NYU | TANDON SCHOOL OF ENGINEERING

Topic 3
Object-Oriented Programming
Binomial Tree Mode Class

What need to be done?

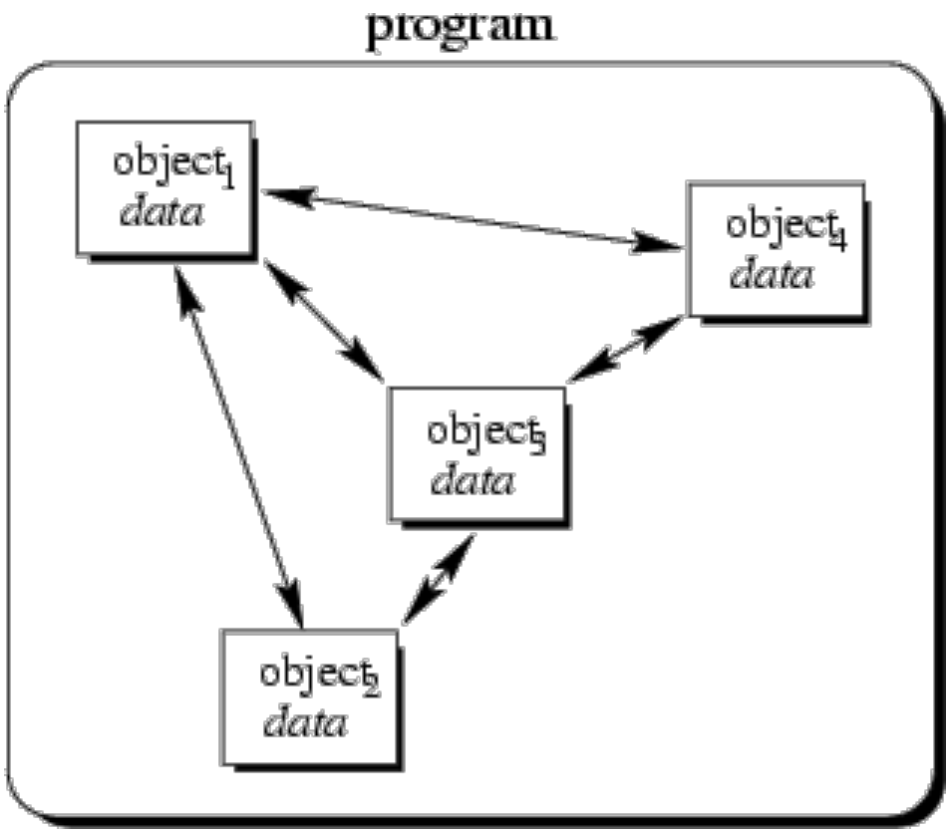
- Pricing other European options such as the double-digital option.
- Pricing American Options.
- Computing the option price based on the Black-Scholes formula.
- Pricing a path-dependent options such as an Asian option?

To Handle the increasing complexity, we need a more powerful way of C++ programming: Object-Oriented Programming

Object-Oriented Programming in C++

- We will implement a C++ object-oriented application based on what we have done in structured programming. In other words, we will recast the option pricer in the style of object-oriented programming. The C++ classes will reflect the relationships between real entities, namely the binomial model and European options of various kinds.

Object-Oriented Concept



Objects

An object is an encapsulation of both functions and data

- ***An object is an instance of a class***
  - *Classes represent real-world entities*
  - *A classes is a data type that define shared common properties or attributes of a real-world entity*
- ***An object has state***
  - *Its data have values at a particular time*
- ***An object has operations***
  - *associated set of operations on its data*

