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 6 月 18 日　第三稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 3 月 13 日	板垣 衛	（初稿）
第二稿	2014 年 4 月 16 日	板垣 衛	constexpr による crc 計算のサンプルプログラムとその説明を大幅改訂。コメントの修正と CRC-32C 対応追加、および、SSE4.2 命令使用コードの追加。
第三項	2014 年 6 月 18 日	板垣 衛	「テンプレートの表記について」を追加。 「メタプログラミング」の「活用例①」～「補足」の構成と内容を大幅改訂。constexpr の説明をもっと分かり易く。 「メタプログラミング」の「活用例②」を大幅改訂。lengthOfArray を extentof に変更。 「メタプログラミング」の「活用例③」を大幅改訂。テンプレート関数版べき乗計算のサンプルを追加。 「メタプログラミング」の「活用例④」を追加。メタプログラミングによる素数計算のサンプル。 「メタプログラミング」の「活用例⑤」を若干改訂。 「高階関数を利用した構造化手法」の標題を、「高階関数を利用した無駄のない処理」から変更。内容も広範囲に改訂し、分かり易さを追求。

■ 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ある。

■ テンプレートに関する表記について

本書では、クラスのテンプレートを「テンプレートクラス」、関数のテンプレートを「テンプレート関数」と表記する。

通常このように表記する場合はテンプレートとして作成されたクラス／関数のことを指す。機能としてのテンプレートを意味する場合は、「クラステンプレート」、「関数テンプレート」と呼ぶ。しかし、本書では両者を区別せず前者の表記を用いる。

また、本書では可変長のテンプレート引数を「可変長テンプレート引数」と表記する。

このように表記する場合は、可変長のテンプレート引数自体を指す。可変長のテンプレート引数が適用されたテンプレート、および、その機能を意味する場合は、「可変長引数テンプレート」と呼ぶ。しかし、本書では両者を区別せず前者の表記を用いる。

■ 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ことにより、若干コンパイル時間が長くなるものの、コードサイズの削減と処理効率の向上といった効果が得られる。

▼ 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。

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()関数と max()関数は比較演算子以外同じであるため、サンプルには max()関数のみを示す。

【サンプル】

テンプレートによる max() 関数のサンプル

```
//max() 関数　※関数のオーバーロードで複数の値に対応
template<typename T> inline T max(T n1, T n2){                  return n1 > n2 ? n1 : n2; }
template<typename T> inline T max(T n1, T n2, T n3){            return n1 > n2 ? n1 : max(n2, n3); }
template<typename T> inline T max(T n1, T n2, T n3, T n4){      return n1 > n2 ? n1 : max(n2, n3, n4); }
template<typename T> inline 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       ←（同上）
010C1CCA  push        5       ←（同上）
010C1CCC  push        2       ←（同上）
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とおり、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）。

また、Visual C++ 2013 で確認したところでは、関数内に 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。

▼ 【補足】テンプレート以外のメタプログラミング手法との比較

メタプログラミング手法の補足として、「活用例①」のテンプレートと同等の処理を、他の手法で実装した場合との違いを比較する。

● 【比較①】使い勝手について

#define マクロでも同様の処理が可能であり、同様に結果を定数化することができる。

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))
```

また、マクロ使用時固有の問題もある。#define マクロでは、引数指定時に演算を行うと、誤った結果になる。テンプレート関数では問題にならない。

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 には 2 が返ることを期待するが、実際には 3 が返される。
```

```
//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い、二重に演算が実行されることが原因。
```

● 【比較②】コンパイル時の定数化について

テンプレートでは期待どおりにコンパイル時に定数化が行えない場合がある。

列挙型の定数やクラス内の static const 型定数にはテンプレート関数を使用することができない。その場合は #define マクロを使用する必要がある。

以下、具体的なサンプルコーデ尾

テンプレートの定数化が期待どおりに行われないケース：

```
//テンプレート関数
template<typename T> inline T t_max(T n1, T n2, T n3, T n4, T n5){ return n1 > n2 ? n1 : max(n2, n3, n4, n5); }
//マクロ
#define m_max(n1, n2, n3, n4, n5) (n1 > n2 ? n1 : max4(n2, n3, n4, 5))

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

//列挙
enum
{
     e1 = t_max(1, 2, 3, 4, 5), //NG:コンパイルエラー
     e2 = m_max(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 = t_max(1, 2, 3, 4, 5); //NG:コンパイルエラー
     static const int sc2 = m_max(1, 2, 3, 4, 5); //OK
     const int c1 = t_max(1, 2, 3, 4, 5);         //OK
     const int c2 = m_max(1, 2, 3, 4, 5);         //OK
     //コンストラクタ
     CClass() :
          v1(t_max(1, 2, 3, 4, 5)), //OK
          v2(m_max(1, 2, 3, 4, 5)) //OK
     {}
     //フィールド
     int v1;
     int v2;
};
```

● 【比較③】C++11 の新仕様について：可変長テンプレート引数

C++11 では、可変長テンプレート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 個の引数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

```
//値が二つの max()
template<typename T1, typename T2>
inline 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>
inline 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           ← （同上）
00821D6B  cmp         eax,esi         ← （同上）
00821D6D  cmovg       esi,eax         ← （同上）
     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         ← （同上）
00821D80  cmovg       eax,esi         ← （同上）
     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             ← （同上）
00821D88  push        9               ← （同上）
00821D8A  push        5               ← （同上）
00821D8C  push        2               ← （同上）
00821D8E  push        849BE4h
00821D93  call        printf (0827767h)
00821D98  add         esp,54h
```

Visual C++ 2013 で確認したところでは、与える引数が多くなると完全には定数化されなくなり、部分的に定数化された値とランタイム時の関数呼び出しとなっていた。

● 【比較④】C++11 の新仕様について：constexpr

定数を算出するためのメタプログラミングの手法は C++11 で更に強化されており、「constexpr」という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、関数の戻り値を定数に与えることができる。前述のテンプレートの再帰では定数化が不完全だったがが、constexpr では完全に定数化される。

constexpr による max() 関数：※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...); }
```

