C++ によるデリバティブ・ライブラリ構築の基礎
4. オブジェクト指向プログラミング

高田勝己*

2018 年 11 月 2 日

1 Reading：C++ Design Patterns and Derivatives Pricing

Ch.3 Inheritance and Virtual Functions

2 Constructor (**コンストラクタ**)

- 各クラスの 1 つまたは複数の constructor (コンストラクタ) と呼ばれる特殊なメンバー関数は、そのクラスのオブジェクトの（データ・メンバーの）初期化を行い、オブジェクトが生成されるときに呼ばれる。
- コンストラクタはクラス名と同じ名前で、戻り値の型はない。コンストラクタはパラメータ・リストをとり（空のこともある)、関数本体がある（空のこともある)。
- **Default constructor**（デフォルト・コンストラクタ）は引数がないコンストラクタで、

  Matrix A;

  は Matrix クラスのデフォルト・コンストラクタが呼ばれている。あるクラスでコンストラクタがなにも定義されていないとき、コンパイラーが勝手にデフォルト・コンストラクタをつくってくれる (これを **synthesized default constructor**（合成デフォルトコンストラクタ）という)。合成デフォルトコンストラクタでは、
  - クラス内初期子 (**in-class initializer**) があれば、これでメンバーを初期化する。
  - そうでなければ、メンバーをデフォルトで初期化（default-initialize) する。

```
class Swap
{
    // No explicit constructor
private:
    std::string counterpartyName  // counterparty name, 空の string で初期化。
    unsigned cashflowNumber = 10;  // number of cashflows
    double fixedRate = 0.01 // fixed rate
}
```

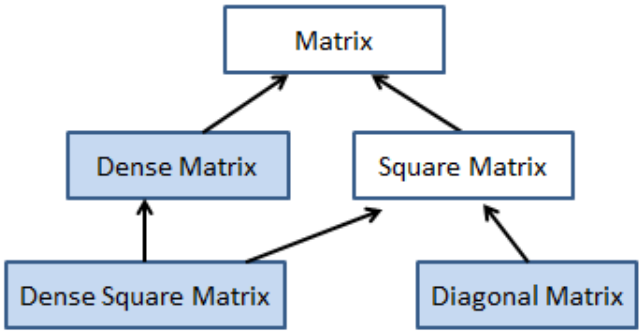
* 株式会社 Diva Analytics, ktakada@divainvest.jp

- 明示的なコンストラクター

```
class Swap
{
    Swap() = default;      // 合成デフォルトコンストラクタと同じもの
    Swap(const std::string &c) : counterpartyName(c) { }
    Swap(const std::string &c, unsigned n, double r) :
        counterpartyName(c), cashflowNumber(n), fixedRate(r) { }      // constructor
initializer list（コンストラクタ初期子リスト）
    private:
        std::string counterpartyName  // counterparty name, 空の string で初期化。
        unsigned cashflowNumber = 10;  // number of cashflows
        double fixedRate = 0.01 // fixed rate
}
```

3 継承 (inheritance)

- 継承 (**inheritance**) と多相性 (**polymorphism**) がオブジェクト指向プログラミングで最も重要な 2 つの要素である。
- "has a"の関係 (コンポジション、**composition**)
  A matrix *has a* certain number of rows and columns.
  An option *has an* underlying asset.
  オブジェクト Y はオブジェクト X を持っている。$\Longleftrightarrow$X は Y の data member である。
  ユーザ定義のクラスもあるクラスの data member に成り得る。
- "is a"の関係 (継承、**inheritance**)
  The identity matrix (単位行列) *is a* matrix.
  A down-and-out option *is an* option.
  オブジェクト Y はオブジェクト X である。$\Longleftrightarrow$Y の型は X の派生クラスである。
  あるクラス (派生クラス、**derived class**) は他のクラス (基底クラス、**base class**) からすべての data members と member functions を引き継ぐことができる。複数の基底クラスから引き継ぐこともできるし、派生クラスが他のクラスの基底クラスにもなることもできる。これら一連のクラス関係をクラス階層 (**class hierarchy**) という。
- 例えば、行列クラスの階層は



(Figure 1) Matrix class hierarchy

- Square matirx を表すクラスを考える。A square matrix "is a" matrix. Matrix クラスで定義したすべての演算は SquareMatirx クラスでも有効である。また、SquareMatirx クラス特有の追加的な演算も定義できる。

```
class SquareMatrix : public Matrix
{
public:
    SquareMatirx(int nrows);
    double determinant() const;
}
```