Object-Oriented Programming Languages

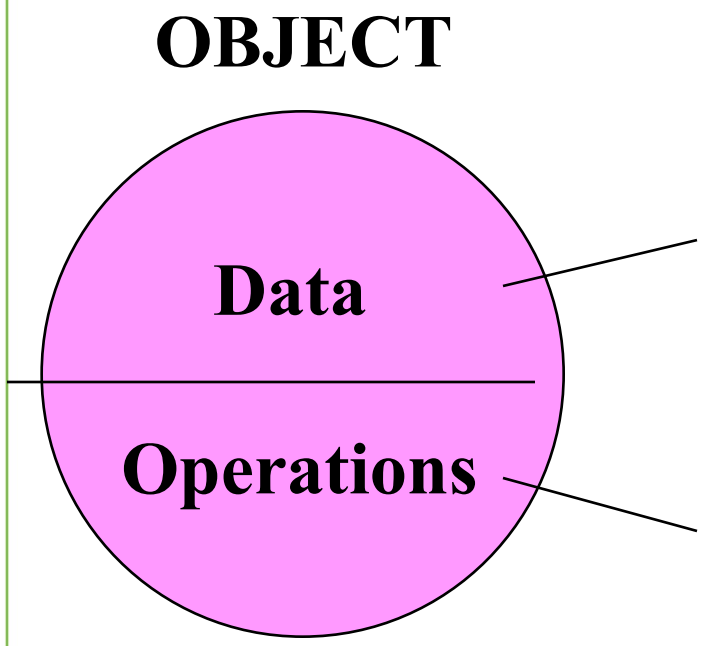
- Characteristics of OOPL:
  - Encapsulation
  - Inheritance
  - Polymorphism

Characteristics of OOPL

- **Encapsulation:** Combining data with operations
  - Data: represents the properties, the state, or characteristics of objects
  - Operations: permissible behaviors that are controlled through the member functions

  Data hiding: Data are only accessible through permissible operations
- **Inheritance:** Ability to derive new objects from old ones
  - permits objects of a more specific class to inherit the properties (data) and behaviors (functions) of a more general/base class
  - ability to define a hierarchical relationship between objects
- **Polymorphism:** Ability for different objects to interpret functions differently

Encapsulation



internal state (values of private data members)

set of methods (member functions)

Encapsulation

Let's look at the Rectangle through object-oriented eyes:

- Define a new type Rectangle (a class)
  - Data
    - width, length
  - Function
    - area()
- Create an instance of the class (an object)
- Calculate the area for an object of Rectangle

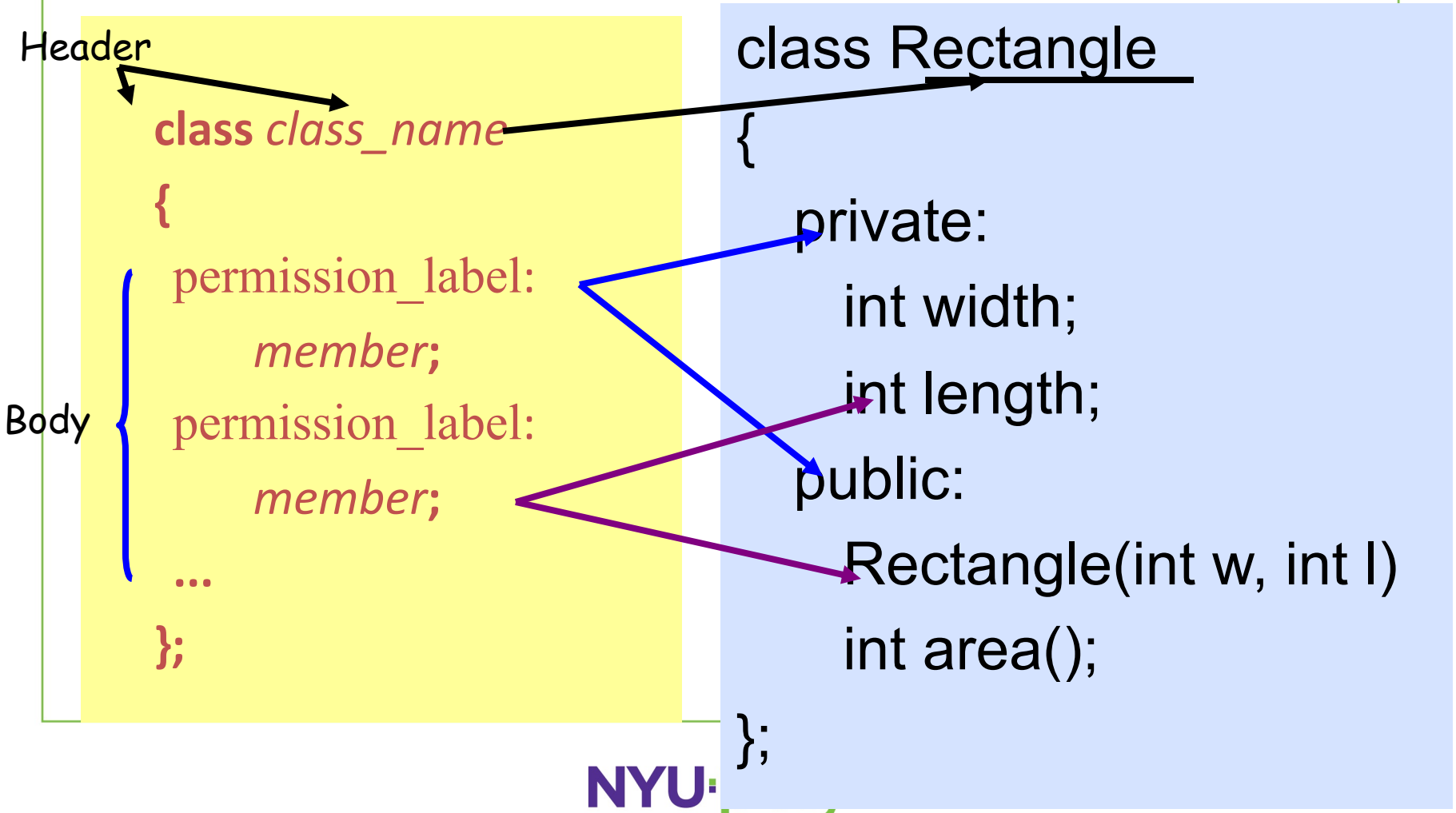
In C++, rather than writing independent functions, we define a class that encapsulates the data and corresponding operations on the data.

