# C++ によるデリバティブ・プライシング
# 2. ポインタと参照

高田勝己*

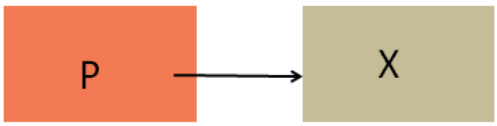
2019 年 10 月 11 日

## 1　ポインタと配列

ポインタと配列は C 及び C++ で最も low-level なデータ構造である。いまや、上位レベルのプログラミングで、ポインタや配列をそのまま使うことはまれである。ポインタは SL や boost で定義されている smart pointer を、配列も SL の vector や list を使うのが普通。とは言っても、smart pointer や vector を学ぶには、ポインタや配列を知らなくてはならない。

### 1.1　ポインタ (pointer)

- ポインタ (**pointer**) とはオブジェクトのアドレス (**address** 場所、番地）を表す値である。




- ある変数を

  ```
  int x;
  ```

  と定義したとき、変数 x が格納されているアドレスは `&x` （ `&` is **address operator**(アドレス演算子）) である。
- あるポインタ変数を

  ```
  int *p; //  p の型は int を指すポインタ。  ここの*は type modifier(型修飾子)
  ```

  と宣言すれば、`*p` （ここの* は dereference operator（逆参照演算子）) は p が指しているオブジェクトのことである。
- 上は

  ```
  int* p; // p の型は int を指すポインタ
  ```

  ともかける。しかし、

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

```
int* p, q;  // p の型は int を指すポインタで q の型は int
```
の場合は
```
int *p, q;
```
としたほうがわかりやすい。

- ポインタの指すオブジェクトのタイプは一致させる
```
double dval;
double *pd = &dval;
int *pi = pd;  //error: types of pi and pd differ
```
- すべてのポインタは初期化したほうがよい。
```
double *dp; //without initialization
double val = *dp; //run-time crash!!!
```
- null pointer で初期化することはよくある。
```
double *val = 0;
double*val2 = nullprt; //最新の方法で、nullptr はポインタ定数でどんなポインタ型にも変換される。
double *val3 = NULL: //NULL はプリプロセッサー変数で、cstdlib ヘッダで 0 と定義している。
```
とできる。

- 例
```
int main()
{
    int x = 5;
    int *p = &x;     //ここでの*は type identifier
    cout << "x = " << x << endl;

    *p = 6;        //ここでの*は dereference operator
    cout << "x = " << x << endl;
    return 0;
}
```
アウトプットは
```
x = 5
x = 6
```
となる。

## 1.2 関数ポインタ (function pointer)

- 宣言
```
int (*fp)(int); // fp は int 型のパラメータをとり、戻り値が int 型である関数のポインタであることを宣言している。
```
- 関数ポインタは、1) 関数のアドレスを取るか、2) 関数を呼ぶかのどちらかである。関数を呼ぶのでな

ければ、アドレスをとることである。例えば、

```
int next (int n)
{
    return n + 1;
}
```

という関数があれば、陽に

```
fp = &next;
```

または

```
fp = next;
```

でもよい。

- 関数をポインタから呼ぶときは、陽に

```
i = (*fp)(i);
```

または

```
i = fp(i);
```

でもわかる。() は関数を呼ぶ演算子で、fp の後ろに () があるので関数を呼んでいることが分かる。

## 1.3 配列

- 配列とポインタは同じ概念である。
- 配列は一種のコンテナ (**container**) で、同じ型のオブジェクトが順に並んでいる。配列の要素 (**element**) 数はコンパイル時にはきまっていなくてはならない。

```
int a[10];
```

とすれば、a という配列が定義されて、その中には 10 個の int 型変数が順に並んでいる。それらは

```
a[0], a[1], ···,a[9]
```

として参照できる。例えば、

```
for(int i =0; i<10; ++i)
{
    a[i] = i;
}
```

- 1 次元配列の初期化は

```
int a[3] = {1,2,3};
```

または、

```
int a[ ] = {1,2,3};
```

- C スタイルの文字列の初期化は、次の 3 つの方法

```
char b[ ] = {'H', 'e','l', 'l', 'o', '\0'};     //文字列の終わりを意味する'\0' を明示的に置く。
char b[ ] = "Hello";     // 最後の要素に'\0' が自動的に付加される。
char *b = "Hello";
```

