C++ によるデリバティブ・プライシング
1.　C++ の基礎

高田勝己*

2019 年 9 月 27 日

1　Hello, world! プログラム

UTMain.cpp ファイル

```
#include <iostream>
int main()
{
    std::cout << "Hello, world!" << std::endl;
    double tmp;
    std::cin >> tmp;
    return 0;
}
```

2　Black-Sholes Model での European option の Monte Carlo による Pricer

2.1　理論

Black-Sholes のアセットのダイナミクスは、配当を無視し、金利を一定とすれば、

$$\frac{dS_t}{S_t} = rdt + \sigma dW_t \ on \ \mathbf{Q} \tag{1}$$

とかける。European option の満期でのペイオフを $f(S_T)$ とすると、オプションの現在価値 V は

$$V = e^{-rT} E\left(f(S_T)\right) \tag{2}$$

で表現できる。ここで、$f(S) = (S-K)^+$ はコールオプションの、$f(S) = (K-S)^+$ はプットオプションの最終ペイオフである。

(1) に Ito を使って、

$$d\log S_t = \left(r - \frac{\sigma^2}{2}\right) dt + \sigma dW_t$$

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

0 からオプション満期 T まで積分して、

$$\begin{aligned}\log\frac{S_T}{S_0} &= \left(r-\frac{\sigma^2}{2}\right)T+\sigma W_T \\ &= \left(r-\frac{\sigma^2}{2}\right)T+\sigma\sqrt{T}Z \end{aligned} \tag{3}$$

ここで、

$$Z \sim N(0,1)$$

(3) を S_T について解いて

$$S_T = S_0 \exp\left\{\left(r-\frac{\sigma^2}{2}\right)T+\sigma\sqrt{T}Z\right\}$$

(2) からコールオプションの現在価値は、

$$V = e^{-rT}E\left[\left(S_0\exp\left\{\left(r-\frac{\sigma^2}{2}\right)T+\sigma\sqrt{T}Z\right\}\right)^+\right]$$

である。

モンテカルロとは、大数の法則、つまり、X_i を $i.i.d.$ な確率変数の列をすれば、

$$\lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N}X_i = E(X)\ with\ prob\ 1$$

から、十分大きな N に対して、$E(X)$ を

$$E(X) \approx \frac{1}{N}\sum_{i=1}^{N}x_i$$

と近似することである。ここで、x_i は X_i の分布からのサンプル。よって、モンテカルロによるコールオプションの現在価値は、z_i を $N(0,1)$ から振ったサンプルとすれば、

$$V \approx \frac{e^{-rT}}{N}\sum_{i=1}^{N}\left(S_0\exp\left\{\left(r-\frac{\sigma^2}{2}\right)T+\sigma\sqrt{T}z_i\right\}\right)^+$$

と計算される。

2.2 コード

2.2.1 UTMain1.cpp

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

using namespace std;

// Monte carlo simulation of call options
double SimpleMonteCarlo1(double Expiry,
      double Strike,
```

```
    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 = thisSpot - Strike;
        thisPayoff = thisPayoff >0 ? thisPayoff : 0;
        runningSum += thisPayoff;
    }

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

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

    // asks for option maturity and input it
    cout << "\nEnter expiry\n";
    cin >> Expiry;

    // asks for strike and input it
    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;

    // do the monte carlo simulation
    double result = SimpleMonteCarlo1(Expiry,
        Strike,
        Spot,
        Vol,
        r,
        NumberOfPaths);

    // outputs price
    cout << "the price is " << result << "\n";

    double tmp;
    cin >> tmp;

    return 0;
}
```

このプログラムは、次の **header file** と **source file** を使う。

2.2.2 UTRandom1.h

```
#ifndef UT_RANDOM1_H
#define UT_RANDOM1_H

double GetOneGaussianBySummation();
double GetOneGaussianByBoxMuller();

#endif
```

2.2.3 UTRandom1.cpp

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

double GetOneGaussianBySummation()
{
    double result = 0;
```

```
    for (unsigned long j = 0; j < 12; j++)
        result += rand() / static_cast<double>(RAND_MAX);
    result -= 6.0;
    return result;
}

double GetOneGaussianByBoxMuller()
{
    double result;
    double x;
    double y;
    double sizeSquared;

    do
    {
        x = 2.0*rand() / static_cast<double>(RAND_MAX)-1;
        y = 2.0*rand() / static_cast<double>(RAND_MAX)-1;
        sizeSquared = x*x + y*y;
    } while
        (sizeSquared >= 1.0);

    result = x*sqrt(-2 * log(sizeSquared) / sizeSquared);
    return result;
}
```

