効果的なテンプレートテクニック

– ゲームプログラミングの最適化手法 –

2014 年 2 月 18 日　初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 2 月 18 日	板垣 衛	（初稿）

■ 目次

■ 概略

テンプレートをゲームプログラミングに効果的に利用するための方法を解説。

■ 目的

本書は、テンプレートの性質を理解し、特に処理速度に重点を置いた活用を行うことを目的とする。一部、生産性の向上についても言及する。

なお、テンプレート自体の解説は目的としないため、細かい仕様の説明は行わない。

■ 参考書籍

本書の内容は、「Ｃ＋＋テンプレートテクニック」（著者：επιστημη＋高橋晶　発行：ソフトバンククリエイティブ）を大いに参考にしている。サンプルを真似ている箇所も多い。

■ プログラムサイズの問題／ソースファイルの書き方

まず、テンプレートはプログラムサイズが肥大化しがちである点とコンパイルに時間がかかる点に注意。

これらの点に関する説明を、別紙の「チーム開発のためのコーディング手法」に示す。

■ メタプログラミング

テンプレートを活用すると、コンパイル時に処理を実行して結果を得る、いわゆる「メタプログラミング」が可能である。

コンパイル時に定数化されるため、若干コンパイル時間がかかるものの、コンパイル時の型安全、プログラムサイズ、処理効率の面で良好な結果が得られる。

▼ 活用例①：min() ／ max() 関数

テンプレート関数による min() 関数のサンプルを示す。
テンプレートを使用すると、オーバーロードにより、3 値以上の比較にも対応し易い。
なお、min()関数はサンプルを示さないが、同様の作り方となる。

【サンプル】

テンプレートによる max() 関数のサンプル：※inline 指定を省略しているので注意

```
//max() 関数　※関数のオーバーロードで複数の値に対応
template<typename T> T max(T n1, T n2){                    return n1 > n2 ? n1 : n2; }
template<typename T> T max(T n1, T n2, T n3){              return n1 > n2 ? n1 : max(n2, n3); }
template<typename T> T max(T n1, T n2, T n3, T n4){        return n1 > n2 ? n1 : max(n2, n3, n4); }
template<typename T> T max(T n1, T n2, T n3, T n4, T n5){ return n1 > n2 ? n1 : max(n2, n3, n4, n5); }
```

使用例：

```
const int v1 = max(1, 2);
const int v2 = max(3, 4, 5);
const int v3 = max(6, 7, 8, 9);
const int v4 = max(10, 11, 12, 13, 14);
printf("{%d, %d, %d, %d}¥n", v1, v2, v3, v4);}
```

↓※コンパイル後のコード結果（Release ビルドの逆アセンブルで確認）

```
010C1CC0  push        ebp
010C1CC1  mov         ebp,esp
010C1CC3  and         esp,0FFFFFFF8h
      const int v1 = max(1, 2);
      const int v2 = max(3, 4, 5);
      const int v3 = max(6, 7, 8, 9);
      const int v4 = max(10, 11, 12, 13, 14);
      printf("{%d, %d, %d, %d}¥n", v1, v2, v3, v4);
010C1CC6  push        0Eh     ←定数化されている（printf への引数は逆順にスタックに積まれる）
010C1CC8  push        9       ←
010C1CCA  push        5       ←
010C1CCC  push        2       ←
010C1CCE  push        10E37B8h
010C1CD3  call        printf (010C74D7h)
010C1CD8  add         esp,14h
```

テンプレート関数に与えられた値が全てリテラル値である場合、コンパイル時に計算が済まされ、定数化される。

サンプルで示している通り、テンプレート関数がネストしていも問題がない（ネスト／再帰の限界はコンパイラによって規定されている）。Visual C++ 2013 で確認したところでは、if 文や while 文のような制御文が含まれていても、コンパイル時に計算可能なら定数化される。

なお、引数が変数の場合は、実行時に関数（もしくはインライン関数）として処理される。

▼ 【比較解説】テンプレート以外のメタプログラミング手法との比較

以下より、若干本筋からそれるが、より効果的なメタプログラミングを意識するために

も、テンプレートと他のメタプログラミングの手法との比較を示す。

▼ 【比較①】#define マクロ（使い勝手）

マクロでも同様の処理が可能であり、同様に結果が定数化されるか、もしくは、計算式としてインライン展開される。

ただし、マクロは関数ではないため、テンプレート関数と比較すると、型チェックが曖昧な点や、オーバーロードができない点、長めの処理が書きにくい点などの不便な面がある。

同様の処理を#deinfe マクロで実装した場合の例：

```
//max() マクロ　※オーバーロードできないため、それぞれの名前が異なる
#define max2(n1, n2)             (n1 > n2 ? n1 : n2)
#define max3(n1, n2, n3)         (n1 > n2 ? n1 : max2(n2, n3))
#define max4(n1, n2, n3, n4)     (n1 > n2 ? n1 : max3(n2, n3, n4))
#define max5(n1, n2, n3, n4, n5) (n1 > n2 ? n1 : max4(n2, n3, n4, 5))
```

マクロ使用時固有の問題もある。下記のような用法はテンプレート関数では問題ないが、マクロでは問題になる。

マクロで問題になる例：

```
#define max(n1, n2) (n1 > n2 ? n1 : n2)
int a = 1;
int b = 1;
int c = max(++a, b);
//結果：c は 2 ではなく 3 になってしまう。
//プリプロセッサがマクロを展開すると、(++a > b ? ++a : b) という式になってしまうためである。
//テンプレート関数では同様の問題は起こらない。
```

▼ 【比較②】#define マクロ（完全な定数化）

テンプレートでは期待通りの定数化が行えない場合がある。

Visual C++ 2013 で確認したところでは、下記のような箇所で問題が確認された。マクロの場合は全て問題がない。

テンプレートの定数化が期待通りに行われないケース：

```
//max()  … テンプレート関数
//max5() … マクロ

//定数
static const int sc1 = max(1, 2, 3, 4, 5);  //OK
static const int sc2 = max5(1, 2, 3, 4, 5); //OK
const int c1 = max(1, 2, 3, 4, 5);          //OK
const int c2 = max5(1, 2, 3, 4, 5);         //OK

//列挙
enum
{
//    e1 = max(1, 2, 3, 4, 5),  //NG:コンパイルエラー
      e2 = max5(1, 2, 3, 4, 5), //OK
};
```

```
//クラス内の定数／初期値
class CClass
{
public:
	//定数
//	static const int sc1 = max(1, 2, 3, 4, 5);  //NG:コンパイルエラー
	static const int sc2 = max5(1, 2, 3, 4, 5); //OK
	const int c1 = max(1, 2, 3, 4, 5);          //OK
	const int c2 = max5(1, 2, 3, 4, 5);         //OK
	//コンストラクタ
	CClass() :
		v1(max(1, 2, 3, 4, 5)), //OK
		v2(max5(1, 2, 3, 4, 5)) //OK
	{}
	//フィールド
	int v1;
	int v2;
};
```

▼ 【比較③】C++11 仕様対応版の min() ／ max() 関数

C++11 では、テンプレートの可変長引数の仕様が追加されている。

「活用例①」のサンプルは、2～5 個の数字にオーバーロードで対応していたが、C++11 の仕様に基づけば、これを可変長にすることができる。

以下、そのサンプルを示す。

【サンプル】

テンプレートによる max() 関数改良版のサンプル：※inline 指定を省略しているので注意

```
//値が二つの max()
template<typename T1, typename T2>
T1 max(T1 n1, T2 n2){ return n1 > n2 ? n1 : n2; }

//値が三つ以上の max() : 再帰処理（注：テンプレートの特殊化ではなく、関数のオーバーロードで再帰を終結させている）
template<typename T1, typename T2, typename T3, typename... Tx>
T1 max(T1 n1, T2 n2, T3 n3, Tx... nx){ return max(max(n1, n2), n3, nx...); } //nx が空になったら値が二つの方が呼ばれる
```

使用例：

```
const int v1 = max(1, 2);
const int v2 = max(3, 4, 5);
const int v3 = max(6, 7, 8, 9);
const int v4 = max(10, 11, 12, 13, 14);
const int v5 = max(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14);
const int v6 = max(6, 5, 4, 3, 2, 1);
const int v7 = max(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
printf("{%d, %d, %d, %d, %d, %d, %d}¥n", v1, v2, v3, v4, v5, v6, v7);
```

↓※コンパイル後のコード結果（Release ビルドの逆アセンブルで確認）

```
00821D60  push        esi
	const int v1 = vmax(1, 2);
	const int v2 = vmax(3, 4, 5);
	const int v3 = vmax(6, 7, 8, 9);
	const int v4 = vmax(10, 11, 12, 13, 14);
	const int v5 = vmax(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14);
	const int v6 = vmax(6, 5, 4, 3, 2, 1);
00821D61  mov         eax,6          ← 定数化しきれずに部分的なインライン展開が起こっているもの
00821D66  mov         esi,2          ←
00821D6B  cmp         eax,esi        ←
00821D6D  cmovg       esi,eax        ←
```

```
    const int v7 = vmax(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
00821D70  sub         esp,34h
00821D73  call        vmax<int,int,int,int,int,int,int,int,int,int,int,int,int,int,int> (0824C30h)
                                ← パラメータが長すぎて定数化されず、ランタイムの関数呼び出し化したもの
    printf("{%d, %d, %d, %d, %d, %d, %d}¥n", v1, v2, v3, v4, v5, v6, v7);
00821D78  push        eax           ← 関数の結果を printf に受け渡し
    const int v1 = vmax(1, 2);
    const int v2 = vmax(3, 4, 5);
    const int v3 = vmax(6, 7, 8, 9);
    const int v4 = vmax(10, 11, 12, 13, 14);
    const int v5 = vmax(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14);
    const int v6 = vmax(6, 5, 4, 3, 2, 1);
00821D79  mov         eax,1         ← 定数化しきれずに部分的なインライン展開が起こっているもの（上部の続き）
00821D7E  cmp         esi,eax       ←
00821D80  cmovg       eax,esi       ←
    printf("{%d, %d, %d, %d, %d, %d, %d}¥n", v1, v2, v3, v4, v5, v6, v7);
00821D83  push        eax           ←インラインで計算した結果を printf に受け渡し
00821D84  push        0Eh           ←他は定数化されている（printf への引数は逆順にスタックに積まれる）
00821D86  push        0Eh           ←
00821D88  push        9             ←
00821D8A  push        5             ←
00821D8C  push        2             ←
00821D8E  push        849BE4h
00821D93  call        printf (0827767h)
00821D98  add         esp,54h
```

Visual C++ 2013 で確認したところでは、少し長めの処理を与えたところ、完全には定数化されなかった。

▼ 【比較④】C++11 の constexpr（今後のメタプログラミングへの期待）

定数を算出する為のメタプログラミングは、C++11 の仕様で更に強化されている。

「constexpr」という、定数式を書くための仕様が追加されている。

constexpr の記述例：※普通の const 型と同様に使える

```
constexpr int x = 10;
constexpr int y = 1 + 2 * 3;
```

constexpr は、テンプレート関数と同様に、関数の形態で定数を記述することができる。

前述のテンプレートの再帰では一部定数化されなかったが、constexpr では完全に定数化される。

constexpr による max() 関数改良版のサンプル：※前述のサンプルに constexpr を付けただけ

```
//値が二つの max()
template<typename T1, typename T2>
constexpr T1 max(T1 n1, T2 n2){ return n1 > n2 ? n1 : n2; }

//値が三つ以上の max() : 再帰処理（注：テンプレートの特殊化ではなく、関数のオーバーロードで再帰を終結させている）
template<typename T1, typename T2, typename T3, typename... Tx>
constexpr T1 max(T1 n1, T2 n2, T3 n3, Tx... nx){ return max(max(n1, n2), n3, nx...); }
```

