NYU | TANDON SCHOOL OF ENGINEERING

Topic 5
Inheritance and Polymorphism – European Option Pricing Framework

Class Call and OptionCalculation

- In addition to the class BinomialTreeModel, we are going to encapsulate the CRR pricer for calculating European Call option into the class Call and OptionCalculation.
- Then we will expand the calculation to European Put and many other European options by using inheritance and polymorphism.

Option04.h

```
#pragma once
#include "BinomialTreeModel02.h"
namespace fre {
        class Call
        {
        private:
                int N;
                double K;
        public:
                Call() :N(0), K(0) {}
                Call(int N_, double K_) : N(N_), K(K_) {}
                Call(const Call& call) : N(call.N), K(call.K) {}
                ~Call() {}
                int GetN() const { return N; }
                double Payoff(double z) const;
        };
```

Option04.h (continue)

```
	class OptionCalculation
	{
	private:
		Call* pOption;
		OptionCalculation() : pOption(0) {}
		OptionCalculation(const OptionCalculation& optionCalculation)
				: pOption(optionCalculation.pOption) {}
	public:
		OptionCalculation(Call* pOption_) : pOption(pOption_) {}
		~OptionCalculation() {}
		double PriceByCRR(const BinomialTreeModel& Model);
	};
}
```

Option04.cpp

```
#pragma once

#include <iostream>
#include <cmath>
#include "Option04.h"
#include "BinomialTreeModel02.h"

using namespace std;

namespace fre {
    double Call::Payoff(double z) const
    {
        if (z > K) return z - K;
        return 0.0;
    }
```

Option04.cpp (continue)

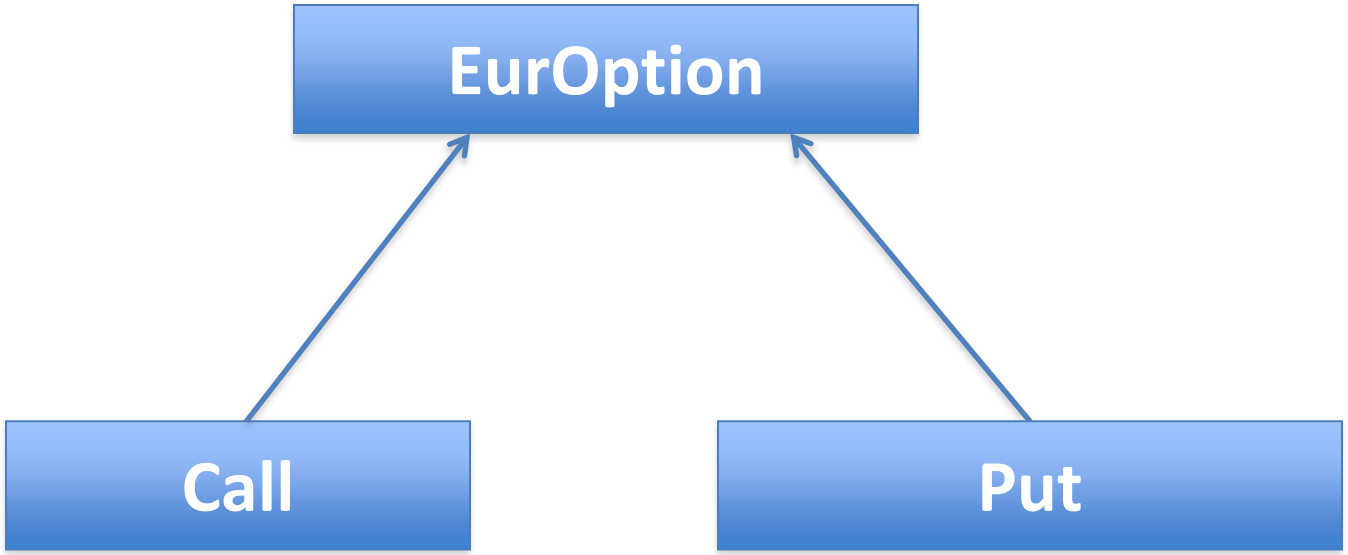
```
double OptionCalculation::PriceByCRR(const BinomialTreeModel& Model)
{   double optionPrice = 0.0;
    double q = Model.RiskNeutProb();
    int N = pOption->GetN();
    double* pPrice = new double[N + 1];
    memset(pPrice, 0, N + 1);
    for (int i = 0; i <= N; i++)
    {           pPrice[i] = pOption->Payoff(Model.CalculateAssetPrice(N, i));    }
    for (int n = N - 1; n >= 0; n--)
    {
                for (int i = 0; i <= n; i++)
                {       pPrice[i] = (q * pPrice[i + 1] + (1 - q) * pPrice[i]) / Model.GetR(); }
    }
    optionPrice = pPrice[0];
    delete [] pPrice;
    pPrice = nullptr;
    return optionPrice;
}
```

}

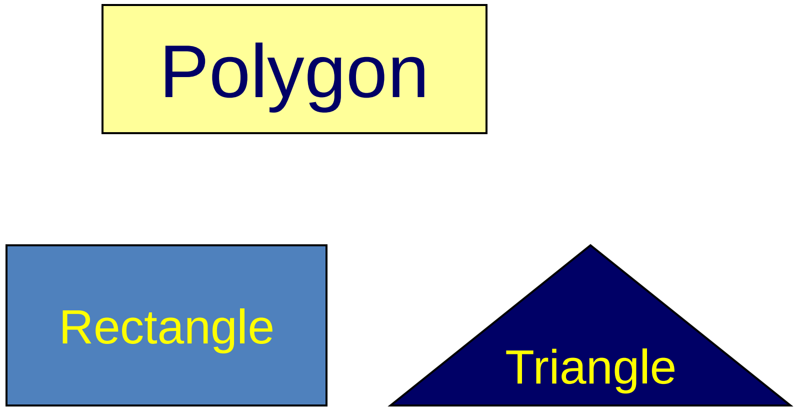
OptionPricer04.cpp

```
#include "BinomialTreeModel02.h"
#include "Option04.h"
#include <iostream>
#include <iomanip>
using namespace std;
using namespace fre;
int main()
{   int N = 8;
    double U = 1.15125, D = 0.86862, R = 1.00545;
    double S0 = 106.00, K = 100.00;
    BinomialTreeModel Model(S0, U, D, R);
    Call* pCall = new Call(N, K);
    OptionCalculation callCalculation(pCall);
    cout << "European call option price = " << fixed << setprecision(2) << callCalculation.PriceByCRR(Model) << endl;
    delete pCall;
    pCall = NULL;
    return 0;
}
// European call option price = 21.68
```

