}

# Prime\_Number

Prime\_Number.h

#pragma once

bool Is\_Prime(int x);

Prime\_Number.cpp

#include "Prime\_Number.h"

bool Is\_Prime(int x)

{

bool indicator = true;

for (int i = 2; i < x; ++i)

{

if (x % i == 0)

{

indicator = false;

break;

}

}

return indicator;

}

Main.cpp

#include <iostream>

#include "Prime\_Number.h"

using namespace std;

int main()

{

int a;

cout << "Choose an integer number" << endl;

cin >> a;

bool result = Is\_Prime(a);

if (result == true)

{

cout << "The number chosen is prime" << endl;

}

if (result == false)

{

cout << "The number chosen is not prime" << endl;

}

return 0;

}

# Fibonacci\_Number\_Finder

Fibonacci\_Number.h

#pragma once

bool Is\_Fibonacci(double x);

Fibonacci\_Number.cpp

#include "Fibonacci\_Number.h"

bool Is\_Fibonacci(double x)

{

double fib = 1; //fibonacci number tracker/builder

double fib\_old = 0; //set the previous fibonacci number

double temp;

bool result = false; //start by assuming that the number isn't fibonacci. if we find it is, switch to result = true

for (int i = 0; i <= x; ++i)

{

if (x == fib || x == fib\_old)

{

result = true;

break;

}

else

{

temp = fib\_old;

fib\_old = fib;

fib = fib + temp;

}

}

return result;

}

Main.cpp

#include <iostream>

#include "Fibonacci\_Number.h"

using namespace std;

int main()

{

int a;

cout << "Choose a positive integer and lets see if it's a Fibonacci number" << endl;

cin >> a;

bool result = Is\_Fibonacci(a);

if (result == true)

{

cout << "The number is a Fibonacci number" << endl;

}

else if (result == false)

{

cout << "The number is not a Fibonacci number" << endl;

}

return 0;

}

# Square\_Root

Square\_Root.h

#pragma once

double babylonian\_sqrt(double x);

Square\_Root.cpp

#include "Square\_Root.h"

double babylonian\_sqrt(double x)

{

double a = x / 2;

double temp;

bool running = true;

do

{

temp = a;

a = (a + x / a) / 2;

double diff = a - temp;

if (diff\*diff <= 0.000001)

{

break;

}

} while (running == true);

return a;

}

Main.cpp

#include <iostream>

#include "Square\_Root.h"

using namespace std;

int main()

{

double x;

cout << "Pick a nonnegative number to find its square-root using the babylonian method" << endl;

cin >> x;

cout << "The square-root of " << x << " is: " << babylonian\_sqrt(x) << endl;

return 0;

}

# Currency\_Converter

Currency\_Converter.h

#pragma once

double Convert\_Currency(int b, int f, double x);

Currency\_Converter.cpp

#include "Currency\_Converter.h"

double Convert\_Currency(int b, int f, double x)

{

// First list currencies in terms of the USD

double USD\_ = 1;

double AUD\_ = 0.70;

double JPY\_ = 0.0081;

double GBP\_ = 1.50;

double EUR\_ = 1.07;

double CAD\_ = 0.75;

double base\_amount;

double foreign\_amount;

double result;

enum currency{USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6};

switch (b)

{

case USD:

base\_amount = USD\_;

break;

case AUD:

base\_amount = AUD\_;

break;

case JPY:

base\_amount = JPY\_;

break;

case GBP:

base\_amount = GBP\_;

break;

case EUR:

base\_amount = EUR\_;

break;

case CAD:

base\_amount = CAD\_;

break;

}

switch (f)

{

case USD:

foreign\_amount = USD\_;

break;

case AUD:

foreign\_amount = AUD\_;

break;

case JPY:

foreign\_amount = JPY\_;

break;

case GBP:

foreign\_amount = GBP\_;

break;

case EUR:

foreign\_amount = EUR\_;

break;

case CAD:

foreign\_amount = CAD\_;

break;

}

return result = (base\_amount/foreign\_amount)\*x;

}

Main.cpp

#include <iostream>

#include "Currency\_Converter.h"

using namespace std;

int main()

{

int base;

int foreign;

double amount;

cout << "Please choose a base currency, where:" << endl;

cout << "{USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6}" << endl;

cin >> base;

cout << "Please choose a foreign currency, where:" << endl;

cout << "{USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6}" << endl;

cin >> foreign;

cout << "Please choose an amount" << endl;

cin >> amount;

cout << "The converted amount is " << Convert\_Currency(base, foreign, amount) << endl;

return 0;

}

# Circle

Circle.h

#pragma once

class Circle

{

public:

Circle();

Circle(double radius);

~Circle();

double GetRadius();

void SetRadius(double radius);

float Circumference();

float Area();

private:

double radius\_;

};

Circle.cpp

#include "Circle.h"

#define \_USE\_MATH\_DEFINES

#include <math.h>

#include <cmath>

Circle::Circle()

: radius\_()

{}

Circle::Circle(double radius)

: radius\_(radius)

{}

Circle::~Circle()

{}

double Circle::GetRadius()

{

return radius\_;

}

void Circle::SetRadius(double radius)

{

radius\_ = radius;

}

float Circle::Circumference()

{

return 2\*M\_PI\*radius\_;

}

float Circle::Area()

{

return M\_PI\*powf(radius\_,2);

}

Main.cpp

#include <iostream>

#include "Circle.h"

using namespace std;

int main()

{

double r;

cout << "Please choose a radius for a circle" << endl;

cin >> r;

Circle c1;

c1.SetRadius(r);

cout << "The circumference of this circle is " << c1.Circumference() << endl;

cout << "The area of this circle is " << c1.Area() << endl;

return 0;

}

# Customer

Customer.h

#pragma once

#include "Account.h"

#include <string>

using std::string;

class Customer

{

public:

// constructors

Customer();

Customer(string name, string address);

// copy constructor

Customer(const Customer& other);

// assignment operator

Customer& operator=(const Customer& other);

~Customer();

string GetName() const;

string GetAddress();

void SetAddress(string address);

// Use the account class

void SetAccount(string account\_number, double balance);

private:

mutable string name\_;

string address\_;

Account account\_;

};

Customer.cpp

#include "Customer.h"

#include <iostream>

using namespace std;

Customer::Customer()

: name\_(""),

address\_("")

{}

Customer::Customer(string name, string address)

: name\_(name),

address\_(address),

account\_("",0.0)

{

cout << "Constructor is run and name, address and account are created" << endl;

}

Customer::Customer(const Customer & other)

:name\_(other.name\_),

address\_(other.address\_)

{}

Customer& Customer::operator=(const Customer & other)

{

if (this != &other) //checking for self assignment

{

name\_ = other.name\_;

address\_ = other.address\_;

}

//return the object on which the function was invoked

return \*this;

}

Customer::~Customer()

{

cout << "Destructor is being run" << endl;

}

string Customer::GetName() const

{

return name\_;

}

string Customer::GetAddress()

{

return address\_;

}

void Customer::SetAddress(string address)

{

address\_ = address;

}

void Customer::SetAccount(string account\_number, double balance)

{

Account acc(account\_number, balance);

account\_ = acc;

}

Account.h

#pragma once

#include <string>

using std::string;

class Account

{

public:

Account();

Account(string account\_number, double balance);

~Account();

string GetAccountNumber();

double GetBalance();

void SetAccount(string account\_number, double balance);

private:

string account\_number\_;

double balance\_;

};

Account.cpp

#include "Account.h"

Account::Account()

:account\_number\_(), balance\_()

{}

Account::Account(string account\_number, double balance)

:account\_number\_(account\_number),

balance\_(balance)

{}

Account::~Account()

{}

string Account::GetAccountNumber()

{

return account\_number\_;

}

double Account::GetBalance()

{

return balance\_;

}

void Account::SetAccount(string account\_number, double balance)

{

account\_number\_ = account\_number;

balance\_ = balance;

}

Main.cpp

#include <iostream>

#include <string>

#include "Customer.h"

using namespace std;

int main()

{

// Normal creation

Customer c1("Ish", "South");

Customer c2("Michael", "North");

string x = c1.GetName();

c2.SetAddress("South Loop");

// Can also use the account class now

c1.SetAccount("11-111-11", 1000.0);

// Const creation

const Customer c3("Zak", "River North");

c3.GetName(); //this can only work if the GetName() function is also made const in the header and cpp files. members of the function can only be

//changed if they are set as mutable

// Copy constructor

Customer c5 = c1;

// Assignment operator

c5 = c2;

// Creation using free store

Customer\* c4 = new Customer("Da", "Hyde Park");

// Accessing free store member

c4->GetName();

c4->SetAddress("55th St");

// Delete creation in free store

delete c4;

return 0;

}

# Currency\_Converter\_Classes

Currency\_Converter.h

#pragma once

class Currency

{

public:

// Constructors

Currency();

Currency(int b, int f, double x);

// Copy constructor

Currency(const Currency& other);

// Assignment operator

Currency& operator=(const Currency& other);

// Destructor

~Currency();

int GetBase();

int GetForeign();

double GetAmount() const;

void SetAmount(double x);

double Convert\_Currency();

private:

int b\_;

int f\_;

mutable double x\_;

};

Currency\_Converter.cpp

#include "Currency\_Converter.h"

Currency::Currency()

: b\_(), f\_(), x\_()

{}

Currency::Currency(int b, int f, double x)

: b\_(b), f\_(f), x\_(x)

{}

Currency::Currency(const Currency & other)

:b\_(other.b\_), f\_(other.f\_), x\_(other.x\_)

{}

Currency & Currency::operator=(const Currency & other)

{

if (this != &other) //checking for self assignment

{

b\_ = other.b\_;

f\_ = other.f\_;

x\_ = other.x\_;

}

// return the object on which the function was invoked

return \*this;

}

Currency::~Currency()

{}

int Currency::GetBase()

{

return b\_;;

}

int Currency::GetForeign()

{

return f\_;

}

double Currency::GetAmount() const

{

return x\_;

}

void Currency::SetAmount(double x)

{

x\_ = x;

}

double Currency::Convert\_Currency()

{

// First list currencies in terms of the USD

double USD\_ = 1;

double AUD\_ = 0.70;

double JPY\_ = 0.0081;

double GBP\_ = 1.50;

double EUR\_ = 1.07;

double CAD\_ = 0.75;

double base\_amount;

double foreign\_amount;

double result;

enum currency { USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6 };

switch (b\_)

{

case USD:

base\_amount = USD\_;

break;

case AUD:

base\_amount = AUD\_;

break;

case JPY:

base\_amount = JPY\_;

break;

case GBP:

base\_amount = GBP\_;

break;

case EUR:

base\_amount = EUR\_;

break;

case CAD:

base\_amount = CAD\_;

break;

}

switch (f\_)

{

case USD:

foreign\_amount = USD\_;

break;

case AUD:

foreign\_amount = AUD\_;

break;

case JPY:

foreign\_amount = JPY\_;

break;

case GBP:

foreign\_amount = GBP\_;

break;

case EUR:

foreign\_amount = EUR\_;

break;

case CAD:

foreign\_amount = CAD\_;

break;

}

return result = (base\_amount / foreign\_amount)\*x\_;

}

Main.cpp

#include <iostream>

#include "Currency\_Converter.h"

using namespace std;

int main()

{

int base;

int foreign;

double amount;

Currency c2(1, 2, 5.3);

Currency c3(4, 5, 1.5);

double a = c2.GetAmount();

c3.SetAmount(3);

// Const creation

const Currency c4(3, 1, 10);

c4.GetAmount(); //this will only work if GetAmount() is also made const in the header and cpp files. members of the function can only be changed

// if they are set to mutable

// Copy constructor

Currency c5 = c2;

// Assignment operator

c5 = c3;

// Creation using free store

Currency\* c6 = new Currency(1, 2, 3.8);

// Accessing free store member

c6->GetBase();

c6->GetForeign();

c6->GetAmount();

c6->SetAmount(7.6);

// Delete creation in free store

delete c6;

cout << "Please choose a base currency, where:" << endl;

cout << "{USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6}" << endl;

cin >> base;

cout << "Please choose a foreign currency, where:" << endl;

cout << "{USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6}" << endl;

cin >> foreign;

cout << "Please choose an amount" << endl;

cin >> amount;

Currency c1(base, foreign, amount);

cout << "The converted amount is " << c1.Convert\_Currency() << endl;

return 0;

}

# Options

NormalUtilities.h

#pragma once

#include <cmath>

class NormalUtil

{

public:

NormalUtil();

NormalUtil(double x);

NormalUtil(double x, double mu, double sigma);

~NormalUtil();

double GetX();

double GetMu();

double GetSigma();

double erf(double x);

double pdf(double x);

double pdf(double x, double mu, double sigma);

double cdf(double x);

double cdf(double x, double mu, double sigma);

private:

double x\_;

double mu\_;

double sigma\_;

};

NormalUtilities.cpp

#include "NormalUtilities.h"

NormalUtil::NormalUtil()

: x\_(), mu\_(), sigma\_()

{}

NormalUtil::NormalUtil(double x)

: x\_(x)

{}

NormalUtil::NormalUtil(double x, double mu, double sigma)

: x\_(x),

mu\_(mu),

sigma\_(sigma)

{}

NormalUtil::~NormalUtil()

{}

double NormalUtil::GetX()

{

return x\_;

}

double NormalUtil::GetMu()

{

return mu\_;

}

double NormalUtil::GetSigma()

{

return sigma\_;

}

double NormalUtil::erf(double x)

{

double y = 1.0 / (1.0 + 0.3275911 \* x);

return 1 - (((((

+1.061405429 \* y

- 1.453152027) \* y

+ 1.421413741) \* y

- 0.284496736) \* y

+ 0.254829592) \* y)

\* exp(-x \* x);

}

double NormalUtil::pdf(double x)

{

//Constants

static const double pi = 3.14159265;

return exp(-1 \* (x) \* (x) / (2)) / (sqrt(2 \* pi));

}

double NormalUtil::pdf(double x, double mu, double sigma)

{

//Constants

static const double pi = 3.14159265;

return exp(-1 \* (x - mu) \* (x - mu) / (2 \* sigma \* sigma)) / (sigma \* sqrt(2 \* pi));

}

double NormalUtil::cdf(double x)

{

return 0.5 \* (1 + erf((x) / (sqrt(2.))));

}

double NormalUtil::cdf(double x, double mu, double sigma)

{

return 0.5 \* (1 + erf((x - mu) / (sigma \* sqrt(2.))));

}

Option.h

#pragma once

#include "NormalUtilities.h"

#include <cmath>

#include <string>

using std::string;

class Option

{

typedef double stock;

typedef double volatility;

typedef double strike;

typedef double maturity;

typedef double time\_lapsed;

typedef double rate;

typedef string put\_or\_call;

typedef double price;

public:

Option();

Option(stock S, volatility sigma, strike K, maturity T, time\_lapsed t, rate r, put\_or\_call type);

~Option();

stock GetS();

volatility GetSigma();

strike GetK();

maturity GetT();

time\_lapsed Gett();

rate GetR();

put\_or\_call GetType();

void SetS(stock S);

void SetSigma(volatility sigma);

void SetK(strike K);

void SetT(maturity T);

void Sett(time\_lapsed t);

void SetR(rate r);

void SetType(put\_or\_call type);

void SetNd1();

void SetNd2();

price P();

double Delta();

double Gamma();

private:

stock S\_;

volatility sigma\_;

strike K\_;

maturity T\_;

time\_lapsed t\_;

rate r\_;

put\_or\_call type\_;