GCC(4.8.2)を使用したところでは、これにより、完全に定数化されることが確認できた。なお、constexpr の実装は、GCC では 4.6 以降。Visual C++は 2013 では対応しておらず、次期バージョンでの実装が予定されている。

▼ 【補足】constexpr の現状と更なる活用

前述の constexpr について補足する。

以下より、constexpr の現状（C++11）の仕様と今後の計画、更に、constexpr を便利に活用する方法について説明する。

● constexpr の現状（C++11 仕様）

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 仕様の constexpr

C++14 仕様では constexpr の制約が緩和され、変数や if 文、クラスが使えるよう

にすることが計画されている。

後の節にて、高度なメタプログラミングの手法を説明するが、それは関数型言語のような難解さがある。constexpr の自由度が高まれば、手続き型言語の処理をそのままコンパイル時に済ませるようになることが期待できる。

● constexpr の活用：CRC 値の算出

constexpr を使用すると、文字列データをコンパイル時に CRC 値に変換して定数化し、元の文字列データは実行コードから削除されるようにすることが可能となる。これができると、ゲームプログラミングでは大変重宝する。

constexpr を扱う上では、少々分かりにくい注意点がある。それは、constexpr 関数の呼び出しが、常にコンパイル時に処理されるわけではないということ。

constexpr 関数の戻り値を、const 型の変数もしくは constexpr 指定された変数で受け取るようにしないと、コンパイル時に定数化されず、実行時の関数呼び出しとなってしまう。

なお、コンパイル時に定数化されたかどうかを確認するには、static_assert（C++11 仕様）を使用して、定数として扱えるかどうかを確認すると良い。

【準備：constexpr による CRC 計算関数】

```
//CRC 計算関数
constexpr unsigned int crc32(const char* s){…省略…}
```

【コンパイル時に定数化される使用方法】

```
//コンパイル時に計算される書き方
const unsigned int val = crc32("123456789");//const 型の変数に代入：コンパイル時に計算される
static_assert(val == 0x261daee5, "calculate mismatch!");//stati_assert も問題なし
```

【実行時の呼び出しになる使用方法】

```
//コンパイル時に計算されず、実行時に計算される書き方
unsigned int val = crc32("123456789");//非 const 型の変数に代入：実行時に計算される
static_assert(val == 0x261daee5, "calculate mismatch!");//stati_assert は、判定以前に評価すらできずエラー
```

この例では、static_assert を使用して計算結果が正しいかどうかをチェックしているが、コンパイル時に定数化されていることを確認するだけなら、「static_assert(val != 0, “dummy”)」のようにするだけでも良い。変数が定数化されていないと、static_assert が処理できず、コンパイルエラーとなる。

● constexpr とユーザー定義リテラル

コンパイル時の CRC 計算のような処理は、C++11 で追加された仕様の「ユーザー定義リテラル」を活用すると更に便利になる。

ユーザー定義リテラルを使用すれば、文字列の CRC 値を表現するために「const

unsigned int crc = "文字列"_CRC;」のように書くことができる。「_CRC」のような独自のサフィックスを付けた表記を用いる。これは、単精度浮動小数点型の例で言えば「12.3f」の「f」の部分に相当するもので、データ形式を指定するためのサフィックスを独自に追加実装できる仕様である。

ユーザー定義リテラルは、特殊な operator として定義する。なお、サフィックスに「_」（アンダースコア）は必ず必要。

【ユーザー定義リテラルによる CRC 計算処理実装】

```
//CRC 計算関数
constexpr unsigned int crc32(const char* s){…省略…}
//ユーザー定義リテラル
constexpr unsigned int operator "" _CRC(const char* s)
{
     return crc32(s);
}
```

【ユーザー定義リテラルを使用した処理】

```
//ユーザー定義リテラルを使用した書き方
const unsigned int val = "123456789"_CRC;//const 型の変数に代入：コンパイル時に計算される
static_assert(val == 0x261daee5, "calculate mismatch!");//stati_assert も問題なし
```

ユーザー定義リテラルは、GCC では 4.7 から対応しており、Visual C++での対応は未定。2013 の次期バージョンでの対応も予定されていない。

GCC(4.8.2)で動作を確認したところでは、ユーザー定義リテラルも constexpr と同様に、const 型の変数に代入しないとコンパイル時に計算されなかった。

● constexpr による CRC 値算出処理のサンプルプログラム

constexpr を使用して CRC 値をコンパイル時に算出するためのサンプルプログラムを示す。メタプログラミングの説明と共に、CRC 計算を SSE 命令で最適化する手法も合わせて説明する。

サンプルプログラムの実行には GCC4.7 以上が必要。

また、constexpr とユーザー定義リテラルの処理を無効化すれば、VisualC++2013 でもコンパイル可能。その場合、ランタイム用の処理だけ動作確認できる。

constexpr の仕様に従い、変数もループも使えず、再帰処理で構成しているので、ランタイム時の処理としては非効率。そのため、ランタイム用の処理は別に設け、ループ処理と SSE4.2 命令を使用することで効率化する。

また、SSE 命令の仕様に合わせて、CRC 計算の生成多項式には、IEEE 勧告の標準的な CRC-32 ではなく、CRC-32C（Castagnoli）を用いる。

なお、256 個の事前計算済み多項式テーブルを用意した場合と、SSE 命令で計算した場合を比較すると、SSE 命令の方が高速に動作することが確認できる。実際の比較

結果はサンプルプログラムの後に記載する。

このサンプルプログラムを GCC でコンパイルする際は、C++11 仕様（constexpr、ユーザー定義リテラル）と SSE4.2 を有効化する必要がある。それには、コンパイラオプションに「-std=c++11」と「-msse4.2」を指定する。