The Rectangle Class

```
class Rectangle
{
    private:
      int width, length;
    public:
      Rectangle(int w, int l)
     {
         width = w;
         length = l;
     }
     int area()
     {
         return width*length;
     }
};
```

```
int main()
{
    Rectangle rect(3, 5);
    cout << rect.area()<<endl;
    return 0;
}
```

Define a Class Type



Class Definition - Data Members

- Can be of any type, built-in or user-defined
- *non-static* data member
  - Each class object has its own copy
- *static* data member
  - Acts as a global variable
  - One copy per class type, e.g., counter
  - We will cover static data members in more details later

Class Definition - Member Functions

- Used to
  - access the values of the data members (**accessor**)
  - perform operations on the data members (**implementor**)
- Are declared inside the class body
- Their definition can be placed inside the class body, or outside the class body
- Can access both public and private members of the class
- Can be referred to using dot or arrow member access operator

Define a Member Function

```
class Rectangle
{
    private:
       int width, length;
    public:
       Rectangle() { width = 0; length = 0; }
       void set (int w, int l);
       int area() {return width*length; }
};
```

class name

member function name

```
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

scope operator

inline

```
r1.set(5,8);

rp->set(8,10);
```

Class Definition-Member Functions

- **const** member function
  - declaration
    - *return_type func_name* (*para_list*) const;
  - definition
    - *return_type func_name* (*para_list*) const { … }
    - *return_type class_name* :: *func_name* (*para_list*) const { … }
  - Makes no modification about the data members (safe function)
  - It is illegal for a const member function to modify a class data member

Const Member Function

```
class  Time
{
  private :
     int hrs, mins, secs ;

  public :
    Time(int h, int m, int s);
     void  Print ( )  const ;
} ;
```

function declaration

function definition