//from normal utilities

double Nd1\_;

double Nd2\_;

};

Option.cpp

#include "Option.h"

#include "NormalUtilities.h"

#include <string>

#include <cmath>

using std::string;

typedef double stock;

typedef double volatility;

typedef double strike;

typedef double maturity;

typedef double time\_lapsed;

typedef double rate;

typedef string put\_or\_call;

typedef double price;

Option::Option()

: sigma\_(), K\_(), T\_(), t\_(), r\_(), type\_()

{}

Option::Option(stock S, volatility sigma, strike K, maturity T, time\_lapsed t, rate r, put\_or\_call type)

:S\_(S),

sigma\_(sigma),

K\_(K),

T\_(T),

t\_(t),

r\_(r),

type\_(type)

{}

Option::~Option()

{}

stock Option::GetS()

{

return S\_;

}

volatility Option::GetSigma()

{

return sigma\_;

}

strike Option::GetK()

{

return K\_;

}

maturity Option::GetT()

{

return T\_;

}

time\_lapsed Option::Gett()

{

return t\_;

}

rate Option::GetR()

{

return r\_;

}

put\_or\_call Option::GetType()

{

return type\_;

}

void Option::SetS(stock S)

{

S\_ = S;

}

void Option::SetSigma(volatility sigma)

{

sigma\_ = sigma;

}

void Option::SetK(strike K)

{

K\_ = K;

}

void Option::SetT(maturity T)

{

T\_ = T;

}

void Option::Sett(time\_lapsed t)

{

t\_ = t;

}

void Option::SetR(rate r)

{

r\_ = r;

}

void Option::SetType(put\_or\_call type)

{

type\_ = type;

}

void Option::SetNd1()

{

double d1 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) + sigma\_\*sqrt(T\_ - t\_)/2;

NormalUtil normu1(d1);

double Nd1 = normu1.cdf(d1);

Nd1\_ = Nd1;

}

void Option::SetNd2()

{

double d2 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) - sigma\_\*sqrt(T\_ - t\_) / 2;

NormalUtil normu2(d2);

double Nd2 = normu2.cdf(d2);

Nd2\_ = Nd2;

}

price Option::P()

{

if (type\_ == "C" || type\_ == "c" || type\_ == "Call" || type\_ == "call")

{

SetNd1();

SetNd2();

return S\_\*Nd1\_ - K\_\*Nd2\_\*exp(-1\*r\_\*(T\_-t\_));

}

else if (type\_ == "P" || type\_ == "p" || type\_ == "Put" || type\_ == "put")

{

SetNd1();

SetNd2();

return K\_\*(1 - Nd2\_)\*exp(-1 \* r\_\*(T\_ - t\_)) - S\_\*(1 - Nd1\_);

}

}

double Option::Delta()

{

if (type\_ == "C" || type\_ == "c" || type\_ == "Call" || type\_ == "call")

{

SetNd1();

return Nd1\_;

}

else if (type\_ == "P" || type\_ == "p" || type\_ == "Put" || type\_ == "put")

{

SetNd1();

return Nd1\_ - 1;

}

}

double Option::Gamma()

{

double d1 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) + sigma\_\*sqrt(T\_ - t\_)/2;

return (1 / sqrt(2 \* 3.14159265359))\*exp(-0.5\*powf(d1,2));

}

Main.cpp

#include <iostream>

#include "Option.h"

#include "NormalUtilities.h"

using namespace std;

int main()

{

Option o1(100.0, 0.3, 100.0, 1.0, 0, 0.05, "Call");

Option o2(120.0, 0.4, 120.0, 2.0, 0, 0.1, "Put");

double call = o1.P();

double call\_delta = o1.Delta();

double call\_gamma = o1.Gamma();

double put = o2.P();

double put\_delta = o2.Delta();

double put\_gamma = o2.Gamma();

cout << "The price of the call is " << call << endl;

cout << "The delta of the call is " << call\_delta << endl;

cout << "The gamma of the call is " << call\_gamma << endl;

cout << "The price of the put is " << put << endl;

cout << "The delta of the put is " << put\_delta << endl;

cout << "The gamma of the put is " << put\_gamma << endl;

return 0;

}

# Bubble Sort

Header.h

#pragma once

int swap(int \*a, int \*b)

{

int t = \*a;

\*a = \*b;

\*b = t;

return 0;

}

Source.cpp

#include<iostream>

#include"Header.h"

using std::cout;

using std::endl;

using std::cin;

int main()

{

int x[5],t[5];

for (int i = 0; i < 5; ++i)

{

cin >> x[i];

}

for (int j = 0; j < 5; j++)

{

for (int i = 0; i < 5; i++)

{

if (x[i] < x[i + 1])

{

swap(&x[i], &x[i + 1]);

}

}

}

for (int i = 0; i < 5; ++i)

{

cout<< x[i]<<endl;

}

}

# Currency Converter Currency Factory Singleton

Currency\_Factory.h

#ifndef \_CURRENCY\_FACTORY\_H

#define \_CURRENCY\_FACTORY\_H

#include "Currency\_Converter.h"

class CurrencyFactory

{

public:

static CurrencyFactory\* Instance();

Currency CreateCurrency();

private:

CurrencyFactory();

static CurrencyFactory\* instance\_;

};

#endif

Currency\_Factory.cpp

#ifndef \_CURRENCY\_FACTORY\_CPP

#define \_CURRENCY\_FACTORY\_CPP

#include "Currency\_Factory.h"

CurrencyFactory::CurrencyFactory()

{}

CurrencyFactory \* CurrencyFactory::Instance()

{

if (!instance\_)

instance\_ = new CurrencyFactory;

return instance\_;

}

Currency CurrencyFactory::CreateCurrency()

{

Currency result;

return result;

}

CurrencyFactory\* CurrencyFactory::instance\_;

#endif

Currency\_Converter.h

#ifndef \_CURRENCY\_CONVERTER\_H

#define \_CURRENCY\_CONVERTER\_H

class Currency

{

public:

// Constructors

Currency();

Currency(int b, int f, double x);

// Copy constructor

Currency(const Currency& other);

// Assignment operator

Currency& operator=(const Currency& other);

// Destructor

~Currency();

int GetBase();

int GetForeign();

double GetAmount() const;

void SetBase(int b);

void SetForeign(int f);

void SetAmount(double x);

double Convert\_Currency();

private:

int b\_;

int f\_;

mutable double x\_;

};

#endif

Currency\_Converter.cpp

#ifndef \_CURRENCY\_CONVERTER\_CPP

#define \_CURRENCY\_CONVERTER\_CPP

#include "Currency\_Converter.h"

Currency::Currency()

: b\_(), f\_(), x\_()

{}

Currency::Currency(int b, int f, double x)

: b\_(b), f\_(f), x\_(x)

{}

Currency::Currency(const Currency & other)

:b\_(other.b\_), f\_(other.f\_), x\_(other.x\_)

{}

Currency & Currency::operator=(const Currency & other)

{

if (this != &other) //checking for self assignment

{

b\_ = other.b\_;

f\_ = other.f\_;

x\_ = other.x\_;

}

// return the object on which the function was invoked

return \*this;

}

Currency::~Currency()

{}

int Currency::GetBase()

{

return b\_;;

}

int Currency::GetForeign()

{

return f\_;

}

double Currency::GetAmount() const

{

return x\_;

}

void Currency::SetBase(int b)

{

b\_ = b;

}

void Currency::SetForeign(int f)

{

f\_ = f;

}

void Currency::SetAmount(double x)

{

x\_ = x;

}

double Currency::Convert\_Currency()

{

// First list currencies in terms of the USD

double USD\_ = 1;

double AUD\_ = 0.70;

double JPY\_ = 0.0081;

double GBP\_ = 1.50;

double EUR\_ = 1.07;

double CAD\_ = 0.75;

double base\_amount;

double foreign\_amount;

double result;

enum currency { USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6 };

switch (b\_)

{

case USD:

base\_amount = USD\_;

break;

case AUD:

base\_amount = AUD\_;

break;

case JPY:

base\_amount = JPY\_;

break;

case GBP:

base\_amount = GBP\_;

break;

case EUR:

base\_amount = EUR\_;

break;

case CAD:

base\_amount = CAD\_;

break;

}

switch (f\_)

{

case USD:

foreign\_amount = USD\_;

break;

case AUD:

foreign\_amount = AUD\_;

break;

case JPY:

foreign\_amount = JPY\_;

break;

case GBP:

foreign\_amount = GBP\_;

break;

case EUR:

foreign\_amount = EUR\_;

break;

case CAD:

foreign\_amount = CAD\_;

break;

}

return result = (base\_amount / foreign\_amount)\*x\_;

}

#endif

Main.cpp

#include <iostream>

#include "Currency\_Factory.h"

#include <string>

using namespace std;

int main()

{

Currency currency =

CurrencyFactory::Instance()->CreateCurrency(); //creating Currency object on free store and returning pointer

cout << "Please choose a base currency, where:" << endl;

cout << "{USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6}" << endl;

// for this assignment, we are asked to show how to convert from USD to foreign currency, therefore please choose 1 here

int base;

cin >> base;

currency.SetBase(base); // set the base currency in terms of the object created above

cout << "Please choose a foreign currency, where:" << endl;

cout << "{USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6}" << endl;

// please choose any currency other than USD, therefore please choose between 2 - 5 inclusive here

int foreign;

cin >> foreign;

currency.SetForeign(foreign); // set the foreign currency in terms of the object created above

cout << "Please choose an amount" << endl;

double amount;

cin >> amount;

currency.SetAmount(amount); // set the amount of the base currency to be converted to the foreign currency

// in terms of the object created above

cout << "The converted amount is: " << currency.Convert\_Currency() << endl;

return 0;

}

# Option with Option Factory Singleton

Normal\_Utilities.h

#pragma once

#include <cmath>

class NormalUtil

{

public:

// constructors

NormalUtil();

NormalUtil(double x); // x is the random variable

NormalUtil(double x, double mu, double sigma);

// desctructor

~NormalUtil();

// get/set

double GetX();

double GetMu();

double GetSigma();

// functions

double erf(double x);

double pdf(double x);

double pdf(double x, double mu, double sigma);

double cdf(double x);

double cdf(double x, double mu, double sigma);

private:

double x\_;

double mu\_;

double sigma\_;

};

NormalUtilities.cpp

#include "NormalUtilities.h"

// constructors

NormalUtil::NormalUtil()

: x\_(), mu\_(), sigma\_()

{}

NormalUtil::NormalUtil(double x)

: x\_(x)

{}

NormalUtil::NormalUtil(double x, double mu, double sigma)

: x\_(x),

mu\_(mu),

sigma\_(sigma)

{}

// destructor

NormalUtil::~NormalUtil()

{}

// get/set

double NormalUtil::GetX()

{

return x\_;

}

double NormalUtil::GetMu()

{

return mu\_;

}

double NormalUtil::GetSigma()

{

return sigma\_;

}

// functions

double NormalUtil::erf(double x)

{

double y = 1.0 / (1.0 + 0.3275911 \* x);

return 1 - (((((

+1.061405429 \* y

- 1.453152027) \* y

+ 1.421413741) \* y

- 0.284496736) \* y

+ 0.254829592) \* y)

\* exp(-x \* x);

}

double NormalUtil::pdf(double x)

{

//Constants

static const double pi = 3.14159265;

return exp(-1 \* (x) \* (x) / (2)) / (sqrt(2 \* pi));

}

double NormalUtil::pdf(double x, double mu, double sigma)

{

//Constants

static const double pi = 3.14159265;

return exp(-1 \* (x - mu) \* (x - mu) / (2 \* sigma \* sigma)) / (sigma \* sqrt(2 \* pi));

}

double NormalUtil::cdf(double x)

{

return 0.5 \* (1 + erf((x) / (sqrt(2.))));

}

double NormalUtil::cdf(double x, double mu, double sigma)

{

return 0.5 \* (1 + erf((x - mu) / (sigma \* sqrt(2.))));

}

Option.h

#ifndef \_OPTION\_H

#define \_OPTION\_H

#include "NormalUtilities.h" // used to calculate Nd1 and Nd2

#include <cmath>

#include <string>

using std::string;

class Option

{

// use typedef to make the arguments more readable

typedef double stock;

typedef double volatility;

typedef double strike;

typedef double maturity;

typedef double time\_lapsed;

typedef double rate;

typedef string put\_or\_call;

typedef double price;

public:

// constructors

Option();

Option(stock S, volatility sigma, strike K, maturity T, time\_lapsed t, rate r, put\_or\_call type);

// destructor

~Option();

// get/set

stock GetS();

volatility GetSigma();

strike GetK();

maturity GetT();

time\_lapsed Gett();

rate GetR();

put\_or\_call GetType();

void SetS(stock S);

void SetSigma(volatility sigma);

void SetK(strike K);

void SetT(maturity T);

void Sett(time\_lapsed t);

void SetR(rate r);

void SetType(put\_or\_call type);

// special set of Nd1 and Nd2, in option price and greeks formulae

void SetNd1();

void SetNd2();

// functions

price P();

double Delta();

double Gamma();

private:

stock S\_;

volatility sigma\_;

strike K\_;

maturity T\_;

time\_lapsed t\_;

rate r\_;

put\_or\_call type\_;

// N() is calculated using normal utilities

double Nd1\_;

double Nd2\_;

};

#endif

Option.cpp

#include "Option.h"

// use typedef to make the arguments more readable

typedef double stock;

typedef double volatility;

typedef double strike;

typedef double maturity;

typedef double time\_lapsed;

typedef double rate;

typedef string put\_or\_call;

typedef double price;

// constructors

Option::Option()

: sigma\_(), K\_(), T\_(), t\_(), r\_(), type\_()

{}

Option::Option(stock S, volatility sigma, strike K, maturity T, time\_lapsed t, rate r, put\_or\_call type)

:S\_(S),

sigma\_(sigma),

K\_(K),

T\_(T),

t\_(t),

r\_(r),

type\_(type)

{}

// destructor

Option::~Option()

{}

// get/set

stock Option::GetS()

{

return S\_;

}

volatility Option::GetSigma()

{

return sigma\_;

}

strike Option::GetK()

{

return K\_;

}

maturity Option::GetT()

{

return T\_;

}

time\_lapsed Option::Gett()

{

return t\_;

}

rate Option::GetR()

{

return r\_;

}

put\_or\_call Option::GetType()

{

return type\_;

}

void Option::SetS(stock S)

{

S\_ = S;

}

void Option::SetSigma(volatility sigma)

{

sigma\_ = sigma;

}

void Option::SetK(strike K)

{

K\_ = K;

}

void Option::SetT(maturity T)

{

T\_ = T;

}

void Option::Sett(time\_lapsed t)

{

t\_ = t;

}

void Option::SetR(rate r)

{

r\_ = r;

}

void Option::SetType(put\_or\_call type)

{

type\_ = type;

}

// special set Nd1 and Nd2 using NormalUtil class from NormalUtilities file

void Option::SetNd1()

{

double d1 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) + sigma\_\*sqrt(T\_ - t\_)/2;

NormalUtil normu1(d1);

double Nd1 = normu1.cdf(d1);

Nd1\_ = Nd1;

}

void Option::SetNd2()

{

double d2 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) - sigma\_\*sqrt(T\_ - t\_) / 2;

NormalUtil normu2(d2);

double Nd2 = normu2.cdf(d2);

Nd2\_ = Nd2;

}

// functions

price Option::P()