とかけば、Matrix クラスのすべての機能をもち、追加的に SquareMatrix に特有な行列式 (determinant) を計算する機能を追加できる。

- 行列の加算を表す operator +() はすでに基底クラスの Matrix で定義済みなので、A と B が SquareMatirx のインスタンスとすれば、式 A+B は、多重定義された関数（または演算子）(overloaded function(opetaror))

```
operator +(const Matrix &, const Matirx &)
```

を呼ぶ。関数が呼ばれる時コンパイラは引数の型が同じ関数を探すが、それが見つからない場合は、base class を引数とするものをみつけようとする。この場合、A+B は SquareMatirx 型ではなく、Matrix 型を返す。よって、

```
SquareMatrix operator +(const SquareMatirx &, const SquareMatirx &)
```

を追加的に overload することを考える。

- 上の

```
class SquareMatrix : public Matrix
```

のキーワード `public` は、基底クラスの Matrix で `public` として宣言されたメンバーは SquareMatrix でも public member であることを意味する。Matrix の data member は `private` なので、SquareMatrix のメンバー関数、例えば、determinant() では行列の成分にアクセスする際には、直接 Matrix の data member を使えず、`public` である Matix::operator()() を使う必要がある。計算スピードの問題で直接 Matrix の data member にアクセスするには、Matrix の data member を `protected` とすれば、その derived class でアクセスできるようになる。しかし、これは Matix の内部表現 (implementation)

を変えると、派生クラスにまで影響をうけてしまうことを意味する。

- SquareMatrix のコンストラクタは

  SquareMatirx::SquareMatirx(int nrows)

  : Matrix(nrows, nrows) { }

  であり、ここではコンストラクタ初期子 (constructor initializer) で base class の Matrix のコンストラクタを呼んでいる。もちろん、

  Matirx::Matrix(nrows, nrows)

  は public なので derived class の SquareMatirx のコンストラクタで使うことができる。

4 多相性 (polymorphism)

- 対角行列を SquareMatrix から派生したクラス DiagonalMatrix として実装することを考える。いまの Matrix クラスのデータ構造からするとこれは効率的とはいえない。なぜなら、Matrix は、DiagonalMatrix ではゼロとなる非対角成分にもメモリを配分してしまう。また、ユーザは、アクセスを表す operator () を通じて DiagonalMatrix の非対角成分をゼロ以外の数にかえてしまうこともできてしまうので、安全ではない。
- この解決策は、DiagonalMatrix は interface（public member function の宣言）だけを base class から引き継ぎ、implementation を上書き (override) することである。具体的には、

  ```
  class DiagonalMatrix : public SquareMatirx
  {
  public:
      DiagonalMatrix(int nrows);
      double &operator() (int i, int j);
      double operator() (int i, int j) const;
  };
  ```

  operator() を DiagonalMatrix のメンバーとして宣言すると、base class で定義された Matrix::operator()() を隠して、DiagonalMatrix 特有の operator () の implementation をすることができる。
- 派生クラスから基底クラスの変換 (Derived-to-base conversion)

  基底クラスを指すポインターや、基底クラスの参照は、派生クラスを指したり参照できる。

  We can **bind** a pointer or ref to a base-class type **to** an object of a type derived from that base class. (**Derived-to-base conversion**)

  ```
  DiagonalMatrix DM;
  Matrix *p1 = &DM;  //ok, 動的な型 (dynamic type) は DiagonalMatrix である
  DiagonalMatrix *p2 = p1;  //error: 基底から派生には変換できない。
  Matrix &refM = DM;  //ok, 動的な型は DiagonalMatrix
  Matrix M;
  DiagonalMatrix &refDM = M;  //error: 基底から派生には変換できない。
  ```
- 次のプログラムを考える。

```
DiagonalMatrix D(10);
Matrix &M = D;  // derived-to-base conversion
M(1,2) = 1.5;     // Matrix::operator()() を呼ぶ。DiagonalMatrix::operator() () は呼ばない。
```

- **Virtual function（仮想関数）**

上で、DiagonalMatrix::operator() を呼ぶには仮想関数を使う。上の様に非仮想関数を再定義することはあまりない。

```
class Matrix
{
    ...
    virtual double &operator() (int i, int j);
    virtual double operator() (int i, int j) const;
    ...
};
class DiagonalMatrix : public SquareMatirx
{
    ...
    virtual double &operator() (int i, int j);
    virtual double operator() (int i, int j) const;
    ...
};
```

このように、演算子宣言の前にキーワード `virtual` を付けると、上の M(1,2) で DiagonalMatrix::operator()() を呼ぶ。基底クラスでは interface だけを定義して、派生クラスで virtual fucntion を再定義すれば、動的な型の仮想関数（つまり implementation) を呼ぶことができる。これが、多相性 (**polymorphism**) である。

この利点は、base class の public member functions(つまり、interface) だけを使ってコードをかくことができ、derived class で implementation が再定義されてもコードを書き直す必要がないということである。(C++ ではとても重要!!)