Visual C++ 2013 では constexpr は未実装なので、GCC(4.8.2)で確認したところでは、完全に定数化されていることが確認できた。（GCC は 4.6 以降の実装とのこと）

constexpr は現状制約が厳しく、凝った計算は行えない。関数内では変数が使用できず、if 文も使えない。変数が使えないので for 文も使えない。やはり再帰でループ処理を行う必要がある。さらに、仮引数と戻り値にはリテラル型しか使えない。

なお、constexpr 関数にリテラル値以外（変数）を渡すと、コンパイル時ではなく、実行時間数として処理される。

C++14 では constexpr に普通に変数や if 文、クラスが使えるようになるらしい。そのような処理もコンパイル時に評価されるらしいので、「コンパイル時に文字列から CRC 値を算出する（できれば同時に文字列リテラルは消滅）」といった定数式が書けそうである。これが使えれば、処理効率の向上とプログラムサイズの削減になる。

なお、現状でもネットを検索すると constexpr と「ユーザー定義リテラル」（C++11 仕様）を活用した CRC 値算出のソースが見つかる。ユーザー定義リテラルと組み合わせると、「unsigned int crc = "文字列"_CRC;」のような書き方ができて便利に使える。（「_CRC」の部分が、「12.3f」の場合の「f」に相当する。「ユーザー定義リテラル」を用いると、このようなリテラル値用のサフィックスを任意に追加することができる。）

▼ 活用例②：lengthOfArray() 関数

テンプレート関数による lengthOfArray() 関数のサンプルを示す。

これは、配列の要素数を取得する方法のサンプルである。

【サンプル】

テンプレートによる lengthOfArray() 関数のサンプル：

```
//一次元の配列要素数を取得
template<typename T, std::size_t N1>
inline std::size_t lengthOfArray1(const T (&var)[N1]) //配列の要素数が渡される
{
    return N1;
}
//二次元の配列要素数を取得
template<typename T, std::size_t N1, std::size_t N2>
inline std::size_t lengthOfArray2(const T (&var)[N1][N2])   //配列の要素数が渡される
{
    return N2;
}
//三次元の配列要素数を取得
template<typename T, std::size_t N1, std::size_t N2, std::size_t N3>
inline std::size_t lengthOfArray3(const T (&var)[N1][N2][N3])    //配列の要素数が渡される
{
    return N3;
}
```

使用例：

```
int var1[10]         = {};
int var2[20][30]     = {};
int var3[40][50][60] = {};
printf("var1[%d]¥n",         lengthOfArray1(var1));
```

```
printf("var2[%d][%d]¥n",     lengthOfArray1(var2), lengthOfArray2(var2));
printf("var3[%d][%d][%d]¥n", lengthOfArray1(var3), lengthOfArray2(var3), lengthOfArray3(var3));
```

↓実行結果

```
var1[10]
var2[20][30]
var3[40][50][60]
```

【参考】lengthOfArray() マクロのサンプル：

```
//一次元の配列要素数を取得
#define lengthOfArray1(var) (sizeof(var) / sizeof(var[0]))
//二次元の配列要素数を取得
#define lengthOfArray2(var) (sizeof(var[0]) / sizeof(var[0][0]))
//三次元の配列要素数を取得
#define lengthOfArray3(var) (sizeof(var[0][0]) / sizeof(var[0][0][0]))
```

▼ 活用例③：テンプレートクラスの特殊化を利用した再帰メタプログラミング

#define マクロでは定数化できないような計算を、テンプレートの活用で実現する方法を解説する。

「べき乗」をコンパイル時に計算する方法をサンプルとして示す。

【サンプル】

テンプレートクラスによるコンパイル時のべき乗算出のサンプル：

```
//べき乗
template<int N, int E>
struct Pow{
    static const int value = N * Pow<N, E - 1>::value;    //再帰
};
//べき乗：再帰終点のための特殊化
template<int N>
struct Pow<N, 0>{
    static const int value = 1;
};
```

使用例：

```
const int n1 = Pow<2, 0>::value;
const int n2 = Pow<2, 1>::value;
const int n3 = Pow<2, 2>::value;
const int n4 = Pow<2, 3>::value;
const int n5 = Pow<10, 4>::value;
printf("{%d, %d, %d, %d, %d}¥n", n1, n2, n3, n4, n5);
```

↓実行結果

```
{1, 2, 4, 8, 10000}
```

↓※コンパイル後のコード結果（Release ビルドの逆アセンブルで確認）

```
    const int n1 = Pow<2, 0>::value;
    const int n2 = Pow<2, 1>::value;
    const int n3 = Pow<2, 2>::value;
    const int n4 = Pow<2, 3>::value;
    const int n5 = Pow<10, 4>::value;
    printf("{%d, %d, %d, %d, %d}¥n", n1, n2, n3, n4, n5);
00292230  push        2710h   //←定数化されている（printf への引数は逆順にスタックに積まれる）
00292235  push        8       //←
00292237  push        4       //←
00292239  push        2       //←
0029223B  push        1       //←
```

```
0029223D  push        2B9E80h
00292242  call        printf (0297787h)
```

この手法を用いると、コンパイル時に確実に定数化される。

処理を解説する。サンプルとして「Pow<2, 3>」が宣言された時の動作を説明する。

宣言に基づいて、構造体定義の実体「struct Pow<2, 3>」が生成されると、その構造体内でまた「Pow<2, 3-1>」が宣言されるため、再帰して「struct Pow<2, 2>」が生成される。

処理はさらに再帰し、同様に「struc Pow<2, 1>」が生成される。

最後に「struct Pow<2, 0>」の生成段階になると、「テンプレートクラスの特殊化」が作用し、「template struct Pow<N, 0>」の方に適合して生成される。このテンプレートは中で再帰していないため、定数「1」が定義された構造体として完結する。

その後は再帰をさかのぼって、順次構造体の定義が実体化されていき、static 定数 value の値が確定する。

なお、クラスではなく構造体を使用しているのは、単にメンバーのデフォルトが public スコープだからである。クラスを使う場合は明示的に「public:」宣言する必要がある。

また、クラス／構造体の static 定数メンバーは、int 型しか初期値を与えることができないので、浮動小数点型の値をこの手法で生成することはできない。

もう一つ注意点として、「特殊化」は、テンプレートクラスにしか適用できない。つまり、テンプレート関数には「特殊化」を用いることができない。テンプレート関数の場合は、「関数のオーバーロード」を用いることで、同様の再帰処理を行う事ができる。ただし、コンパイル時の定数化が確実に行われるとは限らない点に注意。

▼ 活用例④：STATIC_ASSERT（静的アサーション）

実行時にチェックされる assert() ではなく、コンパイル時にチェックされる静的アサーションを説明する。

例えば、何らかの処理の制約により「ある構造体のサイズが 20 を超えたらコンパイルエラーにする」というチェックを行いたいとする。

以下、幾つか静的アサーションの方法を説明する。

【サンプル①：最も一般的な方法】

実装例：

```
//静的アサーションマクロ
#define STATIC_ASSERT(expr) ¥
     typedef char _static_assert_t[ (expr) ]
```

使用例：

```
struct SAMPLE { … }
void func()
```

```
{
    STATIC_ASSERT(sizeof(SAMPLE) <= 20);
```

↓コンパイル結果

```
error C2466: サイズが 0 の配列を割り当てまたは宣言しようとしました。
```

メッセージの意味が分かりにくいが、きちんと問題の箇所でエラーとなる。

条件式の結果が偽（＝0）になると、配列サイズが 0 の型が定義されることによってコンパイルエラーになる。

この方法の利点は、関数の中や外、クラス定義の中など、わりとどんな場所にも書ける事である。

なお、上記のマクロは、Visual C++ 2013 であらかじめ定義されている「_START_ASSERT」マクロと全く同じ内容である。

【サンプル②：Boost C++ のスタイル】

実装例：

```
//静的アサーションマクロ
template <bool> struct static_assertion;
template <> struct static_assertion<true>{ class FAILED; };
#define STATIC_ASSERT(condition) ¥
    typedef static_assertion<condition>::FAILED STATIC_ASSERT_FAILED
```

使用例：

```
struct SAMPLE { … }
void func()
{
    STATIC_ASSERT(sizeof(SAMPLE) <= 20);
```

↓コンパイル結果

```
error C4430: 型指定子がありません - int と仮定しました。メモ: C++ は int を既定値としてサポートしていません
error C2146: 構文エラー : ';' が、識別子 'STATIC_ASSERT_FAILED' の前に必要です。
error C2065: 'STATIC_ASSERT_FAILED' : 定義されていない識別子です。
error C2027: 認識できない型 'static_assertion<false>' が使われています。
```

Boost C++ では、テンプレートクラスの特殊化を利用し、より確実なエラーになる構文を提供しているが、そのエラーメッセージが大量でわかりにくい。

この方法の利点は、マクロと同様に、関数の中や外、クラス定義の中など、わりとどんな場所にも書ける事である。ただし、先に紹介したマクロのほうがまだ扱い易い。

【サンプル③：Loki のスタイル】

実装例：

```
//静的アサーションマクロ
template<bool> struct StaticAssertion;
template<> struct StaticAssertion<true> {};
#define STATIC_ASSERT(expr, msg) ¥
    { StaticAssertion<expr> StaticAssertionFailure_##msg; StaticAssertionFailure_##msg; }
```

使用例：

```
struct SAMPLE { … }
void func()
{
    STATIC_ASSERT(sizeof(SAMPLE) <= 20, 構造体サイズエラー);
```

↓コンパイル結果

```
error C2079: 'StaticAssertionFailure_構造体サイズエラー' が 未定義の struct 'StaticAssertion<false>' で使用しています。
```

Loki というライブラリも、Boost C++と同様に、テンプレートクラスの特殊化を利用している。Boost C++ と比べて、エラーメッセージが簡潔で、かつ、任意のメッセージが指定できる点が優れている。

ただし、下記の制約がある点には注意が必要。

- 関数の中にしか記述できない。
- メッセージ中に空白や「_」（アンダースコア）以外の記号を指定してはいけない。ダブルクォーテーションで囲むのもダメ。
- Visual C++ 2013 では日本語のメッセージが使えたが、普通は使えないものと見るべき。

【サンプル④：C++11 のスタイル】

使用例：

```
struct SAMPLE { … }
void func()
{
    static_assert(sizeof(SAMPLE) <= 20, "Structure size limit overed!");
```

↓コンパイル結果

```
error C2338: Structure size limit overed!
```

C++11 の仕様では、static_assert()が標準で実装している。Visual C++ 2013 でも使用できる。任意のエラーメッセージを指定し、関数の内外どこにでも記述できる。エラーメッセージはダブルクォーテーションで囲んで空白や記号も使用できるが、日本語が正常に扱えないので注意。ソースファイルがユニコードなら日本語も表示される。

■ コーディングの効率化

テンプレートを活用して、コーディングを効率する手法を紹介する。

▼ コンストラクタテンプレートで効率的なコピーコンストラクタ

テンプレートクラスの中のメンバー関数やコンストラクタをテンプレート関数にすることができる。これを利用して、簡潔にコピーできるテンプレートクラスを作ることができる。

以下、サンプルを示す。

例：テンプレートコンストラクタを利用した構造体のサンプル

```
//座標型テンプレート構造体
template<typename T>
struct POINT
{
    T x;
```

```
	T y;
	//通常コンストラクタ
	POINT(T x_, T y_) :
		x(x_),
		y(y_)
	{}
	//コピーテンプレートコンストラクタ
	template<typename U>
	POINT(POINT<U>& o) :
		x(static_cast<T>(o.x)),
		y(static_cast<T>(o.y))
	{}
	//コピーテンプレートオペレータ
	template<typename U>
	POINT<T>& operator=(POINT<U>& o)
	{
		x = static_cast<T>(o.x);
		y = static_cast<T>(o.y);
		return *this;
	}
};
```