{

if (type\_ == "C" || type\_ == "c" || type\_ == "Call" || type\_ == "call")

{

SetNd1();

SetNd2();

return S\_\*Nd1\_ - K\_\*Nd2\_\*exp(-1\*r\_\*(T\_-t\_));

}

else if (type\_ == "P" || type\_ == "p" || type\_ == "Put" || type\_ == "put")

{

SetNd1();

SetNd2();

return K\_\*(1 - Nd2\_)\*exp(-1 \* r\_\*(T\_ - t\_)) - S\_\*(1 - Nd1\_);

}

}

double Option::Delta()

{

if (type\_ == "C" || type\_ == "c" || type\_ == "Call" || type\_ == "call")

{

SetNd1();

return Nd1\_;

}

else if (type\_ == "P" || type\_ == "p" || type\_ == "Put" || type\_ == "put")

{

SetNd1();

return Nd1\_ - 1;

}

}

double Option::Gamma()

{

double d1 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) + sigma\_\*sqrt(T\_ - t\_)/2;

double N\_dash\_d1 = (1 / sqrt(2 \* 3.14159265359))\*exp(-0.5\*powf(d1, 2));

return N\_dash\_d1 / (S\_\*sigma\_\*sqrt(T\_ - t\_));

}

Option\_Factory.h

#ifndef \_OPTION\_FACTORY\_H

#define \_OPTION\_FACTORY\_H

#include "Option.h"

class OptionFactory

{

public:

static OptionFactory\* Instance(); //create single instance of the OptionFactory object via the Instance() static member function

Option CreateOption();

private:

OptionFactory(); //Change access level of constructor to private

static OptionFactory\* instance\_;

};

#endif

Option\_Factory.cpp

#ifndef \_OPTION\_FACTORY\_CPP

#define \_OPTION\_FACTORY\_CPP

#include "Option\_Factory.h"

OptionFactory \* OptionFactory::Instance()

{

if (!instance\_) //if the instance\_ points to an OptionFactory object we return it, otherwise create using OptionFactory, then return

instance\_ = new OptionFactory;

return instance\_;

}

Option OptionFactory::CreateOption()

{

Option result;

return result;

}

OptionFactory::OptionFactory()

{}

OptionFactory \* OptionFactory::instance\_;

#endif

Main.cpp

// Michael Beven

// Option Pricing and Greeks Solution

// Consists of two classes - "NormalUtilities" has normal cdf, erf, pdf calculators. "Option" has functions for option price and greeks.

// 20151112

#include <iostream>

#include "Option\_Factory.h"

#include "NormalUtilities.h"

using namespace std;

int main()

{

Option o1 =

OptionFactory::Instance()->CreateOption(); // creating Option object on free store and returning pointer

//Set option parameters

o1.SetS(100.0);

o1.SetSigma(0.3);

o1.SetK(100.0);

o1.SetT(1.0);

o1.Sett(0);

o1.SetR(0.05);

o1.SetType("Call");

Option o2 =

OptionFactory::Instance()->CreateOption(); // creating Option object on free store and returning pointer

//Set option parameters

o2.SetS(120.0);

o2.SetSigma(0.4);

o2.SetK(120.0);

o2.SetT(2.0);

o2.Sett(0);

o2.SetR(0.1);

o2.SetType("Put");

// calculate call price, delta and gamma

double call = o1.P();

double call\_delta = o1.Delta();

double call\_gamma = o1.Gamma();

// calculate put price, delta and gamma

double put = o2.P();

double put\_delta = o2.Delta();

double put\_gamma = o2.Gamma();

cout << "The price of the call is " << call << endl;

cout << "The delta of the call is " << call\_delta << endl;

cout << "The gamma of the call is " << call\_gamma << endl;

cout << "The price of the put is " << put << endl;

cout << "The delta of the put is " << put\_delta << endl;

cout << "The gamma of the put is " << put\_gamma << endl;

return 0;

}

# Bank Accounts – Inheritance

Account.h

#pragma once

#include <string>

using std::string;

class Account

{

public:

virtual ~Account();

string GetAccountNumber();

double GetBalance();

virtual void ApplyInterest() = 0;

virtual void ApplyBankingFees() = 0;

virtual Account\* Clone() const = 0;

void UpdateAccount(Account\* acc);

protected:

Account(string accountNumber, double accountBalance, double interestRate, double bankingFee);

double accountBalance\_;

double interestRate\_;

double bankingFee\_;

private:

string accountNumber\_;

};

Account.cpp

#include "Account.h"

void Account::UpdateAccount(Account \* acc)

{

acc->ApplyBankingFees();

acc->ApplyInterest();

}

Account::Account(string accountNumber, double accountBalance, double interestRate, double bankingFee)

:accountNumber\_(accountNumber),

accountBalance\_(accountBalance),

interestRate\_(interestRate),

bankingFee\_(bankingFee)

{}

Account::~Account()

{}

string Account::GetAccountNumber()

{

return accountNumber\_;

}

double Account::GetBalance()

{

return accountBalance\_;

}

Customer.h

#pragma once

#include "Account.h"

#include <string>

using std::string;

class Customer

{

public:

Customer(string name, string address);

~Customer();

void Customer::SetAccount(const Account& account);

private:

string name\_;

string address\_;

Account\* bankAccount\_;

};

Customer.cpp

#include "Customer.h"

Customer::Customer(string name, string address)

:name\_(name),

address\_(address)

{}

Customer::~Customer()

{}

void Customer::SetAccount(const Account & account)

{

delete bankAccount\_;

bankAccount\_ = account.Clone();

}

Checking.h

#pragma once

#include "Account.h"

#include <string>

using std::string;

class Checking : public Account

{

public:

Checking(string accountNumber, double accountBalance,

double interestRate, double bankingFee);

~Checking();

void ApplyInterest();

void ApplyBankingFees();

Account\* Clone() const;

private:

double interestRate\_;

double bankingFee\_;

};

Checking.cpp

#include "Checking.h"

Checking::Checking(string accountNumber, double accountBalance, double interestRate, double bankingFee)

:Account(accountNumber,accountBalance,interestRate,bankingFee),

interestRate\_(interestRate),

bankingFee\_(bankingFee)

{}

Checking::~Checking()

{}

void Checking::ApplyInterest()

{

accountBalance\_ = accountBalance\_;

}

void Checking::ApplyBankingFees()

{

accountBalance\_ = accountBalance\_;

}

Account \* Checking::Clone() const

{

return new Checking(\*this);

}

CheckingWithInterest.h

#pragma once

#include "Account.h"

#include <string>

using std::string;

class CheckingWithInterest : public Account

{

public:

CheckingWithInterest(string accountNumber, double accountBalance,

double interestRate, double bankingFee);

~CheckingWithInterest();

void ApplyInterest();

void ApplyBankingFees();

Account\* Clone() const;

private:

double interestRate\_;

double bankingFee\_;

};

CheckingWithInterest.cpp

#include "CheckingWithInterest.h"

CheckingWithInterest::CheckingWithInterest(string accountNumber, double accountBalance, double interestRate, double bankingFee)

:Account(accountNumber,accountBalance,interestRate,bankingFee),

interestRate\_(interestRate),

bankingFee\_(bankingFee)

{}

CheckingWithInterest::~CheckingWithInterest()

{}

void CheckingWithInterest::ApplyInterest()

{

if (accountBalance\_ > 1000.0)

{

accountBalance\_ +=

accountBalance\_ \* (30.0 / 360.0)\*interestRate\_;

}

}

void CheckingWithInterest::ApplyBankingFees()

{

accountBalance\_ -= bankingFee\_;

}

Account \* CheckingWithInterest::Clone() const

{

return new CheckingWithInterest(\*this);

}

Savings.h

#pragma once

#include "Account.h"

#include <string>

using std::string;

class Savings : public Account

{

public:

Savings(string accountNumber, double balance,

double interestRate, double bankingFee);

~Savings();

void ApplyInterest();

void ApplyBankingFees();

Account\* Clone() const;

private:

double interestRate\_;

double bankingFee\_;

};

Savings.cpp

#include "Savings.h"

Savings::Savings(string accountNumber, double accountBalance, double interestRate, double bankingFee)

:Account(accountNumber,accountBalance,interestRate,bankingFee),

interestRate\_(interestRate),

bankingFee\_(bankingFee)

{}

Savings::~Savings()

{}

void Savings::ApplyInterest()

{

accountBalance\_ +=

accountBalance\_ \* (30.0 / 360.0)\*interestRate\_;

}

void Savings::ApplyBankingFees()

{

if (accountBalance\_ < 1000.0)

{

accountBalance\_ -= bankingFee\_;

}

}

Account \* Savings::Clone() const

{

return new Savings(\*this);

}

Main.cpp

#include "Account.h"

#include "Checking.h"

#include "CheckingWithInterest.h"

#include "Savings.h"

#include "Customer.h"

#include <iostream>

#include <string>

using namespace std;

int main()

{

Customer c1("John", "Hyde Park");

Savings acc1("Savings-1111", 1500.0, 0.1, 5.0);

c1.SetAccount(acc1);

Checking acc2("Checking-1111", 1500.0, 0.0, 10.0);

c1.SetAccount(acc2);

acc1.ApplyInterest();

cout << "Test: " << acc1.GetBalance() << endl;

}

# Bank Accounts – Inheritance – 2 Levels

Account.h

#pragma once

#include <string>

using std::string;

class Account

{

public:

virtual ~Account();

string GetAccountNumber();

double GetBalance();

virtual void ApplyInterest() = 0;

virtual void ApplyBankingFees() = 0;

virtual Account\* Clone() const = 0;

void UpdateAccount(Account\* acc);

protected:

Account(string accountNumber, double accountBalance, double interestRate, double bankingFee);

double accountBalance\_;

double interestRate\_;

double bankingFee\_;

private:

string accountNumber\_;

};

Account.cpp

#include "Account.h"

void Account::UpdateAccount(Account \* acc)

{

acc->ApplyBankingFees();

acc->ApplyInterest();

}

Account::Account(string accountNumber, double accountBalance, double interestRate, double bankingFee)

:accountNumber\_(accountNumber),

accountBalance\_(accountBalance),

interestRate\_(interestRate),

bankingFee\_(bankingFee)

{}

Account::~Account()

{}

string Account::GetAccountNumber()

{

return accountNumber\_;

}

double Account::GetBalance()

{

return accountBalance\_;

}

Customer.h

#pragma once

#include "Account.h"

#include <string>

using std::string;

class Customer

{

public:

Customer(string name, string address);

~Customer();

void Customer::SetAccount(const Account& account);

private:

string name\_;

string address\_;

Account\* bankAccount\_;

};

Customer.cpp

#include "Customer.h"

Customer::Customer(string name, string address)

:name\_(name),

address\_(address)

{}

Customer::~Customer()

{}

void Customer::SetAccount(const Account & account)

{

delete bankAccount\_;

bankAccount\_ = account.Clone();

}

Checking.h

#pragma once

#include "Account.h"

#include <string>

using std::string;

class Checking : public Account

{

public:

Checking(string accountNumber, double accountBalance,

double interestRate, double bankingFee);

~Checking();

void ApplyInterest();

void ApplyBankingFees();

Account\* Clone() const;

private:

double interestRate\_;

double bankingFee\_;

};

Checking.cpp

#include "Checking.h"

Checking::Checking(string accountNumber, double accountBalance, double interestRate, double bankingFee)

:Account(accountNumber,accountBalance,interestRate,bankingFee),

interestRate\_(interestRate),

bankingFee\_(bankingFee)

{}

Checking::~Checking()

{}

void Checking::ApplyInterest()

{

accountBalance\_ = accountBalance\_;

}

void Checking::ApplyBankingFees()

{

accountBalance\_ = accountBalance\_;

}

Account \* Checking::Clone() const

{

return new Checking(\*this);

}

InterestBearingAccounts.h

#pragma once

#include "Account.h"

#include <string>

using std::string;

class InterestBearingAccounts : public Account

{

public:

InterestBearingAccounts(string accountNumber, double balance,

double interestRate, double bankingFee);

~InterestBearingAccounts();

virtual void ApplyInterest() = 0;

virtual void ApplyBankingFees() = 0;

};

InterestBearingAccounts.cpp

#include "InterestBearingAccounts.h"

InterestBearingAccounts::InterestBearingAccounts(string accountNumber, double accountBalance, double interestRate, double bankingFee)

:Account(accountNumber, accountBalance, interestRate, bankingFee)

{}

InterestBearingAccounts::~InterestBearingAccounts()

{}

Savings.h

#pragma once

#include "InterestBearingAccounts.h"

#include <string>

using std::string;

class Savings : public InterestBearingAccounts

{

public:

Savings(string accountNumber, double balance,

double interestRate, double bankingFee);

~Savings();

void ApplyInterest();

void ApplyBankingFees();

Account\* Clone() const;

private:

double interestRate\_;

double bankingFee\_;

};

Savings.cpp

#include "Savings.h"

Savings::Savings(string accountNumber, double accountBalance, double interestRate, double bankingFee)

:InterestBearingAccounts(accountNumber,accountBalance,interestRate,bankingFee),

interestRate\_(interestRate),

bankingFee\_(bankingFee)

{}

Savings::~Savings()

{}

void Savings::ApplyInterest()

{

accountBalance\_ +=

accountBalance\_ \* (30.0 / 360.0)\*interestRate\_;

}

void Savings::ApplyBankingFees()

{

if (accountBalance\_ < 1000.0)

{

accountBalance\_ -= bankingFee\_;

}

}

Account \* Savings::Clone() const

{

return new Savings(\*this);

}

CheckingWithInterest.h

#pragma once

#include "InterestBearingAccounts.h"

#include <string>

using std::string;

class CheckingWithInterest : public InterestBearingAccounts

{

public:

CheckingWithInterest(string accountNumber, double accountBalance,

double interestRate, double bankingFee);

~CheckingWithInterest();

void ApplyInterest();

void ApplyBankingFees();

Account\* Clone() const;

private:

double interestRate\_;

double bankingFee\_;

};

CheckingWithInterest.cpp

#include "CheckingWithInterest.h"

CheckingWithInterest::CheckingWithInterest(string accountNumber, double accountBalance, double interestRate, double bankingFee)

:InterestBearingAccounts(accountNumber,accountBalance,interestRate,bankingFee),

interestRate\_(interestRate),

bankingFee\_(bankingFee)

{}

CheckingWithInterest::~CheckingWithInterest()

{}

void CheckingWithInterest::ApplyInterest()

{

if (accountBalance\_ > 1000.0)

{

accountBalance\_ +=

accountBalance\_ \* (30.0 / 360.0)\*interestRate\_;

}

}

void CheckingWithInterest::ApplyBankingFees()

{

accountBalance\_ -= bankingFee\_;

}

Account \* CheckingWithInterest::Clone() const

{

return new CheckingWithInterest(\*this);

}

Main.cpp

#include "Account.h"

#include "Checking.h"

#include "CheckingWithInterest.h"

#include "Savings.h"

#include "Customer.h"

#include <iostream>

#include <string>

using namespace std;

int main()

{

Customer c1("John", "Hyde Park");

Savings acc1("Savings-1111", 1500.0, 0.1, 5.0);

c1.SetAccount(acc1);

Checking acc2("Checking-1111", 1500.0, 0.0, 10.0);

c1.SetAccount(acc2);

acc1.ApplyInterest();

cout << "Test: " << acc1.GetBalance() << endl;

}

# Option with Inheritance

NormalUtilities.h

#pragma once

#include <cmath>

class NormalUtil

{

public:

