C++ によるデリバティブ・ライブラリ構築の基礎
3. クラスとカプセル化

高田勝己*

2019 年 10 月 18 日

1 Reading：C++ Design Patterns and Derivatives Pricing

Ch.2 Encapsulation

2 関数

2.1 呼び出し

パラメータ型 `const`	非参照 f(string s) f(const string s)	参照 f(string &s) f(const string &s)
引数は	passed by value（値渡し）	passed by reference（参照渡し）
関数は	called by value	called by reference
意味	コピーされる	参照される

- Call by value (値渡し)

定義

```
int fact(int val)
{
    int ret = 1;
    while (val > 1)
        ret *= val– –;  // val is changed
    return ret;
}
```

呼び出し

```
int i = 4;
int j = fact(i);
cout << "i =" << i << endl;  //i = 4, j = 24
```

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

定義

```
void reset(int *ip)  //   C ではよくみる。
{
    *ip = 0;
    ip = 0;
}
```

呼び出し

```
int i = 42;
int j = reset(&i);  // i を指すポインタの値は変わらない。
cout << "i =" << i << endl;  //i = 0   (i は変化する)
```

定義

```
void fcn (const int i){ /* fcn 内で i を読むことはできるが、書くことはできない */}
```

呼び出し

```
int j = 3;
fcn(j);
```

- Call by reference (参照渡し)

定義

```
void reset(int &i)   //C++ ではこちら
{
    i = 0;
}
```

呼び出し

```
int j = 42;
reset(j);
cout << "j =" << j << endl;  //j = 0
```

定義

```
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

- 応用例

関数は 1 つの値しか返さないが、複数の値を返したい場合、参照渡しが有効である。
下の関数は与えられた string 型の文字列内の最初の c の位置を返す。同時に c がいくつあるかは参照渡しで渡した変数から取り出せる。

定義

```
// return the index of the first occurance of c in s
string::size_type find_char(const string &s, char c, string::size_type &occurs)
{
    auto ret = s.size();
    occurs = 0;
    for (decltype(ret) i = 0; i != s.size(); ++i)
    {
        if(s[i] == 'c')
        {
            if (i == s.size())
                ret = i;
            ++occurs;
        }
    }
    return ret;
}
```

呼び出し

```
string s("Hello, oops");
string::size_type ctr;
auto index =find_char(s, 'o', ctr);
cout << index << ctr << endl;
```

練習 1 1) 上のコードは 2 か所間違えている。正しくせよ。

2) 関数の定義の 4 行目で、occurs = 0; をいれない場合（コメントアウトした場合）はどういう結果になるか？

2.2 戻り値 (返り値、return value)

変数やパラメータの初期化と同様に、関数が呼ばれた場所の一時オブジェクトが関数の戻り値で初期化される。

- 非参照型を返す場合

定義

```
string make_plural(unsigned int ctr, const string &word, const string &ending)
{
    return (ctr > 1) ? word + ending : word;
}
```

呼び出し

```
string s("apple");
string end("s");
```

string sss = make_plural(1, s, end);
関数 make_plural は名前のない一時的な string を返してそれを sss に代入する。

- 参照型を返す場合

定義

```
const string &shorterString( const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

呼び出し

```
string s3("apple");
string s4("orange");
auto shortStringSize = shortString(s3, s4).size();
shortString(s3, s4)[0] = 'A';  //error
```

- ローカルオブジェクトの参照は返してはいけない

下の関数はローカルオブジェクトの参照を返している。これはやってはいけない。

```
const string &manip()
{
    string ret;
        ⋮
    // ret を操作
    if ( !ret.empty() )
        return ret;  //ローカルオブジェクトの参照をかえしている。
    else
        return "Empty";   // "Empty"もローカルな一時オブジェクト
}
```

練習 2 上の string make_plural() の戻り値の型を const string & とするとどうなるか？

3 クラス (class)