また、Visual C++ 2013 の場合は、constexpr とユーザー定義リテラルをコンパイラスイッチマクロで無効化してコンパイルする必要がある。

サンプルプログラムを以下に示す。

constexpr とユーザー定義リテラル使用箇所を赤色で、SSE 命令使用箇所を黄金色で着色して示す。

【constexpr.h】

```
//C++11 用の constexpr / ユーザー定義リテラルを使ったコンパイル時 CRC 計算および SSE4.2 を使用した CRC 計算

//--------------------
//コンパイラスイッチ用マクロ
#define USE_CONSTEXPR//constexpr を使用する時はこのマクロを有効にする
#define ENABLE_USER_DEFINED_LITERALS//ユーザー定義リテラルを有効にする時はこのマクロを有効にする
#define ENABLE_RUNTIME_FUNC//ランタイム関数を有効にする時はこのマクロを有効にする

#define USE_SSE//SSE4.2 命令を使用する時はこのマクロを有効にする ※CRC32C 専用になるので注意 ※ランタイム時計算専用

#define USE_CRC32C//IEEE 標準の CRC-32 生成多項式ではなく、CRC-32C(Castagnoli)生成多項式使用を使用する時は
                  //このマクロを有効にする
                  //※SSE 使用時は強制的に設定
                  //※本来は、CRC-32 系関数と CRC-32C 系関数の両方を、それぞれ別名の関数で定義したほうがよい
                  //  本サンプルでは、少しでもコードを理解し易くするために、両者を切り替えるスタイルとする

//#define USE_STATIC_TABLE//CRC 計算の際に、事前計算済みの CRC 多項式テーブルを使用する時はこのマクロを有効にする
                          //※通常は有効にしたほうがよい
//#define ENABLE_MAKE_STATIC_TABLE//CRC 多項式テーブル作成関数を有効にする時はこのマクロを有効にする
                                  //※CRC 多項式テーブル未使用時のみ指定可

//--------------------
//コンパイラスイッチ調整
#ifdef USE_SSE
#ifndef USE_CRC32C
#define USE_CRC32C//CRC-32C(Castagnoli)生成多項式使用 ※SSE4.2 命令使用時は CRC32C 生成多項式で確定
#endif//USE_CRC32C
#endif//USE_SSE

#ifdef USE_CONSTEXPR
#ifndef MAKE_CRC_INSTANCE
#define MAKE_CRC_INSTANCE//※constexp 使用時は、ヘッダーで関数を実体化する
#endif//MAKE_CRC_INSTANCE
#endif//USE_CONSTEXPR

#ifdef USE_STATIC_TABLE
#ifdef ENABLE_MAKE_STATIC_TABLE
#undef ENABLE_MAKE_STATIC_TABLE//※CRC 多項式テーブル使用時は、CRC 多項式テーブルを作成（表示）できない
#endif//ENABLE_MAKE_STATIC_TABLE
#endif//USE_STATIC_TABLE

//--------------------
//C++11 互換用マクロ
#ifdef USE_CONSTEXPR
#define CONSTEXPR constexpr//※constexpr を使用（コンパイル時計算が有効になる）
```

```
#else//USE_CONSTEXPR
#define CONSTEXPR const//※constexpr の代わりに const を使用（コンパイル時計算はできない）
#endif//USE_CONSTEXPR

//---------------------
#include <cstddef>//std::size_t 用

//-----------------------------------------------------------------------------
//CRC 算出

//---------------------
//型
typedef unsigned int crc32_t;//CRC32 型

//---------------------
//CRC 算出関数
//※コンパイル時に処理するため、ヘッダーに処理を記述する
namespace crc_sub//直接使用しない処理を隠ぺいするためのネームスペース
{
#ifndef USE_STATIC_TABLE
	//---------------------
	//CRC 多項式計算
	//※constexpr 関数内では変数/定数が使えないため、定数にマクロを使用
	//※constexpr 関数内ではラムダ式が使えないため、関数を分割
	//　（ラムダ式を使用するとコンパイル時に評価されなくなる）
	CONSTEXPR inline crc32_t calcPoly_core(const crc32_t poly)
	{
		//生成多項式
	#ifndef USE_CRC32C
		//IEEE 勧告の一般的な CRC32
		//#define POLYNOMIAL 0x04c11db7u//（標準）
		#define POLYNOMIAL 0xedb88320u//（反転）
	#else//USE_CRC32C
		//CRC-32C(Castagnoli) ※SSE4.2 命令準拠 ※こちらの方が優れている
		//#define POLYNOMIAL 0x1edc6f41u//（標準）
		#define POLYNOMIAL 0x82f63b78u//（反転）
	#endif//USE_CRC32C
		return poly & 1 ? POLYNOMIAL ^ (poly >> 1) : (poly >> 1);
		#undef POLYNOMIAL
	}
	CONSTEXPR inline crc32_t calcPoly(const crc32_t poly)
	{
		//多項式計算
		return calcPoly_core(
			calcPoly_core(
			  calcPoly_core(
			    calcPoly_core(
			      calcPoly_core(
			        calcPoly_core(
			          calcPoly_core(
			            calcPoly_core(poly)
			          )
			        )
			      )
			    )
			  )
			);
	}
#else//USE_STATIC_TABLE
	//---------------------
	//CRC 多項式計算済みテーブル
#ifdef MAKE_CRC_INSTANCE
	CONSTEXPR crc32_t s_polyTable[256] =
	{
```

```
#ifndef USE_CRC32C
        0x00000000u, 0x77073096u, 0xee0e612cu, 0x990951bau, 0x076dc419u, 0x706af48fu, 0xe963a535u, 0x9e6495a3u,
        0x0edb8832u, 0x79dcb8a4u, 0xe0d5e91eu, 0x97d2d988u, 0x09b64c2bu, 0x7eb17cbdu, 0xe7b82d07u, 0x90bf1d91u,
        0x1db71064u, 0x6ab020f2u, 0xf3b97148u, 0x84be41deu, 0x1adad47du, 0x6ddde4ebu, 0xf4d4b551u, 0x83d385c7u,
        0x136c9856u, 0x646ba8c0u, 0xfd62f97au, 0x8a65c9ecu, 0x14015c4fu, 0x63066cd9u, 0xfa0f3d63u, 0x8d080df5u,
        0x3b6e20c8u, 0x4c69105eu, 0xd56041e4u, 0xa2677172u, 0x3c03e4d1u, 0x4b04d447u, 0xd20d85fdu, 0xa50ab56bu,
        0x35b5a8fau, 0x42b2986cu, 0xdbbbc9d6u, 0xacbcf940u, 0x32d86ce3u, 0x45df5c75u, 0xdcd60dcfu, 0xabd13d59u,
        0x26d930acu, 0x51de003au, 0xc8d75180u, 0xbfd06116u, 0x21b4f4b5u, 0x56b3c423u, 0xcfba9599u, 0xb8bda50fu,
        0x2802b89eu, 0x5f058808u, 0xc60cd9b2u, 0xb10be924u, 0x2f6f7c87u, 0x58684c11u, 0xc1611dabu, 0xb6662d3du,
        0x76dc4190u, 0x01db7106u, 0x98d220bcu, 0xefd5102au, 0x71b18589u, 0x06b6b51fu, 0x9fbfe4a5u, 0xe8b8d433u,
        0x7807c9a2u, 0x0f00f934u, 0x9609a88eu, 0xe10e9818u, 0x7f6a0dbbu, 0x086d3d2du, 0x91646c97u, 0xe6635c01u,