Inheritance



Inheritance Concept

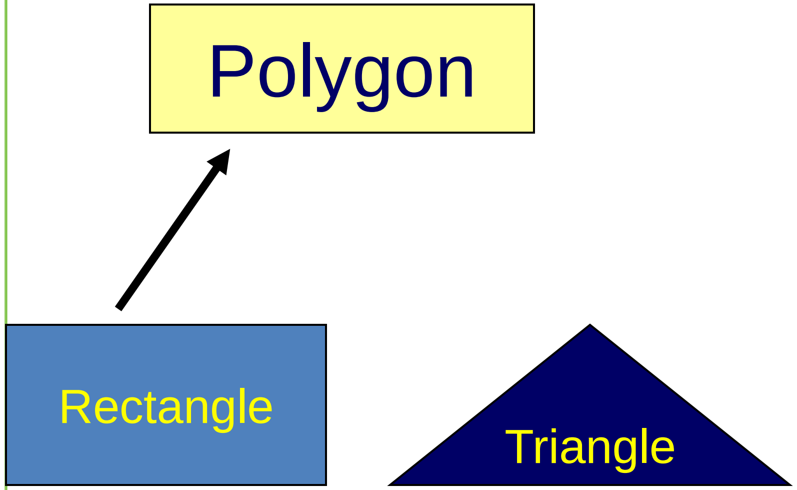


```
class Polygon {
  private:
   int numEdges;
   float xPos, yPos;
  public:
   void set(float x, float y, int nE);
};
```

```
class Rectangle {
   private:
     int numEdges;
     float xPos, yPos;
   public:
     void set(float x, float y, int nE);
     float area();
};
```

```
class Triangle {
   private:
     int numEdges;
    float xPos, yPos;
   public:
    void set(float x, float y, int nE);
     float area();
};
```

Inheritance Concept

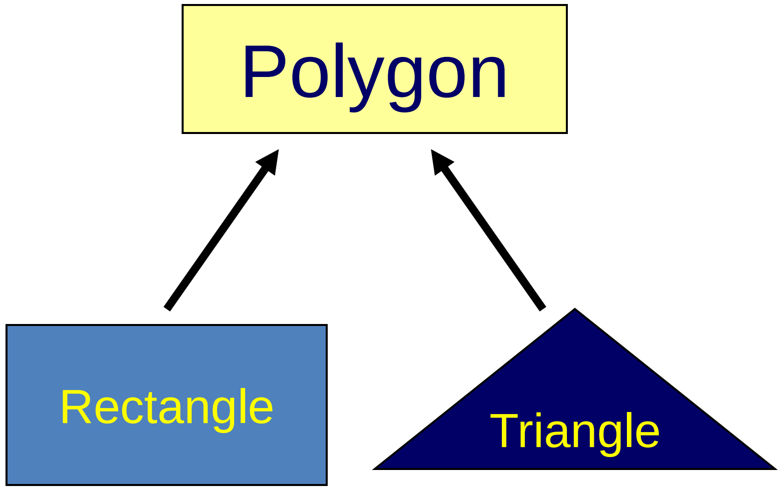


```
class Polygon {
      protected:
        int numEdges;
        float xPos, float yPos;
      public:
         void set(float x, float y, int nE);
};
```

```
class Rectangle : public Polygon {
      public:
         float area();
};
```

```
class Rectangle {
      protected:
        int numEdges;
        float xPos, float yPos;
      public:
         void set(float x, float y, int nE);
         float area();
};
```

Inheritance Concept



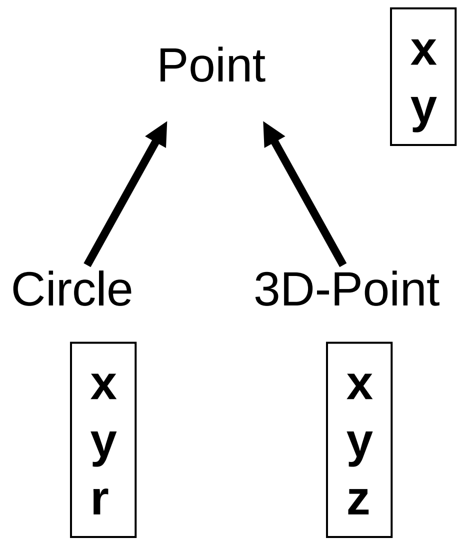
```
class Polygon {
      protected:
        int numEdges;
        float xPos, float yPos;
      public:
         void set(float x, float y, int nE);
};
```

```
class Triangle : public Polygon {
      public:
       float area();
};
```

$\Longleftrightarrow$

```
class Triangle {
      protected:
        int numEdges;
        float xPos, float yPos;
      public:
         void set(float x, float y, int nE);
         float area();
};
```

Inheritance Concept



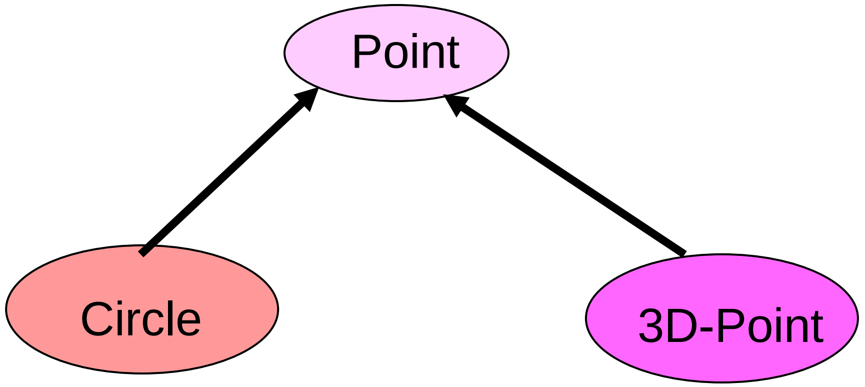
```
class Point
{
    protected:
      int x, y;
    public:
      void set (int a, int b);
};
```

```
class Circle : public Point
{
    private:
      double r;
};
```

