NYU | TANDON SCHOOL OF ENGINEERING

Topic 2
Function Pointers
CRR Pricer for European Call and Put

Pointers

- We will enhance CRR Pricer with function pointer to price European Call and Put options.

- A pointer is a variable used to store an address in computer memory.

Address Operator

- Each variable in a program is stored at a unique address in memory

- Use the address operator (reference operator) **&** to get the address of a variable:

```
double dPrice = 21.68;
cout << &dPrice; // prints address in hexadecimal
```

Pointer Variables

- **Pointer variable (pointer):** variable that holds the address of a variable, i.e., pointers provide a way to access memory locations
- Definition:

```
double *dPtr = &dPrice;
```

- It means that pointer variable **dPtr** holds the address of the double variable dPrice or **dPtr** points to the address of the double variable dPrice.
- Spacing in point variable definition does not matter:

```
double * dPtr;
double*  dPtr;
```

More on Pointer Variables

- Initialization:

```
double dPrice = 21.68;
double * dPtr = &dPrice;
```

- Memory layout:

address of `dPrice`: `0x41B0`

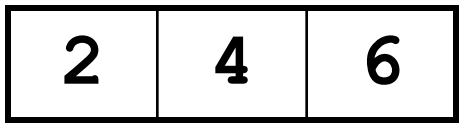
- Access the variable **dPrice** using pointer **dPtr** with **indirection operator (dereference operator) ***:

```
cout << dPtr;        // print 0x41B0
cout << *dPtr;       // print 21.68
```

The Relationship Between Arrays and Pointers

- Array name is starting address of array

```
int iArray[] = {2, 4, 6};
```



starting address of `iArray`: `0x4F80`

```
cout << iArray;    // print 0x4F80
cout << iArray[0]; // print 2
```

The Relationship Between Arrays and Pointers

- Array name can be used as a pointer constant

```
int iArray[] = {2, 4, 6};
cout << *iArray;      // print 2
```

- Pointer can be used as an array name

```
int *arrayPtr = iArray;
cout << arrayPtr[1]; // print 4
```

Pointers in Expressions

- Given:

```
int iArray[]={2, 4, 6};
int *arrayPtr = iArray;
```

- What is **`arrayPtr + 1`**?
  - It means (address in **`arrayPtr`**) + (1 * size of an **`int`**)

```
cout << *(arrayPtr+1); // print 4
cout << *(arrayPtr+2); // print 6
```

  - Must use **`( )`** in expression, comparing with

```
cout << *(arrayPtr)+1;
```

Array Access

- Array elements can be accessed in indexing or pointer expressions:

Array access method	Example
array name and [ ]	`iArray[2] = 12;`
pointer to array and [ ]	`arrayPtr[2] = 12;`
array name and subscript arithmetic	`*(iArray+2) = 12;`
pointer to array and subscript arithmetic	`*(arrayPtr+2) = 12;`

Array Access

- Array notation

```
iArray[i]
```

  is equivalent to the pointer notation

```
*(iArray + i)
```

- No bounds checking performed on array access

Pointer Arithmetic

Some arithmetic operators can be used with pointers:

- Increment and decrement operators **++**, **−−**
- Integers can be added to or subtracted from pointers using the operators **+**, **−**, **+=**, and **−=**
- One pointer can be subtracted from another by using the subtraction operator **−**

Pointer Arithmetic

Assume the variable definitions

```
int iArray[] = {2,4,6};
int *arrayPtr = iArray;
```

Examples of use of **++** and **--**

```
cout << *(++arrayPtr); // print 4
cout << *(--arrayPtr); // now print 2
```

More on Pointer Arithmetic

Assume the variable definitions:

```
int iArray[ ] = {2, 4, 6};
int *arrayPtr = iArray;
```

Example of the use of **+** to add an int to a pointer:

```
cout << *(arrayPtr + 2)
```

This statement will print 6

More on Pointer Arithmetic

Assume the variable definitions:

```
int iArray[ ] = {2, 4, 6};
int * arrayPtr = iArray;
```

Example of use of +=:

```
arrayPtr = iArray;     // point at address of 2
arrayPtr += 2;         // now point to 6
```

More on Pointer Arithmetic

Assume the variable definitions

```
int iArray[ ] = {2, 4, 6};
int * arrayPtr = iArray;
```