        0x6b6b51f4u, 0x1c6c6162u, 0x856530d8u, 0xf262004eu, 0x6c0695edu, 0x1b01a57bu, 0x8208f4c1u, 0xf50fc457u,
        0x65b0d9c6u, 0x12b7e950u, 0x8bbeb8eau, 0xfcb9887cu, 0x62dd1ddfu, 0x15da2d49u, 0x8cd37cf3u, 0xfbd44c65u,
        0x4db26158u, 0x3ab551ceu, 0xa3bc0074u, 0xd4bb30e2u, 0x4adfa541u, 0x3dd895d7u, 0xa4d1c46du, 0xd3d6f4fbu,
        0x4369e96au, 0x346ed9fcu, 0xad678846u, 0xda60b8d0u, 0x44042d73u, 0x33031de5u, 0xaa0a4c5fu, 0xdd0d7cc9u,
        0x5005713cu, 0x270241aau, 0xbe0b1010u, 0xc90c2086u, 0x5768b525u, 0x206f85b3u, 0xb966d409u, 0xce61e49fu,
        0x5edef90eu, 0x29d9c998u, 0xb0d09822u, 0xc7d7a8b4u, 0x59b33d17u, 0x2eb40d81u, 0xb7bd5c3bu, 0xc0ba6cadu,
        0xedb88320u, 0x9abfb3b6u, 0x03b6e20cu, 0x74b1d29au, 0xead54739u, 0x9dd277afu, 0x04db2615u, 0x73dc1683u,
        0xe3630b12u, 0x94643b84u, 0x0d6d6a3eu, 0x7a6a5aa8u, 0xe40ecf0bu, 0x9309ff9du, 0x0a00ae27u, 0x7d079eb1u,
        0xf00f9344u, 0x8708a3d2u, 0x1e01f268u, 0x6906c2feu, 0xf762575du, 0x806567cbu, 0x196c3671u, 0x6e6b06e7u,
        0xfed41b76u, 0x89d32be0u, 0x10da7a5au, 0x67dd4accu, 0xf9b9df6fu, 0x8ebeeff9u, 0x17b7be43u, 0x60b08ed5u,
        0xd6d6a3e8u, 0xa1d1937eu, 0x38d8c2c4u, 0x4fdff252u, 0xd1bb67f1u, 0xa6bc5767u, 0x3fb506ddu, 0x48b2364bu,
        0xd80d2bdau, 0xaf0a1b4cu, 0x36034af6u, 0x41047a60u, 0xdf60efc3u, 0xa867df55u, 0x316e8eefu, 0x4669be79u,
        0xcb61b38cu, 0xbc66831au, 0x256fd2a0u, 0x5268e236u, 0xcc0c7795u, 0xbb0b4703u, 0x220216b9u, 0x5505262fu,
        0xc5ba3bbeu, 0xb2bd0b28u, 0x2bb45a92u, 0x5cb36a04u, 0xc2d7ffa7u, 0xb5d0cf31u, 0x2cd99e8bu, 0x5bdeae1du,
        0x9b64c2b0u, 0xec63f226u, 0x756aa39cu, 0x026d930au, 0x9c0906a9u, 0xeb0e363fu, 0x72076785u, 0x05005713u,
        0x95bf4a82u, 0xe2b87a14u, 0x7bb12baeu, 0x0cb61b38u, 0x92d28e9bu, 0xe5d5be0du, 0x7cdcefb7u, 0x0bdbdf21u,
        0x86d3d2d4u, 0xf1d4e242u, 0x68ddb3f8u, 0x1fda836eu, 0x81be16cdu, 0xf6b9265bu, 0x6fb077e1u, 0x18b74777u,
        0x88085ae6u, 0xff0f6a70u, 0x66063bcau, 0x11010b5cu, 0x8f659effu, 0xf862ae69u, 0x616bffd3u, 0x166ccf45u,
        0xa00ae278u, 0xd70dd2eeu, 0x4e048354u, 0x3903b3c2u, 0xa7672661u, 0xd06016f7u, 0x4969474du, 0x3e6e77dbu,
        0xaed16a4au, 0xd9d65adcu, 0x40df0b66u, 0x37d83bf0u, 0xa9bcae53u, 0xdebb9ec5u, 0x47b2cf7fu, 0x30b5ffe9u,
        0xbdbdf21cu, 0xcabac28au, 0x53b39330u, 0x24b4a3a6u, 0xbad03605u, 0xcdd70693u, 0x54de5729u, 0x23d967bfu,
        0xb3667a2eu, 0xc4614ab8u, 0x5d681b02u, 0x2a6f2b94u, 0xb40bbe37u, 0xc30c8ea1u, 0x5a05df1bu, 0x2d02ef8du
#else//USE_CRC32C
        0x00000000u, 0xf26b8303u, 0xe13b70f7u, 0x1350f3f4u, 0xc79a971fu, 0x35f1141cu, 0x26a1e7e8u, 0xd4ca64ebu,
        0x8ad958cfu, 0x78b2dbccu, 0x6be22838u, 0x9989ab3bu, 0x4d43cfd0u, 0xbf284cd3u, 0xac78bf27u, 0x5e133c24u,
        0x105ec76fu, 0xe235446cu, 0xf165b798u, 0x030e349bu, 0xd7c45070u, 0x25afd373u, 0x36ff2087u, 0xc494a384u,
        0x9a879fa0u, 0x68ec1ca3u, 0x7bbcef57u, 0x89d76c54u, 0x5d1d08bfu, 0xaf768bbcu, 0xbc267848u, 0x4e4dfb4bu,
        0x20bd8edeu, 0xd2d60dddu, 0xc186fe29u, 0x33ed7d2au, 0xe72719c1u, 0x154c9ac2u, 0x061c6936u, 0xf477ea35u,
        0xaa64d611u, 0x580f5512u, 0x4b5fa6e6u, 0xb93425e5u, 0x6dfe410eu, 0x9f95c20du, 0x8cc531f9u, 0x7eaeb2fau,
        0x30e349b1u, 0xc288cab2u, 0xd1d83946u, 0x23b3ba45u, 0xf779deaeu, 0x05125dadu, 0x1642ae59u, 0xe4292d5au,
        0xba3a117eu, 0x4851927du, 0x5b016189u, 0xa96ae28au, 0x7da08661u, 0x8fcb0562u, 0x9c9bf696u, 0x6ef07595u,
        0x417b1dbcu, 0xb3109ebfu, 0xa0406d4bu, 0x522bee48u, 0x86e18aa3u, 0x748a09a0u, 0x67dafa54u, 0x95b17957u,
        0xcba24573u, 0x39c9c670u, 0x2a993584u, 0xd8f2b687u, 0x0c38d26cu, 0xfe53516fu, 0xed03a29bu, 0x1f682198u,
        0x5125dad3u, 0xa34e59d0u, 0xb01eaa24u, 0x42752927u, 0x96bf4dccu, 0x64d4cecfu, 0x77843d3bu, 0x85efbe38u,
        0xdbfc821cu, 0x2997011fu, 0x3ac7f2ebu, 0xc8ac71e8u, 0x1c661503u, 0xee0d9600u, 0xfd5d65f4u, 0x0f36e6f7u,
        0x61c69362u, 0x93ad1061u, 0x80fde395u, 0x72966096u, 0xa65c047du, 0x5437877eu, 0x4767748au, 0xb50cf789u,
        0xeb1fcbadu, 0x197448aeu, 0x0a24bb5au, 0xf84f3859u, 0x2c855cb2u, 0xdeeedfb1u, 0xcdbe2c45u, 0x3fd5af46u,
        0x7198540du, 0x83f3d70eu, 0x90a324fau, 0x62c8a7f9u, 0xb602c312u, 0x44694011u, 0x5739b3e5u, 0xa55230e6u,
        0xfb410cc2u, 0x092a8fc1u, 0x1a7a7c35u, 0xe811ff36u, 0x3cdb9bddu, 0xceb018deu, 0xdde0eb2au, 0x2f8b6829u,
        0x82f63b78u, 0x709db87bu, 0x63cd4b8fu, 0x91a6c88cu, 0x456cac67u, 0xb7072f64u, 0xa457dc90u, 0x563c5f93u,
        0x082f63b7u, 0xfa44e0b4u, 0xe9141340u, 0x1b7f9043u, 0xcfb5f4a8u, 0x3dde77abu, 0x2e8e845fu, 0xdce5075cu,