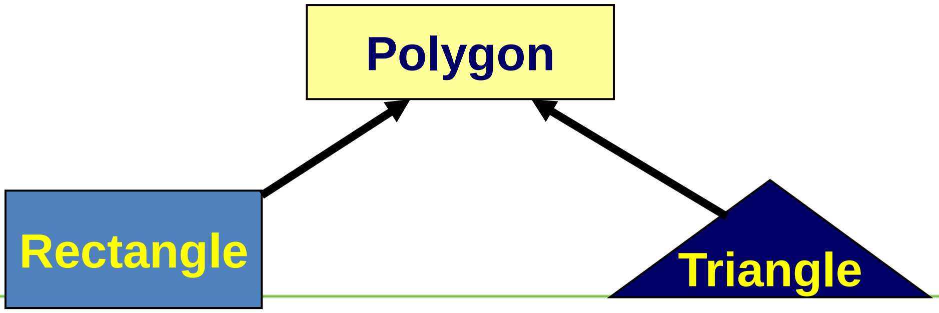
```
class 3D-Point: public Point
{
    private:
      int z;
};
```

Inheritance Concept

- Augmenting the original class



- Specializing the original class



Why Inheritance ?

Inheritance is a mechanism for

- building class types from existing class types
- defining new class types to be a
  - specialization
  - augmentation

  of existing types

Define a Class Hierarchy

- Syntax:

  class *DerivedClassName* : access-level *BaseClassName*

  where
  - access-level specifies the type of derivation
    - private by default, or
    - public
- Any class can serve as a base class
  - Thus a derived class can also be a base class

Class Derivation

Point is the base class of 3D-Point, while 3D-Point is the base class of Sphere

$$\begin{tikzcd}\text{Point} \\ \text{3D-Point} \arrow[u] \\ \text{Sphere} \arrow[u]\end{tikzcd}$$

```
class Point {
    protected:
      int x, y;
    public:
      void set (int a, int b);
};
```

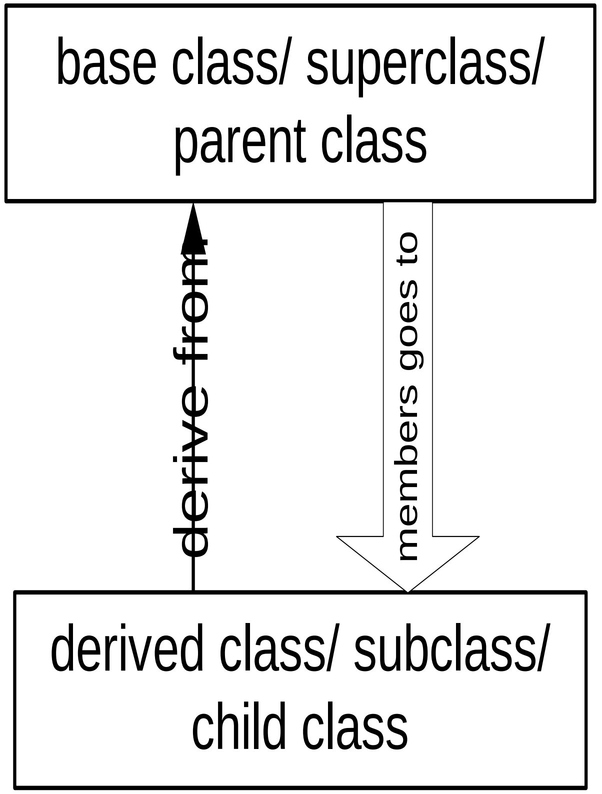
```
class 3D-Point : public Point {
    private:
      double z;
    … …
};
```

```
class Sphere : public 3D-Point {
    private:
      double r;
    … …
};
```

What to inherit?

- In principle, every member of a base class is inherited by a derived class
  - just with different access permission

Access Control Over the Members


base class/ superclass/ parent class
derive from
members goes to
derived class/ subclass/ child class

- Two levels of access control over class members
  - class definition
  - inheritance type

```
class Point {
    protected: int x, y;
    public: void set(int a, int b);
};
```

```
class Circle : public Point {
    … …
};
```

Access Rights of Parent Class

Type of Parent Data Members and Member Functions

	Private	Protected	Public
Child Class	No Access	Public	Public
Outside Parent/Child Classes	No Access	No Access	Public

What to inherit?

- In principle, every member of a base class is inherited by a derived class
  - just with different access permission
- However, there are exceptions for
  - constructor and destructor
  - operator=() member
  - friends

  Since all these functions are class-specific

Constructor Rules for Derived Classes

The default constructor and the destructor of the base class are always called when a new object of a derived class is created or destroyed.

```
class A {
  public:
    A ( )
      {cout<< “A:default”<<endl;}
    A (int a)
      {cout<<“A:parameter”<<endl;}
};
```

```
class B : public A
{
  public:
    B (int a)
        {cout<<“B”<<endl;}
};
```

```
B test(1);
```

output:

```
A:default
B
```

Constructor Rules for Derived Classes

You can also specify a constructor of the base class other than the default constructor

DerivedClassCon (derivedClass args) : BaseClassCon (baseClass args)
{ DerivedClass constructor body }

```
class A {
  public:
    A ( )
     {cout<< “A:default”<<endl;}
    A (int a)
     {cout<<“A:parameter”<<endl;}
};
```

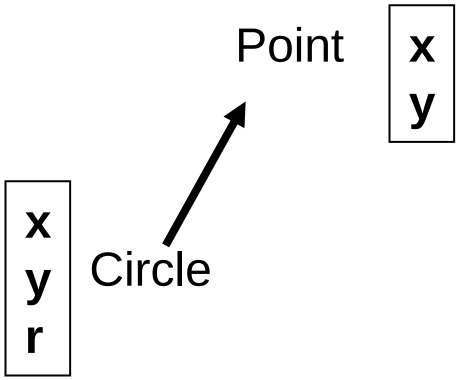
```
class C : public A {
  public:
    C (int a) : A(a)
       {cout<<“C”<<endl;}
};
```

```
C test(1);
```

output:
```
A:parameter
C
```

Define its Own Members

The derived class can also define its own members, in addition to the members inherited from the base class



```
class Circle : public Point {
    private:
      double r;
    public:
      void set_r(double c);
};
```

```
class Point {
    protected:
      int x, y;
    public:
      void set(int a, int b);
};
```

```
class Circle {
    protected:
      int x, y;
    private:
      double r;
    public:
      void set(int a, int b);
      void set_r(double c);
};
```