使用例：

```
POINT<int> p1(1, 2);	//int 型の座標型
POINT<float> p2(p1);	//float 型の座標型に、int 型の座標型のデータを、テンプレートコンストラクタを利用してコピー
POINT<long> p3(0, 0);	//long 型の座標型
p3 = p2;		//long 型の座標型に、float 型の座標型のデータを、テンプレートオペレータを利用してコピー
printf("p1=(%d, %d)¥n", p1.x, p1.y);
printf("p2=(%.1f, %.1f)¥n", p2.x, p2.y);
printf("p3=(%ld, %ld)¥n", p3.x, p3.y);
```

↓実行結果

```
p1=(1, 2)
p2=(1.0, 2.0)
p3=(1, 2)
```

▼ 高階関数を利用した無駄のない処理

「高階関数」とは、「関数を受け渡す関数」のことである。関数を変数に代入できる「関数型言語」でよく使われる手法。「手続き型言語」の C++では、「高階関数」の実現には、「関数ポインター」「関数オブジェクト」、そして、C++11 で追加された「ラムダ式」といった方法を用いる。

ここでは、テンプレートと関数オブジェクトを組み合わせた処理の最適化手法を解説する。

まず、サンプルとして下記の処理要件を実装するものとする。

- int 型の配列とその個数を受け取り、値を 10～100 の範囲に丸めて返す。
- この配列は二つ同時に渡される。
- 丸める前と後の値をそれぞれログ出力する。
- ログ出力の際、配列の合計値、平均値もいっしょに出力する。

サンプルのためのやや強引な要件ではあるが、ゲームプログラミングの現場でも、一時

的なデータ確認のためのログ出力などで、似た要件は実際に発生する。この時、「今ちょっと確認したいだけだから」と、単純にコピペも多用して一気に作ってしまいがちだが、その後、丸めの範囲を変えて再確認したり、Excel にコピペしやすい書式に調整したりなど、意外に長々とメンテすることもある。

　このような背景を踏まえて、サンプルを題材にして、どのように最適化していくかを、順を追って説明する。

● 処理要件サンプル

【サンプル】

呼び出し側：

```
int data1[] = {1, 2, 3, 39, 200, 53, 8, 74, 12};
int data2[] = {13, 6, 76, 43, 23, 125, 1 };
func(data1, lengthOfAray(data1), data2, lengthOfAray(data2));     //←この関数を実装するのが、このサンプルの要件
```

↓実行結果

```
<BEFORE>
data1= 1 2 3 39 200 53 8 74 12 (sum=392, avg=43.6)
data2= 13 6 76 43 23 125 1 (sum=287, avg=41.0)
<AFTER>
data1= 10 10 10 39 100 53 10 74 12 (sum=318, avg=35.3)
data2= 13 10 76 43 23 100 10 (sum=275, avg=39.3)
```

● 最適化前の状態

原型：全く最適化されていない状態

```
//データ丸め処理
void func(int data1[], int n1, int data2[], int n2)
{
	//丸め実行前ログ出力
	printf("<BEFORE>¥n");
	//ログ出力：data1
	printf("data1=");
	int sum1 = 0;
	for (int i = 0; i < n1; ++i)
	{
		sum1 += data1[i];
		printf(" %d", data1[i]);
	}
	printf(" (sum=%d, avg=%.1f)¥n", sum1, static_cast<float>(sum1) / static_cast<float>(n1));
	//ログ出力：data2
	printf("data2=");
	int sum2 = 0;
	for (int i = 0; i < n2; ++i)
	{
		sum2 += data2[i];
		printf(" %d", data2[i]);
	}
	printf(" (sum=%d, avg=%.1f)¥n", sum2, static_cast<float>(sum2) / static_cast<float>(n2));
	//丸め処理：data1
	for (int i = 0; i < n1; ++i)
	{
		if (data1[i] < 10)
			data1[i] = 10;
```

```
            else if (data1[i] > 100)
                  data1[i] = 100;
      }
      //丸め処理：data2
      for (int i = 0; i < n2; ++i)
      {
            if (data2[i] < 10)
                  data2[i] = 10;
            else if (data2[i] > 100)
                  data2[i] = 100;
      }
      //丸め実行後ログ出力
      printf("<AFTER>¥n");
      //ログ出力：data1
      printf("data1=");
      sum1 = 0;
      for (int i = 0; i < n1; ++i)
      {
            sum1 += data1[i];
            printf(" %d", data1[i]);
      }
      printf(" (sum=%d, avg=%.1f)¥n", sum1, static_cast<float>(sum1) / static_cast<float>(n1));
      //ログ出力：data2
      printf("data2=");
      sum2 = 0;
      for (int i = 0; i < n2; ++i)
      {
            sum2 += data2[i];
            printf(" %d", data2[i]);
      }
      printf(" (sum=%d, avg=%.1f)¥n", sum2, static_cast<float>(sum2) / static_cast<float>(n2));
}
```

丸め処理が２箇所に記述されており、範囲の判定もそれぞれに書かれている。丸めの範囲を変更する際に、２箇所の修正となり、ミスを起こしやすい状態である。

● 最適化①：共通関数化

まず、最初の最適化として、この処理の共通化を考える。単純に、共通関数で処理するように変更する。

最適化①：最も重要なロジック部分を共通化する

```
//データ丸め共通処理部分
int round_common(int data)
{
      if (data < 10)
            data = 10;
      else if (data > 100)
            data = 100;
      return data;
}
//データ丸め処理
void func(int data1[], int n1, int data2[], int n2)
{
      //丸め実行前ログ出力
      printf("<BEFORE>¥n");
      //ログ出力：data1
      printf("data1=");
      int sum1 = 0;
      for (int i = 0; i < n1; ++i)
```

```
	{
		sum1 += data1[i];
		printf(" %d", data1[i]);
	}
	printf(" (sum=%d, avg=%.1f)¥n", sum1, static_cast<float>(sum1) / static_cast<float>(n1));
	//ログ出力：data2
	printf("data2=");
	int sum2 = 0;
	for (int i = 0; i < n2; ++i)
	{
		sum2 += data2[i];
		printf(" %d", data2[i]);
	}
	printf(" (sum=%d, avg=%.1f)¥n", sum2, static_cast<float>(sum2) / static_cast<float>(n2));
	//丸め処理：data1
	for (int i = 0; i < n1; ++i)
	{
		data1[i] = round_common(data1[i]);
	}
	//丸め処理：data2
	for (int i = 0; i < n2; ++i)
	{
		data2[i] = round_common(data2[i]);
	}
	//丸め実行後ログ出力
	printf("<AFTER>¥n");
	//ログ出力：data1
	printf("data1=");
	sum1 = 0;
	for (int i = 0; i < n1; ++i)
	{
		sum1 += data1[i];
		printf(" %d", data1[i]);
	}
	printf(" (sum=%d, avg=%.1f)¥n", sum1, static_cast<float>(sum1) / static_cast<float>(n1));
	//ログ出力：data2
	printf("data2=");
	sum2 = 0;
	for (int i = 0; i < n2; ++i)
	{
		sum2 += data2[i];
		printf(" %d", data2[i]);
	}
	printf(" (sum=%d, avg=%.1f)¥n", sum2, static_cast<float>(sum2) / static_cast<float>(n2));
}
```

ひとまずこれで共通化できたが、この関数の中でしか必要としない処理が外に出ているのが気持ち悪い。他の用途で使ってよい関数のようにも見える。

● 最適化②：関数内クラス化（共通関数のスコープを限定）

この対処として、ネームスペースやクラスに隠ぺいする方法も考えられるが、あまり大掛かりにせず、クラス／構造体を利用して、関数内にロジックを定義する方法を取る。

最適化②：共通ロジックをクラスのメンバー関数化して関数内に定義する

```
//データ丸め処理
void func(int data1[], int n1, int data2[], int n2)
{
	//データ丸め処理用クラス
	struct round{
```

```
            static int calc()(int data)
            {
                  if (data < 10)
                        data = 10;
                  else if (data > 100)
                        data = 100;
                  return data;
            }
      };
      //丸め実行前ログ出力
      printf("<BEFORE>¥n");
      //ログ出力：data1
      printf("data1=");
      int sum1 = 0;
      for (int i = 0; i < n1; ++i)
      {
            sum1 += data1[i];
            printf(" %d", data1[i]);
      }
      printf(" (sum=%d, avg=%.1f)¥n", sum1, static_cast<float>(sum1) / static_cast<float>(n1));
      //ログ出力：data2
      printf("data2=");
      int sum2 = 0;
      for (int i = 0; i < n2; ++i)
      {
            sum2 += data2[i];
            printf(" %d", data2[i]);
      }
      printf(" (sum=%d, avg=%.1f)¥n", sum2, static_cast<float>(sum2) / static_cast<float>(n2));
      //丸め処理：data1
      for (int i = 0; i < n1; ++i)
      {
            data1[i] = round::calc(data1[i]);
      }
      //丸め処理：data2
      for (int i = 0; i < n2; ++i)
      {
            data2[i] = round::calc(data2[i]);
      }
      //丸め実行後ログ出力
      printf("<AFTER>¥n");
      //ログ出力：data1
      printf("data1=");
      sum1 = 0;
      for (int i = 0; i < n1; ++i)
      {
            sum1 += data1[i];
            printf(" %d", data1[i]);
      }
      printf(" (sum=%d, avg=%.1f)¥n", sum1, static_cast<float>(sum1) / static_cast<float>(n1));
      //ログ出力：data2
      printf("data2=");
      sum2 = 0;
      for (int i = 0; i < n2; ++i)
      {
            sum2 += data2[i];
            printf(" %d", data2[i]);
      }
      printf(" (sum=%d, avg=%.1f)¥n", sum2, static_cast<float>(sum2) / static_cast<float>(n2));
}
```

関数の中で関数を定義することはできないが、クラス／構造体は定義できる。

これを利用し、関数内のクラス／構造体のメンバー関数としてロジックを定義すること

ができる。

● 最適化③：関数オブジェクト化

この「関数内にクラスを定義できる」という点を利用し、更に共通化を進める。

関数内に何度も登場するループ処理、プリント処理、平均算出処理も共通化する。ここで、ループ処理を共通化するために「関数オブジェクト」の手法を取る。

「関数オブジェクト」は、operator() をオーバーロードしたクラス／構造体である。ループ処理をテンプレート化し、「T 型の参照を関数オブジェクトに受け渡す」と規定することで処理を汎用化する。

最適化③：ループ処理を汎用化し、関数オブジェクトで作成した共通ロジックを実行する

```
//汎用ループ処理テンプレート関数
template<typename T, class F>
inline void for_each_array(T* data, int n, F& functor) //データの配列，データの要素数、関数オブジェクトを受け取る
{
	for (int i = 0; i < n; ++i, ++data) //型Tのデータを、受け取った要素数分のループで処理する
	{
		functor(*data); //関数オブジェクト呼び出し：型Tのデータを受け渡す
	}
}
//データ丸め処理
void func(int data1[], int n1, int data2[], int n2)
{
	//データ表示＆集計関数オブジェクト用クラス
	struct print{
		//全件表示
		void all(const char* name, int data[], int n)
		{
			sum = 0;
			printf("%s=", name);
			for_each_array(data, n, *this); //関数オブジェクトとして自分を渡している
			printf(" (sum=%d, avg=%.1f)¥n", sum, static_cast<float>(sum) / static_cast<float>(n));
		}
		int sum;
		//関数オブジェクトのための ()オペレータ関数
		inline void operator()(int data) //共通関数：型Tのデータを受け取る
		{
			sum += data;
			printf(" %d", data);
		}
	};
	//データ丸め関数オブジェクト用クラス
	struct round{
		//関数オブジェクトのための ()オペレータ関数
		inline void operator()(int& data) //共通関数：型Tのデータを受け取る：値を書き換えるので & を付けている
		{
			if (data < 10)
				data = 10;
			else if (data > 100)
				data = 100;
		}
	};
	//全体表示用のオブジェクト（集計が必要なため）
	print o;
	//丸め実行前ログ出力
```

```
    printf("<BEFORE>¥n");
    //ログ出力：data1
    o.all("data1", data1, n1);
    //ログ出力：data2
    o.all("data2", data2, n2);
    //丸め処理：data1
    for_each_array(data1, n1, round()); //round()は関数オブジェクト（実体を渡している）
    //丸め処理：data2
    for_each_array(data2, n2, round()); //round()は関数オブジェクト（実体を渡している）
    //丸め実行後ログ出力
    printf("<AFTER>¥n");
    //ログ出力：data1
    o.all("data1", data1, n1);
    //ログ出力：data2
    o.all("data2", data2, n2);
}
```