3 解説

3.1 組み込み型 (Build-in Type)

データ型の種類	データ型	意味	大きさ (size of)
整数型 (integral type)	bool（ブール型）	boolean (true/false)	8 bits, 16 bits or 32 bits
	char（文字型）	character	8 bits
	short	short integer	16 bits
	int	integer	16 bits or 32 bits
	long	long integer	32 bits or 64bits
	long long	long integer	64 bits
浮動小数点型 (floating-point type)	float	single-precision floating-point	32 bits
	double	double-precision floating-point	64 bits
	long double	extended-precision floating-point	64, 96 or 128 bits

- 金融数学の一般的なユーザーは short,float,long long, long double を使わないことが多い。
- 浮動小数点型では、float より double を使う。double の有効桁数は 16 ぐらい。最小の正の数

2.2250738585072014e-308、最大値 1.7976931348623158e+308

- 型変換

```
bool b = 42;       // b is true
int i = b;          // i has value 1
i = 3.14;       // i has value 3
double pi = i;   // pi has value 3.0

i = 42;
if (i) i = 0;      // converts to bool
```

- signed/unsigned 型
  - bool 以外の整数型では、signed と unsigned がある。signed はマイナスにもなり得る。unsigned はゼロを含める正数である。
  - int, long とかけば signed である。unsigned にしたければ、unsigned int, unsigned long とする。unsigned は unsigned int のことである。
  - char で unsigned は使わないほうが無難
  - unsigned と signed は混ぜてつかわない。

```
unsigned u = 10;
signed i = -42;
std::cout << u + i << std::endl     //prints 4294967264!!
```

bit 数	表現できる数 (signed)	表現できる数 (unsigned)
8 bits	$-128 \sim 127$ $(-2^7 \sim 2^7 - 1)$	$0 \sim 255$ $(= 2^8 - 1)$
16 bits	$-32768 \sim 32767$ $(-2^{15} \sim 2^{15} - 1)$	$0 \sim 65535$ $(= 2^{16} - 1)$
32 bits	-2147483648〜2147483647 $(-2^{31} \sim 2^{31} - 1)$	0〜4294967295 $(=2^{32} - 1)$
64 bits	-9223372036854775808〜9223372036854775807	0〜18446744073709551615

- 整数型の場合は、
  - 負値にならないことがわかっている場合は、unsigned 型を使う。(unsigned より int を好んで使うのはタイプしやすいから？)
  - int の大きさは以前まで処理系によって 16 bits や 32 bits と異なっていたため、16 bits と想定して、signed なら $-32768 \sim 32767$ を超えれば、long を使う。unsigned なら 65535 を超えるなら unsigned long を使う。そのため、モンテカルロのパスの回数では unsined int ではなく unsigned long を使う。いまでは、int は 32bits であり、実は long と同じ。しかし、慣習上、int と long を区別している？
  - long でも、signed で-2147483648〜2147483647 を超えてしまえば、long long を使う。unsigned なら 4294967295 を超える場合は unsigned long long を使う。Derivatievs のライブラリで long long を使うことはまれ？

3.2 定数 (Literal)

定数も型をもっていて、値でデフォルトの型がきまる。

- ブール定数 (boolean literal)
  true, false
- 文字定数 (Character string literal)
  'a' // character literal, char 型の定数
  "Hello World!" // string literal。 型は char の配列で string 型ではない。コンパイラが Null character '\0' を最後につける。
  \n (new line) や \t (horizontal tab) //制御文字 (escape sequence) も char
  例えば、
  std::cout << '\n'; // 改行
  std::cout << "\tHi!\n"; // タブのスペースの次に "Hi!" そして改行
- 整数定数 (Integer literal)
  123 //decimal （10 進法）
  024 //octal （8 進法）, 0 から始まるのは 8 進法
  0x14 //hexadecimal (16 進法）, 0x または 0X から始まるのは 16 進法
  10L // long
  10u //unsigned int
  012u //8 進法の unsigned int
- 浮動小数点定数 (Floating-point literal)
  デフォルトでは double
  3.14159
  0.
  .3
  4.56e-12 //4.56×10^{-12}
  12E7L //最後に L または l がつくと long double になる。