// constructors

NormalUtil();

NormalUtil(double x); // x is the random variable

NormalUtil(double x, double mu, double sigma);

// desctructor

~NormalUtil();

// get/set

double GetX();

double GetMu();

double GetSigma();

// functions

double erf(double x);

double pdf(double x);

double pdf(double x, double mu, double sigma);

double cdf(double x);

double cdf(double x, double mu, double sigma);

private:

double x\_;

double mu\_;

double sigma\_;

};

NormalUtilities.cpp

#include "NormalUtilities.h"

// constructors

NormalUtil::NormalUtil()

: x\_(), mu\_(), sigma\_()

{}

NormalUtil::NormalUtil(double x)

: x\_(x)

{}

NormalUtil::NormalUtil(double x, double mu, double sigma)

: x\_(x),

mu\_(mu),

sigma\_(sigma)

{}

// destructor

NormalUtil::~NormalUtil()

{}

// get/set

double NormalUtil::GetX()

{

return x\_;

}

double NormalUtil::GetMu()

{

return mu\_;

}

double NormalUtil::GetSigma()

{

return sigma\_;

}

// functions

double NormalUtil::erf(double x)

{

double y = 1.0 / (1.0 + 0.3275911 \* x);

return 1 - (((((

+1.061405429 \* y

- 1.453152027) \* y

+ 1.421413741) \* y

- 0.284496736) \* y

+ 0.254829592) \* y)

\* exp(-x \* x);

}

double NormalUtil::pdf(double x)

{

//Constants

static const double pi = 3.14159265;

return exp(-1 \* (x) \* (x) / (2)) / (sqrt(2 \* pi));

}

double NormalUtil::pdf(double x, double mu, double sigma)

{

//Constants

static const double pi = 3.14159265;

return exp(-1 \* (x - mu) \* (x - mu) / (2 \* sigma \* sigma)) / (sigma \* sqrt(2 \* pi));

}

double NormalUtil::cdf(double x)

{

return 0.5 \* (1 + erf((x) / (sqrt(2.))));

}

double NormalUtil::cdf(double x, double mu, double sigma)

{

return 0.5 \* (1 + erf((x - mu) / (sigma \* sqrt(2.))));

}

Option.h

#pragma once

#include "NormalUtilities.h" // used to calculate Nd1 and Nd2

#include <cmath>

#include <string>

using std::string;

class Option

{

// use typedef to make the arguments more readable

typedef double stock;

typedef double volatility;

typedef double strike;

typedef double maturity;

typedef double time\_lapsed;

typedef double rate;

typedef string put\_or\_call;

typedef double price;

public:

// destructor

virtual ~Option(); // virtual allows derived classes to re-implement member

// get/set

virtual stock GetS();

virtual volatility GetSigma();

virtual strike GetK();

virtual maturity GetT();

virtual time\_lapsed Gett();

virtual rate GetR();

virtual put\_or\_call GetType();

virtual void SetS(stock S);

virtual void SetSigma(volatility sigma);

virtual void SetK(strike K);

virtual void SetT(maturity T);

virtual void Sett(time\_lapsed t);

virtual void SetR(rate r);

virtual void SetType(put\_or\_call type);

// special set of Nd1 and Nd2, in option price and greeks formulae

virtual void SetNd1() = 0;

virtual void SetNd2() = 0;

// functions

virtual price P() = 0; // makes pure virtual, so that different classes can implement different versions of this member

virtual double Delta() = 0;

virtual double Gamma() = 0;

protected:

// constructors

Option();

Option(stock S, volatility sigma, strike K, maturity T, time\_lapsed t, rate r, put\_or\_call type);

stock S\_;

volatility sigma\_;

strike K\_;

maturity T\_;

time\_lapsed t\_;

rate r\_;

put\_or\_call type\_;

};

Option.cpp

#include "Option.h"

// use typedef to make the arguments more readable

typedef double stock;

typedef double volatility;

typedef double strike;

typedef double maturity;

typedef double time\_lapsed;

typedef double rate;

typedef string put\_or\_call;

typedef double price;

// constructors

Option::Option()

: sigma\_(), K\_(), T\_(), t\_(), r\_(), type\_()

{}

Option::Option(stock S, volatility sigma, strike K, maturity T, time\_lapsed t, rate r, put\_or\_call type)

:S\_(S),

sigma\_(sigma),

K\_(K),

T\_(T),

t\_(t),

r\_(r),

type\_(type)

{}

// destructor

Option::~Option()

{}

// get/set

stock Option::GetS()

{

return S\_;

}

volatility Option::GetSigma()

{

return sigma\_;

}

strike Option::GetK()

{

return K\_;

}

maturity Option::GetT()

{

return T\_;

}

time\_lapsed Option::Gett()

{

return t\_;

}

rate Option::GetR()

{

return r\_;

}

put\_or\_call Option::GetType()

{

return type\_;

}

void Option::SetS(stock S)

{

S\_ = S;

}

void Option::SetSigma(volatility sigma)

{

sigma\_ = sigma;

}

void Option::SetK(strike K)

{

K\_ = K;

}

void Option::SetT(maturity T)

{

T\_ = T;

}

void Option::Sett(time\_lapsed t)

{

t\_ = t;

}

void Option::SetR(rate r)

{

r\_ = r;

}

void Option::SetType(put\_or\_call type)

{

type\_ = type;

}

Call.h

#pragma once

#include "Option.h"

#include "NormalUtilities.h"

#include <cmath>

#include <string>

using std::string;

class Call : public Option

{

// use typedef to make the arguments more readable

typedef double stock;

typedef double volatility;

typedef double strike;

typedef double maturity;

typedef double time\_lapsed;

typedef double rate;

typedef string put\_or\_call;

typedef double price;

public:

Call(stock S, volatility sigma, strike K, maturity T, time\_lapsed t, rate r, put\_or\_call type);

~Call();

stock GetS();

volatility GetSigma();

strike GetK();

maturity GetT();

time\_lapsed Gett();

rate GetR();

put\_or\_call GetType();

void SetS(stock S);

void SetSigma(volatility sigma);

void SetK(strike K);

void SetT(maturity T);

void Sett(time\_lapsed t);

void SetR(rate r);

void SetType(put\_or\_call type);

void SetNd1();

void SetNd2();

double P();

double Delta();

double Gamma();

private:

// N() is calculated using normal utilities

double Nd1\_;

double Nd2\_;

};

Call.cpp

#pragma once

#include "Call.h"

// use typedef to make the arguments more readable

typedef double stock;

typedef double volatility;

typedef double strike;

typedef double maturity;

typedef double time\_lapsed;

typedef double rate;

typedef string put\_or\_call;

typedef double price;

Call::Call(stock S, volatility sigma, strike K, maturity T, time\_lapsed t, rate r, put\_or\_call type)

:Option(S, sigma, K, T, t, r, type)

{}

Call::~Call()

{}

stock Call::GetS()

{

return S\_;

}

volatility Call::GetSigma()

{

return sigma\_;

}

strike Call::GetK()

{

return K\_;

}

maturity Call::GetT()

{

return T\_;

}

time\_lapsed Call::Gett()

{

return t\_;

}

rate Call::GetR()

{

return r\_;

}

put\_or\_call Call::GetType()

{

return type\_;

}

void Call::SetS(stock S)

{

S\_ = S;

}

void Call::SetSigma(volatility sigma)

{

sigma\_ = sigma;

}

void Call::SetK(strike K)

{

K\_ = K;

}

void Call::SetT(maturity T)

{

T\_ = T;

}

void Call::Sett(time\_lapsed t)

{

t\_ = t;

}

void Call::SetR(rate r)

{

r\_ = r;

}

void Call::SetType(put\_or\_call type)

{

type\_ = type;

}

void Call::SetNd1()

{

double d1 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) + sigma\_\*sqrt(T\_ - t\_) / 2;

NormalUtil normu1(d1);

double Nd1 = normu1.cdf(d1);

Nd1\_ = Nd1;

}

void Call::SetNd2()

{

double d2 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) - sigma\_\*sqrt(T\_ - t\_) / 2;

NormalUtil normu2(d2);

double Nd2 = normu2.cdf(d2);

Nd2\_ = Nd2;

}

double Call::P()

{

SetNd1();

SetNd2();

return S\_\*Nd1\_ - K\_\*Nd2\_\*exp(-1 \* r\_\*(T\_ - t\_));

}

double Call::Delta()

{

SetNd1();

return Nd1\_;

}

double Call::Gamma()

{

double d1 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) + sigma\_\*sqrt(T\_ - t\_) / 2;

double N\_dash\_d1 = (1 / sqrt(2 \* 3.14159265359))\*exp(-0.5\*powf(d1, 2));

return N\_dash\_d1 / (S\_\*sigma\_\*sqrt(T\_ - t\_));

}

Put.h

#pragma once

#include "Option.h"

#include "NormalUtilities.h"

#include <cmath>

#include <string>

using std::string;

class Put : public Option

{

// use typedef to make the arguments more readable

typedef double stock;

typedef double volatility;

typedef double strike;

typedef double maturity;

typedef double time\_lapsed;

typedef double rate;

typedef string put\_or\_call;

typedef double price;

public:

Put(stock S, volatility sigma, strike K, maturity T, time\_lapsed t, rate r, put\_or\_call type);

~Put();

stock GetS();

volatility GetSigma();

strike GetK();

maturity GetT();

time\_lapsed Gett();

rate GetR();

put\_or\_call GetType();

void SetS(stock S);

void SetSigma(volatility sigma);

void SetK(strike K);

void SetT(maturity T);

void Sett(time\_lapsed t);

void SetR(rate r);

void SetType(put\_or\_call type);

void SetNd1();

void SetNd2();

double P();

double Delta();

double Gamma();

private:

// N() is calculated using normal utilities

double Nd1\_;

double Nd2\_;

};

Put.cpp

#pragma once

#include "Put.h"

// use typedef to make the arguments more readable

typedef double stock;

typedef double volatility;

typedef double strike;

typedef double maturity;

typedef double time\_lapsed;

typedef double rate;

typedef string put\_or\_call;

typedef double price;

Put::Put(stock S, volatility sigma, strike K, maturity T, time\_lapsed t, rate r, put\_or\_call type)

:Option(S, sigma, K, T, t, r, type)

{}

stock Put::GetS()

{

return S\_;

}

volatility Put::GetSigma()

{

return sigma\_;

}

strike Put::GetK()

{

return K\_;

}

maturity Put::GetT()

{

return T\_;

}

time\_lapsed Put::Gett()

{

return t\_;

}

rate Put::GetR()

{

return r\_;

}

put\_or\_call Put::GetType()

{

return type\_;

}

void Put::SetS(stock S)

{

S\_ = S;

}

void Put::SetSigma(volatility sigma)

{

sigma\_ = sigma;

}

void Put::SetK(strike K)

{

K\_ = K;

}

void Put::SetT(maturity T)

{

T\_ = T;

}

void Put::Sett(time\_lapsed t)

{

t\_ = t;

}

void Put::SetR(rate r)

{

r\_ = r;

}

void Put::SetType(put\_or\_call type)

{

type\_ = type;

}

Put::~Put()

{}

double Put::P()

{

SetNd1();

SetNd2();

return K\_\*(1 - Nd2\_)\*exp(-1 \* r\_\*(T\_ - t\_)) - S\_\*(1 - Nd1\_);

}

double Put::Delta()

{

SetNd1();

return Nd1\_ - 1;

}

double Put::Gamma()

{

double d1 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) + sigma\_\*sqrt(T\_ - t\_) / 2;

double N\_dash\_d1 = (1 / sqrt(2 \* 3.14159265359))\*exp(-0.5\*powf(d1, 2));

return N\_dash\_d1 / (S\_\*sigma\_\*sqrt(T\_ - t\_));

}

void Put::SetNd1()

{

double d1 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) + sigma\_\*sqrt(T\_ - t\_) / 2;

NormalUtil normu1(d1);

double Nd1 = normu1.cdf(d1);

Nd1\_ = Nd1;

}

void Put::SetNd2()

{

double d2 = log(S\_\*exp(r\_\*(T\_ - t\_)) / K\_) / (sigma\_\*sqrt(T\_ - t\_)) - sigma\_\*sqrt(T\_ - t\_) / 2;

NormalUtil normu2(d2);

double Nd2 = normu2.cdf(d2);

Nd2\_ = Nd2;

}

Main.cpp

// Michael Beven

// Option Pricing and Greeks Solution

// Consists of two classes - "NormalUtilities" has normal cdf, erf, pdf calculators. "Option" has functions for option price and greeks.

// 20151112

#include <iostream> //input/output

using namespace std;

#include "Option.h" // Base class

#include "Call.h" // Derived class

#include "Put.h" // Derived class

#include "NormalUtilities.h" //normal pdf and cdf tools

int main()

{

// create the options using the Option class

Call o1(100.0, 0.3, 100.0, 1.0, 0, 0.05, "Call");

Put o2(120.0, 0.4, 120.0, 2.0, 0, 0.1, "Put");

// calculate call price, delta and gamma

double call\_price = o1.P();

double call\_delta = o1.Delta();

double call\_gamma = o1.Gamma();

// calculate put price, delta and gamma

double put\_price = o2.P();

double put\_delta = o2.Delta();

double put\_gamma = o2.Gamma();

cout << "The price of the call is " << call\_price << endl;

cout << "The delta of the call is " << call\_delta << endl;

cout << "The gamma of the call is " << call\_gamma << endl;

cout << "The price of the put is " << put\_price << endl;

cout << "The delta of the put is " << put\_delta << endl;

cout << "The gamma of the put is " << put\_gamma << endl;

return 0;

}

# Animal Farm Inheritance

Animal.h

#pragma once

#include "AnimalStats.h"

#include <string>

class Animal

{

public:

Animal(string type, string farmLocation);

~Animal();

void Animal::SetStats(const AnimalStats& stats);

private:

string type\_;

string farmLocation\_;

AnimalStats\* stats\_;

};

Animal.cpp

#include "Animal.h"

Animal::Animal(string type, string farmLocation)

:type\_(type),

farmLocation\_(farmLocation)

{}

Animal::~Animal()

{}

void Animal::SetStats(const AnimalStats & stats)

{

delete stats\_;

stats\_ = stats.Clone();

}

AnimalStats.h

#pragma once

#include <string>

using std::string;

class AnimalStats

{

public:

virtual ~AnimalStats();

string GetcategoryNumber();

virtual double GetPopulation();

virtual void ApplyHarvest() = 0;

virtual void ApplyPopulationGrowth() = 0;

virtual AnimalStats\* Clone() const = 0;

protected:

AnimalStats(string categoryNumber, double averageWeight, double averageAge, double population);

double averageWeight\_;

double averageAge\_;

double population\_;

private:

string categoryNumber\_;

};

AnimalStats.cpp

#include "AnimalStats.h"

AnimalStats::~AnimalStats()

{}

string AnimalStats::GetcategoryNumber()

{

return categoryNumber\_;

}

double AnimalStats::GetPopulation()

{

return population\_;

}

AnimalStats::AnimalStats(string categoryNumber, double averageWeight, double averageAge, double population)

:categoryNumber\_(categoryNumber),

averageWeight\_(averageWeight),

averageAge\_(averageAge),

population\_(population)

{}

Fish.h

#pragma once

#include "AnimalStats.h"

#include <string>

using std::string;

class Fish : public AnimalStats

{

public:

Fish(string categoryNumber, double averageWeight, double averageAge, double population);

~Fish();

void ApplyHarvest();

void ApplyPopulationGrowth();