Even more …

- A derived class can override member functions defined in its parent class. With overriding,
  - the member function in the child class has the identical signature to the member function in the base class.
  - a child class implements its own version of a base class member function.

```
class A {
   protected:
     int x, y;
   public:
     void print ()
      {cout<<“From A”<<endl;}
};
```

```
class B : public A {
   public:
     void print ()
         {cout<<“From B”<<endl;}
};
```

Access a Method

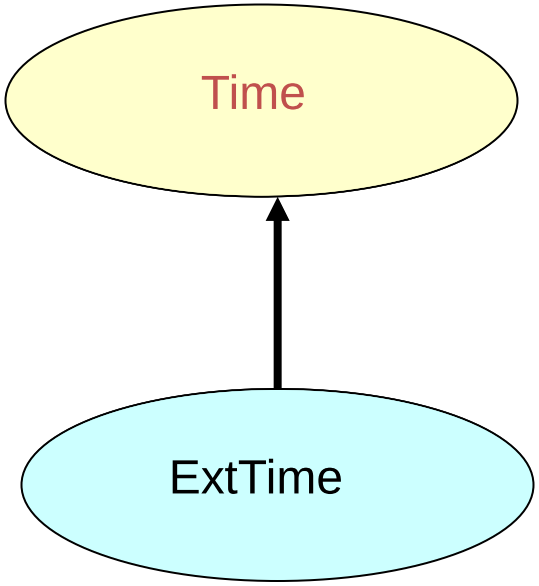
```
class Point {
    protected:
      int x, y;
    public:
      void set(int a, int b)
     {x=a; y=b;}
      void foo ();
      void print();
};
```

```
class Circle : public Point {
  private:  double r;
  public:
     void set (int a, int b, double c) {
         Point :: set(a, b); //same name function call
         r = c;
     }
     void print();  };
```

```
Point A;
A.set(30,50);  // from base class Point
A.print(); // from base class Point
```

```
Circle C;
C.set(10,10,100);   // from class Circle
C.foo ();  // from base class Point
C.print(); // from class Circle
```

Putting Them Together



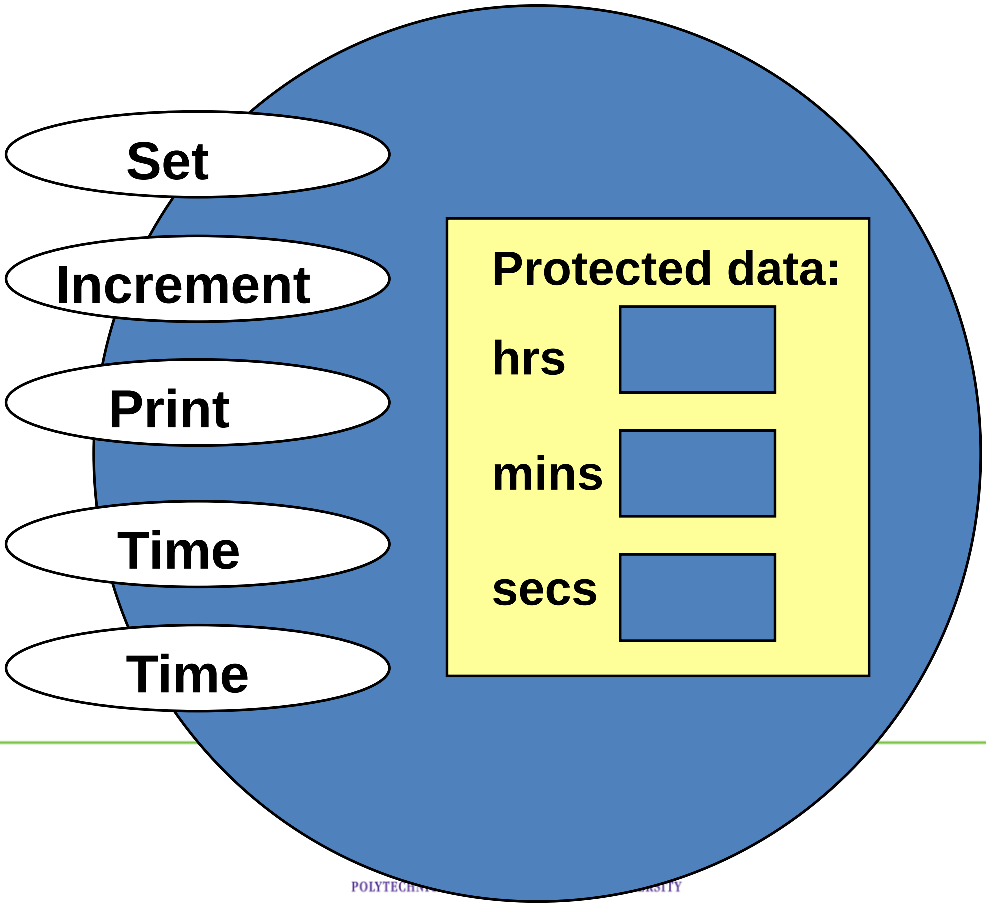
- Time is the base class
- ExtTime is the derived class with public inheritance
- The derived class can
  - inherit all members from the base class, except the constructors and destructor
  - access all public and protected members of the base class
  - define its private data member
  - provide its own constructor
  - define its public member functions
  - override functions inherited from the base class

class Time Specification

```
// SPECIFICATION   FILE              ( time.h)

class  Time
{
protected :
      int     hrs ;
      int     mins ;
      int     secs ;
public :
      Time( ) ;                                  //  default constructor
      Time(int initH, int initM, int initS) ;   //  constructor with parameters
      void  Set ( int h, int m, int s) ;
      void  Increment ( ) ;
      void  Print( )  const ;

} ;
```