```
Time::Time(int h, int m, int s) : hrs(h), mins(m), secs(s)
{ }

void Time :: Print( ) const
{
    cout <<hrs << ":" << mins << ":" << secs << endl;
}
```

Class Definition - Access Control

- Information hiding
  - To prevent the internal representation from direct access from outside the class
- Access Specifiers
  - public
    - may be accessible from anywhere within a program
  - private
    - may be accessed only by the member functions, and friends of this class
  - protected
    - acts as public for derived classes
    - behaves as private for the rest of the program

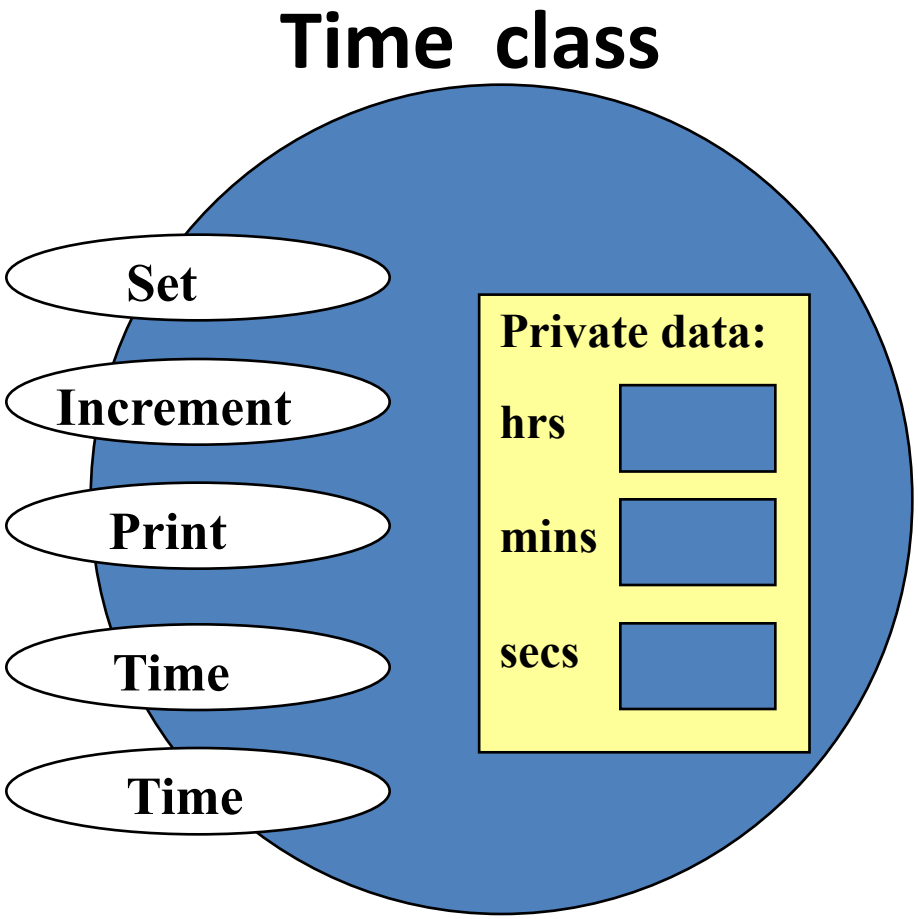
Class Definition - Access Control

- The default access specifier is *private*
- The data members are usually private or protected
- A **private** member function is a helper, may only be accessed by another member function of the same class
- The **public** member functions are part of the *class interface*
- Each access control section is optional, repeatable, and sections may occur in any order

class Time Specification

```
class  Time
{
 private :
     int          hrs ;
     int          mins ;
     int           secs ;
  public :
     Time    ( ) ;                                          // default constructor
     Time    ( int  initHrs, int  initMins,  int  initSecs ) ;   // constructor
     void    Set ( int  hours , int  minutes , int  seconds ) ;
     void    Increment ( ) ;
     void    Print ( )  const ;
} ;
```

Class Interface Diagram



Declaration of an Object

```
class Rectangle
{
    private:
      int width;
      int length;
    public:
     Rectangle() { width = 0; length = 0; };
     void set(int w, int l);
     int area() const { return width * length; }
};
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

```
main()
{
    Rectangle r1;
    Rectangle r2;

    r1.set(5, 8);
    cout<<r1.area()<<endl;

    r2.set(8,10);
    cout<<r2.area()<<endl;
}
```

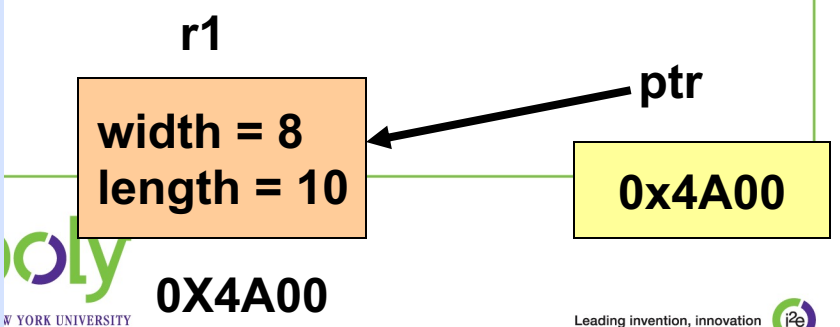
Declaration of an Object

```
class Rectangle
{
    private:
      int width;
      int length;
    public:
     Rectangle() { width = 0; length = 0; };
     void set(int w, int l);
     int area() const { return width * length; }
};
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

ptr is a pointer to a Rectangle object

```
main()
{
    Rectangle r1;
    r1.set(5, 8);     //dot notation

    Rectangle *ptr = nullptr;
    ptr = &r1;
    ptr->set(8,10);  //arrow notation
}
```



Declaration of an Object