3.3 変数 (Variable)

- Variable(変数）とは名前のついた **object** で、object とは型をもつコンピューター上のメモリである。逆に名前のついていない object もある。C++ では、変数とオブジェクトは同義に使うことが多い。
- 宣言（**declaration**) と定義 (**definition**)
  - 宣言：名前と型を指定し、名前をプログラムに知らしめる。あるファイルで、どこかほかのファイルで定義された名前を使う場合は、その変数の宣言をそのファイルに含めなくてはならない。
  - 定義：実体を決める。初期化も定義。
- ある変数を複数のファイルで使うとき、その変数の定義は 1 つのファイルでしかできない。その変数を使う他のファイルでは、使う前にその変数を宣言しなくてはならない。例えば、cout は sandard

header の iostream で宣言されているので、このファイルを#include<iostream> として含める。
初期化をおこなっている宣言は、定義でもある。定義でない宣言を明示的にいう場合、`extern` キーワードを使う。例えば、

```
extern int i;  //declares but does not define i
int j=0;    // declares and defines j
```

- 初期化 (**initialization**)
変数が宣言されたときに、最初の値をあたえその変数を定義すること。例えば、

```
double runningSum = 0;
double variance = Vol*Vol*Expiry;
unsigned long i = 0
```

などは、変数 runningSum, variance, i の初期化である。
Default initialization とは、値を与えないで変数を初期化することを言う。そのとき、変数にはデフォルト値が与えられる。デフォルト値は変数の型とどこで定義されるかによる。例えば、built-in type の変数は、関数の {} の外で定義されたときはゼロがデフォルト値となり、関数内で定義されたときは初期化されず (**uninitialized**)、その値は不定 (**undefined**) となる。不定なものをコピーしたり、これにアクセスするとエラーとなる。

3.4 入出力 (input-output)

- cout << 定数または変数 << 定数または変数 ...
とかき、左の定数や変数から出力する。
- cin >> 変数 >> 変数...
とかき、入力された値は左側の変数から順に入る。
- C++ では、入出力（以下、IO) の機能は Standard Library (SL) の一部で C++ 本来の機能ではない。標準 IO は iostream ライブラリを使うので、#include <iostream> と最初に記述する。
- istream と ostream はそれぞれ、input stream と output stream を表す型である。

型	オブジェクト名 (変数)
istream	cin
ostream	cout, cerr, clog

- インプットには istream 型のオブジェクト cin（このオブジェクトのことを **standard input** という）を使い、アウトプットには ostream 型のオブジェクト cout（このオブジェクトのことを **standard output** という）を使う。
- 他の代表的な 2 つの ostream 型のオブジェクトは、cerr と clog であり、cerr は警告やエラーメッセージを出力するときに用い、clog はプログラムの実行に関する一般的な情報を出力する際に用いる。
- \n (new line) や \t (horizontal tab) 等の制御文字が出力されると、改行やタブスペースが挿入される。
- std::endl も"\n"と同じく改行をする。
- 例えば、

```
cout << "the price is " << result << "\n";
```

とかけば、"the price is "、 result の値が順番に表示され、最後に改行がおこなわれる。

- << は（オーバーロード（多重定義）された）2 項演算子で、左の被演算子 (operand) の ostream に右の被演算子を書き出し、新たに書かれた ostream 型の cout が結果として返る。
- 上の例は、

  cout << ”the price is ”;
  cout << result ;
  cout << ”\n”;

  と同じことである。
- インプットを表す >> は（オーバーロードされた）2 項演算子で、左の被演算子の istream から右の被演算子に格納し、新たに読まれた istream 型の cin が結果として返る。
- 例えば、

  double Vol;
  cin >> Vol;

  はユーザーがインプットした値を変数 Vol に代入する。

3.5 式 (expression)

- 式 (**expression**) は何かを評価する。最も簡単なものは 1 つの定数や変数であり、いろいろな変数や定数が演算子 (**operator**) で結びつき、それが評価されると結果 (result) を産み、あるときは結果とは別の副産物 (**side effect**) を生じさせる。例えば、

  x
  1
  1+2
  x=1
  cout << ”\nEnter strike\n”