元々長く同じような処理が繰り返されていた処理が、かなりすっきした構造になる。処理効率、プログラムサイズの面でも特に問題はない。

汎用ループ処理用テンプレート関数「for_each_array()」は、第３引数に関数オブジェクトを受け取る。「型 T」の値を受け取る関数オブジェクトならどんなクラス／構造体でも渡すことができる。それは、「void operator()(T)」もしくは「void operator()(T&)」がオーバーロードされたクラス／構造体のことである。

なお、プログラム内で関数オブジェクトの変数名を「functor」としているのは、C++では「operator()」が主要メソッドになっているクラス／構造体のことを「ファンクタ」と呼ぶためである。

テンプレートは関数内に定義することができないため、どうしても関数の外に記述する必要がある。しかし、ここで示した「for_each_array()」関数は極めて汎用的であるため、関数外に定義されることには問題がない。

● 最適化④：標準ライブラリの活用

もっと汎用性を高めるには、このような独自の for - each 関数を実装せず、標準的な仕組みを利用することである。

独自の「for_each_array()」関数をやめて、STL 標準の「for_each()」関数に置き換える。

最適化④：ループ処理を STL の for_each に置き換える

```
#inluce <algorithm>     //std::for_each を使う為のインクルード

//データ丸め処理
void func(int data1[], int n1, int data2[], int n2)
{
    //データ表示＆集計関数オブジェクト用クラス
    static int sum;    //for_each()関数に関数オブジェクトが値渡しされるので、メンバー変数を使った集計がうまくいかない。
                       //代わりに static 変数を使う。
```

```
                        //※static 変数の濫用はプログラムサイズが無駄に大きくなるのでよくない。
    struct print{
        //全件表示
        void all(const char* name, int data[], int n)
        {
            sum = 0;
            printf("%s=", name);
            std::for_each(data, data + n, *this);//関数オブジェクトとして自分を渡している（値渡しである点に注意）
            printf(" (sum=%d, avg=%.1f)¥n", sum, static_cast<float>(sum) / static_cast<float>(n));
        }
        //int sum;//for_each()関数に関数オブジェクトが値渡しされるので、メンバー変数を利用できない
        //operator()
        void operator()(int data)      //共通関数：型 T のデータを受け取る
        {
            sum += data;
            printf(" %d", data);
        }
    };
    //データ丸め関数オブジェクト用クラス
    struct round{
        //operator()
        void operator()(int& data)     //共通関数：型 T のデータを受け取る：値を書き換えるので & を付けている
        {
            if (data < 10)
                data = 10;
            else if (data > 100)
                data = 100;
        }
    };
    //全体表示用のオブジェクト（集計が必要なため）
    print o;
    //丸め実行前ログ出力
    printf("<BEFORE>¥n");
    //ログ出力：data1
    o.all("data1", data1, n1);
    //ログ出力：data2
    o.all("data2", data2, n2);
    //丸め処理：data1
    std::for_each(data1, data1 + n1, round()); //round()は関数オブジェクト（実体を渡している）
    //丸め処理：data2
    std::for_each(data2, data2 + n2, round()); //round()は関数オブジェクト（実体を渡している）
    //丸め実行後ログ出力
    printf("<AFTER>¥n");
    //ログ出力：data1
    o.all("data1", data1, n1);
    //ログ出力：data2
    o.all("data2", data2, n2);
}
```

「std::for_each()」関数は、配列の先頭要素のアドレスと、末尾要素＋１のアドレスを渡すことでループできる。

なお、STL の for_each は関数オブジェクトだけではなく、関数ポインターを渡すことも可能。

● 最適化⑤：ラムダ式化（C++11 仕様）

最後に、この処理を C++11 のラムダ式を使用して最適化する方法を示す。

上記の STL 版「for_each()」のサンプルでは、集計のために static 変数を使用してしま

い、メモリ効率のよくない処理になっていた。

　ラムダ式を用いることで、その問題も解消した上、可読性も決して悪くない、かなり効率的なコーディングが可能となる。（STL の「for_each()」にはラムダ式を渡すことができる。）

最適化⑤：C++11 のラムダ式で最適化

```
#inluce <algorithm>     //std::for_each を使う為のインクルード


//データ丸め処理
void func(int data1[], int n1, int data2[], int n2)
{
     //全件表示用ラムダ式
     //※ [外部変数](引数) -> 戻り値の型{ 処理 } が、ラムダ式の書式。戻り値の型は省略可能。
     //※ラムダ式は、auto 型でしか変数に代入できなので注意。
     //  イメージとしては、decltype([x](y)->z{…}) という型になる。
     auto lambda_print = [](int& sum, const char* name, int data[], int n) -> void
     {
          sum = 0;
          printf("%s=", name);
          std::for_each(data, data + n, [&sum](int data)   //ラムダ式は、式の中に直接書くこともできる
               {
                    sum += data;
                    printf(" %d", data);
               }
          );
          printf(" (sum=%d, avg=%.1f)¥n", sum, static_cast<float>(sum) / static_cast<float>(n));
     };
     //データ丸め用ラムダ式
     auto lambda_round = [](int& data)
     {
          if (data < 10)
               data = 10;
          else if (data > 100)
               data = 100;
     };
     //集計用変数
     int sum1 = 0;
     int sum2 = 0;
     //丸め実行前ログ出力
     printf("<BEFORE>¥n");
     //ログ出力：data1
     lambda_print(sum1, "data1", data1, n1);
     //ログ出力：data2
     lambda_print(sum2, "data2", data2, n2);
     //丸め処理：data1
     std::for_each(data1, data1 + n1, lambda_round);
     //丸め処理：data2
     std::for_each(data2, data2 + n2, lambda_round);
     //丸め実行後ログ出力
     printf("<AFTER>¥n");
     //ログ出力：data1
     lambda_print(sum1, "data1", data1, n1);
     //ログ出力：data2
     lambda_print(sum2, "data2", data2, n2);
}
```

● 【参考】C++11 の範囲に基づく for ループ

ここまで、自作の for-each や STL の for-each を使用してきたが、C++11 には「範囲に基づく for ループ」という for-each 構文が追加されている。固定長配列などのループ処理を非常に簡潔に書くことができる。サンプルを示す。

範囲に基づく for ループ：

```
{
	//範囲に基づく for ループ：固定長配列
	int data[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
	printf("data=");
	for (int  elem : data)  // for( 要素型 要素変数 : コンテナ変数 ) でループ処理を記述
	{
		printf(" %d", elem);
	}
	printf("¥n");
}
{
	//範囲に基づく for ループ：STL コンテナ
	std::vector<const char*> data;
	data.push_back("太郎");
	data.push_back("次郎");
	data.push_back("三郎");
	printf("data=");
	for (auto elem : data)  // 要素型に auto 型を使ってもよい
	{
		printf(" %s", elem);
	}
	printf("¥n");
}
{
	//範囲に基づく for ループ：自作コンテナ
	struct DATA
	{
		char* begin(){ return m_data + 0; }
		char* end(){ return m_data + sizeof(m_data) / sizeof(m_data[0]); }
		char m_data[10];
	};
	DATA data = { {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} };
	for (auto& elem : data) //begin(), end() がイテレータを返すものならなんにでも使える
				//値を書き戻したければ要素型に & を付けて参照型にする
	{
		elem += 100;
	}
	printf("data=");
	for (auto elem : data)
	{
		printf(" %d", elem);
	}
	printf("¥n");
}
```