のどれでもよいが、最初のものはあまり使わないであろう。共に要素数は 6 つである。

C では null で終わる文字列で string を表す (**C-style character strings**)。C++ では、ライブラリで定義される string 型を使うのが普通。

- 多次元配列

  例えば、2 次元配列は、1 次元配列をその要素として持つ配列。例えば、

  ```
  int a[3][4];
  ```

  は、4 要素の int 型の配列が 3 個並んだ配列である。`a[0], a[1], a[2]` は 4 要素の int 型の 1 次元配列である。

| | a[0] | | | | a[1] | | | | a[2] | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a: | a[0][0] | a[0][1] | a[0][2] | a[0][3] | a[1][0] | a[1][1] | a[1][2] | a[1][3] | a[2][0] | a[2][1] | a[2][2] | a[2][3] |

- 例えば、10x10 の単位行列を作るには

```
double matrix[10][10];
for (int i = 0; i < 10; ++i)
{
    for (int j =0; j < 10; ++j)  // for の入れ子 (nesting)
    {
        matrix[i][j] = 0.0;
    }
    matrix[i][i] = 1.0;
}
```

- 多次元配列の場合も、一番後ろの添え字の隣り合う要素がメモリ上でも隣り合うように格納される。

## 1.4 ポインタと配列

- ポインタと配列

  ```
  char a[10];
  ```

  で配列名 a は、`&a[0]` という配列先頭のアドレスを示すので、ポインタと配列名は同様に扱える。例えば、

  ```
  char *pa;
  ```

  と宣言すると、

  ```
  pa = a;
  ```

  と

  ```
  pa = &a[0];
  ```

  は同じことである。ゆえに、`*a` **や** `*pa` **は** `a[0]` のことである。

- 文字列定数 (string literal)

  `char` へのポインタが

  ```
  char *pstr;
  ```

  と定義されたとき、

  ```
  pstr = "This is a string.";
  ```

とかけば、右辺の文字列の先頭アドレスが pstr に代入される。

- ポインタと配列の相違点

  例えば、

  ```
  char a[ ] = "Hello";
  char *p = "Hello";
  ```

  でポインタは変数なので値を変えることができるが、a は配列名なので、配列の内容は変えれるがその先頭アドレスは変えれない。

  ```
  p = "Another Hello";
  ```

  で p の値は変わるが、

  ```
  a = p;
  ```

  とはできない。

## 1.5 ポインターの演算

- 1）

  ```
  int arr[ ] = {0,1,2,3,4,5,6,7,8,9};
  int *p = arr;
  ++p;   // p points to arr[1]
  int *e = &arr[10];  //最後の要素の１つ後を指すポインター、これは OK.
  for (int *b = arr; b !=e; ++b)
      cout << *b << endl;
  ```

- 2)

  ```
  int a[ ] = { 3, 4, 7, -1, 9, -6};
  ```

  int *beg = begin(a); // ライブラリ関数 (iterator ヘッダで定義) である begin は最初の要素を指すポインタを返す

  int *last = end(a); // ライブラリ関数 である end は最後の要素の１つ後を指すポインタを返す

  ```
  while (beg != last && *beg >=0)
      ++ beg;
  int b = *beg;   // b=-1
  ```

- 3)

  constexpr size_t sz = 5; // 定数式 (**constant expression**)、size_t は長さ、大きさ、サイズを表現する SL で定義される変数の型

  ```
  int arr[sz] = {1,2,3,4,5};
  int *ip = arr + sz;  // OK, しかし、参照先を取得する (dereference) ことはできない!
  int *ip2 = arr + 10;  // error:  ip2 は不定値
  int last = *(arr + 4); // last = 5, arr[4] のこと
  last = *arr + 4;     //last = 5, arr[0] + 4
  int k = ip[-3]   // k = 3, *(ip-3) のこと
  ```

Constant expression（定数式）とは評価値が変更できない式で、コンパイル時に評価される。

# 2 参照（Reference)

- ポインタとともに複合タイプ (**compound type**) の 1 つである。複合タイプの宣言は、base type + list of declarators(type modifier +name)

```
int *p, q, r;
```

- 参照はオブジェクトではなく、変数の別名 (**alias**) である。参照型の `d` を定義するときは、`&d` と宣言して初期化する。

```
int ival = 1024;
int &refval = ival; // refval は ival を参照する。（refval は ival の別名）
int & refval2;  // error: 参照型は初期化されなくてはならない。
```