  はすべて式である。これらはすべて値を持ち、それぞれ、x の値、1, 3, 1(x の値), cout のオブジェクトで、最後の例だけ、標準出力で改行して、”Enter strike”を書き込み、また改行するという副産物を生成する。
- 演算子 (operator) : 式の演算を行う。
  - 単項演算子 (**unary operator**): +, -, ˜, &, *, ++, –, ˜, !, new, delete
    被演算子 (**operand**) を演算子の右にもつ。
    ++x　　//x をインクリメントする
  - 二項演算子 (**binary operator**): +, -, *, /, %, <<, >>, =, +=, -=, *=, /=, %=, ==, !=, <, >, <=, >=, &, | ,˜, &&, ||
    被演算子を演算子の右と左にもつ。
  - 算術演算子 (**arithmetic operator**)
    左結合で優先順位は上から下

Operator	Function	Use
+	unary plus	+ expr
−	unary minus	- expr
*	multiplication	expr * expr
/	division	expr / expr
%	remainder	expr % expr
+	addition	expr + expr
-	subtraction	expr - expr

-3+6　　//+3
1.0*3/2　//1.5
1+2.0*3　　//7.0
2/(4–1)+1.2　　//1.2
2%3+5/2　　//2+2=4

– 関係演算子 (**relational operator**): ==, !=, >, >=, <, <=
データの関係を判定し、bool 型を返す。左結合
i<j<k　//true if k>1

– 論理演算子 (**logical operator**): &&, ||, !
「かつ」や「あるいは」といった論理的な演算をおこなう。
i<j && j<k　// true if i<j<k
A || B && C || D　　// A || (B && C) || D のこと
A && B　//A を最初に評価して false であれば、B は評価されない
A || B　// A を最初に評価して true であれば、B は評価されない

– 左結合で優先順位は上から下

Operator	Function	Use
!	logical NOT	! expr
<	less than	expr < expr
<=	less than or equal	expr <= expr
>	greater than	expr > expr
>=	greater than or equal	expr >= expr
==	equality	expr == expr
!=	inequlity	expr != expr
&&	logical AND	expr && expr
||	logical OR	expr || expr

– 条件演算子 (**conditional operator**)
expression1 ? expression2 : expression3　　expression1 の評価値が真（true) なら expression2 の評価値を、expression1 が偽 (false) なら expression3 評価値を返す。
例えば、

```
double thisPayoff = thisSpot - Strike;
```

thisPayoff = thisPayoff >0 ? thisPayoff : 0; // thisPayoff = max(thisPayoff, 0)

- 演算子の例

  int i = 1;

  int j = i^2; //i*i とはならない。^は排他的論理和を表す演算子である。

  int k = ++i; //k=2, i=2 となる。prefix increment operator （前値インクリメント演算子）

  int l = i++; // l=2, i=3 となる。i++ は 1 を加算するが、戻り値はもとのまま。postfix increment operator （後置インクリメント演算子） は必要な時だけ使うようにする。

  int a =1, b = 2;

  int c = (a ==b); //c=0, ==は等価演算子 (**eqality operator**)

  c = (a=b); //c=2, =は 代入演算子 (**assignment operator**)

  runningSum += thisPayoff; // runningSum = runningSum + thisPayoff の意, 複合代入演算子 (**compound assignment operator**)

  mean *= exp(-r*Expiry); // mean = mean * exp(-r*Expiry) の意

- 結合 (**associative**) : 同じ演算子の演算順序

  a+b+c //左結合 (**left-associative**): まず a+b を評価し、次にその結果で +c を評価。(a+b)+c のこと。

  a=b=c //右結合 (**right-associative**): まず b=c を評価し、次にその結果で a=を評価。a=(b=c) のこと。

  cout << "the price is " << result << "\n" //左結合。まず cout << "the price is "を評価し cout を返す。次に cout << result を評価し cout を返す。cout << "\n"を評価して cout を返す。あるところで"the price is "を出力してから次に result を出力、最後に改行。

- 優先順位 (**precedence**): 異なる演算子の演算順序

  3+4*5 //まず、4*5 を評価し、3+ その評価値を評価する。3+(4*5) のこと

  6+3*4/2+2

  自信がないとき、または他人が分かりにくいと思うときは () で優先順位を確定すること!