■ テンプレートクラスによる多態性

クラスの多態性は、virtual による仮想クラス（インターフェースクラス）を使用する方

法だけではなく、テンプレートクラスを使用した方法でも実現できる。

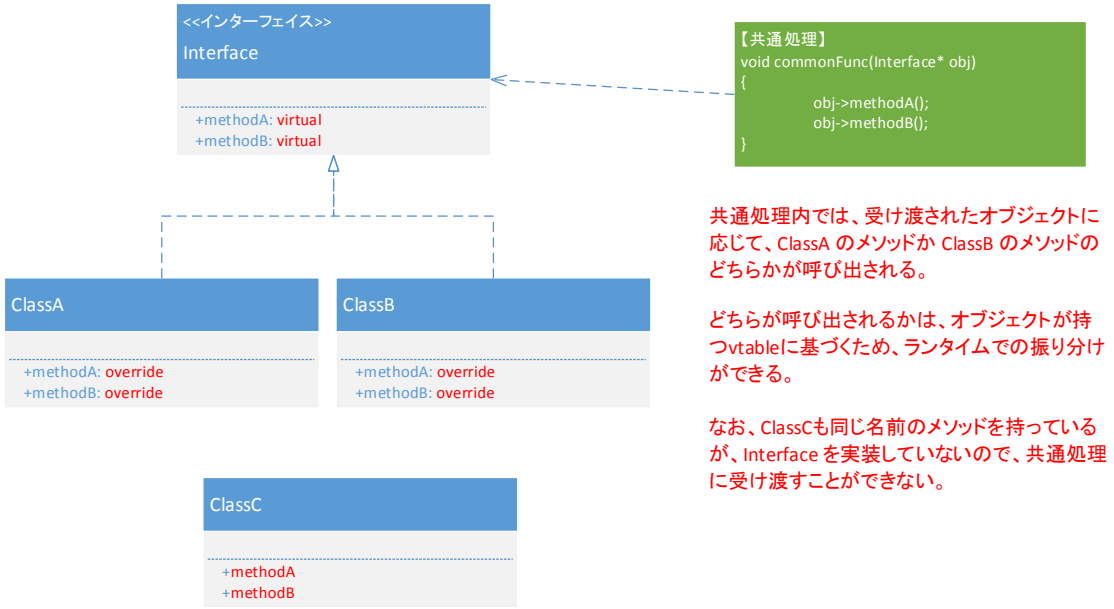
▼ 動的な多態性と静的な多態性

まず、仮想クラスを用いる方法が「動的な多態性」であるのに対して、テンプレートクラスで実現出来るのは「静的な多態性」であることを強調する。

「動的な多態性」は、ランタイムでインスタンスに基づいて振る舞いが変わるのに対して、「静的な多態性」はコンパイル時に振る舞いが確定する。

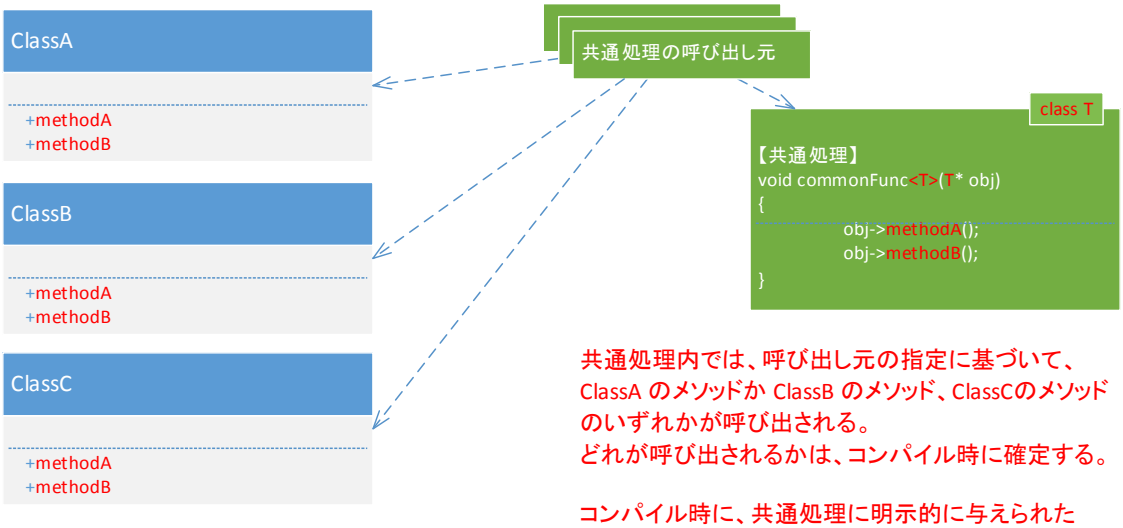
▼ 仮想クラスの場合：動的な多態性

仮想クラスを用いた場合のクラス図を示す。



▼ テンプレートクラスの場合：静的な多態性

テンプレートクラスを用いた場合のクラス図を示す。



共通処理内では、呼び出し元の指定に基づいて、ClassA のメソッドか ClassB のメソッド、ClassCのメソッドのいずれかが呼び出される。
どれが呼び出されるかは、コンパイル時に確定する。

コンパイル時に、共通処理に明示的に与えられた「型」(クラス T)に基づいて、その種類の分だけ処理が実体化される

▼ 仮想クラス VS テンプレートクラス

静的な多態性にしても、動的な多態性にしても、その目的は処理の共通化である。
以下に両者の違いを示す。

【仮想クラス：動的な多態性】

＜良い点＞

- ランタイムで処理を振り分けできる。
- テンプレートクラスと比べて、プログラムサイズへの影響が小さい。

＜悪い点＞

- データサイズが若干大きくなる。
  - クラスのメンバーに、vtable への参照が自動的に追加される。
  - 多重継承の場合は、継承しているクラスの数だけ vtable の参照が追加される。
- vtable を通して関数を呼び出すため、処理が遅くなる。
  - 単純なアクセッサでも virtual 化されると、インライン展開の恩恵を受けることができないため、場合によってはかなりのパフォーマンス劣化を招く。
  - 「関数呼び出し」はコンピュータが行う処理の中でも特に重い部類。「いかに関数呼び出しを減らすか」は高速化のために気をつかうべきポイント。

【テンプレートクラス：静的な多態性】

＜良い点＞

・ 処理が高速。

・ データサイズへの影響がない。

＜悪い点＞

・ ランタイムで処理を振り分けできない。

  - vtable を独自実装することで解決する手法もある。

・ プログラムサイズが大きくなる。

  - テンプレートに与えられた型（クラス）の種類数分、処理の実体（コピー）が作られることになるため、処理が大きいと爆発的なプログラムサイズの肥大化を招くことがある。
  - 十分に短い処理なら、メタプログラミングの成果やインライン展開によって、逆に小さくなる可能性もある。

▼ 動的な多態性と静的な多態性の折衷案

仮想クラスはどうしても必須となる場面がある。

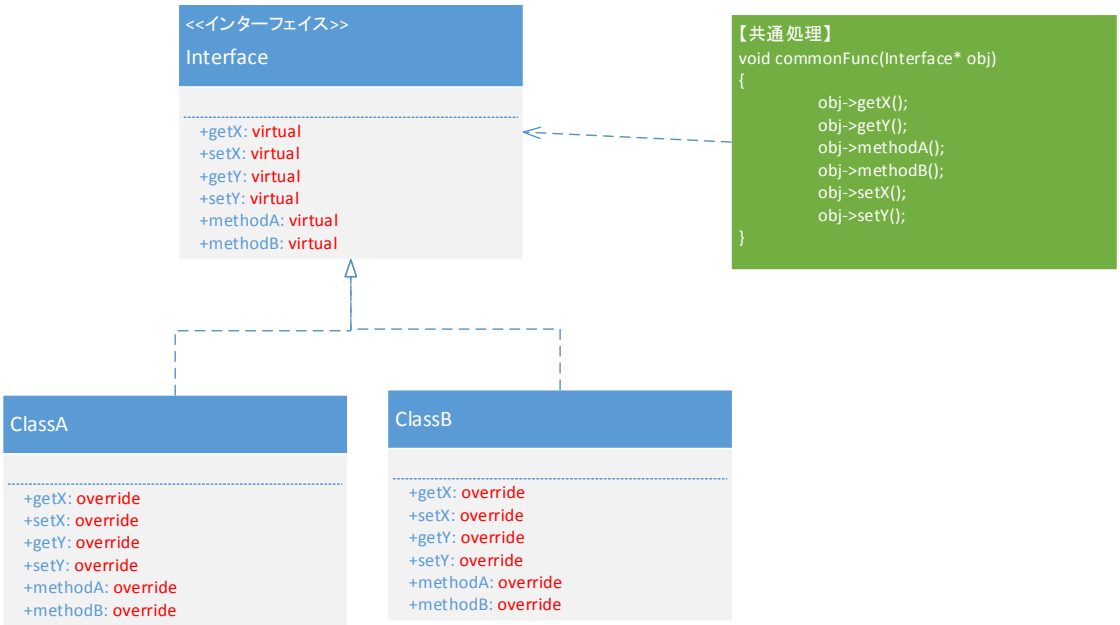
例えば、シーングラフに登録されている「主人公」「敵」「ミサイル」「爆発エフェクト」などの様々なオブジェクトに対して、逐次 update 処理を実行するには、update を仮想関数にして各オブジェクトが update をオーバーライドしているのが良い。

このような要件が発生すると、update 以外にも「座標を返す関数」「オブジェクトの種類を返す関数」など、便利に使えそうな仮想関数を次々と登録してしまいがちである。

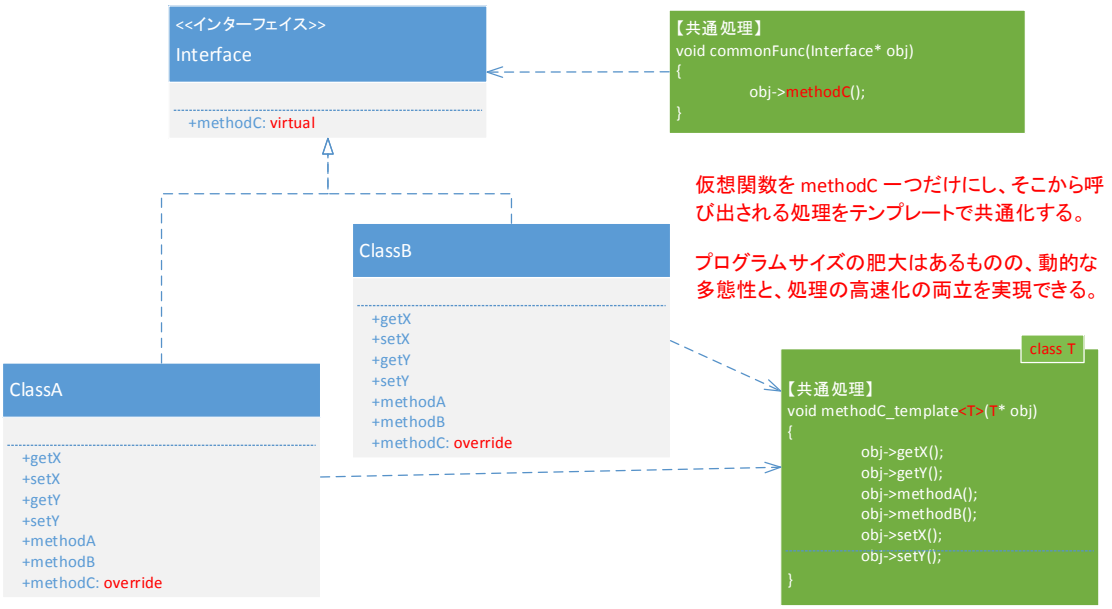
しかし、安易な対応はパフォーマンスの劣化を招くため、仮想関数は必要最小限のまとまった処理に対してのみ適用すべきである。その上で、処理の共通化にテンプレートを組み合わせると効果的である。

以下、具体的な改善例を示す。

【改善前】



【改善後】



▼ ポリシー（ストラテジーパターン）

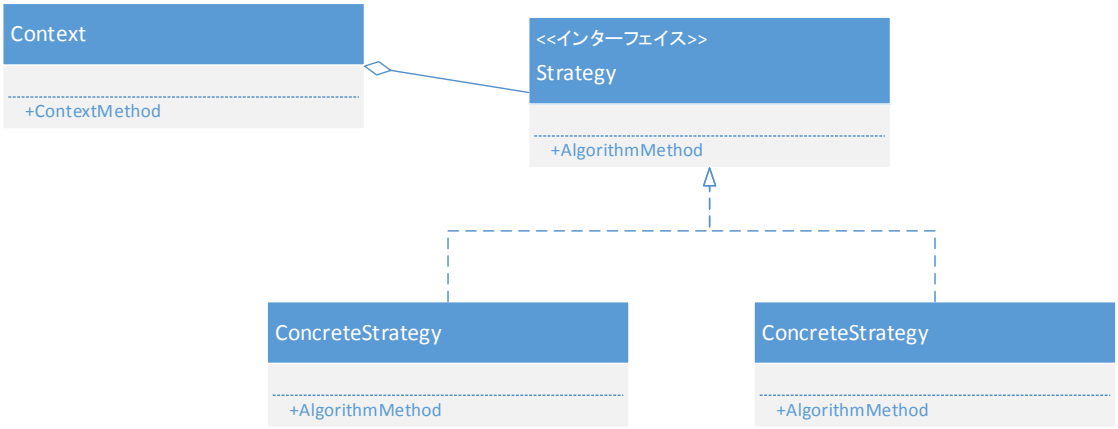
「ポリシー」という手法を説明する。

「ポリシー」は、デザインパターンでいうところの「ストラテジーパターン」と同様の

もので、共通処理に受け渡すオブジェクトもしくはクラスによって、その振る舞い（処理）を変える手法である。

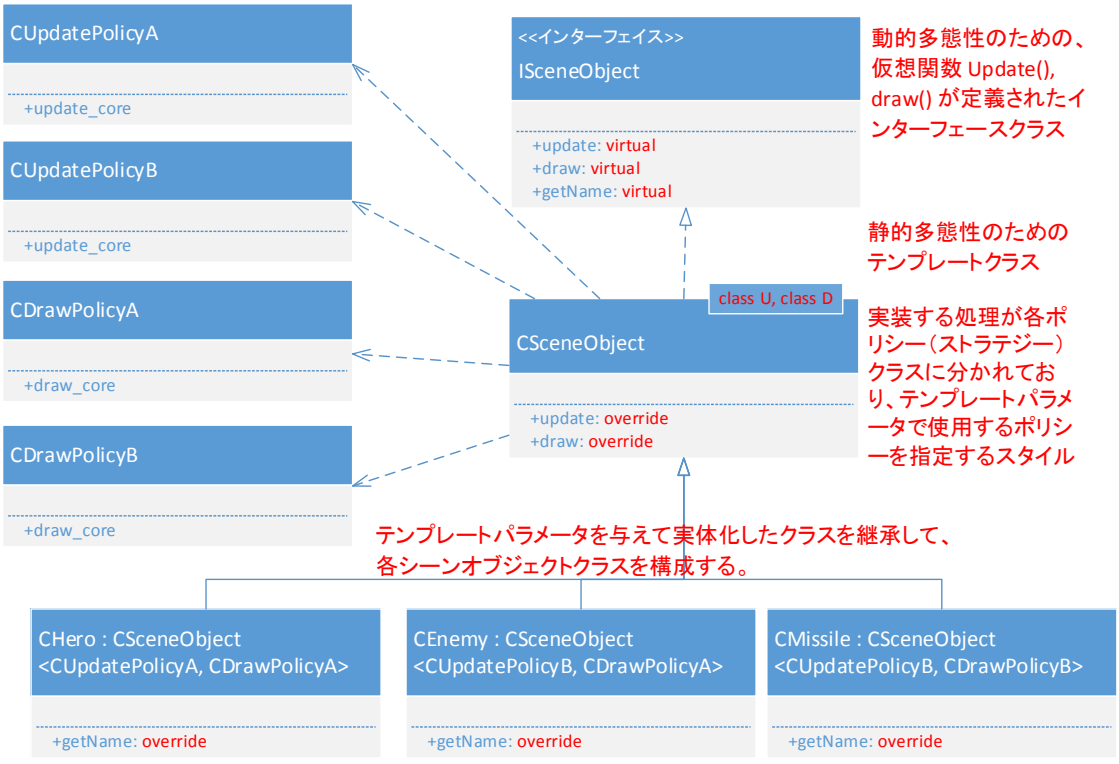
STL の各種コンテナが受け取る Allocator クラスは、ポリシー（ストラテジー）の実例の一つである。