Class Interface Diagram

Time class

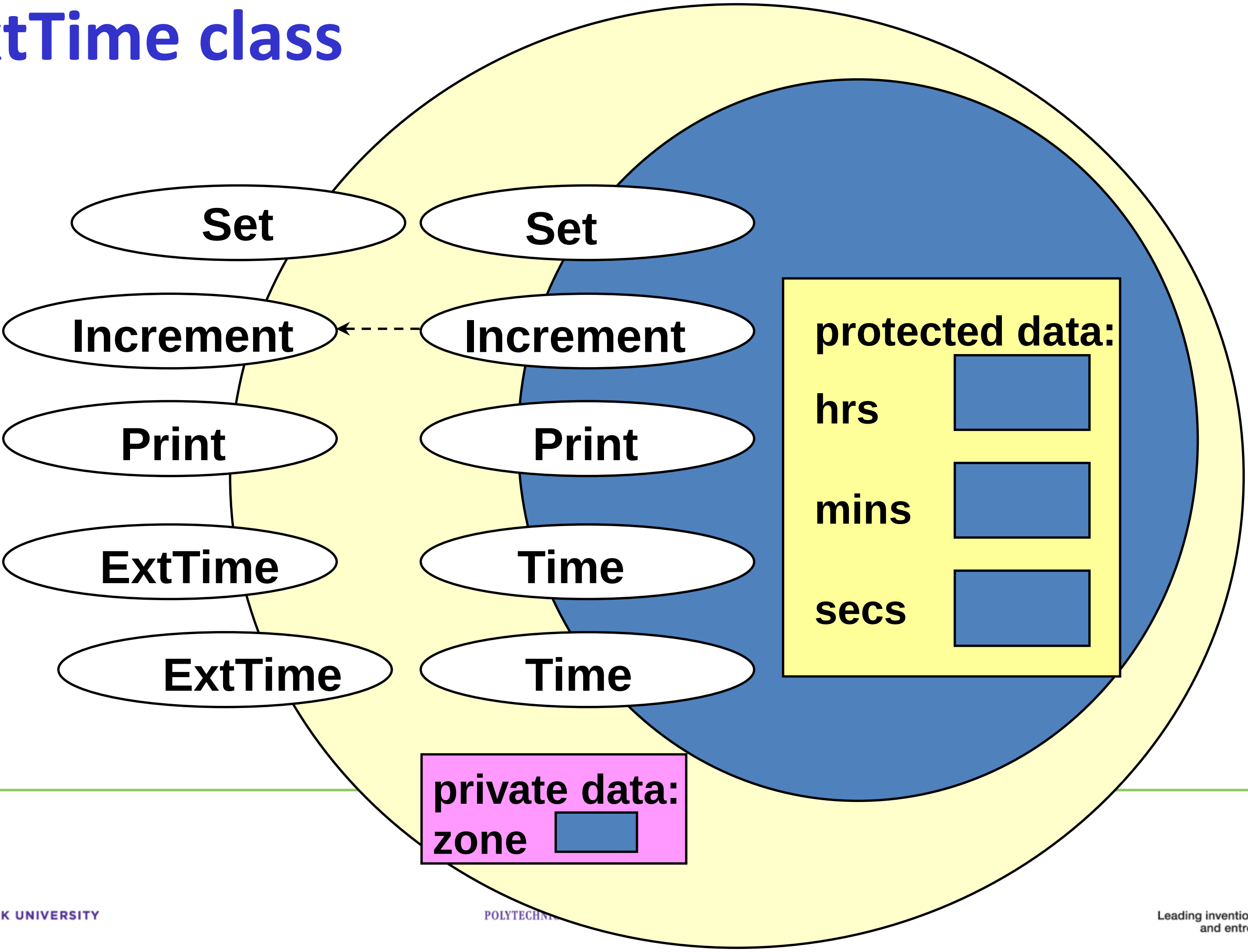


Derived Class ExtTime

```
// SPECIFICATION   FILE                 ( exttime.h)
#include "time.h"

enum  ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT } ;

class  ExtTime  :  public  Time  // Time is the base class and use public inheritance
{
public :

     void  Set (int h, int m, int s, ZoneType timeZone) ;
     void  Print( )  const;      //overridden
     ExtTime(int initH, int initM, int initS, ZoneType initZone) ;
     ExtTime();

private :
     ZoneType  zone ;   //  added data member

} ;
```

Class Interface Diagram

ExtTime class



Implementation of `ExtTime`

Default Constructor

```
ExtTime :: ExtTime (  )
{
      zone  =  EST ;
}
```

```
ExtTime et1;
```

et1

```
hrs = 0
mins = 0
secs = 0
zone = EST
```

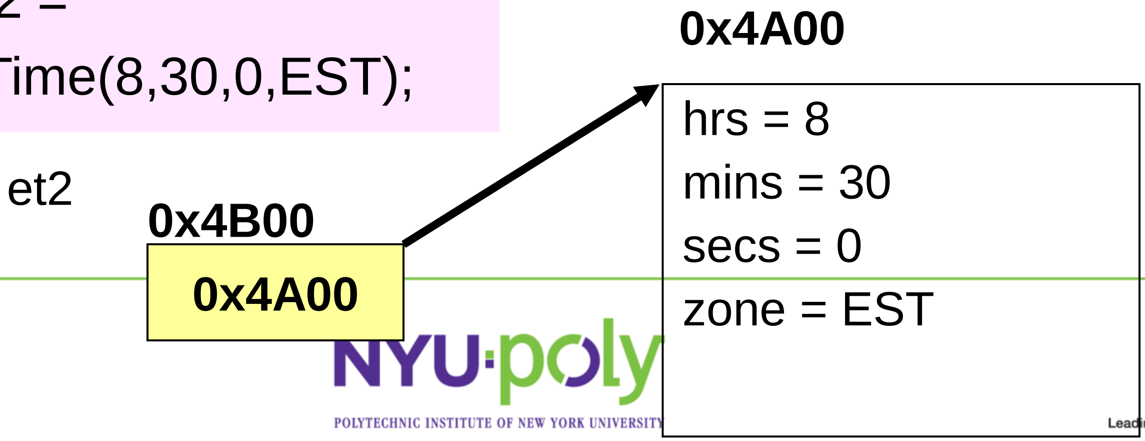
The default constructor of base class, Time(), is automatically called, when an ExtTime object is created.