- 評価順序 (order of evaluation)

  被演算子の評価順序はわからない。

  f()+g()*h()+j(); //f(), g(), h(), j() の評価順序はわからない

  int i = 3;

  cout << i << " " << ++i <<endl; //undefined!

- 演算子の優先順位と結合をまとめたもの：

Operator	Associativity	Precedence
( ) [ ] . ->	left	high
++ −− (unary)(* & + -) ~! sizeof	right	
* / %	left	
+ -	left	
<< >>	left	
< > <= >=	left	
== !=	left	
&	left	
^	left	
|	left	
&&	left	
||	left	
?:	right	
= += -= *= /= %= <<= >>= &= ^= |=	right	
,	left	low

3.6 文 (statement)

;(セミコロン）で文を終える。

- 単文 (simple statement)
  ival + 5 という式はセミコロンを付けるとそのまま式文 (expression statement) となる。

```
ival + 5;          //useless expression statement
cout << ival;  //useful expression statement
;                      // null statement
```

- 複文 (compound statement, block)
  { 単文または複文 }
  複数の文が { } で囲まれている場合は、1 つの単位であり、記述通りの順序で処理される。
- 流れ制御文 (flow of control statement)

3.7 流れ制御文 (flow of control statement)

- 条件文 (conditional statement)
  - if 文
  - switch 文
- 繰り返し文 (iterateve statement)
  - while 文
  - for 文
  - do while 文

- ジャンプ分 (jump statement)
  - break 文
  - continue 文
  - return 文

3.7.1 if 文

- if(条件式) 文

```
if( ivalue > imax )
      imax = ivalue;
```

```
if( iref > maxref )
{
    std::cout << "Error: invalid index" << std::endl;
}
```

- if(条件式) 文 1 else 文 2

```
if( ivalue > 0 ) { ... }
else if( ivalue < 0 ) {...}
else {...}
```

3.7.2 for 文

- for(初期化文; 判定式; 更新文) 実行文

```
for (unsigned long i = 0; i < NumberOfPaths; ++i)
  {
      double thisGaussian = GetOneGaussianByBoxMuller();
      thisSpot = movedSpot*exp(rootVariance*thisGaussian);
      double thisPayoff = thisSpot - Strike;
      thisPayoff = thisPayoff >0 ? thisPayoff : 0;
      runningSum += thisPayoff;
  }
```

3.7.3 while 文

- while(条件式) 文

```
int sum = 0, val = 1;
while (val <= 10)
{
    sum += val;
    ++val;
}
```

- break と continue

```
while( iflag ) {
    ...
    if( i > 0 ) continue;
    if( i < 0 ) break;
    ....
}
```

3.7.4 do while 文

- do 文　while(条件式);

```
do{
    x = 2.0*rand() / static_cast<double>(RAND_MAX)-1;

      y = 2.0*rand() / static_cast<double>(RAND_MAX)-1;
      sizeSquared = x*x + y*y;
  }  while (sizeSquared >= 1.0);
```

3.8 関数 (function)

- 関数 (**function**) とは、名前のついているプログラムの固まりであり、プログラムの他のところでそれが呼ばれる。
- プログラムを関数を使ってモジュール化すると、ソースがよみやすくなる。例えば、main() のなかでモンテカルロ計算だけを関数として切り出すと、main() だけをみて何をおこなっているのかがよくわかる。また、同じ関数をプログラムの複数のところで呼び出す場合は、1 つの関数を再利用 (reuse) でき、何回も同じコードを書かなくて済む。
- 関数の定義

int fsum(int n)　//関数の戻り値の型 (return type)　関数名 (パラメータ (**parameter**)1 の型　パラメータ 1 の名前、・・・)

```
{
    static unsigned k = 0;  //ローカル静的変数
    ++k;
    cout <<  k << endl;

    int isum = 0;  //関数内部でのローカル変数 isum
    for( int i = 0; i < = n; i ++ )  //for loop だけのローカル変数　i
    {
        isum += i;
    }
    return isum ; // return 文
}
```