Example of pointer subtraction

```
arrayPtr += 2;
cout << arrayPtr - vals;
```

This statement prints **2**: the number of **int**s between **arrayPtr** and **iArray**

Pointer Initialization

- Can initialize to NULL or 0 (zero)

```
int *iPtr = NULL;
```

- Can initialize to addresses of other variables

```
int iNum = 25, *numPtr = &iNum;
int iArray[3], *arrayPtr = iArray;
```

- Initial value must have correct type

```
float dPrice = 21.83;
int *iPtr = &dPrice; //  wrong due to mismatch
                         data types
```

Comparing Pointers

- Relational operators can be used to compare addresses in pointers
- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2) // compares
                  // pointers(addresses)

if (*ptr1 == *ptr2) // compares
                    // values at the
                    // pointers
```

Pointers as Function Parameters (Call by Pointer)

- A pointer can be a parameter
- Works like a reference parameter to allow change to argument from within function
- A pointer parameter must be explicitly dereferenced to access the contents at that address

Pointers as Function Parameters (Call by Pointers)

- Declare pointer variables in parameter list:

  void getNum(int *iPtr)

- Access value at the pointer after dereferencing the pointer:

  cin >> *iPtr;

- Pass the argument address to the corresponding parameter of the function:

  getNum(&iNum);

Pointers as Function Parameters (Call by Pointers)

```
#include <iostream>
using namespace std;
void getNum(int* iPtr)       // call by pointer
{     cin >> *iPtr;  // dereference operator, assign 9 to the value at iPtr,
                     // the value of iNum will be also changed to 9
}
int main()
{     int iNum = 1;
      getNum(&iNum);
      cout << "iNum = " << iNum << endl;// 9
      return 0;
}
```

Pointers as Function Parameters (Call by Pointers)

```
void swap(int *iPtrX, int *iPtrY)
{
      int iTemp;
      iTemp = *iPtrX;
      *iPtrX = *iPtrY;
      *iPtrY = iTemp;
}
int main()
{  int iNum1 = 2, iNum2 = -3;
   swap(&iNum1, &iNum2);
   return 0;
}
```

Pointers to Constants and Constant Pointers

- Pointer to a constant: cannot change the value that is pointed at. But the pointer variable itself (the address pointed by the pointer) is changeable.

- Constant pointer: address in pointer cannot change once pointer is initialized. But the content pointed by the pointer is changeable.

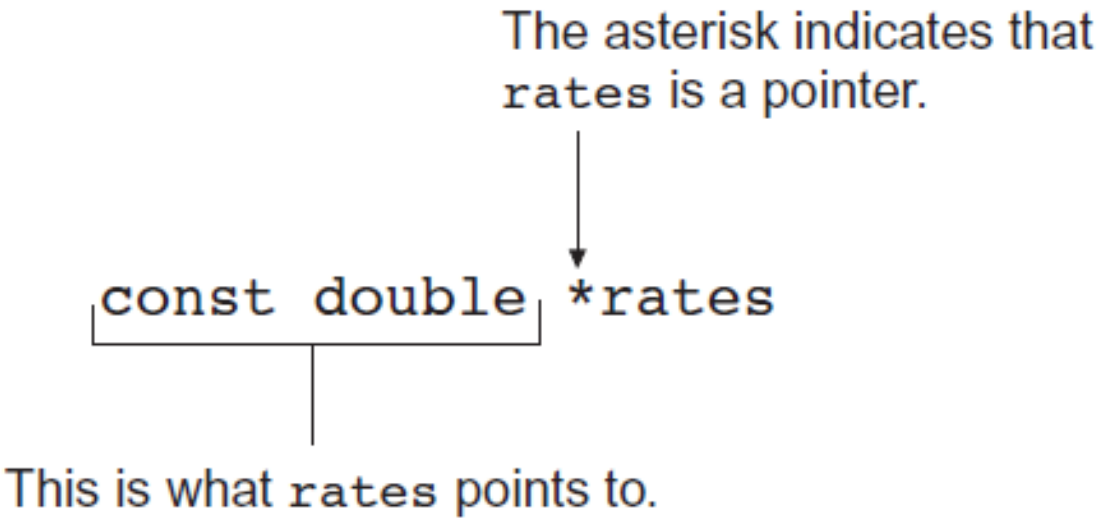
Ponters to Constant

- Must use **`const`** keyword in pointer definition:

```
const double dTaxRates[] =
            {0.65, 0.8, 0.75};
const double *rates = dTaxRates;
```

- Use **`const`** keyword for pointers in function headers to protect data from modification from within function

Pointer to Constant – What does the Definition Mean?



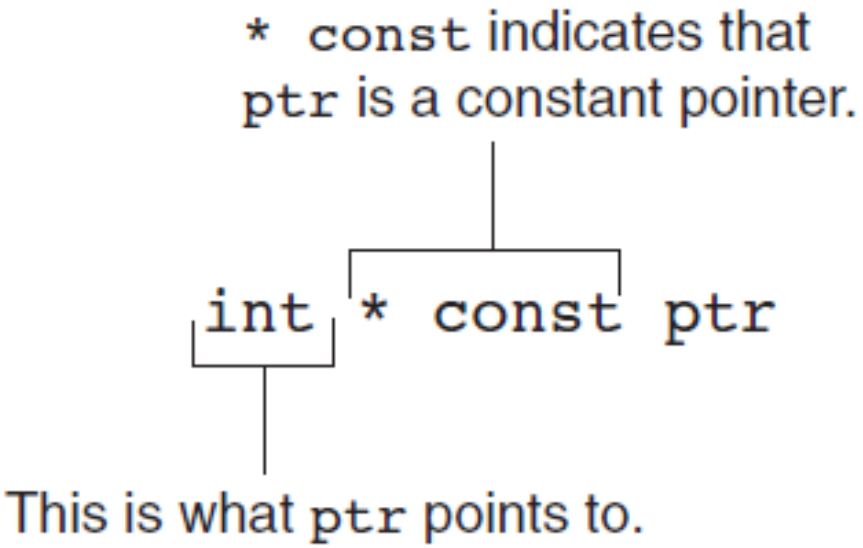
Constant Pointers

- Defined with **const** keyword adjacent to variable name:

```
int classSize = 24;
int * const ptr = &classSize;
```

- Must be initialized when defined
- Can be used without initialization as a function parameter
  - Initialized by argument when function is called
  - Function can receive different arguments on different calls
- While the address in the pointer cannot change, the data at that address may be changed

Constant Pointer – What does the Definition Mean?



Dynamic Memory Allocation

- Can allocate storage for a variable while program is running
- Uses **new** operator to allocate memory

```
double *dPtr =  NULL;
dPtr = new double;
*dPtr = 21.68;
```

- **new** returns address of memory location

Dynamic Memory Allocation

- Can also use **new** to allocate an array:

  double * arrayPtr = new double[5];

  - Program often terminates if there is not sufficient memory
  - Can then use indexing or pointer arithmetic to access array after dynamic allocation.

Releasing Dynamic Memory

- Use **delete** to free dynamic memory

  delete dPtr;

- Use **delete []** to free dynamic array memory

  delete [] arrayPtr;

- Only use **delete** with dynamic memory!

Dangling Pointers and Memory Leaks

- A memory leak occurs if no-longer-needed dynamic memory is not freed. The memory is unavailable for reuse within the program.
  - Solution: free up dynamic memory after use

- A pointer is dangling if it contains the address of memory that has been freed by a call to **`delete`**.
  - Solution: set such pointers to 0 as soon as memory is freed.

Returning Pointers from Functions

- Pointer can be return type of function

```
int* newNum();
```

- Function must not return a pointer to a local variable in the function
- Function should only return a pointer
  - to data that was passed to the function as an argument
  - to dynamically allocated memory

Example of Dynamic Allocation & Call by Pointers

```
#include <iostream>
using namespace std;
int* getNum1()
{       int num = 10;
        num += 1;
        return &num;
}

int getNum2()
{       int num = 10;
        num += 1;
        return num;
}
```

```
void getNum3(int* ptr)
{
        *ptr += 1;
}

int* getNum4()
{
        int* ptr = new int;
        *ptr = 10;
        *ptr += 1;
        delete ptr;
        return ptr;
}
```