変数を初期化するときは、初期値であるオブジェクトがコピーされる。参照型の初期化では、オブジェクトをつくるのではなく、初期値のオブジェクトを参照する。英語では、We bind the reference to its initializer. 参照型は必ず初期化され、その後に他のものを再参照することは許されない。

```
int i = 1024, i2 = 2048;
int &r = i, r2 = i2; //r is a ref bound to i, r2 は int 型.
r = 1025;           // i = 1025
r = i2;               // i = 2048
&r = i2;          //error: 参照型は再定義されない。そもそもここの& は？
int &refval3 = 10;  // error: 初期値は定数でなく変数でなくてはならない。rvalue でなく、lvalue でなくてはならない。
double dval = 3.14;
int &refval4 = dval; // error: 初期値の型は参照の型と同じでなくてはならない。
```

# 3 const 型修飾子 (const type modifier)

## 3.1 初期化

- 変数を後から変えたくなければ、

```
const int bufSize = 312;     //あとから変えることができないので、初期化が必要。initialized at compile time
const int bufSize2 = get_size(); //ランタイムでの初期化
const int k; //error
bufSize = 500; //error
```

- 演算子は型に応じて定義されるが、const 型が被演算子の場合それを変えてしまう演算はできない。

```
int i = 42; //plain nonconst int
const int ci = i;
int j = ci;  //ok, オブジェクトのコピーでコピーする方のオブジェクトは変わらない。
```

```
ci = j;   //error: const 型は左側の代入被演算子にはなれない。
```

## 3.2 const の参照 (reference to const, const ref)

- `const` 型のオブジェクトを参照する場合
  なるべく const ref (ref to const) を使うようにする。

```
int i = 42;
const int &r1 = i; // ok, we can bind a const int& to a plain int object.
int & r2 = i;
r1 = 0;  //error: r1 は ref to const. i は r1 を通じては変えれない。
r2 = 0; // i = 0, i は r2 を通じて変えることができる。
const int &r3 = 42; // we can bind a const int& to a literal.
const int &r4 = r1 * 2; // we can bind a const int& to an expression
int &r5 = r1 * 2;   //error
i = 45;
int j = r1;  // j = 45, r1 は i の別名。
```

- 関数パラメータとしての参照型

```
int i = 45;
int j = f (i); //
cout << i << endl; // もし int f(const int &r) なら、i=45 が保障される。int f( int &r) なら通常 i は関数 f の中で変えられていることを意味する。パラメータ型が int, double, bool の場合は、 int f(const int &r) は int f(int r) でも大差はない。
```

## 3.3 const とポインタ

- const 型を指すポインタ (pointer to const) からは、ref to const と同様、指すオブジェクトを変えれない。

```
const double pi = 3.14;
double *ptr = &pi;  //error: const double* から double *への変換はおこなわない。変換できると、ptr から pi を変えれてしまう
const double *cptr = &pi;     //ok
*cptr = 42;   // error
double dval = 3.14;
cptr = &dval; // ok なぜなら cptr そのものは const ではないから。しかし、const 型を指すポインタなので cptr を通じて dval を変えることはできない.
```

- const pointer
  ポインタは参照と異なり、オブジェクトである。よって、他のオブジェクトと同様、ポインタ自身が const ということだってあり得る。

```
int errNumb = 0;
```

```
int *const curErr = &errNumb;  // curErr は常に errNumb を指す。
int errNumb2 = 1;
currErr = &errNumb2; //error: pointer は const
*currErr = 1; //ok, pointer to nonconst
const double pi = 3.14159;
const double *const pip = &pi; // pip は const オブジェクトを指す const pointer
*pip = 3.0 // errror
```

- 2 つの const, Top-level const と low-level const

  **top-level const** とはオブジェクトそのものが const であること。**low-level const** とはポインターや参照などの複合タイプ (compound type) のベース型 (base type) の const のことを言う。top-level と low-level の 2 つをもっているのはポインタだけである。参照はオブジェクトではないので low-level しかもたない。

  コピーや代入するときには top-level const は無視される。low-level では nonconst から const への変換を行うが、逆はエラーとなる。

```
int i = 0;
int *const p1 = &i;  // p1 はtop-level const
const int ci = 42;
const int *p2 = &ci;  // p2 はlow-level const
const int *const p3 = p2; // p3 はtop-level と low-level で const
const int &r = ci;  // r はlow-level const
i = ci; // ok, ci の top-level const は無視される。
p2 = p3; // ok, p3 の top-level const は無視される。
int *p = p3;  //error: p3 は low-level const だが p は nonconst
p2 = &i; // ok, int* から const int*の変換はよい。
int &r2 = ci; // error: cannot bind int& to a const int object
const int &r3 = i;  //ok, can bind const int& to plain int
```