- 関数の呼び出し

  int iSumTo = 10;

  int iSum = fsum(iSumTo);　// iSumTo は関数の引数 (**argument**)
- Local objects（ローカル変数）
  - name は有効範囲があり、object には存続期間がある。
  - 関数の中で定義された変数を **local variable**(ローカル変数）という。
  - **automatic object**（自動オブジェクト）:静的 (static) でないローカル変数は関数の中で変数が定義されたときにオブジェクトが生成され、処理が関数ブロックを抜けると、オブジェクトは自動的に消滅する。
  - **local static object**（ローカル静的オブジェクト）: 静的 (static) なローカル変数は関数内で最初に定義される前にオブジェクトが生成され、プログラムが終わるまで存続する。

3.9 main 関数

- すべての C++ プログラムは main と名のついた関数を含めなくてはならない。プログラムを実行すると、main 関数が呼ばれる。この関数は int 型を返し、ゼロは成功をその他の値は、なにか問題があったことを示す。よって、

  int main()

  から始め、return 文でおわる。main の後の () (parentheses) はパラメータ (**parameter)** を囲むものだが、ここではパラメータはない。
- 中括弧 (**curly braces** （または、単に braces)) は一連の文を囲む。

```
int main()
{           // left brace
           // the statements go here
}           // right brace
```

3.10 コメント

- 単一行コメント (**single-line comment**) : //以降、その行だけをコメントアウトする。
- コメントペアー (Comment pair): コメントペア /*· · · */ で挟んだものをコメントアウトする。複数の行でもコメントアウトできる。ただし、コメントペアーの中にコメントペアーはかけない。

```
/*
* Comment pairs /*   */ cannot nest.
*/
```

- Visual Studio で複数の行をコメントアウトしたい場合はそれらを選択して、Ctrl+K 続けて Ctrl+C。戻したいときは、Ctrl+K 続けて Ctrl+U。

3.11 #include

- C++ では、入出力 (**input-output** 以下、**IO**) の機能は標準ライブラリ (**standard library** 以下、SL) の一部で本来の C++ の機能ではない。C++ の機能は常にプログラマーは利用可能だが、SL の機能はそれが定義されている SL のファイル (**standard header**) を#include directive で含めなくてはならない。
- 例えば、SL の input-output の機能をつかうには、

  #include <iostream>

  と最初に宣言する。また、exp や log などの math function は、standard header の cmath で定義されているので、このファイルを

  #include <cmath>

  として含める。
- 自作の header を含めるには

  #include "· · · .h"

  とする。

3.12 名前の範囲 (scope of a name)

- 変数や関数に名前 (**name**) をつける。名前はそれが宣言された範囲 (scope) でしか有効でない。1 つの scope は { } で規定されることが多い。
- main{ } で定義されている Expiry, Strike,... は main の { } でしか有効ではない。また、for 文の j も for 文の { } でしか有効でない。j は for 文以外のところで新たに定義して使うことができる。
- main という関数名は関数の外で定義されているので、これはグローバル範囲 (**global scope**) で定義されていることになる。Global scope で一旦名前が定義されていれば、プログラムのどこでもそれを使うことができる。
- SL で定義されている名前を使うには、std::を名前の前につける。SL で定義されている名前は std という名の名前空間 (**name space**) の中にある。std::cout は std 名前空間で定義された cout のことを示す。::をスコープ演算子 (**scope operator**) という。

3.13 using 宣言 (using declaration)

- std という名前空間で定義されている名前を使う場合は、例えば、std::cin とする。
- これが面倒な場合、

  using namespace std; //すべての SL で定義されている名前に適用。ただし該当する標準ヘッダーをインクルードしなくてはならない。

  または、

  using std::cin; // cin だけに適用

  と宣言すればそれ以降、cin だけは何もつけずに名前を使うことができる。
- SL で使われる名前とそれが定義されているヘッダーは、C++ の参考書をみよ。

3.14 個別コンパイル

- 関数を使ってモジュール化すると、コードがよみやすくなる。例えば、main() のなかでモンテカルロ計算だけを関数として切り出すと、main() で何をおこなっているのかがよくわかる。
- 関数でモジュール化したとしても、関数がたくさんあり、main() のあるソースファイルが大きるなると、チームで作業がしずらい。ソースファイルを複数に分けて作業して、個別のソースファイルごとにコンパイルする。
- double GetOneGaussianBySummation() や double GetOneGaussianByBoxMuller() という関数は、main の上に double SimpleMonteCarlo1() と同様に定義をかくことができる。main のソースファイルとは違うファイルで、誰かが double GetOneGaussianBySummation() や double GetOneGaussianByBoxMuller() を使うかもしれない。その場合、double GetOneGaussianBySummation() や double GetOneGaussianByBoxMuller() の定義をソースファイルに書き、ヘッダーファイルに宣言をかく。ユーザーはヘッダーファイルだけを#include "· · · .h"でソースファイルに取り込めばよい。