```
class Rectangle
{
    private:
      int width;
      int length;
    public:
     Rectangle() { width = 0; length = 0; };
     void set(int w, int l);
     int area() const { return width * length; }
};
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

ptr is dynamically allocated

```
main()
{
    Rectangle *ptr = nullptr;
     ptr = new Rectangle();

     ptr->set(80,100);  //arrow notation

     delete ptr;
     ptr = nullptr;
}
```

ptr

nullptr

Object Initialization

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       Rectangle();
       Rectangle(int w, int l);
       Rectangle(const Rectangle &r);
       void set(int w, int l);
       int area();
}
```

- Default constructor
- Copy constructor
- Constructor with parameters

They are publicly accessible

Have the same name as the class

There is no return type

Are used to initialize class data members

They have different signatures

Object Initialization

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

When a class is declared with no constructors, the compiler automatically assumes default constructor and copy constructor for it.

- Default constructor

```
Rectangle :: Rectangle() : width(0),
                           length(0)
{ }
```

- Copy constructor

```
Rectangle :: Rectangle (const
    Rectangle & r) :  width(r.width),
                      length(r.length)
{ };
```

Object Initialization

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
}
```

- Initialize with default constructor

```
Rectangle r1;
Rectangle *r3 = new Rectangle();
```

- Initialize with copy constructor

```
Rectangle r4;
r4.set(60,80);

Rectangle r5 = r4;
Rectangle r6(r4);

Rectangle *r7 = new Rectangle(r4);
```

Object Initialization

```
class Rectangle
{
    private:
      int width;
      int length;
    public:
      Rectangle(int w, int l);
      void set(int w, int l);
      int area();
}
```

If any constructor with any number of parameters is declared, no default constructor will exist, unless you define it.

```
Rectangle r4;     // error
```

- Initialize with constructor

```
Rectangle r5(60,80);
Rectangle *r6 = new Rectangle(60,80);
```

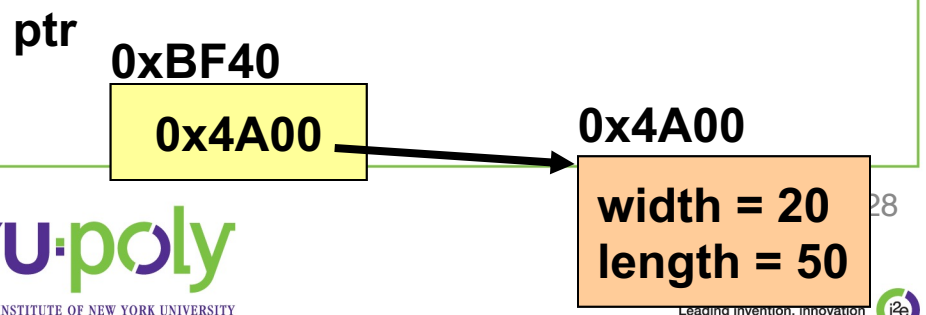
Object Initialization

```
class Rectangle
{
    private:
       int width;
       int length;
    public:
       Rectangle();
       Rectangle(int w, int l);
       void set(int w, int l);
       int area();
}
```

Write your own constructors

```
Rectangle :: Rectangle()
{
    width = 20;
    length = 50;
};
```

```
Rectangle  *ptr = new Rectangle();
```



Object Initialization

```
class Account
{
    private:
      char *name;
      double balance;
      unsigned int id;
    public:
      Account();
      Account(const Account &a);
      Account(char * const person);
}
```

With constructors, we have more control over the data members

```
Account :: Account(const Account &a)
    : balance(a.balance), id(a.id)
{
    name = new char[strlen(a.name)+1];
    strcpy (name, a.name);
};
```

```
Account :: Account() : name(NULL),
                balance(0.0), id(0)
{};
```

```
Account :: Account(char *const person)
: balance(0.0), id(0)
{
    name = new char[strlen(person)+1];
    strcpy (name, person);
};
```

So far, …

- An object can be initialized by a class constructor
  - default constructor
  - copy constructor
  - constructor with parameters
- Resources are allocated when an object is initialized
- Resources should be revoked when an object is about to end its lifetime

Cleanup of An Object

```
class Account
{
    private:
      char *name;
      double balance;
      unsigned int id;  //unique
    public:
      Account();
      Account(const Account &a);
      Account(char * const person);
      ~Account();
}
```

