テンプレートプログラミング

－ テンプレートの活用で生産性の向適化を －

2014 年 1 月 20 日　初版

板垣 衛

■ 改訂履歴

版	リリース	担当	改訂内容
初版	2014 年 1 月 20 日	板垣 衛	（初版）

■ 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：επισστημη＋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「コンパイルを効率化する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() / max() 関数のサンプルを示す。
オーバーロードにより、3 値以上の比較にも対応し易い。

【サンプル】

テンプレートによる max() 関数のサンプル：　※min() 関数は省略

```
//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： ※min() 関数は省略

```
//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おり、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。（プログラム例は割愛）

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 では完全な定数化されることが期待できる。

また、他には「コンパイル時に文字列から CRC 値を算出する（同時に文字列リテラルは消滅）」といった定数式が書けると、ゲームプログラミングの生産性と処理効率、プログラムサイズが非常によくなる。

ただし、Visual C++ 2013 では constexpr の仕様が未実装であるため、constexpr がコンパイル時に文字列を処理できるかどうかは確認できていない。GCC も Ver.4.6 からの実装のようで、環境の用意が間に合っておらず、確認ができていない。少なくとも、テンプレート関数ではコンパイル時に文字列を処理することができない。

▼ 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>
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>
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>
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。

■ 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」「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、このサンプル処理の単純な実装から、どのように最適化していくかを、順を追って説明する。

【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)
```

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。

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。

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。

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>
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;
        //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;
            }
    };
    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。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、標準の for - each 関数を使用することである。

最後に、この独自の「for_each_array()」関数を、STL 標準の「for_each()」関数に置き換える方法を示す。

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、メンバー変数の代わりに static 変数を使う
    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;
        }
    };
    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。

■ テンプレートにクラスによる多態性

▼ 特殊化

▼ 部分特殊化

▼ SINAE

▼ ポリシー

▼ パラメータ化継承と CRTP

■■以上■■

■ 索引

索引項目が見つかりません。

テンプレートプログラミング

以　上