# 4 ライブラリ string 型

string とは SL で定義される可変長な文字列。ライブラリ string 型を使うときは、

```
#include <string>
```

として string ヘッダーをインクルードしなくてはならない。

## 4.1 定義と初期化

- 定義と初期化

```
string s1;              // デフォルトの初期化 (default initialization); s1 は空の string
string s2 = s1;          // s2 は s1 のコピー (copy initialization)
string s3(s1);               // s3 は s1 のコピー、直接の初期化 (direct initialization)
```

```
string s4 = "hello";          // s4 は string literal のコピー
string s5("hello");           // direct initialization
string s6(3, 'c');           // s6 は ccc, direct initializaton
string s7 = string(3, 'c');   //一時オブジェクトをつくりこれを s7 にコピー。
```
直接の初期化 (direct initializaton) の方がよい。

## 4.2 演算

- `string` の演算

| 演算 | 意味 |
|---|---|
| `os << s` | s を出力ストリームに書く。os が返る。 |
| `is >> s` | 空白で区切られた string を is から s へ読み込む。is が返る。 |
| `s.empty()` | s が空なら true が返る。そうでなければ、false が返る。 |
| `s.size()` | 文字数を返す。 |
| `s[n]` | s の (0 から始まる）n 番目の要素の参照を返す。 |
| `s1 + s2` | s1 と s2 の連結した string が返る。 |
| `s1 = s2` | s2 を s1 にコピーする。 |
| `s1 == s2` | s1 と s2 の文字がすべて同じなら true。大子文字を区別。 |
| `s1 != s2` | 上の逆。 |
| `<, <=, >, >=` | 辞書的順番による比較 |

- 例

```
using namespace std;
int main()
{
    string word;
    while (cin >> world)        // ”end of file”まで読みこむ。”end of file”は Ctrl + z
        cout << word << endl;
    return 0;
}
```

- `string` class では処理系によらないいくつかの型を定義している。`string::size_type` は `size()` の戻り値の型であるので、`unsigned` であることはわかる。昔は

```
string line ("line");
string::size_type len = line.size();
```

と書いていたが、今では、`size()` **の型が** `string::size_type` であるということはしらなくても

```
auto len = line.size();
```

とかける。型を `auto` にするとコンパイラーが初期値から型を決定してくれる。`auto` は型指定子 (**type specifier**) である。

- 2 項演算子はオーバーロード (overloaded, 多重定義) されている。

```
string s1 = "hello", s2 = "world";
```

```
stirng s3 = s1 + s2;
string s4 = s1 + ", " + s2 + '\n';  // s1 + ", " で ", "は string 型に変換される。
string s5 =  "hello" + ", ";  // error: 2 項演算子 + は string literal や character literal
```
（ともに string 型ではない）では定義されていない。

- **range for** と string の文字操作

for (要素の宣言 : コンテナを表すオブジェクト)　文

```
string s("Hello World!!!");
for (auto &c :  s)  // char &c = s[0] で初期化。 s の要素を上書きするので &c と参照
```
型。更新文はかかなくとも列の成分を最初から最後まで繰り返す。

```
    c = toupper(c);  //大文字にして代入。s[i] が上書きされる。
cout << s << endl;
```
とすれば、

```
HELLO WORLD!!!
```
と出力される。これを **traditional for** を使うと

```
for (decltype(s.size()) i = 0; index != s.size() && !isspace(s[i]) ; ++i)  //
```
decltype(式) で評価値の型を返す。decltype **はデクルタイプと読む**

```
    s[i] = toupper(s[i]);
```
であり、出力は

```
HELLO world!!!
```
である。

# 5 ライブラリ vector 型

- vector は同じ型のオブジェクトの集合である。vector は配列を拡張した可変長なコンテナ (**container**) の 1 つである。string は char 型の vector であるとも考えられる。vector 型を使うときは、

```
#include <vector>
```
として vector ヘッダーをインクルードしなくてはならない。