Implementation of `ExtTime`

Another Constructor

```
ExtTime :: ExtTime (int initH, int initM, int initS, ZoneType initZone)
        : Time (initH, initM, initS) , zone(initZone)
        // constructor initializer
{
}
```

```
ExtTime *et2 =
    new ExtTime(8,30,0,EST);
```



NEW YORK UNIVERSITY

NYU·poly POLYTECHNIC INSTITUTE OF NEW YORK UNIVERSITY

Leading invention, innovation and entrepreneurship

Implementation of ExtTime

```
void  ExtTime :: Set (int h, int m, int s, ZoneType timeZone)
{
      Time :: Set (h, m, s);  // same name function call
      zone  = timeZone ;
}
void  ExtTime :: Print( )  const  // function overriding
{
   string  zoneString[8] =
        {“EST”, “CST”, MST”, “PST”, “EDT”, “CDT”, “MDT”, “PDT”} ;

  Time :: Print( ) ;
  cout  << zoneString[zone] << endl;
}
```

Working with ExtTime

```
#include  “exttime.h”
 … …
 int main()
{
  ExtTime    thisTime ( 8, 35, 0, PST ) ;
  ExtTime    thatTime ;                 // default constructor called

  thatTime.Print( ) ;                   // outputs 00:00:00 EST

  thatTime.Set (16, 49, 23, CDT) ;
  thatTime.Print( ) ;                   // outputs 16:49:23 CDT

  thisTime.Increment ( ) ;
  thisTime.Increment ( ) ;
  thisTime.Print( ) ;                   // outputs 08:35:02  PST
 }
```

Virtual Functions and Polymorphism

- By default, C++ matches a function call with the correct function definition at compile time. This is called ***static binding***. You can specify that the compiler match a function call with the correct function definition at run time; this is called ***dynamic binding***. You declare a function with the keyword ***virtual*** if you want the compiler to use dynamic binding for that specific function.
- A virtual function is a member function you may redefine for other derived classes and can ensure that the compiler will call the redefined virtual function for an object of the corresponding derived class, when you call that function with a pointer or reference to a base class of the object.
- A class that declares or inherits a virtual function is called a ***polymorphic*** class.

Why Polymorphism?--Review:
Time and ExtTime Example by Inheritance

```
void  Print  (Time   someTime )  //pass an object by value
{
        cout  <<  "Time is  " ;
        someTime.Print(  ) ;                          // Time :: Print()
        cout  <<  endl ;
}
```

Example of using Print() function:

```
Time        startTime ( 8, 30, 0 ) ;
ExtTime    endTime (10, 45, 0, CST) ;

Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

```
Time is  08:30:00
Time is  10:45:00
```

Static Binding

- When the type of a parameter is a parent class, the argument used can be:

  the same type as the parameter,

  or,

  any derived class type.

- Static binding is the compile-time determination of which function to call for a particular object based on the type of the formal parameter

- When pass-by-value is used, static binding occurs

Can we do better?

```
void  Print  (Time   someTime )  //pass an object by value
{
        cout  <<  “Time is  “ ;
        someTime.Print (  ) ;                                          // Time :: Print()
        cout  <<  endl ;
}
```

Example of using Print() function:

```
Time        startTime ( 8, 30, 0 ) ;
ExtTime    endTime (10, 45, 0, CST) ;

Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

```
Time is  08:30:00
Time is  10:45:00
```

Polymorphism – An Introduction

- *noun, the quality or state of being able to assume different forms* - Webster
- An essential feature of an OO Language
- It builds upon Inheritance
- Allows run-time interpretation of object type for a given class hierarchy
  - Also Known as "Late Binding"
- Implemented in C++ by using virtual functions

Dynamic Binding

- Is the run-time determination of which function to call for a particular object of a derived class based on the type of the argument
- Declaring a member function to be virtual instructs the compiler to generate code that guarantees dynamic binding
- Dynamic binding requires pass-by-reference or call by pointer.

Virtual Member Function

```
//  SPECIFICATION FILE                    ( time.h )

class  Time
{

public :

      . . .

      virtual   void   Print( ) const  ;             //  for dynamic binding
      virtual ~Time();                               // destructor

private :

      int          hrs ;
      int          mins ;
      int          secs ;
} ;
```

This is the way we like to see…

```
void  Print  (Time *   someTime )
{
     cout  <<  "Time is  " ;
     someTime->Print(  ) ;
     cout  <<  endl ;
}
```

```
Time         startTime( 8, 30, 0 ) ;
ExtTime      endTime(10, 45, 0, CST) ;
```

OUTPUT

```
Time is  08:30:00
Time is  10:45:00  CST
```

$$\text{Print (\&startTime) ;} \longrightarrow \text{Time::Print()}$$

$$\text{Print (\&endTime) ;} \longrightarrow \text{ExtTime::Print()}$$

Virtual Functions

- Virtual Functions overcome the problem of run time object determination
- Keyword virtual instructs the compiler to use late binding and delay the object interpretation
- How ?
  - Define a virtual function in the base class. The word virtual appears only in the base class
  - If a base class declares a virtual function, it must implement that function, even if the body is empty
  - Virtual function in base class stays virtual in all the derived classes
  - It can be overridden in the derived classes
  - But a derived class is not required to re-implement a virtual function. If it does not, the base class version is used

Polymorphism Summary:

- When you use virtual functions, compiler store additional information about the types of object available and created
- Polymorphism is supported with this additional overhead
- Important :
  - virtual functions work only with pointers/references
  - Not with objects even if the function is virtual
  - If a class declares any virtual functions, the destructor of the class should be declared as virtual as well.