AnimalStats\* Clone() const;

private:

};

Fish.cpp

#include "Fish.h"

Fish::Fish(string categoryNumber, double averageWeight, double averageAge, double population)

:AnimalStats(categoryNumber, averageWeight, averageAge, population)

{}

Fish::~Fish()

{}

void Fish::ApplyHarvest()

{

population\_ = population\_ \* 0.75;

}

void Fish::ApplyPopulationGrowth()

{

population\_ = population\_ \* 1.05;

}

AnimalStats \* Fish::Clone() const

{

return new Fish(\*this);

}

Mammal.h

#pragma once

#include "AnimalStats.h"

#include <string>

using std::string;

class Mammal : public AnimalStats

{

public:

Mammal(string categoryNumber, double averageWeight, double averageAge, double population);

~Mammal();

void ApplyHarvest();

void ApplyPopulationGrowth();

AnimalStats\* Clone() const;

private:

};

Mammal.cpp

#include "Mammal.h"

Mammal::Mammal(string categoryNumber, double averageWeight, double averageAge, double population)

:AnimalStats(categoryNumber, averageWeight, averageAge, population)

{}

Mammal::~Mammal()

{}

void Mammal::ApplyHarvest()

{

population\_ = population\_ \* 0.9;

}

void Mammal::ApplyPopulationGrowth()

{

population\_ = population\_ \* 1.02;

}

AnimalStats \* Mammal::Clone() const

{

return new Mammal(\*this);

}

Main.cpp

#include "Animal.h"

#include "AnimalStats.h"

#include "Fish.h"

#include "Mammal.h"

#include <iostream>

#include <string>

using namespace std;

int main()

{

Animal a1("Sheep", "ChicagoFarm");

Mammal stats1("5\_Sheep", 50.0, 2.5, 1000);

a1.SetStats(stats1);

Animal a2("Tuna", "DenverFarm");

Fish stats2("2\_Tuna", 25.0, 1.2, 30000);

a2.SetStats(stats2);

stats1.ApplyHarvest();

stats2.ApplyPopulationGrowth();

cout << "The population of sheep is: " << stats1.GetPopulation() << endl;

cout << "The population of tuna is: " << stats2.GetPopulation() << endl;

return 0;

}

# Polygon

Polygon.h

#pragma once

class Polygon

{

public:

~Polygon();

void SetL(double l);

void SetW(double w);

virtual double Area() = 0;

protected:

Polygon(double l, double w);

double l\_;

double w\_;

};

Polygon.cpp

#include "Polygon.h"

Polygon::~Polygon()

{}

void Polygon::SetL(double l)

{

l\_ = l;

}

void Polygon::SetW(double w)

{

w\_ = w;

}

Polygon::Polygon(double l, double w)

:l\_(l), w\_(w)

{}

Rectangle.h

#pragma once

#include "Polygon.h"

class Rectangle : public Polygon

{

public:

Rectangle(double l, double w);

~Rectangle();

double Area();

};

Rectangle.cpp

#include "Rectangle.h"

Rectangle::Rectangle(double l, double w)

:Polygon(l,w)

{}

Rectangle::~Rectangle()

{}

double Rectangle::Area()

{

return l\_\*w\_;

}

Triangle.h

#pragma once

#include "Polygon.h"

class Triangle : public Polygon

{

public:

Triangle(double l, double w);

~Triangle();

double Area();

};

Triangle.cpp

#include "Triangle.h"

Triangle::Triangle(double l, double w)

:Polygon(l,w)

{}

Triangle::~Triangle()

{}

double Triangle::Area()

{

return 0.5 \* l\_ \* w\_;

}

Main.cpp

#include "Polygon.h"

#include "Triangle.h"

#include "Rectangle.h"

#include <iostream>

using namespace std;

int main()

{

Rectangle r1(5, 4);

Triangle t1(10, 2);

t1.SetW(4);

cout << "Rectangle area: " << r1.Area() << endl;

cout << "Triangle area: " << t1.Area() << endl;

return 0;

}

# Adder Template Class

Adder.h

#pragma once

template <typename T>

class Adder

{

public:

Adder(const T& initValue) : sum\_(initValue) {}

void AddValue(const T& value)

{

sum\_ += value;

}

T GetSum()

{

return sum\_;

}

private:

T sum\_;

};

Main.cpp

#include <iostream>

#include "Adder.h"

#include <complex>

using namespace std;

int main()

{

// normal stuff

Adder<double> a1(1.40);

a1.AddValue(10.2);

a1.AddValue(20.1);

// complex stuff

Adder<complex<int>> a2(complex<int>(0, 0)); // initialise a2

complex<int> c1(1, 2);

a2.AddValue(c1);

cout << "Normal solution: " << a1.GetSum() << endl;

cout << "Complex solution: " << a2.GetSum() << endl;

# Vector STL

Main.cpp

#include <vector>

#include <iostream>

using std::vector;

using namespace std;

int main()

{

vector<int> numbers;

for (int i = 0; i <= 10; ++i)

{

numbers.push\_back(i);

}

cout << numbers[2] << endl;

}

# Iterator STL

Main.cpp

#include <vector>

#include <iostream>

using namespace std;

int main()

{

vector<int> numbers;

for (int i = 0; i <= 10; ++i)

{

numbers.push\_back(i);

}

/\*vector<int>::iterator iter; // use the iterator to go through numbers in a vector

for (iter = numbers.begin(); iter != numbers.end(); ++iter)

{

cout << \*iter << endl;

}\*/

/\*vector<int>::const\_iterator iter; // we are not changing the elements in the vector in this case

for (iter = numbers.cbegin(); iter != numbers.cend(); ++iter)

{

cout << \*iter << endl;

}\*/

vector<int>::reverse\_iterator iter; // we can also iterate in reverse direction

for (auto iter = numbers.rbegin(); iter != numbers.rend(); ++iter)

{

cout << \*iter << endl;

}

}

# List STL

Main.cpp

#include <list>

#include <iostream>

using namespace std;

int main()

{

list<int> numbers;

for (int i = 0; i <= 10; ++i)

{

numbers.push\_back(i);

}

list<int>::iterator iter;

for (iter = numbers.begin(); iter != numbers.end(); ++iter)

{

cout << \*iter << endl;

}

}

# Set STL

Main.cpp

#include <set>

#include <iostream>

using namespace std;

using std::set;

int main()

{

set<int> numbers;

for (int i = 0; i <= 10; ++i)

{

numbers.insert(i);

}

for (auto iter = numbers.cbegin(); iter != numbers.cend(); ++iter)

{

cout << \*iter << endl;

}

}

# Map STL

Main.cpp

#include <map>

#include <string>

using std::map;

using std::string;

int main()

{

map<unsigned long, string> zipcodes;

zipcodes.insert(std::make\_pair<unsigned long, string>(60604, "Chicago"));

zipcodes[60637] = "Hyde Park";

}

# Map STL 2

Main.cpp

#include <map>

#include <string>

#include <iostream>

using std::map;

using std::string;

int main()

{

map<unsigned long, string> zipcodes;

zipcodes.insert(std::make\_pair<unsigned long, string>(60604, "Chicago"));

zipcodes[60637] = "Hyde Park";

// to look up an element we can use the find method

auto iter = zipcodes.find(60604);

if (iter != zipcodes.end())

{

std::cout << "key: " << iter->first << ",value: " << iter->second << std::endl;

}

else

{

std::cout << "Key not found" << std::endl;

}

}

# Bank Accounts Sequence Container

Same as Bank Accounts code except Main.cpp is different

Main.cpp

#include "Account.h"

#include "Checking.h"

#include "CheckingWithInterest.h"

#include "Savings.h"

#include "Customer.h"

#include <iostream>

#include <string>

#include <vector>

using namespace std;

int main()

{

vector<string> details;

vector<string>::iterator iter;

Customer c1("John", "Hyde Park");

Savings acc1("Savings-1111", 1500.0, 0.1, 5.0);

c1.SetAccount(acc1);

Checking acc2("Checking-1111", 1500.0, 0.0, 10.0);

c1.SetAccount(acc2);

acc1.ApplyInterest();

details.push\_back(acc1.GetAccountNumber());

details.push\_back(acc2.GetAccountNumber());

for (iter = details.begin(); iter != details.end(); ++iter)

{

cout << \*iter << endl;

}

//cout << "test: " << details[0] << endl;

//cout << "test: " << details[1] << endl;

}

# Bank Accounts Multiple Accounts per Customer

Customer.h

#pragma once

#include "Account.h"

#include <string>

#include <vector>

using std::vector;

using std::string;

class Customer

{

public:

Customer(string name, string address);

~Customer();

void Customer::SetAccount(Account& account);

string Customer::GetName();

vector<Account\*> Customer::GetAccountPorfolio();

private:

string name\_;

string address\_;

Account\* bankAccount\_;

vector<Account\*> bankAccounts\_;

};

Customer.cpp

#include "Customer.h"

#include <vector>

using std::vector;

Customer::Customer(string name, string address)

:name\_(name),

address\_(address)

{}

Customer::~Customer()

{}

void Customer::SetAccount(Account& account)

{

//delete bankAccount\_;

bankAccount\_ = &account;

bankAccounts\_.push\_back(bankAccount\_);

}

string Customer::GetName()

{

return name\_;

}

vector<Account\*> Customer::GetAccountPorfolio()

{

return bankAccounts\_;

}

Main.cpp

#include "Account.h"

#include "Checking.h"

#include "CheckingWithInterest.h"

#include "Savings.h"

#include "Customer.h"

#include <iostream>

#include <string>

#include <vector>

using namespace std;

int main()

{

Customer c1("John", "Hyde Park");

Savings acc1("Savings-1111", 2500.0, 0.1, 5.0);

c1.SetAccount(acc1);

Checking acc2("Checking-1111", 1500.0, 0.0, 10.0);

c1.SetAccount(acc2);

vector<Account\*> details = c1.GetAccountPorfolio();

vector<Account\*>::iterator iter;

for (iter = details.begin(); iter != details.end(); ++iter)

{

cout << (\*iter)->GetBalance() << endl;

}

for (iter = details.begin(); iter != details.end(); ++iter)

{

cout << (\*iter)->GetAccountNumber() << endl;

}

//acc1.ApplyInterest();

//details.push\_back(acc1.GetAccountNumber());

//details.push\_back(acc2.GetAccountNumber());

//for (iter = details.begin(); iter != details.end(); ++iter)

/\*

{

cout << \*iter << endl;

}

\*/

//cout << "test: " << details[0] << endl;

//cout << "test: " << details[1] << endl;

}

# Currency Converter with Map

Only modified Main.cpp

Main.cpp

#include <iostream>

#include <map>

#include "Currency\_Converter.h"

using namespace std;

int main()

{

int base;

int foreign;

double amount;

map<int, double> converted\_currencies;

Currency c2(1, 2, 5.3);

Currency c3(4, 5, 1.5);

double a = c2.GetAmount();

c3.SetAmount(3);

// Const creation

const Currency c4(3, 1, 10);

c4.GetAmount(); //this will only work if GetAmount() is also made const in the header and cpp files. members of the function can only be changed

// if they are set to mutable

// Copy constructor

Currency c5 = c2;

// Assignment operator

c5 = c3;

// Creation using free store

Currency\* c6 = new Currency(1, 2, 3.8);

// Accessing free store member

c6->GetBase();

c6->GetForeign();

c6->GetAmount();

c6->SetAmount(7.6);

// Delete creation in free store

delete c6;

cout << "Please choose a base currency, where:" << endl;

cout << "{USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6}" << endl;

cin >> base;

cout << "Please choose a foreign currency, where:" << endl;

cout << "{USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6}" << endl;

cin >> foreign;

cout << "Please choose an amount" << endl;

cin >> amount;

Currency c1(base, foreign, amount);

cout << "The converted amount is " << c1.Convert\_Currency() << endl;

pair<int, double>p1(c1.GetForeign(), c1.Convert\_Currency());

converted\_currencies.insert(p1);

auto iter = converted\_currencies.find(1);

if (iter != converted\_currencies.end())

{

cout << "key: " << iter->first << ",value:" << iter->second << endl;

}

else

{

cout << "key not found" << endl;

}

return 0;

}

# Currency Converter with Maps

Modified Currency\_Converter class only

Currency\_Converter.h

#ifndef \_CURRENCY\_CONVERTER\_H

#define \_CURRENCY\_CONVERTER\_H

#include <map>

using std::map;

using std::make\_pair;

class Currency

{

public:

// Constructors

Currency();

Currency(int b, int f, double x);

// Copy constructor

Currency(const Currency& other);

// Assignment operator

Currency& operator=(const Currency& other);

// Destructor

~Currency();

int GetBase();

int GetForeign();

double GetAmount() const;

void SetBase(int b);

void SetForeign(int f);

void SetAmount(double x);

double Convert\_Currency();

private:

int b\_;

int f\_;

mutable double x\_;

map<int, double> Currencies\_;

};

#endif

Currency\_Converter.cpp

#ifndef \_CURRENCY\_CONVERTER\_CPP

#define \_CURRENCY\_CONVERTER\_CPP

#include "Currency\_Converter.h"

Currency::Currency()

: b\_(), f\_(), x\_()

{}

Currency::Currency(int b, int f, double x)

: b\_(b), f\_(f), x\_(x)

{}

Currency::Currency(const Currency & other)

:b\_(other.b\_), f\_(other.f\_), x\_(other.x\_)

{}

Currency & Currency::operator=(const Currency & other)

{

if (this != &other) //checking for self assignment

{

b\_ = other.b\_;

f\_ = other.f\_;

x\_ = other.x\_;

}

// return the object on which the function was invoked

return \*this;

}

Currency::~Currency()

{}

int Currency::GetBase()

{

return b\_;;

}

int Currency::GetForeign()

{

return f\_;

}

double Currency::GetAmount() const

{

return x\_;

}

void Currency::SetBase(int b)

{

b\_ = b;

}

void Currency::SetForeign(int f)

{

f\_ = f;

}

void Currency::SetAmount(double x)

{

x\_ = x;

}

double Currency::Convert\_Currency()

{

// First list currencies in terms of the USD

double USD\_ = 1;

double AUD\_ = 0.70;

double JPY\_ = 0.0081;

double GBP\_ = 1.50;

double EUR\_ = 1.07;

double CAD\_ = 0.75;

double base\_amount;

double foreign\_amount;

double result;

enum currency { USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6 };

auto iter\_b = Currencies\_.find(b\_);

if (iter\_b != Currencies\_.end())

{

base\_amount = iter\_b->second;

return iter\_b -> second;

}

else

{

switch (b\_)

{

case USD:

base\_amount = USD\_;

break;

case AUD:

base\_amount = AUD\_;

break;

case JPY:

base\_amount = JPY\_;

break;

case GBP:

base\_amount = GBP\_;

break;

case EUR:

base\_amount = EUR\_;

break;

case CAD:

base\_amount = CAD\_;

break;

}

}

auto iter\_f = Currencies\_.find(f\_);

if (iter\_f != Currencies\_.end())

{

foreign\_amount = iter\_f->second;

return foreign\_amount;

}

else

{

switch (f\_)

{

case USD:

foreign\_amount = USD\_;

break;

case AUD:

foreign\_amount = AUD\_;

break;

case JPY:

foreign\_amount = JPY\_;

break;

case GBP:

foreign\_amount = GBP\_;

break;

case EUR:

foreign\_amount = EUR\_;

break;

case CAD:

foreign\_amount = CAD\_;

break;