- vector は **class template** なので、要素の型を <> で指定する。
- 定義と初期化

```
vector<int>  ivec;  // empty
vector<int> ivec2(ivec); // ivec の要素を ivec2 にコピー
vector<int> ivec3 = ivec;
vector<string> svec(ivec2); //error: 型違い
vector<string> svec2 = {"a", "an", "the"}; //リストによる初期子 (list initialization)
```
をコピー

```
vector<string> svec3{"a", "an", "the"};  // 直接の list initialization
vector<int> ivec4(10, -1); // (-1, -1, ···, -1)
vector<int> ivec5(10); // サイズは 10。 各要素はゼロでデフォルト初期化
vector<string> svec4(10); // サイズは 10。 各要素は空でデフォルト初期化
```

- vector の演算

| 演算 | 意味 |
|---|---|
| v.empty() | v が空なら true が返る。そうでなければ、false が返る。 |
| v.size() | 要素数を返す。 |
| v.push_back(t) | v の最後に t の要素を加える。 |
| v[n] | v の (0 から始まる）n 番目の要素の参照を返す。 |
| v1 = v2 | v2 を v1 にコピーする。 |
| v1 == v2 | v1 と v2 の要素がすべて同じなら true。 |
| v1 != v2 | 上の逆。 |
| <, <=, >, >= | 辞書的順番による比較 |

- push_back()

```
vector<int> v;  //空の vector, 100 つの要素があると事前にわかっていてもこの方が効率的!!
for (int i =0; i != 100; ++i) // range-for は使えない。 renge-for は固定長なコンテイナが必要
    v.push_back(i); // ランタイム時に要素を後ろから追加
```

これを初心者の人は

```
for (decltype(v.size()) i = 0; i !=10; ++i)
    v[i]=i;
```

としてしまう。このエラーはコンパイル時にわからない。

**問題 1** なぜ、いけないのか？

- 例

```
vector<int> v{1,2,3,4,5,6,7,8,9,10}
for (auto &i :  v)
    i *=i;
for (auto i :  v)
    cout << i << " ";
cout << endl;
```

- size_type は型を伴った vector で定義されるので、例えば、vector<int>::size_type である。

# 6 Iterator

- SL では、vector や string 等の多くのコンテナ (container) を提供している。イタレータは SL のコンテナの要素であるオブジェクトにアクセスするときに、ポインタのように使われる。
- イタレータ (**iterator**) はポインタと同様、オブジェクトに間接的にアクセスするオブジェクト。ポインタのように、&b でそのアドレスを取得できない。iterator を持っているコンテナは、iterator を返すメンバー関数（例えば、begin や end）をもっている。
- iterator はコンテナ内の要素を指すか、最後の次の要素（dereference はできない）を指すかである。

- iterator の演算に関しては、ポインタの演算とほぼ同じである。

```
string s("some string");
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++i)
    *it = toupper(*it); //iterator から文字にアクセス
```

  上で、it < s.end() としてもよいが、他のコンテナの iteratror で演算子 < が定義さているとは限らない。これが、C++ のプログラマーは subscript の場合でも < よりも != を好んで使う理由である。

- vector や string の siza_type が実際にはどういう型か知る必要がないのと同様に、iterator の実際の型を知る必要はない。ライブラリでは iterator の 2 つの型を定義している。iterator と const_iterator である。

```
vector<int> v(2,0);
vector<int>::iterator it=v.begin(); //vector<int> の要素の読み書きができる。
*it =3;
string s("This class is boring.");
string::iterator it2 = s.begin();  // string の文字の読み書きができる。
*it2 = 't';
vector<int>::const_iterator it3 = v.begin(); //読むだけ。
int i = *it3;
vector<string>::const_iterator it4  = s.begin(); // string の文字を読むだけ。
bool isEmpty = (*it).empty();  // it->empty() でもよい
```

- begin と end

```
vector<int> v;
const vector<int> cv;
auto it1 = v.begin(); // it1 の型は vector<int>::iterator
auto it2 = cv.begin(); // it2 の型は vector<int>::const_iterator
auto it3 = v.cbegin(); // it3 の型は vector<int>::const_iterator
```

- iterator の算術演算

  Binary serch: text は順序付けされたコンテナで sought は求めたいコンテナの要素。

```
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg ) / 2;
while (mid != end && *mid !=sought )
{
    if (sought < *mid)
        end = mid;
    else
        beg = mid + 1;
    mid = beg + (end - beg ) / 2;
}
```

## 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。