Abstract Classes & Pure Virtual Functions

- Some classes exist logically but not physically.
- Example : Shape
  - `Shape s; // Legal but silly..!! : "Shapeless shape"`
  - Shape makes sense only as a base of some classes derived from it. Serves as a "category"
  - Hence instantiation of such a class must be prevented

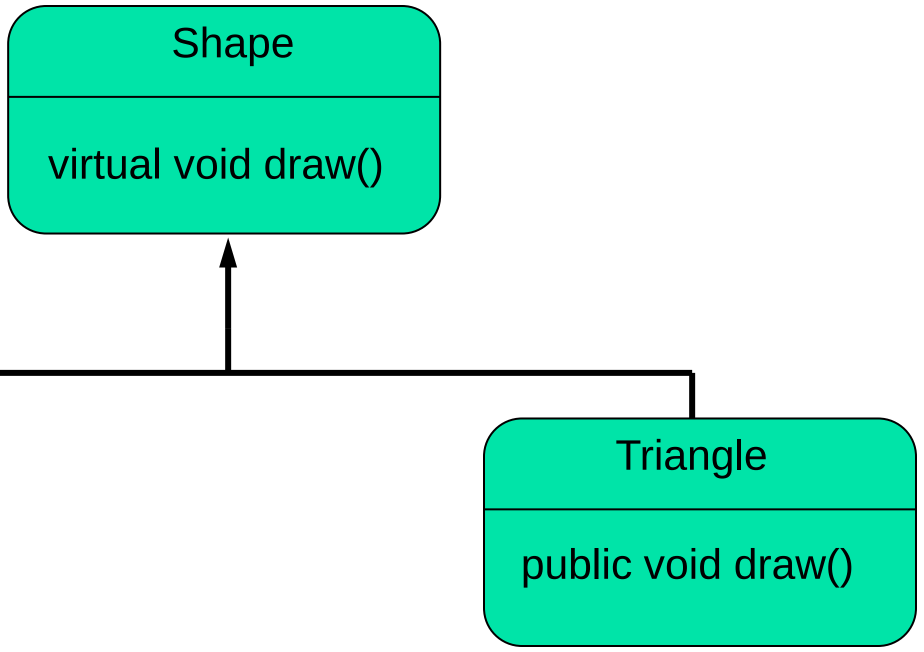
```
class Shape     //Abstract
{
  public :
  //Pure virtual Function
  virtual void draw() = 0;
}
```

A class with one or more pure virtual functions is an Abstract Class

Objects of abstract class can't be created

```
Shape s; // error : variable of an abstract class
```

Example


Shape
virtual void draw()
Circle
public void draw()
Triangle
public void draw()

- A pure virtual function not defined in the derived class remains a pure virtual function.
- Hence derived class also becomes abstract

```
class Circle : public Shape { //No draw() - Abstract
   public :
   void print(){
     cout << “I am a circle” << endl;
   }

class Rectangle : public Shape {
   public :
   void draw(){ // Override Shape::draw()
     cout << “Drawing Rectangle” << endl;
   }
```

```
Rectangle r;  // Valid: Rectangle is a concrete class
Circle c;     // error: Circle is an abstract class
```

Pure virtual functions : Summary

- Pure virtual functions are useful because they make explicit the abstractness of a class
- Tell both the user and the compiler how it was intended to be used
- Note : It is a good idea to keep the common code as close as possible to the root of you hierarchy

Summary ..continued

- It is still possible to provide definition of a pure virtual function in the base class
- The class will remain abstract, and functions must be redefined in the derived classes, but a common piece of code can be kept there to facilitate reuse
- In this case, they can not be declared inline

```
class Shape { //Abstract
public :
  virtual void draw() = 0;
};

// OK, not defined inline
void Shape::draw(){
 cout << "Shape" << endl;
}
```

```
class Rectangle : public Shape
{
  public :
   void draw(){
     Shape::draw(); //Reuse
     cout <<"Rectangle"<< endl;
}
```

Option05.h

```
#pragma once
#include "BinomialTreeModel02.h"
namespace fre {
    class EurOption
    {
    private:
        EurOption() : N(0) {}
        EurOption(const EurOption& option) : N(option.N) {}
    protected:
        int N;
    public:
        EurOption(int N_) : N(N_) {}
        int GetN() const { return N; }
        virtual double Payoff(double z) const = 0;
        virtual ~EurOption() = 0;
    };
```

```
class Call : public EurOption
{
private:
  double K;
public:
  Call(int N_, double K_) : EurOption(N_), K(K_) {}
  ~Call() {}
  double Payoff(double z) const;
};
class Put : public EurOption
{
private:
  double K;
public:
  Put(int N_, double K_) : EurOption(N_) , K(K_) {}
  ~Put() {}
  double Payoff(double z) const;
};
```

```
class OptionCalculation
{
private:
  EurOption* pOption;
  OptionCalculation() : pOption(0) {}
  OptionCalculation(const OptionCalculation& optionCalculation)
                                        : pOption(optionCalculation.pOption) {}
public:
  OptionCalculation(EurOption* pOption_) : pOption(pOption_) {}
  ~OptionCalculation() {}
  double PriceByCRR(const BinomialTreeModel& model);
};
}
```

Option05.cpp

```
#pragma once
#include <iostream>
#include <cmath>
#include "Option05.h"
#include "BinomialTreeModel02.h"
using namespace std;
namespace fre {
    EurOption::~EurOption() {}
    double Call::Payoff(double z) const
    {   if (z > K) return z - K;
        return 0.0;
    }
    double Put::Payoff(double z) const
    {   if (z < K) return K - z;
        return 0.0;
    }
    double OptionCalculation::PriceByCRR(const BinomialTreeModel& Model)
    {   double optionPrice = 0.0;
        double q = Model.RiskNeutProb();
        int N = pOption->GetN();
        double* pPrice = new double[N + 1];
        for (int i = 0; i <= N; i++)
        {       pPrice[i] = pOption->Payoff(Model.CalculateAssetPrice(N, i));
        }
        for (int n = N - 1; n >= 0; n--)
        {       for (int i = 0; i <= n; i++)
                {       pPrice[i] = (q * pPrice[i + 1] + (1 - q) * pPrice[i]) / Model.GetR();      }
        }
        optionPrice = pPrice[0];
        delete[] pPrice;
        pPrice = nullptr;
        return optionPrice;
    }
}
```

OptionPricer05.cpp

```
#include <iostream>
#include <iomanip>
#include "BinomialTreeModel02.h"
#include "Option05.h"
using namespace std;
using namespace fre;
int main()
{   int N = 8;
    double U = 1.15125, D = 0.86862, R = 1.00545;
    double S0 = 106.00, K = 100.00;
    BinomialTreeModel Model(S0, U, D, R);
    Call call(N, K);
    OptionCalculation callCalculation(&call);
    cout << "European call = " << fixed << setprecision(2) << callCalculation.PriceByCRR(Model) << endl;
    Put put(N, K);
    OptionCalculation putCalculation(&put);
    cout << "European put =  "<< fixed << setprecision(2) << putCalculation.PriceByCRR(Model) << endl;
    return 0;
}
// European call = 21.68
// European put = 11.43
```

Details of class EurOption:

- The **EurOption** class has a protected data member:
  - **N** of type **int** to hold the number of steps to expiry date.
- The **EurOption** class 4 public member functions:
  - Constructor with parameter
  - Pure virtual destructor.
  - Pure virtual function Payoff().
  - GetN() function to access protected data member N in the class OptionCalculation.
- The class containing a pure virtual function is called an ***abstract class***.
  - We have the abstract of European option represented by the abstract EurOption class!