}

}

return result = (base\_amount / foreign\_amount)\*x\_;

}

#endif

# Pointers Demo

Currency.h

#pragma once

#include <string>

using std::string;

using std::endl;

class Currency

{

public:

Currency(string symbol, double exchangeRate);

Currency(const Currency& other);

~Currency();

double ConvertFromUSD(double amount);

double ConvertToUSD(double amount);

string GetSymbol();

double GetExchangeRate();

void SetExchangeRate(double rate);

private:

string symbol\_;

double exchangeRate\_;

};

Currency.cpp

#include "Currency.h"

#include <iostream>

using std::string;

using std::cout;

using std::endl;

Currency::Currency(string symbol, double exchangeRate)

:symbol\_(symbol), exchangeRate\_(exchangeRate)

{

cout << "Currency Constructor: " << symbol\_ << endl;

}

Currency::Currency(const Currency & other)

:symbol\_(other.symbol\_), exchangeRate\_(other.exchangeRate\_)

{

std::cout << "Copy Constructor: " << symbol\_ << endl;

}

Currency::~Currency()

{

cout << "Currency Destructor: " << symbol\_ << endl;

}

double Currency::ConvertFromUSD(double amount)

{

return amount \* exchangeRate\_;

}

double Currency::ConvertToUSD(double amount)

{

return 1.0 \* amount / exchangeRate\_;

}

string Currency::GetSymbol()

{

return symbol\_;

}

double Currency::GetExchangeRate()

{

return exchangeRate\_;

}

void Currency::SetExchangeRate(double rate)

{

exchangeRate\_ = rate;

}

Main.cpp

#include "Currency.h"

#include <iostream>

#include <memory> //for std::shared\_ptr, std::unique\_ptr, std::make\_shared and std::make\_unique

#include <string>

#include <utility> // for std::move

using std::cout;

using std::endl;

using std::unique\_ptr;

using std::shared\_ptr;

using std::weak\_ptr;

using std::make\_unique;

int main()

{

//unique\_ptr

unique\_ptr<Currency> u1(new Currency("USD", 1.0));

unique\_ptr<Currency> u2 = make\_unique<Currency>("JPY", 120); // creates into a unique pointer

u1.reset(new Currency("JPY", 120)); // reset the pointer

u1->GetSymbol();

u1->GetExchangeRate();

unique\_ptr<Currency> u3 = std::move(u1); // move the unique\_ptr from u3 to u1

u1 = nullptr; // u1 is now free to change

//shared\_ptr

shared\_ptr<Currency> sp1(new Currency("USD", 1.0));

shared\_ptr<Currency> sp2 = std::make\_shared<Currency>("USD", 1.0);

int sp1\_refCount = sp1.use\_count(); // counting the number of references

{

shared\_ptr<Currency> sp3 = sp1; // shared pointers can be copied

sp1\_refCount = sp1.use\_count(); // reference count goes up by 1

}

sp1\_refCount = sp1.use\_count(); // reference inside scope gets deleted

{

shared\_ptr<Currency> sp4 = sp1;

sp1\_refCount = sp1.use\_count();

int sp4\_refCount = sp4.use\_count(); // has the same reference count as sp1

sp4.reset(new Currency(string("JPY"), 115.0));

sp4\_refCount = sp4.use\_count(); // will now only have reference count of 1 since it is reset

sp1\_refCount = sp1.use\_count(); // will now only have reference count of 1 since sp4 is reset

}

//weak\_ptr

shared\_ptr<Currency> sp5 = std::make\_shared<Currency>("USD", 1.0);

int sp5\_refCount = sp5.use\_count();

weak\_ptr<Currency> wp1 = sp5;

sp5\_refCount = sp5.use\_count(); // weak pointer doesn't add any count to the reference count

if (!wp1.expired())

{

shared\_ptr<Currency> sp6 = wp1.lock(); // converting a weak pointer to a shared pointer before using it

sp6->GetSymbol();

}

}

# Bank Account with Pointers

Account.h

#pragma once

#include <string>

using std::string;

class Account

{

public:

Account();

Account(string account\_number, double balance);

~Account();

string GetAccountNumber();

double GetBalance();

void SetAccount(string account\_number, double balance);

private:

string account\_number\_;

double balance\_;

};

Account.cpp

#include "Account.h"

Account::Account()

:account\_number\_(), balance\_()

{}

Account::Account(string account\_number, double balance)

: account\_number\_(account\_number),

balance\_(balance)

{}

Account::~Account()

{}

string Account::GetAccountNumber()

{

return account\_number\_;

}

double Account::GetBalance()

{

return balance\_;

}

void Account::SetAccount(string account\_number, double balance)

{

account\_number\_ = account\_number;

balance\_ = balance;

}

Customer.h

#pragma once

#include <string>

#include <memory>

#include "Account.h"

using std::string;

using std::unique\_ptr;

class Customer

{

public:

Customer(string name, string address);

~Customer();

Customer(const Customer& other); // copy constructor

Customer& operator=(const Customer& other); // assignment operator

void SetAccount(string accNumber, double balance);

private:

string name\_;

string address\_;

unique\_ptr<Account> account\_; // account object is on the free store

};

Customer.cpp

#include "Customer.h"

Customer::Customer(string name, string address)

:name\_(name), address\_(address)

{}

Customer::~Customer()

{

//delete account\_; // no need for a delete account member anymore. when out of scope account\_ is deleted automatically

}

Customer::Customer(const Customer & other)

:name\_(other.name\_),address\_(other.address\_),

account\_(std::make\_unique<Account>(

other.account\_->GetAccountNumber(),

other.account\_->GetBalance()))

{}

Customer & Customer::operator=(const Customer & other)

{

if (this != &other)

{

name\_ = other.name\_;

address\_ = other.address\_;

account\_.reset(

new Account(other.account\_->GetAccountNumber(),

other.account\_->GetBalance()));

}

return \*this;

}

void Customer::SetAccount(string accNumber, double balance)

{

account\_.reset(new Account(accNumber, balance));

}

# Currency Converter Pointers

Currency\_Converter.h

#pragma once

#include <memory>

using std::shared\_ptr;

class Currency

{

public:

// Constructors

Currency();

Currency(int b, int f, double x);

// Copy constructor

Currency(const Currency& other);

// Assignment operator

Currency& operator=(const Currency& other);

// Destructor

~Currency();

int GetBase();

int GetForeign();

double GetAmount() const;

void SetAmount(double x);

double Convert\_Currency();

private:

shared\_ptr<int> b\_;

shared\_ptr<int> f\_;

mutable shared\_ptr<double> x\_;

};

Currency\_Converter.cpp

#include "Currency\_Converter.h"

Currency::Currency()

: b\_(), f\_(), x\_()

{}

Currency::Currency(int b, int f, double x)

: b\_(std::make\_shared<int>(b))

, f\_(std::make\_shared<int>(f))

, x\_(std::make\_shared<double>(x))

{}

Currency::Currency(const Currency & other) // copy constructor is as default

{}

Currency & Currency::operator=(const Currency & other)

{

if (this != &other) //checking for self assignment

{

b\_ = other.b\_;

f\_ = other.f\_;

x\_ = other.x\_;

}

// return the object on which the function was invoked

return \*this;

}

Currency::~Currency()

{}

int Currency::GetBase()

{

return \*b\_;

}

int Currency::GetForeign()

{

return \*f\_;

}

double Currency::GetAmount() const

{

return \*x\_;

}

void Currency::SetAmount(double x)

{

\*x\_ = x;

}

double Currency::Convert\_Currency()

{

// First list currencies in terms of the USD

double USD\_ = 1;

double AUD\_ = 0.70;

double JPY\_ = 0.0081;

double GBP\_ = 1.50;

double EUR\_ = 1.07;

double CAD\_ = 0.75;

double base\_amount;

double foreign\_amount;

double result;

enum currency { USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6 };

switch (\*b\_)

{

case USD:

base\_amount = USD\_;

break;

case AUD:

base\_amount = AUD\_;

break;

case JPY:

base\_amount = JPY\_;

break;

case GBP:

base\_amount = GBP\_;

break;

case EUR:

base\_amount = EUR\_;

break;

case CAD:

base\_amount = CAD\_;

break;

}

switch (\*f\_)

{

case USD:

foreign\_amount = USD\_;

break;

case AUD:

foreign\_amount = AUD\_;

break;

case JPY:

foreign\_amount = JPY\_;

break;

case GBP:

foreign\_amount = GBP\_;

break;

case EUR:

foreign\_amount = EUR\_;

break;

case CAD:

foreign\_amount = CAD\_;

break;

}

return result = (base\_amount / foreign\_amount)\* \*x\_;

}

Main.cpp

#include <iostream>

#include "Currency\_Converter.h"

using namespace std;

int main()

{

int base;

int foreign;

double amount;

Currency c2(1, 2, 5.3);

Currency c3(4, 5, 1.5);

double a = c2.GetAmount();

c3.SetAmount(3);

// Const creation

const Currency c4(3, 1, 10);

c4.GetAmount(); //this will only work if GetAmount() is also made const in the header and cpp files. members of the function can only be changed

// if they are set to mutable

// Copy constructor

Currency c5 = c2;

// Assignment operator

c5 = c3;

// Creation using free store

Currency\* c6 = new Currency(1, 2, 3.8);

// Accessing free store member

c6->GetBase();

c6->GetForeign();

c6->GetAmount();

c6->SetAmount(7.6);

// Delete creation in free store

delete c6;

cout << "Please choose a base currency, where:" << endl;

cout << "{USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6}" << endl;

cin >> base;

cout << "Please choose a foreign currency, where:" << endl;

cout << "{USD = 1, AUD = 2, JPY = 3, GBP = 4, EUR = 5, CAD = 6}" << endl;

cin >> foreign;

cout << "Please choose an amount" << endl;

cin >> amount;

Currency c1(base, foreign, amount);

cout << "The converted amount is " << c1.Convert\_Currency() << endl;

return 0;

}

# Jarrow Rudd Binomial Tree

Option.h

#ifndef OPTION\_H

#define OPTION\_H

#include <string>

#include <memory>

#include "Payoff.h"

using std::string;

using std::shared\_ptr;

class Option

{

public:

Option(); // default constructor

Option(double T, const shared\_ptr<Payoff>& payoff);

~Option(); // destructor

virtual double ExpirationPayoff(double ST) const = 0;

virtual double IntermediatePayoff(double St, double discountedExpectation) const = 0;

//Gets

double GetS();

double Getsigma();

double GetK() const;

double GetT();

double Gett();

double Getr();

string Gettype();

//Sets

void SetS(double S);

void Setsigma(double sigma);

void SetK(double K);

void SetT(double T);

void Sett(double t);

void Setr(double r);

void Settype(string type);

protected:

shared\_ptr<Payoff> payoff\_;

private:

double S\_;

double sigma\_;

double K\_;

double T\_;

double t\_;

double r\_;

string type\_;

};

# endif

Option.cpp

#include "Option.h"

Option::Option()

{}

Option::Option(double T, const shared\_ptr<Payoff>& payoff)

{

T\_ = T;

payoff\_ = payoff;

}

Option::~Option()

{}

double Option::GetS()

{

return S\_;

}

double Option::Getsigma()

{

return sigma\_;

}

double Option::GetK() const

{

return K\_;

}

double Option::GetT()

{

return T\_;

}

double Option::Gett()

{

return t\_;

}

double Option::Getr()

{

return r\_;

}

string Option::Gettype()

{

return type\_;

}

void Option::SetS(double S)

{

S\_ = S;

}

void Option::Setsigma(double sigma)

{

sigma\_ = sigma;

}

void Option::SetK(double K)

{

K\_ = K;

}

void Option::SetT(double T)

{

T\_ = T;

}

void Option::Sett(double t)

{

t\_ = t;

}

void Option::Setr(double r)

{

r\_ = r;

}

void Option::Settype(string type)

{

type\_ = type;

}

EuropeanOption.h

#pragma once

#include "Option.h"

#include <cmath>

class EuropeanOption : public Option

{

public:

EuropeanOption(double T, const shared\_ptr<Payoff>& payoff);

~EuropeanOption();

double ExpirationPayoff(double ST) const;

double IntermediatePayoff(double St, double discountedExpectation) const;

};

EuropeanOption.cpp

#include "EuropeanOption.h"

EuropeanOption::EuropeanOption(double T, const shared\_ptr<Payoff>& payoff)

:Option(T,payoff)

{}

EuropeanOption::~EuropeanOption()

{}

double EuropeanOption::ExpirationPayoff(double ST) const

{

return payoff\_->GetPayoff(ST);

}

double EuropeanOption::IntermediatePayoff(double St, double discountedExpectation) const

{

return discountedExpectation;

}

AmericanOption.h

#pragma once

#include "Option.h"

#include <cmath>

class AmericanOption : public Option

{

public:

AmericanOption(double T, const shared\_ptr<Payoff>& payoff);

~AmericanOption();

double ExpirationPayoff(double ST) const;

double IntermediatePayoff(double St, double discountedExpectation) const;

};

AmericanOption.cpp

#include "AmericanOption.h"

AmericanOption::AmericanOption(double T, const shared\_ptr<Payoff>& payoff)

:Option(T, payoff)

{}

AmericanOption::~AmericanOption()

{}

double AmericanOption::ExpirationPayoff(double ST) const

{

return payoff\_->GetPayoff(ST);

}

double AmericanOption::IntermediatePayoff(double St, double discountedExpectation) const

{

double earlyExercise = payoff\_->GetPayoff(St);

return earlyExercise > discountedExpectation ? earlyExercise : discountedExpectation;

}

Payoff.h

#ifndef \_PAYOFF\_H

#define \_PAYOFF\_H

#include <algorithm>

class Payoff

{

public:

Payoff();

virtual ~Payoff();

// Overloaded operator(), turn the Payoff into an abstract function object

virtual double GetPayoff(double) const = 0; // pure virtual

protected:

double K\_;

};

#endif

Payoff.cpp

#include "Payoff.h"

Payoff::Payoff()

{}

Payoff::~Payoff()

{}

PayoffCall.h

#ifndef \_PAYOFFCALL\_H

#define \_PAYOFFCALL\_H

#include "Payoff.h"

class PayoffCall : public Payoff

{

public:

PayoffCall(double K, double r, double T);

~PayoffCall();

double GetPayoff(double ST) const;

private:

double r\_;

double T\_;

};

#endif

PayoffCall.cpp

#ifndef \_PAYOFFCALL\_CPP

#define \_PAYOFFCALL\_CPP

#define \_USE\_MATH\_DEFINES

#include <math.h>

#include "PayoffCall.h"

PayoffCall::PayoffCall(double K, double r, double T)

{

K\_ = K;

r\_ = r;

T\_ = T;

}

PayoffCall::~PayoffCall()

{}

double PayoffCall::GetPayoff(double ST) const

{

double Ci = exp(-r\_\*T\_) \* (ST > K\_ ? ST - K\_ : 0.0);

return Ci;

}

#endif

PayoffPut.h

#ifndef \_PAYOFFPUT\_H

#define \_PAYOFFPUT\_H

#include "Payoff.h"

class PayoffPut : public Payoff

{

public:

PayoffPut(double K, double r, double T);

virtual ~PayoffPut();

double GetPayoff(double ST) const;

private:

double r\_;

double T\_;

};

#endif

PayoffPut.cpp

#ifndef \_PAYOFFPUT\_CPP

#define \_PAYOFFPUT\_CPP

#define \_USE\_MATH\_DEFINES

#include <math.h>

#include "PayoffPut.h"