ストラテジーパターンのクラス図：



「ポリシー」のサンプルとして、前述の「動的多態性＋静的多態性」の実装を、ポリシーを使って対応したケースを示す。

例：ポリシーを活用したシーンオブジェクトのクラス図



実際のコードのサンプルを示す。

例：

```
//シーンオブジェクト用インターフェースクラス
class ISceneObject
{
public:
	virtual void update() = 0;
	virtual void draw() = 0;
	virtual const char* getName() = 0;	//末端のシーンオブジェクト固有の情報にアクセスするためのアクセッサ
};

//シーンオブジェクト用テンプレートクラス
template<class U, class D>
class CSceneObject : public ISceneObject
{
public:
	void update() override
	{
		U policy;
		policy.update_core(this);
	}
	void draw() override
	{
		D policy;
		policy.draw_core(this);
	}

};

//update 処理用ポリシークラス：パターンA
class CUpdatePolicyA
{
public:
	void update_core(ISceneObject* obj)
	{
		printf("%s->CUpdatePolicyA::update_core()¥n", obj->getName());
	}
};

//update 処理用ポリシークラス：パターンB
class CUpdatePolicyB
{
public:
	void update_core(ISceneObject* obj)
	{
		printf("%s->CUpdatePolicyB::update_core()¥n", obj->getName());
	}
};

//draw 処理用ポリシークラス：パターンA
class CDrawPolicyA
{
public:
	void draw_core(ISceneObject* obj)
	{
		printf("%s->CDrawPolicyA::draw_core()¥n", obj->getName());
	}
};

//draw 処理用ポリシークラス：パターンB
class CDrawPolicyB
{
public:
	void draw_core(ISceneObject* obj)
```

```
	{
		printf("%s->CDrawPolicyB::draw_core()¥n", obj->getName());
	}
};

//主人公
class CHero : public CSceneObject<CUpdatePolicyA, CDrawPolicyA>
{
public:
	const char* getName() override { return "CHero"; }
};

//敵
class CEnemy : public CSceneObject<CUpdatePolicyB, CDrawPolicyA>
{
public:
	const char* getName() override { return "CEnemy"; }
};

//ミサイル
class CMissile : public CSceneObject<CUpdatePolicyB, CDrawPolicyB>
{
public:
	const char* getName() override { return "CMissile"; }
};

//for_each
template<class T, std::size_t N, class F>
inline void for_each(T* (&obj)[N], F& functor)
{
	T** p = obj;
	for (int i = 0; i < N; ++i, ++p)
	{
		functor(*p);
	}
}

//ポリシーテスト
void testPolicy()
{
	//シーンオブジェクトのリスト生成
	ISceneObject* scene_objs[] = {
		new CHero,
		new CEnemy,
		new CMissile
	};
	//update 実行用関数オブジェクト
	struct update_functor{ inline void operator()(ISceneObject* obj){ obj->update(); } };//←この呼び出しは動的多態性
	//draw 実行用関数オブジェクト
	struct draw_functor{ inline void operator()(ISceneObject* obj){ obj->draw(); } };	//←この呼び出しは動的多態性
	//delete 用関数オブジェクト
	struct delete_functor{ inline void operator()(ISceneObject* obj){ delete obj; } };
	//全シーンオブジェクトの update 実行
	printf("[update]¥n");
	for_each(scene_objs, update_functor());
	//全シーンオブジェクトの draw 実行
	printf("[draw]¥n");
	for_each(scene_objs, draw_functor());
	//全シーンオブジェクトの delete
	for_each(scene_objs, delete_functor());
}
```

↓実行結果

```
[update]
CHero->CUpdatePolicyA::update_core()	← CHero の update 処理は、ポリシーパターンＡの処理
```

```
CEnemy->CUpdatePolicyB::update_core()     ← CEnemy の update 処理は、ポリシーパターンBの処理
CMissile->CUpdatePolicyB::update_core()   ← CMissile の update 処理は、ポリシーパターンBの処理
[draw]
CHero->CDrawPolicyA::draw_core()          ← CHero の draw 処理は、ポリシーパターンAの処理
CEnemy->CDrawPolicyA::draw_core()         ← CEnemy の draw 処理は、ポリシーパターンAの処理
CMissile->CDrawPolicyB::draw_core()       ← CMissile の draw 処理は、ポリシーパターンBの処理
```

▼ CRTP（テンプレートメソッドパターン）

先の「ポリシー」のサンプルプログラムでは、せっかく静的多態性を実装しているにもかかわらず、末端のシーンオブジェクトクラス固有の情報にアクセスするために、getName() という virtual 関数に頼っている点がやや不便である。処理効率にも影響がある。

そこで、ポリシーに定義した共通処理から、直接末端のクラスの情報にアクセスできるように修正する。これには、「CRTP」と呼ばれるテンプレートクラスのテクニックを使用する。

「CRTP」とは、「Curiously Recursive Template Pattern」（奇妙に再帰したテンプレートパターン）の略語である。この「奇妙な再帰」を説明するために、まずは CRTP の簡単な実装サンプルを示す。

例：CRTP のサンプル

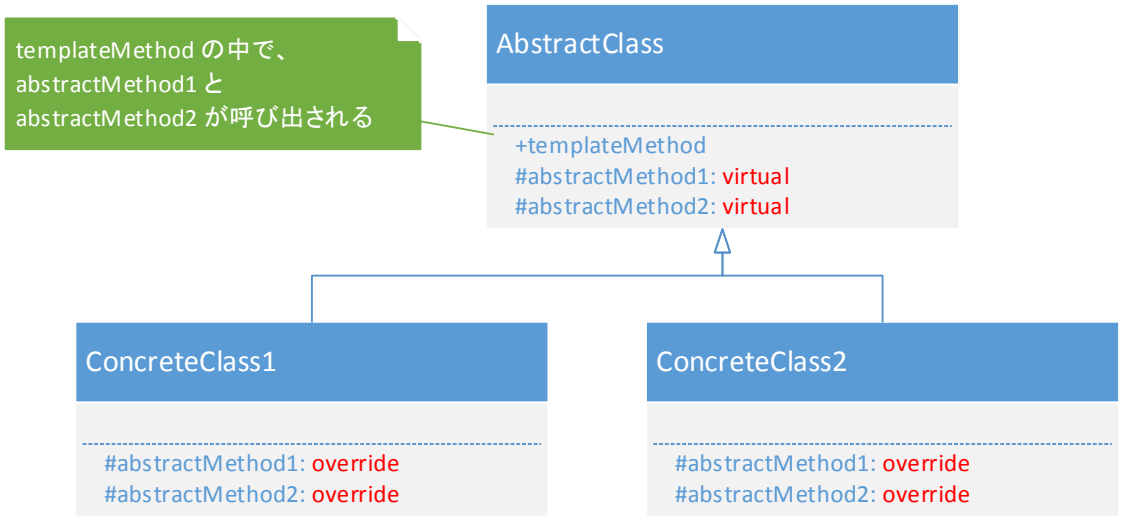
```
//親クラス（テンプレート）
template<class D>
class CBase
{
public:
	void foo() const
	{
		printf("CBase<D>::foo()¥n");
		static_cast<const D&>(*this).bar(); //子クラスのメンバーを呼び出し
	}
};

//子クラス
class CDerived : public CBase<CDerived>   //親クラスのテンプレートパラメータに自分（子クラス）を渡している
{
	friend class CBase<CDerived>;
private:
	void bar() const
	{
		printf("CDerived::bar()¥n");
	}
};

//CRTP のテスト
void test()
{
	CDerived obj;
	obj.foo();
}
```

↓実行結果

```
CBase<D>::foo()
CDerived::bar()
```

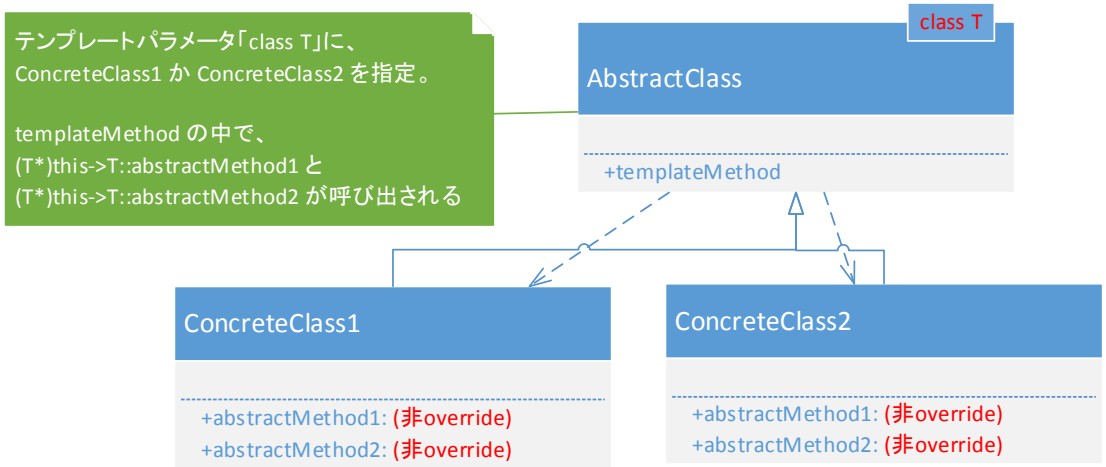
「CRTP」の主な用途は、「テンプレートメソッドパターン」の静的な実装である。「テンプレートメソッドパターン」とは、クラスのあるメソッド（「テンプレートメソッド」と呼ぶ）の中の一部のロジックを、同クラス内の抽象メソッド（仮想メソッド）に依存する、プログラミング手法である。つまり、メソッドの中の一部の処理を抽象化して、クラスを継承した子クラスの方でその一部のロジックを実装・置き換えする。