        0x92a8fc17u, 0x60c37f14u, 0x73938ce0u, 0x81f80fe3u, 0x55326b08u, 0xa759e80bu, 0xb4091bffu, 0x466298fcu,
        0x1871a4d8u, 0xea1a27dbu, 0xf94ad42fu, 0x0b21572cu, 0xdfeb33c7u, 0x2d80b0c4u, 0x3ed04330u, 0xccbbc033u,
        0xa24bb5a6u, 0x502036a5u, 0x4370c551u, 0xb11b4652u, 0x65d122b9u, 0x97baa1bau, 0x84ea524eu, 0x7681d14du,
        0x2892ed69u, 0xdaf96e6au, 0xc9a99d9eu, 0x3bc21e9du, 0xef087a76u, 0x1d63f975u, 0x0e330a81u, 0xfc588982u,
        0xb21572c9u, 0x407ef1cau, 0x532e023eu, 0xa145813du, 0x758fe5d6u, 0x87e466d5u, 0x94b49521u, 0x66df1622u,
        0x38cc2a06u, 0xcaa7a905u, 0xd9f75af1u, 0x2b9cd9f2u, 0xff56bd19u, 0x0d3d3e1au, 0x1e6dcdeeu, 0xec064eedu,
        0xc38d26c4u, 0x31e6a5c7u, 0x22b65633u, 0xd0ddd530u, 0x0417b1dbu, 0xf67c32d8u, 0xe52cc12cu, 0x1747422fu,
        0x49547e0bu, 0xbb3ffd08u, 0xa86f0efcu, 0x5a048dffu, 0x8ecee914u, 0x7ca56a17u, 0x6ff599e3u, 0x9d9e1ae0u,
        0xd3d3e1abu, 0x21b862a8u, 0x32e8915cu, 0xc083125fu, 0x144976b4u, 0xe622f5b7u, 0xf5720643u, 0x07198540u,
        0x590ab964u, 0xab613a67u, 0xb831c993u, 0x4a5a4a90u, 0x9e902e7bu, 0x6cfbad78u, 0x7fab5e8cu, 0x8dc0dd8fu,
        0xe330a81au, 0x115b2b19u, 0x020bd8edu, 0xf0605beeu, 0x24aa3f05u, 0xd6c1bc06u, 0xc5914ff2u, 0x37faccf1u,
        0x69e9f0d5u, 0x9b8273d6u, 0x88d28022u, 0x7ab90321u, 0xae7367cau, 0x5c18e4c9u, 0x4f48173du, 0xbd23943eu,
        0xf36e6f75u, 0x0105ec76u, 0x12551f82u, 0xe03e9c81u, 0x34f4f86au, 0xc69f7b69u, 0xd5cf889du, 0x27a40b9eu,
```

```
            0x79b737bau, 0x8bdcb4b9u, 0x988c474du, 0x6ae7c44eu, 0xbe2da0a5u, 0x4c4623a6u, 0x5f16d052u, 0xad7d5351u
      #endif//USE_CRC32C
      };
#endif//MAKE_CRC_INSTANCE
#endif//USE_STATIC_TABLE
      //--------------------
      //文字列から CRC 算出用（再帰処理）
      //※constexpr 関数内では SSE 命令に非対応（使用するとコンパイル時に評価されなくなる）
      CONSTEXPR crc32_t calcStr(const crc32_t crc, const char* str)
#ifdef MAKE_CRC_INSTANCE
      {
      #ifndef USE_STATIC_TABLE
            return *str == '¥0' ?
                  crc :
                  calcStr(calcPoly(static_cast<crc32_t>((crc ^ *str) & 0xffu)) ^ (crc >> 8), str + 1);
                                                                  //CRC 多項式(生成多項式から計算)を合成
      #else//USE_STATIC_TABLE
            return *str == '¥0' ?
                  crc :
                  calcStr(s_polyTable[(crc ^ *str) & 0xffu] ^ (crc >> 8), str + 1);
                                                                  //CRC 多項式(計算済みテーブルの値)を合成
      #endif//USE_STATIC_TABLE
      }
#else//MAKE_CRC_INSTANCE
      ;
#endif//MAKE_CRC_INSTANCE
      //--------------------
      //データ長を指定して CRC 算出用（再帰処理）
      //※constexpr 関数内では SSE 命令に非対応（使用するとコンパイル時に評価されなくなる）
      CONSTEXPR crc32_t calcData(const crc32_t crc, const char* data, const std::size_t len)
#ifdef MAKE_CRC_INSTANCE
      {
      #ifndef USE_STATIC_TABLE
            return len == 0 ?
                  crc :
                  calcData(calcPoly(static_cast<crc32_t>((crc ^ *data) & 0xffu)) ^ (crc >> 8), data + 1, len - 1);
                                                                  //CRC 多項式(生成多項式から計算)を合成
      #else//USE_STATIC_TABLE
            return len == 0 ?
                  crc :
                  calcData(s_polyTable[(crc ^ *data) & 0xffu] ^ (crc >> 8), data + 1, len - 1);
                                                                  //CRC 多項式(計算済みテーブルの値)を合成
      #endif//USE_STATIC_TABLE
      }
#else//MAKE_CRC_INSTANCE
      ;
#endif//MAKE_CRC_INSTANCE
}//namespace crc_sub

//--------------------
//【constexpr 版】文字列から CRC 算出
CONSTEXPR inline crc32_t calcConstCRC32(const char* str)
{
      return ~crc_sub::calcStr(~0u, str);
}
//--------------------
//【constexpr 版】データ長を指定して CRC 算出
CONSTEXPR inline crc32_t calcConstCRC32(const char* data, const std::size_t len)
{
      return ~crc_sub::calcData(~0u, data, len);
}
#ifdef ENABLE_USER_DEFINED_LITERALS
//--------------------
//【ユーザー定義リテラル版】文字列と文字列長を指定して CRC 算出
//※operator "" の後に空白が必要なことに注意
```

```
CONSTEXPR inline crc32_t operator "" _crc32(const char* str, const std::size_t len)
{
	return calcConstCRC32(str, len);
}
#endif//ENABLE_USER_DEFINED_LITERALS

#ifdef ENABLE_RUNTIME_FUNC
//--------------------
//【通常関数版】文字列から CRC 算出
crc32_t calcCRC32(const char* str);
//--------------------
//【通常関数版】データ長を指定して CRC 算出
crc32_t calcCRC32(const char* data, const std::size_t len);
#endif//ENABLE_RUNTIME_FUNC
#ifdef ENABLE_MAKE_STATIC_TABLE
//--------------------
//CRC 多項式テーブルを作成
void makePolyTable();
#endif//ENABLE_MAKE_STATIC_TABLE
```