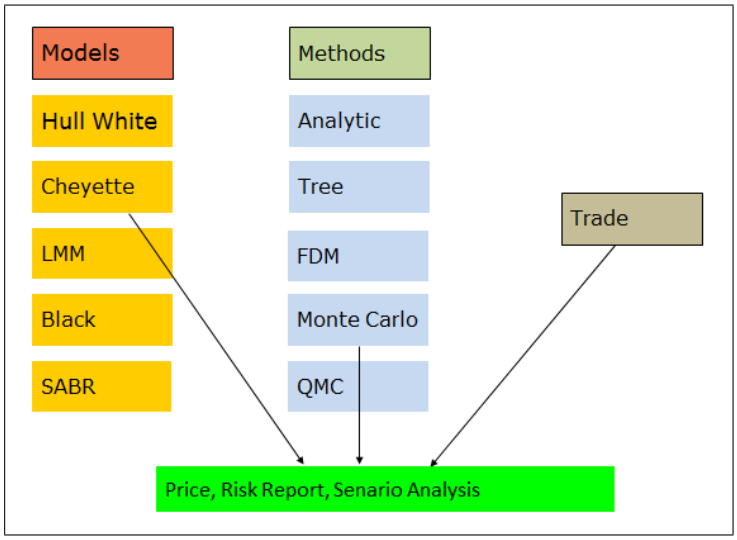
3.15 プリプロセッサ (preprocessor)

- コンパイルする前に、プリプロセッサがソースを書き換える。
- プリプロセッサが#include をみると、指定されたヘッダのコードでそれを置き換えようとする。
- define directive (#define) は後にくる名前をプリプロセッサ変数 (preprocessor variable) として定義する。
- 次の 2 つの directive は後にくる名前がプリプロセッサ変数として定義されているかをテストする。#ifdef は定義されていたら true を、#ifndef は定義されていなければ true となる。テストが true ならば、そこから#endif までが処理される。
- UTRandom1.h のヘッダがインクルードされたら、#ifndef テストが ture なら、つまり、UT_RANDOM1_H が定義されていなければ、 UT_RANDOM1_H がプリプロセッサ変数として定義され、それ以下から#endif の前までがインクルードしたファイルに書き込まれる。
- さらに、同じファイルでまた UTRandom1.h がインクルードされても、今度は UT_RANDOM1_H がが定義されているので、なにも書き込まれない。
- この仕組みにより、同じヘッダを何回もインクルードしても、その内容は 1 回だけしかインクルードされない。これを、**header guards** といい、すべてのヘッダで、最初に#ifndef · · · , #define· · · を、最後に #endif をおまじないのように書く。

```
#ifndef UT_RANDOM1_H
#define UT_RANDOM1_H
double GetOneGaussianBySummation();
double GetOneGaussianByBoxMuller();
#endif
```

4　商品、モデル及び解法

商品、モデル、解法はそれぞれ、独立に実装されなくてはならない。数値解法には Tree(Lattice), Finite difference, Monte Carlo がある。商品の属性により、Model と Method が選ばれ、これらの組み合わせで、評価、リスクの算出、シナリオ分析等が行われる。



(Figure 1) Model, Method and Trade

- 金利スワップは、モデルは静的なイールドカーブモデル、解法は Analytic(解析解）である。
- Swaption や Cap/Floor の金利オプションは、モデルは (静的な)SABR モデル、解法は Analytic(解析解）である。
- Bermudan Callable Swap は、例えば、モデルはダイナミックな Cheyette モデル、解法は Finite difference である。
- Bermudan Callable Swap は、

$$\text{Bermudan callable swap} = \text{Swap} + \text{反対の Bermudan swap}$$

に分解できるので、Swap 部分は静的なイールドカーブモデル、解法は Analytic(解析解) で、Bermudan swap だけにモデルをダイナミックな Cheyette モデル、解法は Finite difference とすることもできる。