```
int main()
{       int* ptr1 = getNum1();
        *ptr1 += 1;
        cout << "*ptr1=" << *ptr1 << endl;
        int x = getNum2();
        x += 1;
        cout << "x=" << x << endl;
        int y = 10;
        getNum3(&y);
        y += 1;
        cout << "y=" << y << endl;
        int* ptr2 = nullptr;
        ptr2 = getNum4();
        *ptr2 += 1;
        cout << "*ptr2=" << *ptr2 << endl;
        return 0;
}
```

Function Pointers

- A function pointer points to function, i.e., holds the starting address of executable codes of a function.
- In Functions Pointers, function's name can be used to get function's address.
- A function pointer can be used a parameter for call by pointer.
- A function name can be passed as an argument to a pointer function and can be returned from a function.
  - Function declaration with Function Pointer:
  - Return_type (*Function_pointer_name)(Parameter_list)

Function Pointer Example

```
#include<iostream>
using namespace std;
int addition(int a, int b)
{
    return a + b;
}
int main()
{
    int x = 1, y = 2;
    int (*fp) (int, int) = addition;
    int result = fp(x, y);
    cout << "Addition of x and y = " << result << endl;
    return 0;
}
```

Using Function Pointers in PriceByCRR

- Enhance PriceByCRR() by adding new payoff functions
- We want to make only minor change of PriceByCRR().
- Function pointers offer a way to achieve it:
  - double PriceByCRR(double S0, double U, double D,
    double R, int N, double K,
    double (*Payoff)(double z, double K));

Options02.h

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

        //pricing European option
        double * PriceByCRR(double S0, double U, double D, double R, 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);
}
```

Options02.cpp

```
#include "Option02.h"
#include "BinomialTreeModel.h"
#include <iostream>
#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;
   }
```

```
double * PriceByCRR(double S0, double U, double D, double R,  int N, double K,
                    double (*Payoff)(double z, double K) )
  {
     double q = RiskNeutProb(U, D, R);
     double *Price = new double[N+1];
     for (int i = 0; i <= N; i++)
     {
        Price[i] = Payoff(CalculateAssetPrice(S0, U, D, 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]) / R;
        }
     }
     return Price;
  }
```

```
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;
  }
}
```

OptionPricer02.cpp

```
#include "BinomialTreeModel.h"
#include "Option02.h"
#include <iostream>
#include <iomanip>
using namespace std;
using namespace fre;
int main()
{
   double S0 = 0.0, U = 0.0, D = 0.0, R = 0.0;
   if (GetInputData(S0, U, D, R) != 0)
        return -1;
   double K = 0.0;        //strike price
   int N = 0;             //steps to expiry
   cout << "Enter call option data:" << endl;
   GetInputData(N, K);
   double* optionPrice = NULL;
```

OptionPricer02.cpp (Continue)

```
    optionPrice = PriceByCRR(S0, U, D, R, N, K, CallPayoff);
    cout << "European Call option price = " << fixed << setprecision(2)
            << optionPrice[0] << endl;

    optionPrice = PriceByCRR(S0, U, D, R, N, K, PutPayoff);
    cout << "European Put option price = " << fixed << setprecision(2)
            << optionPrice[0] << endl;

    delete optionPrice;
    optionPrice = NULL;
    return 0;
}
```

OptionPricer02.cpp (Result)

```
/*
Enter S0: 106
Enter U:  1.15125
Enter D:  0.86862
Enter R:  1.00545

Input data checked
There is no arbitrage

Enter call option data:
Enter steps to expiry N: 8
Enter strike price K:    100

European Call option price = 21.68
European Put option price = 11.43

*/
```

Function Overloading

- BinomialTreeModel.h
  - int GetInputData(double& S0, double& U, double& D, double& R);
- Option02.h
  - int GetInputData(int& N, double& K);
- In CRR Pricer, we use two GetInputData() functions (shown as above), which have the same function name, but different parameter lists, which is called **Function Overloading.**
- Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters, considered as a polymorphism feature in C++.

Homework Assignment

- Include the ability to price digital calls and puts in the program developed in the present section by adding new payoff functions to the files Option02.h, Option02.cpp and OptionPrice02.cpp
  - double DigitCallPayoff(double z, double K)

```
{
        if (z>K) return 1.0;
        return 0.0;
}
```

  - double DigitPutPayoff(double z, double K)

```
{
        if (z<K) return 1.0;
        return 0.0;
}
```

All what we have done so far

is know as the

Procedural style of programming.

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/