3.1 解説

- クラスの定義

クラスはユーザ定義の型。定義は

```
class Matrix
{
    ...
}
```

として、行列を表す Matrix 型の変数 (またはオブジェクト、またはインスタンス）を

```
Matrix mat;
```

と宣言する。

- Data member

  クラスは **data member** と **member function(**または **method)** を持ち、data member はオブジェクトの具体的内容を表す。Matrix class の data member は

```
class Matrix
{
    int r;  // number of rows
    int c; // number of columns
    double *d  // array of double for the matrix elements
    ...
}
```

これらのデータで行列を内部的に表現できる。

- Constructor

  コンストラクタ (**constructor**) は、Matrix 型の変数をつくるときに、data member を初期化する。コンストラクタはクラスの名前と同じ特別な member function である。

```
class Matrix
{
    // constructor
    Matrix(int nrows, int ncols, double ini = 0);
        ...
    int r;  // number of rows
    int c; // number of columns
    double *d  // array of double for the matrix elements
}
```

  このコンストラクタで、data member である行数を表す r をパラメータの nrows が関数呼び出しの引数で初期化されたもので、data member である列数を表す c をパラメータの ncols が関数呼び出しの引数で初期化されたもので、それぞれ初期化して、行列成分すべてを同じ ini とする。この関数定義はクラス定義の外でおこなう。

```
Matrix::Matrix(int nrows, int ncols, double ini)
: r(nrows), c(ncols)      // constructor initializer, member はクラス定義の順に初期化される
{
    d = new double [nrows*ncols];
    double *p = d;
    for (int i = 0; i < nrows*ncols; ++i) *p++ = ini;
}
```

- メモリの種類は以下の 3 つ。1) static memory, 2) **stack** memory: 関数の中で定義された static でない local object のメモリでコンパイラによって自動的にその領域が確保され、解放される。, 3) **free**

store または **heap**: run-time で動的にオブジェクトのメモリが確保され、プログラムで明示的にオブジェクトを削除しメモリを解放する。

new 演算子は、メモリを動的に確保 (allocate) して、delete 演算子はそのメモリを解放する。new は heap に作られた unnamed object（名前なしのオブジェクト）を指すポインタをかえす。

```
string *ps = new string;
int *pi = new int;
double *pd = new double[10]  //allocate memory for an array
...
string *ps2 = ps;
delete ps; // ps2 は無効 (invalid) となる (dangling pointer)。
delete ps2; //同じメモリを 2 回消せない。free store may be corrrupted.
delete pi;
delete[ ] pd;   //deallocate memory for the array..
```

- Default argument

  コンストラクタの宣言の 3 番目のパラメータをデフォルト引数 (**default argument**)(=0.0) としている。これは、

  ```
  Matrix mat(3, 3);
  ```

  とコンストラクターを呼び出すときに 3 番目の引数を指定しない場合は、すべての行列要素がゼロとなる。関数の宣言でデフォルト引数をもつパラメータはすべて最後に寄せる。

- Instance

  あるクラスの変数をつくることをクラスを具体化する (instantiate) といい、上の mat という具体化されたオブジェクトをクラス Matrix のインスタンス (**instance**) という。

- Destructor

  `new` を使って割り当てられたメモリは、それがいらなくなったときは陽に `delete` 演算子でシステムに返却しなくてはならない。これを怠ると、**memory leak** の原因となる。Matrix 型のオブジェクトが scope から抜けると、**destructor** が自動的に呼ばれ、そのオブジェクトが破棄される。destructor はメンバ関数で

  ```
  Matrix::~Matrix()
  {
      delete[ ] d;
  }
  ```

  と定義する。

- member へのアクセス

  public な function や data は instance に **dot operator** . をつけてアクセスできる。例えば、

  ```
  i <A.r
  ```

  または、

  ```
  i <A.rows()
  ```

  のようである。

  追加の member function を必要に応じて付け加える。例えば、行列の転置を計算したければ、

  ```
  Matrix transpose() const;
  ```

をクラス定義に付け加える。

```
Matrix A(3,2);
Marix A_T =A.transpose();
```

というように、public な member (function や data) は instance に **dot operator** . をつけて呼ぶことができる。また、

```
Matrix *pA = &A;
Marix A_T =pA->transpose();
```