テンプレートメソッドパターンのクラス図：



このテンプレートメソッドパターンを、CRTP を使用して静的に実装するとこうなる。

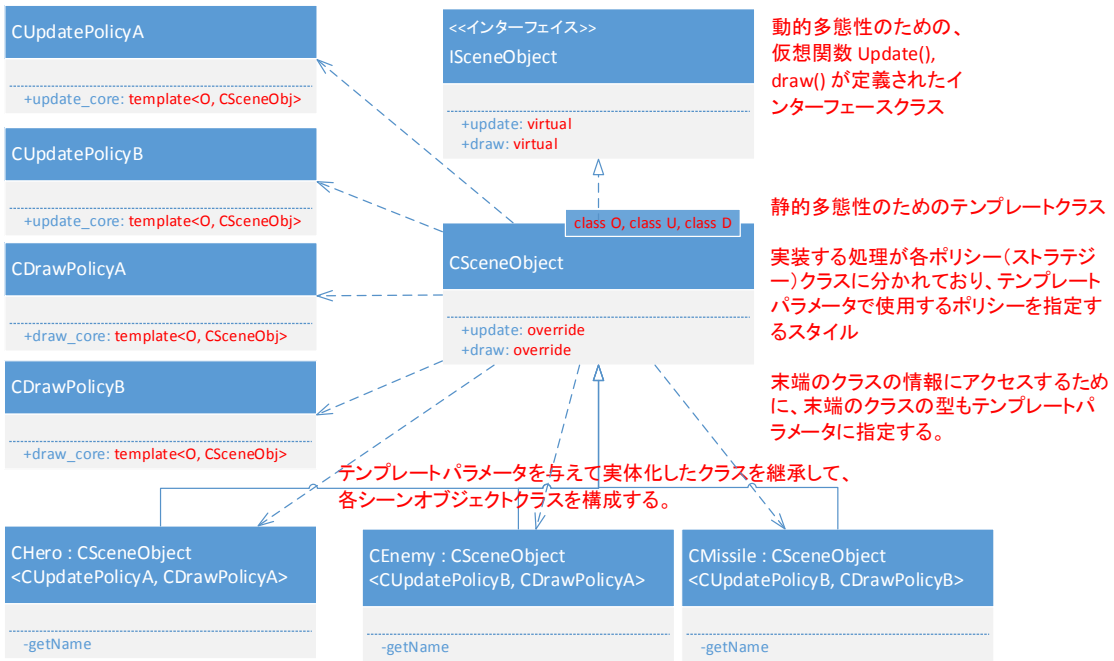
CRTP によるテンプレートメソッドパターンのクラス図：



二つのクラス図にも示している通り、本来のスタイルでは abstractMethod をプロテクテッドスコープにしているのに対して、CRTP 版では public スコープにしている。そうしないと templateMethod からアクセスできないためである。この問題は、子クラス側で親クラスを friend クラスに設定することで回避できる。その場合は、private スコープにできる。

以上を踏まえて、先の「ポリシー」のサンプルプログラムを改良し、virtual 関数に頼らず、共通処理から末端のクラスの情報にアクセスするようにする。

例：ポリシー＋CRTP を活用したシーンオブジェクトのクラス図



実際のコードのサンプルを示す。

例：　※赤字はポリシーのサンプルから特に変わっている箇所を示す。

```
//シーンオブジェクト用インターフェースクラス
class ISceneObject
{
public:
	virtual void update() = 0;
	virtual void draw() = 0;
//	virtual const char* getName() = 0;//削除
};

//シーンオブジェクト用テンプレートクラス
template<class O, class U, class D>
class CSceneObject : public ISceneObject
{
public:
	void update() override
	{
		U policy;
		policy.update_core<O, CSceneObject<O, U, D> >(this);
	}
```

```
	void draw() override
	{
		D policy;
		policy.draw_core<O, CSceneObject<O, U, D> >(this);
	}

};

//update 処理用ポリシークラス：パターンA
class CUpdatePolicyA
{
public:
	template<class O, class T>
	void update_core(T* obj)
	{
		printf("%s->CUpdatePolicyA::update_core()¥n", static_cast<O*>(obj)->getName());
	}
};

//update 処理用ポリシークラス：パターンB
class CUpdatePolicyB
{
public:
	template<class O, class T>
	void update_core(T* obj)
	{
		printf("%s->CUpdatePolicyB::update_core()¥n", static_cast<O*>(obj)->getName());
	}
};

//draw 処理用ポリシークラス：パターンA
class CDrawPolicyA
{
public:
	template<class O, class T>
	void draw_core(T* obj)
	{
		printf("%s->CDrawPolicyA::draw_core()¥n", static_cast<O*>(obj)->getName());
	}
};

//draw 処理用ポリシークラス：パターンB
class CDrawPolicyB
{
public:
	template<class O, class T>
	void draw_core(T* obj)
	{
		printf("%s->CDrawPolicyB::draw_core()¥n", static_cast<O*>(obj)->getName());
	}
};

//主人公
class CHero : public CSceneObject<CHero, CUpdatePolicyA, CDrawPolicyA>
{
	friend class CUpdatePolicyA;
	friend class CDrawPolicyA;
private:
	const char* getName() { return "CHero"; }//非 opverride
};

//敵
class CEnemy : public CSceneObject<CEnemy, CUpdatePolicyB, CDrawPolicyA>
{
	friend class CUpdatePolicyB;
```

```
	friend class CDrawPolicyA;
private:
	const char* getName() { return "CEnemy"; }//非opverride
};

//ミサイル
class CMissile : public CSceneObject<CMissile, CUpdatePolicyB, CDrawPolicyB>
{
	friend class CUpdatePolicyB;
	friend class CDrawPolicyB;
private:
	const char* getName() { return "CMissile"; }//非opverride
};

//for_each
template<class T, size_t N, class F>
inline void for_each(T* (&obj)[N], F& functor)
{
	T** p = obj;
	for (int i = 0; i < N; ++i, ++p)
	{
		functor(*p);
	}
}

//ポリシー&CRTPテスト
void testPolicyAndCRTP()
{
	printf("¥n- testPolicyAndCRTP() -¥n¥n");

	//シーンオブジェクトのリスト生成
	ISceneObject* scene_objs[] = {
		new CHero,
		new CEnemy,
		new CMissile
	};
	//update実行用関数オブジェクト
	struct update_functor{ inline void operator()(ISceneObject* obj){ obj->update(); } };
	//draw実行用関数オブジェクト
	struct draw_functor{ inline void operator()(ISceneObject* obj){ obj->draw(); } };
	//delete用関数オブジェクト
	struct delete_functor{ inline void operator()(ISceneObject* obj){ delete obj; } };
	//全シーンオブジェクトのupdate実行
	printf("[update]¥n");
	for_each(scene_objs, update_functor());
	//全シーンオブジェクトのdraw実行
	printf("[draw]¥n");
	for_each(scene_objs, draw_functor());
	//全シーンオブジェクトのdelete
	for_each(scene_objs, delete_functor());
}
```

↓実行結果　※先のポリシーのサンプルと全く同じ結果

```
[update]
CHero->CUpdatePolicyA::update_core()
CEnemy->CUpdatePolicyB::update_core()
CMissile->CUpdatePolicyB::update_core()
[draw]
CHero->CDrawPolicyA::draw_core()
CEnemy->CDrawPolicyA::draw_core()
CMissile->CDrawPolicyB::draw_core()
```

なお、各ポリシークラスのメンバー関数が template になるため、getName() の virtual をやめることと引き換えに、プログラムサイズが大きくなっている点に注意。処理速度は

上がっている。

▼ テンプレートクラスによる動的な多態性（vtable の独自実装）

ここまで来ると、もう virtual を完全に廃止した上で、動的な多態性も実現出来ないかと考えるが、vtable と同じ仕組みを独自に実装することになる。

vtable の独自実装はあらゆる面で効果的ではないので、行わないことが賢明である。「あらゆる面で」とは、処理速度、データサイズ、プログラムサイズ、コーディングの手間・可読性の面である。

わずかな利点としては、クラス／構造体への暗黙的なメンバー追加が発生しないので、完全に自分で状態を把握できることと、「override したつもりがされていなかった」という事故を防ぐことができるぐらいである。

後者の「override」の問題は、C++11 仕様からは override キーワードを使用する事で対処できる。

参考までに、vtable を独自実装した場合のサンプルを示す。どれぐらい「使えないか」が理解できる。なお、テンプレートは全く使っていない。

例：

```
//----------
//基底クラス
class CBase
{
	//----------仮想関数テーブル準備（基底クラスだけに必要）----------
protected:
	//仮想関数テーブル構造体定義
	struct VTABLE
	{
		//static 関数ポインターのテーブル ※this ポインターを受け取るため、必ず第一引数は void* 型
		void(*v_methodA)(void*);
		int(*v_methodB)(void*);
		int(*v_methodC)(void*, int, char);
	};
public:
	//仮想関数呼び出しテーブル構造体定義
	struct CALL_TABLE
	{
		//【仮想関数対象】通常メソッドと同じ名前と引数のメソッドを定義し、
		//仮想関数テーブルのメソッドを this ポインター付きで呼び出す
		void methodA(){ m_vtable->v_methodA(m_this); }
		int methodB(){ return m_vtable->v_methodB(m_this); }
		int methodC(int par1, char par2){ return m_vtable->v_methodC(m_this, par1, par2); }
		//コンストラクタ
		CALL_TABLE(void* this_, VTABLE* vtable) :
			m_this(this_),
			m_vtable(vtable)
		{}
	private:
		//フィールド
		void* m_this;	//this ポインター
		VTABLE* m_vtable;//仮想関数テーブル
	};
```

```
public:
	//アクセッサ
	CALL_TABLE& getCallTable() { return m_callTable; } //仮想関数呼び出しテーブル取得
protected:
	//フィールド
	CALL_TABLE m_callTable;//仮想関数呼び出しテーブル
protected:
	//コンストラクタ
	CBase(VTABLE* vtable) :                          //子クラス用に、パラメータを受け取るコンストラクタも用意
		m_callTable(static_cast<void*>(this), vtable) //必ず仮想関数呼び出しテーブルを初期化する
	{}

	//----------仮想関数用処理----------
public:
	//【仮想関数対象】通常メソッド
	void methodA(){ printf("CBase::methoA()¥n"); }
	int methodB(){ printf("CBase::methoB()¥n"); return 0; }
	int methodC(int par1, char par2){ printf("CBase::methoC(%d, %d)¥n", par1, par2); return 0; }
public:
	//【仮想関数対象】通常メソッドを呼び出す static メソッド
	//※ this ポインター＋通常メソッドの引数を受け取り、this ポインターの通常メソッドを呼び出す
	static void s_methodA(void* this_){ static_cast<CBase*>(this_)->methodA(); }
	static int s_methodB(void* this_){ return static_cast<CBase*>(this_)->methodB(); }
	static int s_methodC(void* this_, int par1, char par2){ return static_cast<CBase*>(this_)->methodC(par1, par2); }
private:
	//static フィールド
	static VTABLE s_vtalble;//仮想関数テーブル

	//----------通常処理----------
public:
	//.....

public:
	//コンストラクタ
	CBase() :
		CBase(&s_vtalble) //必ず仮想関数呼び出しテーブルを初期化する（初期化のためのコンストラクタを呼び出す）
	{}
};

//----------仮想関数用処理----------
//static 仮想関数テーブルを初期化
CBase::VTABLE CBase::s_vtalble = {
	&CBase::s_methodA,
	&CBase::s_methodB,
	&CBase::s_methodC
};

//----------
//子クラス A
class CClassA : public CBase
{
	//----------仮想関数用処理----------
public:
	//【仮想関数対象】通常メソッド
	void methodA(){ printf("CClassA::methoA()¥n"); }
	int methodB(){ printf("CClassA::methoB()¥n"); return 0; }
	int methodC(int par1, char par2){ printf("CClassA::methoC(%d, %d)¥n", par1, par2); return 0; }
public:
	//【仮想関数対象】通常メソッドを呼び出す static メソッド
	//※ this ポインター＋通常メソッドの引数を受け取り、this ポインターの通常メソッドを呼び出す
	static void s_methodA(void* this_){ static_cast<CClassA*>(this_)->methodA(); }
	static int s_methodB(void* this_){ return static_cast<CClassA*>(this_)->methodB(); }
	static int s_methodC(void* this_, int par1, char par2){ return static_cast<CClassA*>(this_)->methodC(par1, par2); }
private:
	//static フィールド
```

```
	static VTABLE s_vtalble;//仮想関数テーブル

	//----------通常処理----------
public:
	//.....

public:
	//コンストラクタ
	CClassA() :
		CBase(&s_vtalble) //親クラスのコンストラクタを呼び出し、必ず仮想関数呼び出しテーブルを初期化する
	{}
};

//----------仮想関数用処理----------
//static 仮想関数テーブルを初期化
CBase::VTABLE CClassA::s_vtalble = {
	&CClassA::s_methodA,
	&CClassA::s_methodB,
	&CClassA::s_methodC
};

//----------
//子クラス B
class CClassB : public CBase
{
	//----------仮想関数用処理----------
public:
	//【仮想関数対象】通常メソッド
	void methodA(){ printf("CClassB::methoA()¥n"); }
	int methodB(){ printf("CClassB::methoB()¥n"); return 0; }
	//methodC はオーバーライドしない
public:
	//【仮想関数対象】通常メソッドを呼び出す static メソッド
	//※ this ポインター＋通常メソッドの引数を受け取り、this ポインターの通常メソッドを呼び出す
	static void s_methodA(void* this_){ static_cast<CClassB*>(this_)->methodA(); }
	static int s_methodB(void* this_){ return static_cast<CClassB*>(this_)->methodB(); }
	//methodC はオーバーライドしない
private:
	//static フィールド
	static VTABLE s_vtalble;//仮想関数テーブル

	//----------通常処理----------
public:
	//.....

public:
	//コンストラクタ
	CClassB() :
		CBase(&s_vtalble) //親クラスのコンストラクタを呼び出し、必ず仮想関数呼び出しテーブルを初期化する
	{}
};

//----------仮想関数用処理----------
//static 仮想関数テーブルを初期化
CBase::VTABLE CClassB::s_vtalble = {
	&CClassB::s_methodA,
	&CClassB::s_methodB,
	&CBase::s_methodC   //オーバーライドしない場合は、親のメソッドを指定
};

//for_each
template<class T, size_t N, class F>
inline void for_each(T* (&obj)[N], F& functor)
{
	T** p = obj;
```

```
	for (int i = 0; i < N; ++i, ++p)
	{
		functor(*p);
	}
}

//自作 vtable テスト
void test ()
{
	//オブジェクトのリスト生成
	CBase* objs[] = {
		new CBase,
		new CClassA,
		new CClassB
	};
	//関数オブジェクト
	struct functor{
		inline void operator()(CBase* obj){
			CBase::CALL_TABLE& ctbl = obj->getCallTable(); //仮想関数呼び出しテーブルを取得してから、
														//仮想関数を呼び出す
			ctbl.methodA();
			ctbl.methodB();
			ctbl.methodC(12, 34);
		}
	};
	//全オブジェクトの処理実行
	for_each(objs, functor());
}
```