【constexpr.cpp】

```
#define MAKE_CRC_INSTANCE//"constexpr.h"内に定義されている関数を実体化
#include "constexpr.h"

#include <stdio.h>//printf 用

#ifdef USE_SSE
#include <nmmintrin.h>//SSE4.2
#endif//USE_SSE

//-----------------------------------------------------------------------------
//CRC 算出

#ifdef ENABLE_RUNTIME_FUNC
//--------------------
//【通常関数版】文字列から CRC 算出
crc32_t calcCRC32(const char* str)
{
	crc32_t crc = ~0u;
	const char* p = str;
	while (*p)
	{
	#ifndef USE_SSE
	#ifndef USE_STATIC_TABLE
		crc = crc_sub::calcPoly(static_cast<crc32_t>((crc ^ *(p++)) & 0xffu)) ^ (crc >> 8);
                                                                    //CRC 多項式(生成多項式から計算)を合成
	#else//USE_STATIC_TABLE
		crc = crc_sub::s_polyTable[(crc ^ *(p++)) & 0xffu] ^ (crc >> 8);
                                                                    //CRC 多項式(計算済みテーブルの値)を合成
	#endif//USE_STATIC_TABLE
	#else//USE_SSE
		crc = _mm_crc32_u8(crc, *(p++));//CRC 多項式(SSE4.2 による計算)を合成
	#endif//USE_SSE
	}
	return ~crc;
}
//--------------------
//【通常関数版】データ長を指定して CRC 算出
crc32_t calcCRC32(const char* data, const std::size_t len)
{
	crc32_t crc = ~0u;
	const char* p = data;
	for (std::size_t pos = 0; pos < len; ++pos)
	{
	#ifndef USE_SSE
```

```
	#ifndef USE_STATIC_TABLE
		crc = crc_sub::calcPoly(static_cast<crc32_t>((crc ^ *(p++)) & 0xffu)) ^ (crc >> 8);
                                                                    //CRC 多項式(生成多項式から計算)を合成
	#else//USE_STATIC_TABLE
		crc = crc_sub::s_polyTable[(crc ^ *(p++)) & 0xffu] ^ (crc >> 8);
                                                                    //CRC 多項式(計算済みテーブルの値)を合成
	#endif//USE_STATIC_TABLE
	#else//USE_SSE
		crc = _mm_crc32_u8(crc, *(p++));//CRC 多項式(SSE4.2 による計算)を合成
	#endif//USE_SSE
	}
	return ~crc;
}
#endif//ENABLE_RUNTIME_FUNC
#ifdef ENABLE_MAKE_STATIC_TABLE
//--------------------
//CRC 多項式テーブルを作成
void makePolyTable()
{
	printf("¥tCONSTEXPR crc32_t s_polyTable[256] =¥n");
	printf("¥t{¥n");
	for (int i = 0; i < 256; ++i)
	{
		if (i > 0)
		{
			printf(",");
			if (i % 8 == 0)
				printf("¥n");
			else
				printf(" ");
		}
		if (i % 8 == 0)
			printf("¥t¥t");
		printf("0x%08xu", crc_sub::calcPoly(i));
	}
	printf("¥n¥t};¥n");
}
#endif//ENABLE_MAKE_STATIC_TABLE
```