Destructor

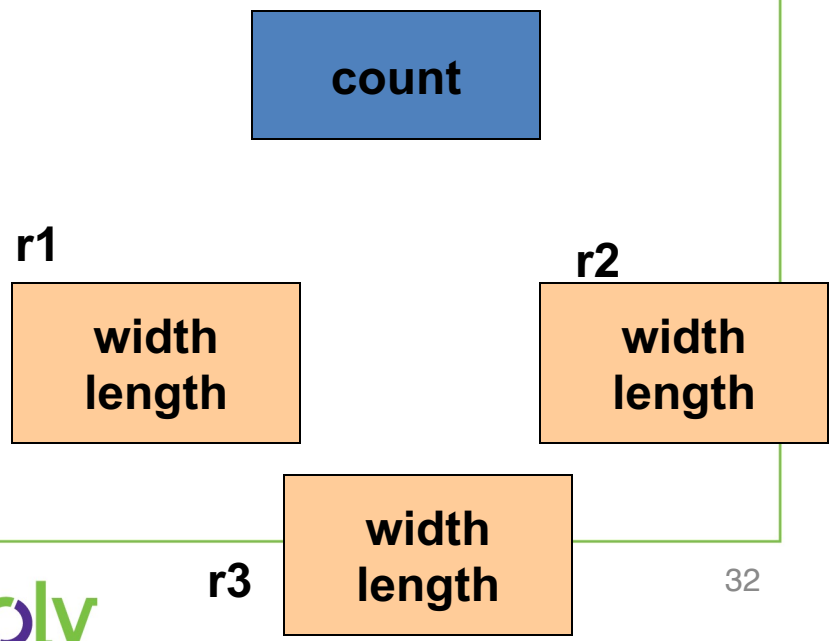
```
Account :: ~Account()
{
    if ( name != NULL)
          delete[] name;
}
```

- Its name is the class name preceded by a ~ (tilde)
- It has no argument
- It is used to release dynamically allocated memory and to perform other "cleanup" activities
- It is executed automatically when the object goes out of scope
- If a class has no destructor, the compiler automatically adds destructor

Static Data Member

```
class Rectangle
{
   private:
      int width;
      int length;
      static int count;
   public:
      Rectangle();
      void set(int w, int l);
      int area();
}
```

```
Rectangle  r1;
Rectangle  r2;
Rectangle  r3;
```



```
#include <iostream>
using namespace std;
class Rectangle
{
private:
   int length, width;     // non static data members
   static int count;      // static data member
public:
   // default constructor
   Rectangle() :length(0), width(0)
   {
      count++;
      cout << "Length = " << length << " Width = " << width << " Count = " << count << endl;
   }

  // constructor with parameters
   Rectangle(int length_, int width_) :length(length_), width(width_)
   {
      count++;
      cout << "Length = " << length << " Width = " << width << " Count = " << count << endl;
   }
```

```
    // copy constructor
    Rectangle(const Rectangle& R) :length(R.length), width(R.width)
    {
       count++;
       cout << "Length = " << length << " Width = " << width << " Count = " << count << endl;
    }
    // destructor
    ~Rectangle()
    {
       count--;
       cout << "Length = " << length << " Width = " << width << " Count = " << count << endl;
    }
    static int GetCount() { return count; }     // count is private
};
```

```
void RectangleQuestion(void)
{
   Rectangle R1, R2(10, 2), R3 = R2;      // R1 is created default constructor,
                                          // R2 by constructor with parameters,
                                          // R3 by copy constructor
   cout << "The number of rectangles = " << Rectangle::GetCount() << endl;   // 3
}  // destructor will be invoked for each object; last created object will be the 1st to destroy

int Rectangle::count = 0;      // static data member must be initialized outside constructor

int main(void)
{
   RectangleQuestion();    // after the function is completed, all 3 objects are destroyed
   return 0;
}
```

```
Length = 0 Width = 0 Count = 1      <- R1
Length = 10 Width = 2 Count = 2     <- R2
Length = 10 Width = 2 Count = 3     <- R3
The number of rectangles = 3

//The rest 3 lines are due to destructor is invoked for each object
Length = 10 Width = 2 Count = 2     ~Rectangle() for R3
Length = 10 Width = 2 Count = 1     ~Rectangle() for R2
Length = 0 Width = 0 Count = 0      ~Rectangle() for R1
Program ended with exit code: 0
```

Pointers to Class Objects and Structures

- Can create pointers to objects and structure variables

```
struct Student {…};
class Square {…};
Student stu1;
Student *stuPtr = &stu1;
Square sq1[4];
Square *squarePtr = &sq1[0];
```