↓実行結果

```
CBase::methoA()
CBase::methoB()
CBase::methoC(12, 34)
CClassA::methoA()
CClassA::methoB()
CClassA::methoC(12, 34)
CClassB::methoA()
CClassB::methoB()
CBase::methoC(12, 34)
```

▼ SFINAE による柔軟なテンプレートオーバーロード

SFINAE とは、「Substitution Failure Is Nt An Error」（置き換え失敗はエラーではない）の略で、テンプレートのコンパイル時の挙動を示すものである。

オーバーロードされた多数のテンプレートがある場合、実体化の際に適切なテンプレートが選択される。この時、実体化に失敗したテンプレートは候補から外され、適合するテンプレートが見つかるまで試行し、一つもみつからなかった時にエラーとなる。SFINAE はこのような挙動を表すものである。

なお、「SFINAE」の読み方は、「C++テンプレートテクニック」の著者の読み方にならえば「スフィナエ」である。

SFINAE を活用する事で、関数のオーバーロードと同様に、同じ名前を付けて様々な処理をオーバーロードすることができる。これにより、名前さえ知っていれば、深く考えずに使う事ができる。

具体的なサンプルとして、for_each のバリエーションを作成してみたものを示す。

例：

```
#include <stdio.h>
#include <iostream>
#include <vector>

//SFINAE テスト（様々な for_each）

//独自コンテナ
template<typename T, int N>
class MY_ARRAY
{
public:
	typedef MY_ARRAY<T, N> MY_TYPE;
	typedef T DATA_TYPE;//テンプレート引数の型 T を、for_each に知らせるためのテクニック
	static const int NUM = N;
public:
	T* begin(){ return &m_array[0]; }
	T* end(){ return &m_array[N]; }
	T& operator[](int i){ return m_array[i]; }
private:
	T m_array[N];
};

//固定配列版 for_each
template<class T, size_t N, class F>
inline void for_each(T (&obj)[N], F& functor)
{
	bool is_first = true;
	T* p = obj;
	for (int i = 0; i < N; ++i, ++p)
	{
		functor(*p, is_first);
	}
}

//ポインター型の固定配列版 for_each　※部分特殊化
template<class T, std::size_t N, class F>
inline void for_each(T*(&obj)[N], F& functor)
{
	bool is_first = true;
	T** p = obj;
	for (int i = 0; i < N; ++i, ++p)
	{
		functor(**p, is_first);
	}
}

//動的配列版 for_each①：配列要素数を渡す
template<class T, class F>
inline void for_each(T* p, int n, F& functor)
{
	bool is_first = true;
	for (int i = 0; i < n; ++i, ++p)
	{
		functor(*p, is_first);
	}
}

//動的配列版 for_each②：配列の終端+1 を渡す
template<class T, class F>
inline void for_each(T* p, T* end, F& functor)
{
	bool is_first = true;
```

```
	while (p != end)
	{
		functor(*p, is_first);
		++p;
	}
}

//STL コンテナ版 for_each
template<class T, class F>
inline void for_each(T& con, F& functor)
{
	bool is_first = true;
	typename T::iterator ite = con.begin();
	typename T::iterator end = con.end();
	while (ite != end)
	{
		functor(*ite, is_first);
		++ite;
	}
}

//独自コンテナ版 for_each
template<class T, class F>
inline void for_each(typename T::MY_TYPE& con, F& functor)//独自コンテナには ::MY_TYPE メンバーが定義されている
{
	bool is_first = true;
	typename T::DATA_TYPE* p = con.begin();//コンテナ型変数 con が扱う値の型を::DATA_TYPE で取得
	typename T::DATA_TYPE* end = con.end();//typename は DATA_TYPE がデータ型であることを示している。
	                                       //typename を付けなくても問題ないが、付けておくと、
	                                       //実体がデータ型ではなく定数だったときに確実にエラーになる。
	while (p != end)
	{
		functor(*p, is_first);
		++p;
	}
}

//関数オブジェクト　※template なので関数の外で定義
template<typename T>
struct functor {
	inline void operator()(T& o, bool& is_first)
	{
		std::cout << (is_first ? "" : ", ") << o;
		is_first = false;
	}
};

//SFINAE テスト
void test ()
{
	//int の配列
	int data[] = { 1, 22, 333, 44444, 55555 };

	//int* の配列
	int* data_p[] = { &data[4], &data[3], &data[2], &data[1], &data[0] };

	//STL の vector
	std::vector<char> vec;
	vec.push_back('T');
	vec.push_back('E');
	vec.push_back('S');
	vec.push_back('T');

	//自作コンテナ
	MY_ARRAY<float, 6> my_con;
```

```
    my_con[0] = 1.2f;
    my_con[1] = 2.3f;
    my_con[2] = 3.4f;
    my_con[3] = 5.6f;
    my_con[4] = 7.8f;
    my_con[5] = 9.1f;

    //固定配列版 for_each
    std::cout << "data[(all)]=";
    for_each(data, functor<int>());
    std::cout << std::endl;

    //ポインター型の固定配列版 for_each
    std::cout << "data_p[(all)]=";
    for_each(data_p, functor<int>());
    std::cout << std::endl;

    //動的配列版 for_each①：配列要素数を渡す
    std::cout << "data[3]=";
    for_each(data, 3, functor<int>());
    std::cout << std::endl;

    //動的配列版 for_each②：配列の終端+1 を渡す
    std::cout << "data[2～3]=";
    for_each(data + 2, data + 3 + 1, functor<int>());
    std::cout << std::endl;

    //STL コンテナ版 for_each
    std::cout << "vec=";
    for_each(vec, functor<char>());
    std::cout << std::endl;

    //独自コンテナ版 for_each
    std::cout << "my_con=";
    for_each<MY_ARRAY<float, 6> >(my_con, functor<float>());//STL コンテナ版と区別するために明示的に型を指定
    std::cout << std::endl;
}
```

↓実行結果

```
data[(all)]=1, 22, 333, 44444, 55555
data_p[(all)]=55555, 44444, 333, 22, 1
data[3]=1, 22, 333
data[2～3]=333, 44444
vec=T, E, S, T
my_con=1.2, 2.3, 3.4, 5.6, 7.8, 9.1
```

独自コンテナと STL コンテナを区別するために、関数の引数を T::MY_TYPE 型で受け取る for_each を用意している。与えられた型が ::MY_TYPE をメンバーに持っていれば適合する。

ただし、このような型指定ゆえに、残念ながら変数から型を推論することができず、for_each に明示的に型を与えて for_each<MY_ARRAY<float, 6> > と記述している。

■ 様々なテンプレートライブラリ／その他のテクニック

▼ STL／Boost C++／Loki ライブラリ

STL, Boost C++, Loki といったライブラリは、テンプレートをふんだんに活用しており、有用なものも多い。

ただし、内部で自動的にメモリを確保しているものも多いため、個人の判断で安易に利用せず、きちんとプロジェクトの方針に従って利用するべきである。

▼ コンテナの自作

ゲーム開発の現場では、メモリ管理を徹底しつつ生産性を向上させるためにも、STL の vector などに相当するコンテナを自作するのが最適である。開発プロジェクトに合わせたメモリ制御に適合したコンテナを用意すると、安全かつ便利である。

▼ Expression Template による高速算術演算／Blitz++ライブラリ

Expression Template と呼ばれるテクニックを利用し、ベクトル算術演算などを高速化する手法がある。

例えば、ベクトル型変数 a＋b＋c＋d のような計算を行う際、演算子ごとに計算して結果をスタックに積み直すという普通の処理手順だと、一つ一つのデータ量が大きいこともあり、けっこうロスが大きい。Expression Template では、値をまとめてスタックした後で一気に計算する手法で高速化を実現している。

このような高速化手法で作成された算術ライブラリが Blitz である。

ベクトルや行列の演算は SIMD 演算の活用も重要なので、Blitz を参考にハードウェアに合わせた演算ライブラリを構築するのも有効かもしれない。

▼ その他のテンプレートテクニック

先に紹介した書籍「C++テンプレートテクニック」には、まだまだ様々なテンプレートのテクニックが紹介されている。

特に、本書で示さなかった「パラメータ化継承」や「型変換演算子（キャストオペレーター）を利用した戻り値型に合わせたオーバーロード」、「複数の戻り値を返す関数」、C++11（当時の呼び方では C++0x）の新仕様の解説など、有用な要素がまだまだ多い。

■■以上■■

■ 索引

お

か

き

こ

さ

し

す

せ

た

て

と

は

ふ

ほ

ま

め

ら

効果的なテンプレートテクニック

以　上