というように、オブジェクトを指すポインタから member (function や data) へのアクセスは **arrow operator** -> をつかうことで実現できる。これは、(*pA).mem のことである。

- Overloaded operator

  行列の 2 行 1 列の成分を、mat(2,1) のように () 演算子で指定することを考える。すでに C++ で割り当てられている演算子を再定義することで、クラスのインスタンス対する演算子による演算を可能にする。これを**演算子の多重定義 (operator overloading)** という。ここでは、添え字演算子の [ ] は、1 つの引数しかとらないので、overload する演算子を () にしている。**overloaded operator** も member function の特別な形にすぎない。

```
class Matix
{
    ...
    inline double operator() (int i, int j) const;
    inline double& operator() (int i, int j);
};
```

  演算子 () は 2 回 overload されている。最初のバージョンは要素を読むもので、

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

  書き込むバージョンは

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

  である。読むバージョンは値を返しているのに対して、書き込むバージョンは **lvalue** である行列要素の参照を返している。

- `const` member function

  読むバージョンは、instance の data member を変えないので、**const** キーワードがパラメータリストの後についている。`const` で宣言された instance は const member function しか呼べない。

- Inline function

  `inline` を関数の最初につけることで、インライン関数 (**inline function**) となる。プログラムで inline function を呼ぶとコンパイラがそこで関数の中身を展開する。例えば、下のコードで、

```
A(i,j) = 0.1*i*j;
```

が、コンパイル中に

```
d[i*c + j] = 0.1*i*j;
```

と置き換わるので、run-time 時の関数を呼ぶコスト (overhead) が削減される。inline 関数は小さく簡潔で、loop の中でよく呼ばれるものが候補である。`inline` と宣言してもコンパイラーが inline にしない場合もある。例えば、再帰的な関数や文が多い関数の場合である。

- Matrix 型のユーザは、例えばつぎのような main 関数を書く。

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

int main()
{
    int i,j;
    Matrix A(3,2);
    // assign values to matrix elements
    for (i = 0; i < A.rows(); ++i)
    {
        for (j = 0; j < A.columns(); ++j)
        {
            A(i,j) = 0.1*i*j;
        }
    }

    // access matrix elements
    double sum = 0.0;
    for (i = 0; i < A.rows(); ++i)
    {
        for (j = 0; j < A.columns(); ++j)
        {
            sum += A(i,j);
        }
    }

    //writes a sum of all the elements
    cout << "The sum of the matrix elements is " << sum << endl;

    return 0;
}
```

- メンバ関数でないがクラスに関係する関数

メンバ関数でない通常の関数や演算子もオーバーロードされる (overloaded)。例えば、2 つの Matrix

型の instances の要素ごとの足し算は、

Matrix operator+ (const Matirx &A, const Matrix &B);

で、Matrix 型と double の足し算は

Matrix operator+ (const Matirx &A, double x);

となる。算術演算子や関係演算子は、通常の非メンバー関数とするのが普通である。これらの関数は class member ではないが、概念的にはクラス Matrix の interface の一部であるので、クラス Matrix が定義してあるヘッダーファイルのクラス定義の外でそれらの宣言することが多い。例えば、Matrix 型の output 演算子

std::ostream& operator << (std::ostream& os,const Matrix& A);

はそうである。

- Overloaded function

  演算子も含めた関数は、パラメータリストのパラメータの型が違えば何度でも overloaded される。クラスの member function の場合は、const かそうでないかでも区別される。ただし、関数の戻り値の型だけが違うときは overload されない。また、デフォルト引数をもつパラメータがある場合は、そのパラメータを除いても overload されない。例えば、

  double myFunction(int i);

  double& myFunction(int i);

  や

  double myFunction2(string str, string str2, int i = 0);

  double myFunction2(string str, string str2);

  は overload されない。

- **Data abstraction**: data member,member function や overloaded operator を private と public に分けることで、**interface** と **implementation** の区別を実現する。
- **Encapsulation**: 典型的なのは、data member や実装をモジュール化した member fuction を private にして、クラスユーザが interface として使う constructor, member function や overloaded operator は public にする。

  Matrx クラスの場合、ユーザが constructor で Matrix 型の instance をつくったり、member function や overloaded operator でそのオブジェクトを操作する等の interface がわかれば、ユーザは、data member や member function の実装等のクラスの内部構造や interface の具体的な実現方法である implementation までは知る必要がない。例えば、

  mat(1,2)=3;

  と行列の要素を変えるオペレーションをするユーザは内部の行列が、double の配列でその配列にアクセスするときは、 d[i*c + j] と配列の添え字演算子 [ ] を使っているとかという具合的な implementation は知る必要がない。

  クラスを書く人 (class author) は、interface を変えずにもっといい implementation があれば、クラスのコードを書き換えることが可能である。
- キーワード `private` と `public`

  Data abstraction や Encapsulation はキーワード `public` と `private` でアクセスの違いを設けることで可能となる。

`pubic` として宣言された、data member, function はクラスの外 (class user) からでもアクセスが可能。private として宣言された、data member や function はクラスの member function だけで使用できる。

- キーワード `friend`

  多重定義された (overloaded) クラスメンバーでない operator + は、効率的に計算できるよう、直接 data member にアクセスさせたい。このようなときは、クラス定義内で関数や演算子を宣言するときキーワード `friend` を付けると、private である member にもアクセスできる。

- 演算子 + を定義したので、

  Matrix C(A+B); // A と B の大きさは同じ

  や

  C = A + B*C; // A, B, C の大きさは同じ

  のような演算も定義したい。それぞれの宣言を加える。