PayoffPut::PayoffPut(double K, double r, double T)

{

K\_ = K;

r\_ = r;

T\_ = T;

}

PayoffPut::~PayoffPut()

{}

double PayoffPut::GetPayoff(double ST) const

{

double Ci = exp(-r\_\*T\_) \* (ST < K\_ ? K\_ - ST : 0.0);

return Ci;

}

#endif

BinomialTree.h

//#ifndef BINOMIAL\_TREE\_H

//#define BINOMIAL\_TREE\_H

#pragma once

#include "Option.h"

#include <vector>

using std::vector;

using std::pair;

class BinomialTree

{

public:

BinomialTree(double S, double rate, double div, double vol, double time, int steps);

double Price(const Option& option);

private:

void initializeTree();

bool treeInitialized\_;

typedef double StockPrice;

typedef double OptionPrice;

typedef pair<StockPrice, OptionPrice> Node;

typedef vector<Node> VerticalNodes;

typedef vector<VerticalNodes> Tree;

Tree tree\_;

double S0;

double r;

double q; //you can add dividends

double v;

double T;

double N;

double dt;

};

BinomialTree.cpp

#include "BinomialTree.h"

#include <iostream>

#include <omp.h>

using std::cout;

using std::endl;

BinomialTree::BinomialTree(double S, double rate,

double div, double vol, double expiry, int steps)

: S0(S),

r(rate),

q(div),

v(vol),

T(expiry),

N(steps),

treeInitialized\_(false)

{

dt = T / N;

initializeTree();

}

void BinomialTree::initializeTree()

{

//Step 1

//Create/initialize the Tree

//we have N+1 time steps i.e. N+1 VerticalNodes

tree\_.resize(N + 1);

//each VerticalNodes vector should have (i+1) Nodes where i is

//the time index

for (long i = 0; i <= N; ++i)

{

tree\_[i].resize(i + 1);

}

//Step 2

//Populate stock prices

//we access the first and second elements in a pair using first and second fields

tree\_[0][0].first = S0;

double nu = (r - 0.5\*v\*v);

double sqrt\_dt = sqrt(dt);

//i is horizontal time index

// we go from left to right

for (long i = 1; i <= N; ++i)

{

//Eqn 6

//St(D, t+dt) = St\*exp((nu)dt - sigma\*sqrt(dt)), where nu = r-0.5\*sigma\*sigma

tree\_[i][0].first = tree\_[i - 1][0].first\*exp(nu\*dt - v\*sqrt\_dt);

for (int j = 1; j <= i; ++j)

{

tree\_[i][j].first = tree\_[i][0].first\*exp(j \* 2 \* v\*sqrt\_dt); //Using factor SU/SD

}

}

treeInitialized\_ = true;

}

double BinomialTree::Price(const Option &theOption)

{

if (!treeInitialized\_) initializeTree();

//Step 3

//Calculate option prices at t=T

for (long j = 0; j <= N; ++j)

{

tree\_[N][j].second = theOption.ExpirationPayoff(tree\_[N][j].first);

}

double disc = exp(-r\*dt);

//Step 4

//Back propagation

//ir is index i in reverse direction

for (long ir = N - 1; ir >= 0; --ir)

{

for (long j = 0; j <= ir; ++j)

{

//use Equation 8 to find the discounted expectation of the two adjacent option prices

double discountedExpectation = disc\*0.5\*(tree\_[ir + 1][j].second + tree\_[ir + 1][j + 1].second);

//find the payoff at the node:

// a)euroepan: the intermediate payoff is the same as discounted expectation

// b)american: the intermediate payoff is the more profitable value between

// immediate exercise and discounted expectation

// c)barrier: we have to see if the barrier is hit and use appropriate in/out rules

tree\_[ir][j].second = theOption.IntermediatePayoff(tree\_[ir][j].first, discountedExpectation);

}

}

return tree\_[0][0].second;

}

Main.cpp

#include <iostream>

#include <memory>

#include <utility>

#include "Option.h"

#include "AmericanOption.h"

#include "EuropeanOption.h"

#include "BinomialTree.h"

#include "Payoff.h"

#include "PayoffCall.h"

#include "PayoffPut.h"

using std::cout;

using std::cin;

using std::endl;

using std::shared\_ptr;

using std::make\_shared;

using std::vector;

using std::pair;

int main()

{

double K = 100.0;

double S0 = 100.0;

double r = 0.02;

double q = 0.0;

double sigma = 0.3;

double T = 1.0;

const int N = 100;

shared\_ptr<Payoff> putPayOff = make\_shared<PayoffPut>(K, r, T);

shared\_ptr<Payoff> callPayOff = make\_shared<PayoffCall>(K, r, T);

EuropeanOption europeanPut(T, putPayOff);

EuropeanOption europeanCall(T, callPayOff);

AmericanOption americanPut(T, putPayOff);

BinomialTree tree(S0, r, q, sigma, T, N);

double callPrice1 = tree.Price(europeanCall);

double putPrice1 = tree.Price(europeanPut);

double putPrice2 = tree.Price(americanPut);

cout << "European Call Price= " << callPrice1 << endl;

cout << "European Put Price= " << putPrice1 << endl;

cout << "American Put Price= " << putPrice2 << endl;

}

# Assertions

Main.cpp

#include <cassert>

#include <iostream>

#include <string>

void AssertTest()

{

int x = 0;

int y = 4;

assert(x != 0);

int res = y / x;

}

int main()

{

AssertTest();

}

# Chrono

Main.cpp

#include <chrono>

#include <iostream>

using namespace std::chrono;

using std::cout;

using std::endl;

int main()

{

//measure time now

auto t1 = high\_resolution\_clock::now();

//do something

//measure time again

auto t2 = high\_resolution\_clock::now();

//time taken given by time difference

cout << duration\_cast<milliseconds>(t2 - t1).count();

}

# Exceptions

Main.cpp

#include <vector>

#include <iostream>

#include <algorithm>

#include <stdexcept>

using std::vector;

using std::cout;

using std::endl;

using std::cin;

int main()

{

try // encloses a block of code (one or more statements) that might throw an exception

{

vector<int> values{ 2, 3, 6 };

for (int idx = 0; idx <= 3; ++idx)

{

cout << values.at(idx) << endl;

cout << "index " << idx << endl;

cout << "Element at " << idx << " is:" << values.at(idx) << endl;;

}

}

catch (std::out\_of\_range& e) // one or more catch block should immediately follow a try block. each catch block specifies the type of exception it can handle

// when an exception is thrown from a try block, the control reaches the appropriate catch block

{

cout << "caught exception " << endl;

}

cout << "doing some more work ...." << endl;

cout << "gracefully shutting down" << endl;

}

# FileOps

Main.cpp

#include <iostream>

#include <string>

#include <fstream>

#include <sstream>

#include <iterator>

#include <vector>

using std::string;

using std::ofstream;

using std::ifstream;

using std::endl;

using std::cout;

void WriteToFile()

{

string filename = "greetings.txt";

//open output file for writing

ofstream outfile(filename);

//ofstream outfile(filename, std::ios::trunc);

//outfile.open(filename);

if (outfile) //testing to see if the file is open

{

outfile << "hello world 1" << std::endl;

outfile << "hello world 2" << std::endl;

//not needed, closes when it goes out of scope

outfile.close();

}

else //for some reason were were not able to open file

{

cout << "Unable to open file " << filename << endl;

}

}

void ReadFromFile()

{

//name of the input file

string filename = "greetings.txt";

//open file for reading

ifstream infile(filename);

if (!infile) //testing to see if the file is open

{

cout << "Unable to open file " << filename << endl;

return;

}

//read the file

string data;

while (getline(infile, data))

{

//reads characters from infile (input stream) and stores them into data (string)

//until the newline character ('\n') is found.

cout << data << endl;

}

}

int main()

{

WriteToFile();

ReadFromFile();

}

/\*

The default mode for opening a file with ofstream's constructor is to create it if it does not exist, or delete everything in it if something does exist in it.

If necessary, you can give a second argument that specifies how the file should be handled. They are listed below:

ios::app -- Append to the file

ios::ate -- Set the current position to the end

ios::trunc -- Delete everything in the file

For example:

ofstream a\_file ( "test.txt", ios::app );

This will open the file without destroying the current contents and allow you to append new data. When opening files, be very careful not to use them if the file

could not be opened. This can be tested for very easily:

ifstream a\_file ( "example.txt" );

if ( !a\_file.is\_open() ) {

// The file could not be opened

}

else {

// Safely use the file stream

}

\*/

# IOManipulators

Main.cpp

#include <iostream>

#include <iomanip>

using std::cout;

using std::endl;

int main()

{

double val = 22.0 / 7.0;

cout << val << endl;

cout << std::fixed << std::setprecision(10) << val << endl;

}

# IOStreams

Main.cpp

#include <iostream>

#include <fstream>

#include <iterator>

#include <vector>

void streams()

{

std::ofstream ofs("cout.txt");

std::cout.rdbuf(ofs.rdbuf());

std::ofstream efs("cerr.txt");

std::cerr.rdbuf(efs.rdbuf());

std::ofstream lfs("clog.txt");

std::clog.rdbuf(lfs.rdbuf());

std::cout << "cout message1 " << std::endl;

std::cout << "cout message2 " << std::endl;

std::cerr << "cerr message1 " << std::endl;

std::cerr << "cerr message2 " << std::endl;

std::clog << "clog message1 " << std::endl;

std::clog << "clog message2 " << std::endl;

}

void stream\_iterators()

{

std::istream\_iterator<int> v1(std::cin);

std::vector<int> v2;

//use default constructor to test equal to end of stream

copy(v1, std::istream\_iterator<int>(), std::back\_inserter(v2));

copy(v2.begin(), v2.end(), std::ostream\_iterator<int>(std::cout, " "));

}

int main()

{

streams();

}

# StringStream

Main.cpp

#include <sstream>

#include <iostream>

#include <string>

using std::cout;

using std::endl;

using std::ends;

using std::string;

int main()

{

// ostringstream example:

std::ostringstream oss;

oss << "hello,";

oss << " world" << ends;

cout << oss.str() << endl;

// istringstream example:

string s = "USD 1.0";

std::istringstream iss(s);

string symbol; double rate;

iss >> symbol >> rate;

}

# Deque

Main.cpp

#ifndef LIST\_H

#define LIST\_H

#include <iostream>

using std::cout;

using std::endl;

namespace finm323

{

template <typename T>

class forward\_list

{

public:

//member functions

private:

struct node

{

node(T val, node\* n)

: data(val), next(n)

{}

T data;

node \* next;

};

typedef node\* link;

link head;

};

}

#endif

# Singly Linked List

Forward\_list.h

#ifndef LIST\_H

#define LIST\_H

#include <iostream>

using std::cout;

using std::endl;

namespace finm323

{

template <typename T>

class forward\_list

{

public:

//member functions

private:

struct node

{

node(T val, node\* n)

: data(val), next(n)

{}

T data;

node \* next;

};

typedef node\* link;

link head;

};

}

#endif

main.cpp

#include "forward\_list.h"

#include <iostream>

#include <string>

int main()

{

finm323::forward\_list<int> l;

}

# Node

Main.cpp

#include <iostream>

using std::cout;

using std::endl;

struct node

{

//write class definition here

};

typedef node\* link;

void create\_list()

{

//create first node

//link head = write code here

// \_\_\_\_\_\_\_\_\_\_\_\_\_

// head | | |

//------> | 12 | |-->nullptr

// |\_\_\_\_\_|\_\_\_\_\_|

//

//create second node

//link n = write code here

// \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_

// head | | | n | | |

//------> | 12 | |-->nullptr ---->| 21 | |--->nullptr

// |\_\_\_\_\_|\_\_\_\_\_| |\_\_\_\_\_\_|\_\_\_\_\_|

//

//link them

//head->next = write code here

// \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_

// head | | | n | | |

//-------> | 12 | |--------->| 21 | |--->nullptr

// |\_\_\_\_\_|\_\_\_\_\_| |\_\_\_\_\_\_|\_\_\_\_\_|

//

//continue ...

//write code here

// \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_

// head | | | | | | n | | |

//-----> | 12 | |------>| 21 | |---->nullptr --->|37 | |-->nullptr

// |\_\_\_\_\_|\_\_\_| |\_\_\_\_\_|\_\_\_\_| |\_\_\_\_|\_\_\_\_|

//

// \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_

// head | | | | | | n | | |

//-----> | 12 | |------>| 21 | |------>|37 | |-->nullptr

// |\_\_\_\_\_|\_\_\_| |\_\_\_\_\_|\_\_\_\_| |\_\_\_\_|\_\_\_\_|

//

//traverse the list (and print)

//write code here

}

int main()

{

create\_list();

}

# Slicing

Main.cpp

#include <iostream>

using std::endl;

using std::cout;

//Here I have written the class definition and implementation in the same place

//for "demo" convenience.

//Writing definition and implementation like below is legal in C++.

//I'll to say more on this later when we discuss "inlining"

class Base

{

public:

Base() {}

virtual void DoSomething()

{

cout << "Base: DoSomething" << endl; //<= Output A

}

};

class Derived : public Base

{

public:

Derived() {}

void DoSomething()

{

cout << "Derived: DoSomething" << endl; //<= Output B

}

};

void SomeFunction(Base b)

{

b.DoSomething();

}

int main()

{

Derived d;

SomeFunction(d);

}

# Vector

Main.cpp

#include "Student.h"

#include <vector>

#include <iostream>

#include <chrono>

#include <thread>

using std::vector;

using std::cout;

using std::endl;

void test\_size\_capacity()

{

vector<int> v0; // constructing empty vector

cout << v0.size() << ", " << v0.capacity() << endl;

vector<int> v1(10); // constructing vector of specific capacity

cout << v1.size() << ", " << v1.capacity() << endl;

vector<int> v2;

v2.reserve(10); // specific capacity

cout << v2.size() << ", " << v2.capacity() << endl;

}

void test\_resize()

{

vector<int> v;

cout << v.size() << ", " << v.capacity() << endl;

v.resize(10); // resizing capacity

cout << v.size() << ", " << v.capacity() << endl;

}

void test\_appending\_end()

{

vector<Student> v(10);

Student s("aaa");

v.push\_back(s); // add element

cout << v.size() << ", " << v.capacity() << endl;

v.insert(v.end(), s); // add element

}

void test\_appending\_middle()

{

vector<Student> v(10);

Student s("aaa");

v.push\_back(s);

cout << v.size() << ", " << v.capacity() << endl;

auto mid = v.begin() + 2;

v.insert(mid, s);

}

void test\_allocating\_capacity()

{

using namespace std::chrono;

auto t1 = high\_resolution\_clock::now();

for (int count = 0; count < 1000; ++count)

{

vector<int> v;

for (int i = 0; i < 100; ++i)

{

v.push\_back(i);

}

}

auto t2 = high\_resolution\_clock::now();

std::this\_thread::sleep\_for(60s);

auto t3 = high\_resolution\_clock::now();

for (int count = 0; count < 1000; ++count)

{

vector<int> v;

v.reserve(100);

for (int i = 0; i < 100; ++i)

{

v.push\_back(i);

}

}

auto t4 = high\_resolution\_clock::now();

cout << "test 1 elapsed time: " << duration\_cast<microseconds>(t2 - t1).count() << endl;

cout << "test 2 elapsed time: " << duration\_cast<microseconds>(t4 - t3).count() << endl;

}

void test\_move()

{

using namespace std::chrono;

auto t1 = high\_resolution\_clock::now();

for (int count = 0; count < 1000; ++count)