- Need `()` when using `*` and `.`

```
(*stuPtr).studentID = 12204;
```

Structure Pointer Operator

- Simpler notation than **`(*ptr).member`**
- Use the form **`ptr->member:`**

```
stuPtr->studentID = 12204;
squarePtr->setSide(14);
```

in place of the form **`(*ptr).member:`**

```
(*stuPtr).studentID = 12204;
(*squarePtr).setSide(14);
```

Dynamic Memory with Objects

- Can allocate dynamic structure variables and objects using pointers:

```
stuPtr = new Student;
```

- Can pass values to constructor:

```
squarePtr = new Square(17);
```

- **`delete`** causes destructor to be invoked:

```
delete squarePtr;
```

Selecting Members of Objects

- Situation: A structure/object contains a pointer as a member. There is also a pointer to the structure/object.
- Problem: How do we access the pointer member via the structure/object pointer?

```
struct GradeList
{   string courseNum;
    int * grades;
}
GradeList test1, *testPtr = &test1;
```

Selecting Members of Objects

Expression	Meaning
`testPtr->grades`	Access the grades pointer in `test1`. This is the same as `(*testPtr).grades`
`*testPtr->grades`	Access the value pointed at by `testPtr->grades`. This is the same as `*(*testPtr).grades`
`*test1.grades`	Access the value pointed at by `test1.grades`

Our First Class, BinomialTreeModel02.h

```
#pragma once
namespace fre {
    class BinomialTreeModel
    {
    private:
      double S0;
      double U;
      double D;
      double R;
```

BinomialTreeModel02.h (Continue)

```
public:
    BinomialTreeModel()  :S0(0), U(0), D(0), R(0) {}
    BinomialTreeModel(double S0_, double U_, double D_, double R_)
                                :S0(S0_), U(U_), D(D_), R(R_) {}
    BinomialTreeModel(const BinomialTreeModel& B)
                                :S0(B.S0), U(B.U), D(B.D), R(B.R) {}
    ~BinomialTreeModel() {}
    double RiskNeutProb() const;
    double CalculateAssetPrice(int n, int i) const;
```

BinomialTreeModel02.h (Continue)

```
void UpdateBinomialTreeModel(double S0_, double U_,
                                        double D_, double R_);
 int ValidateInputData() const;
 int GetInputData();
 double GetS0() const { return S0; }
 double GetU() const { return U; }
 double GetD() const { return D; }
 double GetR() const { return R; }
};
```

BinomialTreeModel02.cpp

```
#include "BinomialTreeModel02.h"
#include <iostream>
#include <cmath>
using namespace std;
namespace fre {
   double BinomialTreeModel::RiskNeutProb() const
   {   return (R - D) / (U - D);
   }
   double BinomialTreeModel::CalculateAssetPrice(int n, int i) const
   {   return S0 * pow(U, i) * pow(D, n - i);
   }
   void BinomialTreeModel::UpdateBinomialTreeModel(double S0_, double U_,
                                                   double D_, double R_)
   {   S0 = S0_; U = U_; D = D_; R = R_;
   }
```

BinomialTreeModel02.cpp (Continue)

```
int BinomialTreeModel::GetInputData()
{
  //entering data
  cout << "Enter S0: "; cin >> S0;
  cout << "Enter U:  "; cin >> U;
  cout << "Enter D:  "; cin >> D;
  cout << "Enter R:  "; cin >> R;
  cout << endl;

  //making sure that 0<S0, -1<D<U, -1<R
  if (S0<=0.0 || U<=-1.0 || D<=-1.0 || U<=D ||
      R<=-1.0)
  {
    cout << "Illegal data ranges" << endl;
    cout << "Terminating program" << endl;
    return -1;
  }
```

```
  //checking for arbitrage
  if (R>=U || R<=D)
  {
    cout << "Arbitrage exists" << endl;
    cout << "Terminating program" << endl;
    return -1;
  }

  cout << "Input data checked" << endl;
  cout << "There is no arbitrage" << endl << endl;

  return 0;
}
```

BinomialTreeModel02.cpp (Continue)

```
int BinomialTreeModel::ValidateInputData() const
{
      //making sure that S0>0, U>D>0, R>0
      if (S0 <= 0.0 || U <= 0.0 || D <= 0.0
                    || U <= D || R <= 0.0)
      {
         cout << "Illegal data ranges" << endl;
         cout << "Terminating program" << endl;
         return -1;
      }
```

```
      //checking for arbitrage
       if (R >= U || U <= D)
       {
          cout << "Arbitrage exists" << endl;
          cout << "Terminating program" << endl;
          return -1;
       }

       cout << "Input data checked" << endl;
       cout << "There is no arbitrage" << endl <<
      endl;

       return 0;
   }
}
```