- An ***abstract class*** is a class that is designed to be specifically used as a base class. An abstract class contains at least one ***pure virtual function***. You declare a pure virtual function by using a *pure specifier* (= 0) in the declaration of a virtual member function in the class declaration.
- You cannot declare an instance of an abstract base class; you can use it only as a base class when declaring other classes.
- An abstract class is used to define an implementation and is intended to be inherited from by concrete classes. It's a way of forcing a contract between the class designer and the users of that class.

- *The **protected** class members:*
  - *A protected nonstatic base class member can be accessed by members and friends of any classes derived from that base class by using one of the following:*
    - *A pointer to a directly or indirectly derived class*
    - *A reference to a directly or indirectly derived class*
    - *An object of a directly or indirectly derived class*
    - *If a class is derived privately from a base class, all protected base class members become private members of the derived class.*

Derived Classes: Call and Put

- class *Call:public EurOption*
  - *Introduce* ***Call*** *as a* ***child class*** *of the EurOption class. There is also a similar class for puts.*
  - *The child class Call overrides the pure virtual function Payoff(), so the class Call is a concrete class.*
  - *The derived classes, Call and Put, redefine the virtual function for its corresponding functionality:*

```
double Call::Payoff(double z)
{  if (z>K) return z-K;
   return 0.0;  }
```

```
double Put::Payoff(double z)
{  if (z<K) return K-z;
   return 0.0;  }
```

The class OptionCalculation

- The class ***OptionCalculation*** class has a pointer pointing to the base class EurOption.
- The pointer will be associated with either the address of a Call object or Put object when an object of OptionCalculation is created.
- In this way, the corresponding of Payoff function will be used in the function OptionCalculation::PriceByCRR() – ***Polymorphism.***

Homework Assignment

- Add the ability to price **bull spreads** and **bear spreads** by introducing new child classes BullSpread and BearSpread of EurOption class defined in Option05.h and Option05.cpp. Use the new child classes to price European bull and bear spreads in the OptionPricer program

$$h^{bull}(z) = \begin{cases} 0 & if\ z \le K_1 \\ z - K_1 & if\ K_1 < z < K_2 \\ K_2 - K_1 & if\ K_2 \le z \end{cases} \qquad h^{bear}(z) = \begin{cases} K_2 - K_1 & if\ z \le K_1 \\ K_2 - z & if\ K_1 < z < K_2 \\ 0 & if\ K_2 \le z \end{cases}$$

Pricing Double-digital Option

```
#pragma once
#include "Option05.h"
namespace fre {
    class DoubDigitOpt : public EurOption
    {
    private:
        double K1;
        double K2;
    public:
        DoubDigitOpt(int N_, double K1_, double K2_) :
                              EurOption(N_), K1(K1_), K2(K2_) {};
        ~DoubDigitOpt() {}
        double Payoff(double z) const;
    };
}
```

DoubleDigitOpt.cpp

```
#include "DoubDigitOpt.h"
namespace fre {
    double DoubDigitOpt::Payoff(double z) const
    {
      if (K1 < z && z < K2)
            return 1.0;
      return 0.0;
    }
}
```

OptionPricer05.cpp

```
#include <iostream>
#include <iomanip>
#include "BinomialTreeModel02.h"
#include "Option05.h“
#include "DoubDigitOpt.h"
using namespace std;
using namespace fre;
int main()
{
   int N = 8;
   double U = 1.15125, D = 0.86862, R = 1.00545;
   double S0 = 106.00, K1 = 100, K2 = 110;
   BinomialTreeModel Model(S0, U, D, R);
   DoubDigitOpt doubDigitOpt(N, K1, K2);
   OptionCalculation optCalculation(&doubDigitOpt);
   cout << "European double digit option price = "
      << fixed << setprecision(2) << optCalculation.PriceByCRR(Model) << endl;
   return 0;
}
```

Homework Assignment

- Add further payoffs, namely, **strangle**, and **butterfly** spreads, by means of child classes of the EurOption class placed in separate files, without changing anything in existing code, except for the necessary changes in cpp file for the main function, to price options with the newly introduced payoffs.

$$h^{strangle}(z) = \begin{cases} K_1 - z & if\ z \le K_1 \\ 0 & if\ K_1 < z \le K_2 \\ z - K_2 & if\ K_2 < z \end{cases} \qquad h^{butterfly} = \begin{cases} z - K_1 & if \quad K_1 < z \le \frac{K_1 + K_2}{2} \\ K_2 - z & if \quad \frac{K_1 + K_2}{2} < z \le K_2 \\ 0 & otherwise \end{cases}$$

References

- Numerical Methods in Finance with C++ (Mastering Mathematical Finance), by Maciej J. Capinski and Tomasz Zastawniak, Cambridge University Press, 2012, ISBN-10: 0521177162
- Introduction to software development, David Megias Jimenz, Jordi Mas, Josep Anton Perez Lopez and Lluis Ribas Xirgo, www.uoc.edu
- IBM XL C/C++ V8.0 for AIX, publib.boulder.ibm.com/infocenter/comphelp/v8v101
- Starting Out with C++ Early Objects, Seventh Edition, by Tony Gaddis, Judy Walters, and Godfrey Muganda, ISBN 0-13-607774-9, Addison-Wesley, 2010
- web.cse.ohio-state.edu/~neelam/courses/45922/Au05Somasund/