【constexpr_test.h】

```
//C++11 用の constexpr / ユーザー定義リテラルを使ったコンパイル時 CRC 計算および SSE4.2 を使用した CRC 計算テスト

//--------------------
#include "constexpr.h"

//--------------------
//コンパイラスイッチ用マクロ
#define ENABLE_MAIN//メイン関数を有効にする時はこのマクロを有効にする

//#define USE_MAKE_STATIC_TABLE//CRC 多項式テーブル作成を使用する時はこのマクロを有効にする
#define ENABLE_CONSTEXPR_TEST//constexpr テストを有効にする時はこのマクロを有効にする
#define ENABLE_USER_DEFINED_LITERALS_TEST//ユーザー定義リテラルテストを有効にする時はこのマクロを有効にする
#define ENABLE_RUNTIME_TEST//ランタイム時 CRC 計算テストを有効にする時はこのマクロを有効にする
//#define ENABLE_MORE_TEST//追加テストを有効にする時はこのマクロを有効にする
//#define ENABLE_PERFORMANCE_TEST//パフォーマンステストを有効にする時はこのマクロを有効にする

#define USE_STATIC_ASSERT//static_assert を使用する時はこのマクロを有効にする

#define CONST_TO_CONST//CONST マクロを const にする時はこのマクロを有効にする
//#define NOCONST_TO_CONST//NOCONST マクロを const にする時はこのマクロを有効にする

//※「ランタイム時 CRC 計算テスト」と「追加テスト」、「パフォーマンステスト」を無効化すると、
//  constexpr とユーザー定義リテラルのにより、文字列リテラル "1234567890" が
//  コンパイル時に消滅して、CRC のリテラル値で扱われることが、アセンブラコードから確認できる。
//  なお、const 型で計算結果を受けないと、コンパイル時に計算されず、ランタイム時の処理になってしまうので注意。
```

```
//--------------------
//コンパイラスイッチ調整
#ifndef ENABLE_MAKE_STATIC_TABLE
#ifdef USE_MAKE_STATIC_TABLE
#undef USE_MAKE_STATIC_TABLE//※CRC 多項式テーブルを作成が無効なら、呼び出せない
#endif//USE_MAKE_STATIC_TABLE
#endif//ENABLE_MAKE_STATIC_TABLE

#ifndef ENABLE_USER_DEFINED_LITERALS
#ifdef ENABLE_USER_DEFINED_LITERALS_TEST
#undef ENABLE_USER_DEFINED_LITERALS_TEST//※ユーザー定義リテラルが無効なら、テストできない
#endif//ENABLE_USER_DEFINED_LITERALS_TEST
#endif//ENABLE_USER_DEFINED_LITERALS

#ifndef ENABLE_RUNTIME_FUNC
#ifdef ENABLE_RUNTIME_TEST
#undef ENABLE_RUNTIME_TEST//※ランタイム関数が無効なら、テストできない
#endif//ENABLE_RUNTIME_TEST
#ifdef ENABLE_PERFORMANCE_TEST
#undef ENABLE_PERFORMANCE_TEST//※ランタイム関数が無効なら、パフォーマンステストできない
#endif//ENABLE_PERFORMANCE_TEST
#endif//ENABLE_RUNTIME_FUNC

//--------------------
//CONST マクロ
#ifdef CONST_TO_CONST
#define CONST const//※CONST を const として扱う
#else//CONST_TO_CONST
#define CONST//※CONST を非 const として扱う
#endif//CONST_TO_CONST
//NOCONST マクロ
#ifdef NOCONST_TO_CONST
#define NOCONST const//※NOCONST を const として扱う
#else//NOCONST_TO_CONST
#define NOCONST//※NOCONST を非 const として扱う
#endif//NOCONST_TO_CONST
```