{

vector<Student> v;

v.reserve(100);

for (int i = 0; i < 100; ++i)

{

v.push\_back(Student("viktor"));

}

}

auto t2 = high\_resolution\_clock::now();

std::this\_thread::sleep\_for(60s);

auto t3 = high\_resolution\_clock::now();

for (int count = 0; count < 1000; ++count)

{

vector<Student> v;

v.reserve(100);

for (int i = 0; i < 100; ++i)

{

v.emplace\_back(Student("viktor"));

}

}

auto t4 = high\_resolution\_clock::now();

cout << "test 1 elapsed time: " << duration\_cast<microseconds>(t2 - t1).count() << endl;

cout << "test 2 elapsed time: " << duration\_cast<microseconds>(t4 - t3).count() << endl;

}

void test\_capacity\_step\_size()

{

std::vector<int> v;

std::cout << "----" << std::endl;

for (int i = 1; i < 200; ++i)

{

v.push\_back(i);

if (i % 25 == 0) cout << "adding element: " << i

<< " size: " << v.size()

<< " capacity: " << v.capacity() << endl;

}

}

int main()

{

//test\_size\_capacity();

//test\_resize();

//test\_appending\_end(); //remove

//test\_appending\_middle(); //remove

//test\_allocating\_capacity();

//test\_move();

//test\_capacity\_step\_size();

}

# Vector vs. List

Currency.h

#include <iostream>

#include <string>

using std::cout;

using std::endl;

using std::string;

class Currency

{

public:

Currency() {};

Currency(string symbol, double factor);

Currency(const Currency& other);

Currency(const Currency&& other);

~Currency();

double ConvertFromUSD(double amount);

string GetSymbol() const;

double GetConversionRate() const;

void SetConversionRate(double rate);

private:

string symbol\_;

double conversionRate\_;

};

Currency.cpp

#include "Currency.h"

Currency::Currency(string symbol, double conversionRate)

: symbol\_(symbol),

conversionRate\_(conversionRate)

{

cout << "Constructing Currency for " << symbol\_ << endl;

}

Currency::Currency(const Currency& other)

: symbol\_(other.symbol\_),

conversionRate\_(other.conversionRate\_)

{

cout << "Copy Constructing Currency for " << symbol\_ << endl;

}

Currency::Currency(const Currency&& other)

: symbol\_(std::move(other.symbol\_)),

conversionRate\_(std::move(other.conversionRate\_))

{

cout << "Move Copy Constructing Currency for " << symbol\_ << endl;

}

Currency::~Currency()

{

cout << "Destructing Currency for " << symbol\_ << endl;

}

double Currency::ConvertFromUSD(double amount)

{

double convertedAmount = amount \* conversionRate\_;

return convertedAmount;

}

string Currency::GetSymbol() const

{

return symbol\_;

}

double Currency::GetConversionRate() const

{

return conversionRate\_;

}

void Currency::SetConversionRate(double rate)

{

conversionRate\_ = rate;

}

Main.cpp

#include "Currency.h"

#include <vector>

#include <list>

#include <algorithm>

using std::vector;

using std::list;

using std::endl;

using std::cout;

int main()

{

vector<Currency> currencies;

Currency c1("JPY", 81.1);

Currency c2("USD", 1.0);

Currency c3("CAD", 1.1);

currencies.push\_back(c1);

currencies.push\_back(c2);

currencies.push\_back(c3);

}

# Throw Exception

InvalidCurrency.h

#pragma once

#include<stdexcept>

#include<string>

class InvalidCurrency :public std::runtime\_error

{

public:

InvalidCurrency(const std::string&msg)

:runtime\_error(msg)

{

}

};

Currency.cpp

switch (currencyType)

{

case EUR:

c = new Currency("EUR", 0.7901);

break;

case GBP:

c = new Currency("GBP", 0.6201);

break;

case CAD:

c = new Currency("CAD", 1.1150);

break;

case AUD:

c = new Currency("AUD", 1.1378);

break;

default:

{

std::ostringstream s;

s << "Currency Type" << currencyType << "is Invalid" << endl;

throw InvalidCurrency(s.str());

} }

# Circle Monte Carlo

Montecarlo.h

#ifndef \_MonteCarlo\_H

#define \_MonteCarlo\_H

using namespace std;

#include<cmath>

class MonteCarlo

{

public:

//construct params

MonteCarlo(const int& \_num\_sim);

//destructor

~MonteCarlo();

//monte carlo sim

double MCValue();

//params (only number of simulations here, given dimensions of circle/square

int num\_sim;

};

#endif

Montecarlo.cpp

#include "MonteCarlo.h"

#include <cmath>

#include <math.h>

//construct params

MonteCarlo::MonteCarlo(const int& \_num\_sim)

{

num\_sim = \_num\_sim;

}

//destructor

MonteCarlo::~MonteCarlo() {}

//in or out of the circle function

int InOrOut(double x, double y)

{

if (sqrt(x\*x + y\*y) < 1) // checking distance of coordinate from zero

{

return 1;

}

else

{

return 0;

}

}

double MonteCarlo::MCValue()

{

int z = 0;

double sum = 0.0;

for (int i = 0; i < num\_sim; i++)

{

double x = static\_cast<double>(std::rand()) / RAND\_MAX; // rand[0,1]

double y = static\_cast<double>(std::rand()) / RAND\_MAX; // rand[0,1]

int z = InOrOut(x, y); // use the function set above

sum = sum + z; // build up a sum of points found within circle

}

return (sum/num\_sim); // take the ratio to get pi/4

}

main.cpp

#include "MonteCarlo.h"

#include <iostream>

using namespace std;

int main()

{

// set number of simulations

double N1 = 100;

double N2 = 1000;

double N3 = 10000;

// initialise simulations

MonteCarlo mc1(N1);

MonteCarlo mc2(N2);

MonteCarlo mc3(N3);

cout << "N = 100: " << 4\*mc1.MCValue() << endl; // multiply by 4, as per formula

cout << "N = 1000: " << 4\*mc2.MCValue() << endl;

cout << "N = 10000: " << 4\*mc3.MCValue() << endl; // note: by 10000 simulation our approximation of pi is almost 3.14

}

# Vector Midterm

Main.cpp

#include <iostream>

#include <vector>

#include <cmath>

#include <math.h>

#include <numeric>

#include <algorithm> // used sorting for parts (c) to (e)

using std::cout;

using std::endl;

using std::vector;

using std::copy\_if; // used for part (f)

using std::find;

using std::for\_each; // used in part (g)

using std::accumulate; // used in part (g)

vector<int> v{5,4,3,2,1,0,1,2,3,4,5};

// i didn't use this given format for the questions. my answers are placed in the main() file.

// this function is used for part (f)

bool Greater3(int number)

{

if (number > 3) return true;

else return false;

}

// this function is used for part (g)

void cube(int elem)

{

int sum = 0;

int cubed = elem\*elem\*elem;

sum = sum + cubed;

cout << sum << endl;

}

int main()

{

///////////////////////////////////////////////////////////////////////////

//part (a)

cout << "Part (a) " << endl;

vector<int>::iterator iter;

for (iter = v.begin(); iter != v.end(); ++iter)

{

cout << \*iter << endl;

}

///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////

//part (b)

cout << "Part (b) " << endl;

// sort based on the range

vector<int>::reverse\_iterator r\_iter;

for (auto r\_iter = v.rbegin(); r\_iter != v.rend(); ++r\_iter)

{

cout << \*r\_iter << endl;

}

///////////////////////////////////////////////////////////////////////////

//for parts (c) to (e), sort the values to find max min and median

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

int N = v.size(); // get size of vector

///////////////////////////////////////////////////////////////////////////

//part (c)

cout << "Part (c) " << endl;

cout << v[N-1] << endl; // last value of the sorted array is the maximum

///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////

//part (d)

cout << "Part (d) " << endl;

cout << v[0] << endl; // first value of the sorted array is the minimum

///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////

//part (e)

cout << "Part (e) " << endl;

cout << v[(N-1)/2] << endl; // middle value of the sorted array is the median

///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////

//part (f)

cout << "Part (f) " << endl;

vector<int> unique;

for (int i = 0; i < N; i++)

{

if (v[i] > 3) // if the value in the vector is greater than 3, look into another 'unique' vector to see if it's already there

{

int M = unique.size();

if (M == 0) // there's nothing in the unique vector yet

{

unique.push\_back(v[i]);

}

else

{

for (int j = 0; j < M; j++) // for the size of the unique vector

{

if (v[i] == unique[j])

{

break; // break because it already exists

}

else if (j == M) // if we reach the end and code hasn't broken

{

unique.push\_back(v[i]); // add it to the unique vector

}

}

}

}

}

vector<int>::iterator uiter; // create new iterator

for (uiter = unique.begin(); uiter != unique.end(); ++uiter)

{

cout << \*uiter << endl; // output

}

///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////

//part (g)

cout << "Part (g) " << endl;

//for\_each(v.begin(), v.end(), cube);

int sum = 0;

for (int i = 0; i < N; i++) // N is found above using v.size()

{

sum = sum + v[i] \* v[i] \* v[i];

}

cout << sum << endl;

///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////

//part (h)

cout << "Part (h) " << endl;

int sum2 = 0;

for (int i = 0; i < N; i++) // N is found above using v.size()

{

sum2 = sum2 + v[i];

}

cout << "Mean: " << sum2/N << endl;

int sum3 = 0;

for (int i = 0; i < N; i++) // N is found above using v.size()

{

sum3 = sum3 + (v[i] - sum2/2)\* (v[i] - sum2 / 2);

}

cout << "Variance: " << sum3 / (N - 1) << endl;

///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////

//part (i)

cout << "Part (i) " << endl;

/\*for (iter = v.begin(); iter != v.end(); ++iter)

{

int sum = 0;

if (sum == 0 && \*iter == 4)

{

replace(v.begin(), v.end(), //range

4, // old value

44); // new value)

sum = sum + 1;

}

cout << \*iter << endl;

}\*/

vector<int> v{ 5,4,3,2,1,0,1,2,3,4,5 }; // reset vector v so we can do this part

int sum4 = 0;

for (int i = 0; i < N; i++) // N is found above using v.size()

{

if (sum4 == 0 && v[i] == 4)

{

v[i] = 44;

sum4 = sum4 + 1;

}

}

for (iter = v.begin(); iter != v.end(); ++iter)

{

cout << \*iter << endl;

}

///////////////////////////////////////////////////////////////////////////

}

# Date Class

Date.h

#ifndef DATE\_H

#define DATE\_H

#include <ostream>

class Date

{

friend std::ostream& operator<<(std::ostream& os, const Date& d);

public:

// constructors

Date(int mm, int dd, int yyyy);

Date(std::string dateString);

// overloading

std::ostream& operator<<(std::ostream& os); // overloading operator - option 1

int operator-(Date& other); // difference between two dates

Date operator+(int n);

// gets

int GetMonth();

int GetDay();

int GetYear();

// sets

void SetMonth(int mm);

void SetDay(int dd);

void SetYear(int yyyy);

// functions

Date today();

private:

int day\_;

int month\_;

int year\_;

};

std::ostream& operator<<(std::ostream& os, const Date& d); // overloading operator option 2

//Date operator+(int n, Date& d); // symmetry of operator

//Date operator+(Date& d, int n); // symmetry of operator

#endif

Date.cpp

#include "Date.h"

#include <stdexcept>

Date::Date(int mm, int dd, int yyyy)

:month\_(mm), day\_(dd), year\_(yyyy)

{

if (yyyy >= 1900 && yyyy <= 2050 && mm >= 1 && mm <= 12) // possible values of year month

{

if (mm == 2) // february

{

if (yyyy % 4 == 0) // leap year

{

if (dd <= 29) // days in a leap year in february is 29

{

month\_ = mm;

day\_ = dd;

year\_ = yyyy;

}

else

{

throw std::exception("invalid month");

}

}

else

{

if (dd <= 28)

{

month\_ = mm;

day\_ = dd;

year\_ = yyyy;

}

else

{

throw std::exception("invalid month");

}

}

}

if (mm == 4 || mm == 6 || mm == 9 || mm == 11) // months with 30 days

{

if (dd <= 30)

{

month\_ = mm;

day\_ = dd;

year\_ = yyyy;

}

else

{

throw std::exception("invalid month");

}

}

if (mm == 1 || mm == 3 || mm == 5 || mm == 7 || mm == 8 || mm == 10 || mm == 12) // months with 31 days

{

if (dd <= 31)

{

month\_ = mm;

day\_ = dd;

year\_ = yyyy;

}

else

{

throw std::exception("invalid month");

}

}

}

else

{

throw std::exception("invalid month");

}

}

Date::Date(std::string dateString)

{

}

std::ostream & Date::operator<<(std::ostream & os)

{

os << month\_ << "/" << day\_ << "/" << year\_;

return os;

}

int Date::operator-(Date & other)

{

if (month\_ < 3)

year\_--, month\_ += 12;

int epoch = 365 \* year\_ + year\_ / 4 - year\_ / 100 + year\_ / 400 + (153 \* month\_ - 457) / 5 + day\_ - 306;

if (other.month\_ < 3)

other.year\_--, other.month\_ += 12;

int epoch\_other = 365 \* other.year\_ + other.year\_ / 4 - other.year\_ / 100 + other.year\_ / 400 + (153 \* other.month\_ - 457) / 5 + other.day\_ - 306;

return epoch - epoch\_other;

}

Date Date::operator+(int n)

{

int y = year\_;

int m = month\_;

int d = day\_;

int jd = (1461 \* (y + 4800 + (m - 14) / 12)) / 4 + (367 \* (m - 2 - 12 \* ((m - 14) / 12))) / 12 - (3 \* ((y + 4900 + (m - 14) / 12) / 100)) / 4 + d - 32075;

double l = jd + 68569;

double nn = (4 \* l) / 146097;

l = l - (146097 \* nn + 3) / 4;

double i = (4000 \* (l + 1)) / 1461001;

l = l - (1461 \* i) / 4 + 31;

double j = (80 \* l) / 2447;

d = l - (2447 \* j) / 80;

l = j / 11;

m = j + 2 - (12 \* l);

y = 100 \* (nn - 49) + i + l;

return Date(m, d, y);

}

int Date::GetMonth()

{

return month\_;

}

int Date::GetDay()

{

return day\_;

}

int Date::GetYear()

{

return year\_;

}

void Date::SetMonth(int mm)

{

month\_ = mm;

}

void Date::SetDay(int dd)

{

day\_ = dd;

}

void Date::SetYear(int yyyy)

{

year\_ = yyyy;

}

Date Date::today()

{

return Date(month\_,day\_,year\_);

}

std::ostream& operator<<(std::ostream& os, const Date& d)

{

os << d.month\_ << "/" << d.day\_ << "/" << d.year\_;

return os;

}

Main.cpp

#include "Date.h"

#include <iostream>

#include <string>

#include <string.h>

using namespace std;

int main()

{

try

{

Date d1(4, 12, 2016);

//d1.operator<<(std::cout); // same as d1 << cout; // for friend function cout << d1 << endl;

d1 << cout << endl;

//Date d2(49, 45, -1);

Date d3(5, 20, 2016);

Date d5 = d3 + 25;

cout << "number of days difference: " << d3 - d1 << endl;

cout << "day plus " << 25 << " is: " << d5 << endl;

}

catch (std::exception& ex)

{

std::cout << "exception caught" << std::endl;

}

std::string dateString("4/15/2016");

Date d4(dateString);

}