```
class Matrix
{
public:
    ...
    Matrix(const Matrix &mat);  // copy constructor
    Matirx &operator=(const Matrix &mat);  //assignment operator
    ...
}
```

- Copy constructor

  copy constructor はコピーによる初期化で、 1) オブジェクトが値渡しされるとき (例えば、上の A+B の演算子の戻り値）や 2) オブジェクトが同じクラスの instance から陽に初期化されるとき (例えば、上の C(A+B)) に呼ばれる。

- Assignment operator

  assignment operator の戻り値の型は、ref to a Matrix であり、実装の最後で

  return *this;

  となっている。`this` は関数がよばれたオブジェクト（assignment operator の場合は左被演算子）を指すポインタであり、上は `this` が指すオブジェクトを返している。上の C = A + B*C は Matrix 型の instance C の ref を返している。これは、built-in 型の assignment operator と整合的である。

  a=b=c; // b=c; a=b と同じこと。=は右結合。

  (a = b) = c; // a = b; a=c; と同じこと。あまり、こんなことはしないが、const を返すとコンパイルできない。nonref を返すと、a が c とはならない（なぜか？)。

  assignment operator では、this と右被演算子のポインタを比べることで、self-assignment をチェックしている。同じ場合は、なにもせず自分自身のオブジェクトを返す。このチェックがないと、self-assignment の場合、delete[] d; で指すオブジェクトの member data である配列が消されてしまい、配列の要素をコピーする際に、d[i]=mat.d[i] の mat.d[i] はエラーとなる。

- Destructor(デストラクタ)

  ~Matrix();

  戻り値はなく、パラメータはない。コンストラクタと逆のことをする。function body { } が実行され、

data member がコンストラクタとは逆の順序で破棄される。クラス型のメンバーはそのデストラクタが呼ばれる。built-in 型にはデストラクタはない。ポインタ (naked pointer) がメンバーのときは、指しているオブジェクトは破棄されない。よって、data member であるポインターが `new` で作った動的メモリを指している場合、陽に `delete` で指しているオブジェクトを破棄しなくてはならない。

- Rule of Three
  以上の 1) copy constructor, 2) assignment operator 及び 3) destructor は陽に書かなくてもよい。その場合は、コンパイラが synthesized version（合成されたもの）をつくってくれる。synthesized version はメンバーごとのコピー、代入、破棄である。”Rule of Three”とは、3 つの関数のうち 1 つでも陽に書かなくてはならないときは、3 つとも陽に書く必要があるという Rule of thumb である。
  C++11 で move という概念が新たに加わり、上の 3 つと Move constructor と Move assignment operator で”rule of five”となった。

3.2 コード

3.2.1 Matrix.h