- **Pure virtual function（純仮想関数）**

DiagonalMatrix クラスの非対角要素はゼロなので、Matrix クラスの内部の表現方式は非効率的であるかもしれない。ここでは、DiagonalMatrix クラスで独自のデータの持ち方をすることを考える。1 つの方法は Matrix を **abstract base class（抽象基底クラス）**として定義することである。

```
class Matrix
{
public:
    virtual ~Matrix();       // virtual destructor（仮想デストラクタ）
    virtual double operator() (int i, int j) const = 0;
    virtual double &operator() (int i, int j) = 0;
    virtual Matrix transpose() = 0;
    ...
}
```

Matrix のクラス定義は、任意の行列に関する演算を表すメンバー関数で構成され、これらの関数は Matrix では実装 (implement) されない。これを示すために、=0 を関数宣言の最後につけることで、**pure virtual function（純仮想関数）**を表す。純仮想関数をもっているクラスは抽象クラスでこのクラスのインスタンスは生成されない。具体的な派生クラス (concrete derived class) で純仮想関数が具体的に実装（implement）され、オブジェクトが生成される。

- SquareMatrix を次のように定義する。

```
class SquareMatrix : public Matrix
{
public:
    virtual double determinant() const = 0;
};
```

追加で、determinant() という純仮想関数を宣言し、また、ここでは基底クラスである Matix で宣言された純仮想関数は実装されないので、SquareMatrix も抽象クラスである。

- DenseMatrix は具体クラス（**concrete class**）で、クラスの実装は前回の Matrix クラスと同じである。DenseSquareMatrix は DenseMatrix と SquareMatrix からの多重継承 (**multiple inheritance**) である。DiagonalMatrix は DenseMatrix とは異なる内部表現をつかっている。
- それぞれの行列の表現方法の違いにより、コンストラクタで確保 (allocate) され、デストラクタで解放 (deallocate) されるメモリ領域の大きさは異なる。よって、ディストラクタは virtual function（仮想関数）として、派生クラスのそれが呼ばれるようにする。

5 コード （Matrix)

5.1 Matrix.h

```
#include <iostream>

// Abstract base class
class Matrix
{
public:
    virtual ~Matrix();
    virtual int rows() const = 0;
    virtual int columns() const = 0;
    virtual double operator()(int i,int j) const = 0;
    virtual double& operator()(int i,int j) = 0;
/* ... */
};
```

// Matrix から派生したクラス SquareMatrix と DenseMatrix から多重継承されるために、SquareMatrix と DenseMatrix の基底クラスである Matrix は”virtual”として宣言されなくてはならない。

```
class SquareMatrix : public virtual Matrix{
```

```
public:
    virtual double determinant() const = 0;
    /* ... */
};

class DenseMatrix : public virtual Matrix{
private:
    int r; //number of rows
    int c; // number of columns
    double* d; // array of doubles for matrix contents
public:
    DenseMatrix(int nrows,int ncols,double ini = 0.0);
    DenseMatrix(const DenseMatrix& mat);
    DenseMatrix(const Matrix& mat);
    DenseMatrix& operator=(const Matrix& mat);
    virtual ~DenseMatrix();
    virtual int rows() const;
    virtual int columns() const;
    virtual double operator()(int i,int j) const;
    virtual double& operator()(int i,int j);
    /* ... */
};

class DenseSquareMatrix : public DenseMatrix, public SquareMatrix
{
public:
    inline DenseSquareMatrix(int nrows,double ini = 0.0) : DenseMatrix(nrows,nrows,ini) { };
    virtual double determinant() const;
    /* ... */
};

class DiagonalMatrix : public SquareMatrix
{
private:
    int r; // number of rows and columns
    double* d; // array of doubles for diagonal element
public:
    DiagonalMatrix(int nrows,double ini = 0.0);
    virtual ~DiagonalMatrix();
    virtual int rows() const;
    virtual int columns() const;
    virtual double operator()(int i,int j) const;
```

```
    virtual double& operator()(int i,int j);
    virtual double determinant() const;
    /* ... */
};

DenseMatrix operator+(const Matrix& A,const Matrix& B);
std::ostream& operator<<(std::ostream& os,const Matrix& A);
```