Option03.h

```
#pragma once
#include "BinomialTreeModel02.h“

namespace fre {
      //inputting and displaying option data
      int GetInputData(int& N, double& K);

      //pricing European option
      double * PriceByCRR(const BinomialTreeModel & Model, int N, double K,
                                    double (*Payoff)(double z, double K));

      //computing Call Payoff
      double CallPayoff(double z, double K);

      //computing Put Payoff
      double PutPayoff(double z, double K);
}
```

Option03.cpp

```
#include <cmath>
using namespace std;

namespace fre {
   int GetInputData(int& N, double& K)
   {
      cout << "Enter steps to expiry N: "; cin >> N;
      cout << "Enter strike price K:    "; cin >> K;
      cout << endl;
      return 0;
   }
```

Option03.cpp (Continue)

```
double * PriceByCRR(const BinomialTreeModel & Model, int N, double K,
                    double (*Payoff)(double z, double K))
{
   double q = Model.RiskNeutProb();
   double *Price = new double[N+1];
   memset(Price, 0, sizeof(Price));
   for (int i = 0; i <= N; i++)
   {
      Price[i] = Payoff(Model.CalculateAssetPrice(N, i), K);
   }
   for (int n = N - 1; n >= 0; n--)
   {
      for (int i = 0; i <= n; i++)
      {
         Price[i] = (q * Price[i + 1] + (1 - q) * Price[i]) / Model.GetR();
      }
   }
   return Price;
}
```

Option03.cpp (Continue)

```
    double CallPayoff(double z, double K)
    {
        if (z > K) return z - K;
        return 0.0;
    }

    double PutPayoff(double z, double K)
    {
        if (z < K) return K - z;
        return 0.0;
    }
}
```

Notes for Option03.cpp

- A reference to an **object** of class BinomialTreeModel is passed to the function instead of the four variables, S0, U, D, and R of type double – ***data encapsulation.***
- The ***Model*** is read-only in the function PriceByCRR().
- Dot operator such as ***Model. RiskNeutProb()*** is used on Model in the function PriceByCRR().

OptionPricer03.cpp

```
// OptionPricer03.cpp
#include "BinomialTreeModel02.h"
#include "Option03.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 BinModel(S0, U, D, R);
   if (BinModel.ValidateInputData() != 0) return -1;
```

OptionPricer03.cpp (Continue)

```
    double* optionPrice = NULL;
    optionPrice = PriceByCRR(BinModel, N, K, CallPayoff);
    cout << "European Call option price = " << optionPrice[0] << endl;
    optionPrice = PriceByCRR(BinModel, N, K, PutPayoff);
    cout << "European Put option price = " << optionPrice[0] << endl;
    delete optionPrice;
    optionPrice = NULL;
    return 0;
}
/*
Input data checked
There is no arbitrage

European Call option price = 21.6811
European Put option price = 11.4261
*/
```

Homework Assignment

- A definite integral can be computed numerically by trapezoidal approximation:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\sum_{k=1}^{N}\left(f(x_{k+1}) + f(x_k)\right)$$
$$= \frac{b-a}{2N}\left(f(x_1) + 2f(x_2) + 2f(x_3) + \ldots + 2f(x_N) + f(x_{N+1})\right).$$

Where $x_k = a + kh$ for $k = 1, \ldots N$. Write a class called ***DefInt*** to compute the trapezoidal approximation for a given function f(x). The class should contain the following:

1)Private members to hold the values of the integration limits a and b and a pointer to the function f.

2)A constructor function such that the integration limits a, b and the pointer to the function f can be initiated at the time of creating an object of the class such as ***DefInt MyInt(a, b, f)***

3)A public function ***ByTrapzoid()*** taking N as an argument and returning the trapezoidal approximation to the integral when called by ***MyInt.ByTrapzoidal(N).***

4)You may also want to include another public function ***BySimpson()*** to compute the Simpson approximation to the integral (look it up in the literature).

References

- Numerical Methods in Finance with C++ (Mastering Mathematical Finance), by Maciej J. Capinski and Tomasz Zastawniak, Cambridge University Press, 2012, ISBN-10: 0521177162
- 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/