```
#include <iostream>

class Matrix
{
public:
     Matrix(int nrows,int ncols,double ini = 0.0); //constructor
     Matrix(const Matrix &mat);                    //copy constructor
     Matrix &operator=(const Matrix &mat);         //assignment operator
     ~Matrix();                                    //destructor

     inline int rows() const { return r; };
     inline int columns() const { return c; };
     inline double operator()(int i,int j) const;
     inline double& operator()(int i,int j);

     friend Matrix operator+(const Matrix& A,const Matrix& B);
     friend Matrix operator+(const Matrix& A,double x);
     friend Matrix operator*(const Matrix& A,const Matrix& B);
     friend Matrix operator*(const Matrix& A,double x);

private:
     int r; // number of rows
     int c; // number of columns
     double* d; // array of doubles for matrix contents
};

// output operator
std::ostream& operator << (std::ostream& os,const Matrix& A);
```

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

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

3.2.2 Matrx.cpp

```
#include "Matrix.h"

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

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

Matrix &Matrix::operator=(const Matrix& mat)
{
    // make sure we are not assigning a matrix to itself
    if (this != &mat)
    {
        delete[] d;
        d = new double[mat.r*mat.c];
        r = mat.r;
        c = mat.c;
        // copy data
        for (int i=0; i<r*c; ++i) d[i] = mat.d[i];
    }
```

```
    return *this;
}

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

Matrix operator+(const Matrix& A,const Matrix& B)
{
    int i;
    Matrix result(A);
    double* p = result.d;
    double* pB = B.d;
    for (i=0;i<A.r*A.c;i++) *p++ += *pB++;
    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;
}
```

4 Pay-Off クラス

Vanilla option のペイオフの概念の encapsulation を考える。

4.1 Pay-off クラスの implementation

4.1.1 UTPayOff1.h

```
#ifndef UT_PayOff_H
#define UT_PayOff_H

class UTPayOff
{
public:
    enum UTOptionType {call, put};
```

```
    UTPayOff(double strike, UTOptionType optionType);
    double operator()(double Spot) const;
private:
    double myStrike;
    UTOptionType myOptionType;
};
#endif
```

4.1.2 UTPayOff1.cpp

```
#include <MinMax.h>
#include "UTPayOff1.h"

UTPayOff::UTPayOff(double strike, UTOptionType optionType)
: myStrike(strike), myOptionType(optionType)
{
}

double UTPayOff::operator ()(double spot) const
{
    switch (myOptionType)
    {
    case call :
        return max(spot-myStrike,0.0);
    case put:
        return max(myStrike-spot,0.0);
    default:
        throw("unknown option type found.");
    }
}
```

4.2 PayOff クラスを使う

4.2.1 SimpleMC.h

```
#ifndef UT_SIMPLE_MC_H
#define UT_SIMPLE_MC_H
#include "UTPayOff1.h"

double SimpleMonteCarlo2(const UTPayOff& payOff,
                         double expiry,
                         double spot,
                         double vol,
                         double r,
```

```
                                unsigned long NumberOfPaths);
#endif
```

4.2.2 SimpleMC.cpp

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

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;
}
```

4.2.3 Main.cpp

```
#include<iostream>
#include"UTSimpleMC.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;

    UTPayOff callPayOff(strike, UTPayOff::call);
    UTPayOff putPayOff(strike, UTPayOff::put);

    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;
  }
```

著作権と免責事項

- 当資料（本文及びデータ等）の著作権を含む知的所有権は（株）Diva Analytics に帰属し、事前に（株）Diva Analytics への書面による承諾を得ることなく、本資料およびその複製物に修正・加工することは堅く禁じられています。また、本資料およびその複製物を送信および配布・譲渡することは堅く禁じられています。
- 当資料（本文及びデータ等）は主として（株）Diva Analytics が入手したデータ、もしくは信頼できると判断した情報に基づき作成されていますが、情報の正確性、完全性、適宜性、将来性およびパフォーマンスについて（株）Diva Analytics は保証を行っておらず、またいかなる責任を持つものではありません。