5.2 Matrix.cpp

```
#include "Matrix.h"

DenseMatrix::DenseMatrix(int nrows,int ncols,double ini)
{
    int i;
    r = nrows;
    c = ncols;
    d = new double [nrows*ncols];
    double* p = d;
    for (i=0;i<nrows*ncols;i++) *p++ = ini;
}

DenseMatrix::DenseMatrix(const DenseMatrix& mat)
{
    int i;
    r = mat.r;
    c = mat.c;
    d = new double [r*c];
    double* p = d;
    double* pm = mat.d;
    for (i=0;i<r*c;i++) *p++ = *pm++;
}

DenseMatrix::DenseMatrix(const Matrix& mat)
{
    int i,j;
    r = mat.rows();
    c = mat.columns();
    d = new double [r*c];
    for (i=0;i<r;i++) {
        for (j=0;j<c;j++) d[i*c + j] = mat(i,j); }
}

DenseMatrix& DenseMatrix::operator=(const Matrix& mat)
```

```
{
    int i,j;
    if (this != &mat) {
    for (i=0;i<r;i++) {
        for (j=0;j<c;j++) d[i*c + j] = mat(i,j); }}
    return *this;
}

DenseMatrix::~DenseMatrix()
{
    delete[] d;
}

double DenseMatrix::operator()(int i,int j) const
{
    return d[i*c + j];
}

double& DenseMatrix::operator()(int i,int j)
{
    return d[i*c + j];
}

int DenseMatrix::rows() const
{
    return r;
}

int DenseMatrix::columns() const
{
    return c;
}

DenseMatrix operator+(const Matrix& A,const Matrix& B)
{
    int i,j;
    DenseMatrix result(A);
    for (i=0;i<A.rows();i++)
    {
        for (j=0;j<A.columns();j++)
            result(i,j) = A(i,j) + B(i,j);
    }
    return result;
}
```

```
std::ostream& operator<<(std::ostream& os,const Matrix& A)
{
    int i,j;
    for (i=0;i<A.rows();i++)
    {
        for (j=0;j<A.columns()-1;j++)
            os << A(i,j) << ',';
        os << A(i,j) << std::endl;
    }
    return os;
}

DiagonalMatrix::DiagonalMatrix(int nrows,double ini)
{
    int i;
    r = nrows;
    d = new double [nrows];
    double* p = d;
    for (i=0;i<nrows;i++) *p++ = ini;
}

DiagonalMatrix::~DiagonalMatrix()
{
    delete[] d;
}

double DiagonalMatrix::operator()(int i,int j) const
{
    return (i==j) ? d[i] : 0.0;
}

double& DiagonalMatrix::operator()(int i,int j)
{
    if (i!=j) throw std::logic_error("Write access to off-diagonal elements of DiagonalMatrix not permitted");
    return d[i];
}

int DiagonalMatrix::rows() const
{
    return r;
}

int DiagonalMatrix::columns() const
```

```
{
    return r;
}
```

5.3 main.cpp

```
#include <iostream>
#include "Matrix.h"
using namespace std;

int main()
{
    int i,j;
    DenseMatrix A(3,5),B(3,5),C(3,5),D(3,5);

    // assign values to matrix elements
    for (i=0;i<3;i++)
    {
        for (j=0;j<5;j++)
            A(i,j) = B(i,j) = C(i,j) = 0.1*i*j;
    }

    D = A + B + C;
    cout << "Matrix D is: " << D << endl;
    DenseSquareMatrix E(4);
    DiagonalMatrix F(4);
  for (i=0;i<4;i++)
    {
        F(i,i) = -1.5;
        for (j=0;j<4;j++)
            E(i,j) = 1.3*i+j;
    }
    cout << "The sum of E and F is " << E+F << endl;
    return 0;
}
```

6 コード 2

6.1 UTPayOff2.h

```
#ifndef UT_PayOff2_H
#define UT_PayOff2_H

class UTPayOff
```

```
{
public:
    UTPayOff(){};
    virtual double operator()(double Spot) const = 0;
    virtual ~UTPayOff(){}
private:
};

class UTPayOffCall : public UTPayOff
{
public:
    UTPayOffCall(double Strike_);
    virtual double operator()(double Spot) const;
    virtual ~UTPayOffCall(){}
private:
    double Strike;
};

class UTPayOffPut : public UTPayOff
{
public:
    UTPayOffPut(double Strike_);
    virtual double operator()(double Spot) const;
    virtual ~UTPayOffPut(){}
private:
    double Strike;
};

#endif
```

6.2 UTPayOff2.cpp

```
#include <minmax.h>
#include "UTPayOff2.h"

UTPayOffCall::UTPayOffCall(double Strike_) : Strike(Strike_)
{
}

double UTPayOffCall::operator () (double Spot) const
{
    return max(Spot - Strike, 0.0);
}
```

```
double UTPayOffPut::operator () (double Spot) const
{
    return max(Strike - Spot, 0.0);
}

UTPayOffPut::UTPayOffPut(double Strike_) : Strike(Strike_)
{
}
```

6.3 UTSimpleMC2.h

```
#ifndef UT_SIMPLE_MC2_H
#define UT_SIMPLE_MC2_H

#include "UTPayOff2.h"

double SimpleMonteCarlo2(const UTPayOff& thePayOff,
    double Expiry,
    double Spot,
    double Vol,
    double r,
    unsigned long NumberOfPaths);

#endif
```

6.4 UTSimpleMC2.cpp

```
#include <cmath>
#include"UTSimpleMC2.h"
#include "UTRandom1.h"

double SimpleMonteCarlo2(const UTPayOff& thePayOff,
    double Expiry,
    double Spot,
    double Vol,
    double r,
    unsigned long NumberOfPaths)
{
    double variance = Vol*Vol*Expiry;
    double rootVariance = sqrt(variance);
    double itoCorrection = -0.5*variance;
    double movedSpot = Spot*exp(r*Expiry + itoCorrection);
    double thisSpot;
    double runningSum = 0;
```

```
    for (unsigned long i = 0; i < NumberOfPaths; i++)
    {
        double thisGaussian = GetOneGaussianByBoxMuller();
        thisSpot = movedSpot*exp(rootVariance*thisGaussian);
        double thisPayOff = thePayOff(thisSpot);
        runningSum += thisPayOff;
    }

    double mean = runningSum / NumberOfPaths;
    mean *= exp(-r*Expiry);
    return mean;
}
```

6.5 UTSimpleMCMain3.cpp

```
#include<iostream>
#include "UTSimpleMC2.h"
using namespace std;
int main()
{
    double Expiry;
    double Strike;
    double Spot;
    double Vol;
    double r;

    unsigned long NumberOfPaths;
    cout << "\nEnter expiry\n";
    cin >> Expiry;

    cout << "\nEnter strike\n";
    cin >> Strike;

    cout << "\nEnter spot\n";
    cin >> Spot;

    cout << "\nEnter vol\n";
    cin >> Vol;

    cout << "\nEnter r\n";
    cin >> r;

    cout << "\nNumber of paths\n";
    cin >> NumberOfPaths;

    UTPayOffCall callPayOff(Strike);
```

```
    UTPayOffPut putPayOff(Strike);

    double resultCall = SimpleMonteCarlo2(callPayOff,
        Expiry,
        Spot,
        Vol,
        r,
        NumberOfPaths);

    double resultPut = SimpleMonteCarlo2(putPayOff,
        Expiry,
        Spot,
        Vol,
        r,
        NumberOfPaths);

    cout << "the prices are " << resultCall << " for the call and "
        << resultPut << " for the put\n";

    double tmp;
    cin >> tmp;

    return 0;
}
```

6.6 UTSimpleMCMain4.cpp

```
#include<iostream>
#include"UTSimpleMC2.h"
using namespace std;

int main()
{
    double Expiry;
    double Strike;
    double Spot;
    double Vol;
    double r;
    unsigned long NumberOfPaths;

    cout << "\nEnter expiry\n";
    cin >> Expiry;
    cout << "\nEnter strike\n";
    cin >> Strike;
    cout << "\nEnter spot\n";
```

```
    cin >> Spot;
    cout << "\nEnter vol\n";
    cin >> Vol;
    cout << "\nr\n";
    cin >> r;
    cout << "\nNumber of paths\n";
    cin >> NumberOfPaths;
    unsigned long optionType;
    cout << "\nenter 0 for call, otherwise put ";
    cin >> optionType;

    UTPayOff* thePayOffPtr;
    if (optionType == 0)
        thePayOffPtr = new UTPayOffCall(Strike);
    else
        thePayOffPtr = new UTPayOffPut(Strike);

    double result = SimpleMonteCarlo2(*thePayOffPtr,
        Expiry,
        Spot,
        Vol,
        r,
        NumberOfPaths);

    cout << "\nthe price is " << result << "\n";

    double tmp;
    cin >> tmp;

    delete thePayOffPtr;

    return 0;
}
```

著作権と免責事項