【constexpr_test.cpp】

```
#include "constexpr_test.h"

#include <stdio.h>
#include <stdlib.h>

#include <assert.h>//assert 用

#include <chrono>//処理時間計測用

//--------------------------------------------------------------------------------
//テスト

#ifdef ENABLE_MORE_TEST
//--------------------
//追加テスト用データ定義
static const char* TEXT1 = "1";
static const char* TEXT2 = "12";
static const char* TEXT3 = "123";
static const char* TEXT4 = "1234";
static const char* TEXT5 = "12345";
static const char* TEXT6 = "123456";
static const char* TEXT7 = "1234567";
static const char* TEXT8 = "12345678";
static const char* TEXT9 = "123456789";
static const char* TEXT10 = "1234567890";
#endif//ENABLE_MORE_TEST
```

```
//--------------------
//constexpr / ユーザー定義リテラルによる CRC 計算のテスト
void test_constexpr()
{
#ifdef USE_MAKE_STATIC_TABLE
	{
		printf("¥n");
		makePolyTable();//CRC 多項式テーブルを作成
	}
#endif//USE_MAKE_STATIC_TABLE
#ifdef ENABLE_CONSTEXPR_TEST
	{
		printf("¥n");
		CONST crc32_t crc = calcConstCRC32("1234567890");//constexpr でコンパイル時に計算
							//※文字列リテラルも消滅
							//※const 変数に代入しないとコンパイル時に計算されないので注意
	#ifdef USE_STATIC_ASSERT
	#ifndef USE_CRC32C
		static_assert(crc == 0x261daee5u, "invalid crc");//OK：コンパイル時に評価
	#else//USE_CRC32C
		static_assert(crc == 0xf3dbd4feu, "invalid crc");//OK：コンパイル時に評価
	#endif//USE_CRC32C
	#else//USE_STATIC_ASSERT
	#ifndef USE_CRC32C
		assert(crc == 0x261daee5u);//OK：ランタイム時に評価
	#else//USE_CRC32C
		assert(crc == 0xf3dbd4feu);//OK：ランタイム時に評価
	#endif//USE_CRC32C
	#endif//USE_STATIC_ASSERT
		printf("constexpr による CRC 計算結果=0x%08x¥n", crc);
	#ifdef ENABLE_MORE_TEST
		CONST crc32_t TEXT1_CRC = calcConstCRC32(TEXT1);
		CONST crc32_t TEXT2_CRC = calcConstCRC32(TEXT2);
		CONST crc32_t TEXT3_CRC = calcConstCRC32(TEXT3);
		CONST crc32_t TEXT4_CRC = calcConstCRC32(TEXT4);
		CONST crc32_t TEXT5_CRC = calcConstCRC32(TEXT5);
		CONST crc32_t TEXT6_CRC = calcConstCRC32(TEXT6);
		CONST crc32_t TEXT7_CRC = calcConstCRC32(TEXT7);
		CONST crc32_t TEXT8_CRC = calcConstCRC32(TEXT8);
		CONST crc32_t TEXT9_CRC = calcConstCRC32(TEXT9);
		CONST crc32_t TEXT10_CRC = calcConstCRC32(TEXT10);
		printf(" CRC32:TEXT1:¥"%s¥" = 0x%08x(%u)¥n", TEXT1, TEXT1_CRC, TEXT1_CRC);
		printf(" CRC32:TEXT2:¥"%s¥" = 0x%08x(%u)¥n", TEXT2, TEXT2_CRC, TEXT2_CRC);
		printf(" CRC32:TEXT3:¥"%s¥" = 0x%08x(%u)¥n", TEXT3, TEXT3_CRC, TEXT3_CRC);
		printf(" CRC32:TEXT4:¥"%s¥" = 0x%08x(%u)¥n", TEXT4, TEXT4_CRC, TEXT4_CRC);
		printf(" CRC32:TEXT5:¥"%s¥" = 0x%08x(%u)¥n", TEXT5, TEXT5_CRC, TEXT5_CRC);
		printf(" CRC32:TEXT6:¥"%s¥" = 0x%08x(%u)¥n", TEXT6, TEXT6_CRC, TEXT6_CRC);
		printf(" CRC32:TEXT7:¥"%s¥" = 0x%08x(%u)¥n", TEXT7, TEXT7_CRC, TEXT7_CRC);
		printf(" CRC32:TEXT8:¥"%s¥" = 0x%08x(%u)¥n", TEXT8, TEXT8_CRC, TEXT8_CRC);
		printf(" CRC32:TEXT9:¥"%s¥" = 0x%08x(%u)¥n", TEXT9, TEXT9_CRC, TEXT9_CRC);
		printf(" CRC32:TEXT10:¥"%s¥" = 0x%08x(%u)¥n", TEXT10, TEXT10_CRC, TEXT10_CRC);
	#endif//ENABLE_MORE_TEST
	}
#endif//ENABLE_CONSTEXPR_TEST
#ifdef ENABLE_USER_DEFINED_LITERALS_TEST
	{
		printf("¥n");
		CONST crc32_t crc = "1234567890"_crc32;//ユーザー定義リテラルでコンパイル時に計算
							//※文字列リテラルも消滅
							//※const 変数に代入しないとコンパイル時に計算されないので注意
	#ifdef USE_STATIC_ASSERT
	#ifndef USE_CRC32C
		static_assert(crc == 0x261daee5u, "invalid crc");//OK：コンパイル時に評価
	#else//USE_CRC32C
```

```
            static_assert(crc == 0xf3dbd4feu, "invalid crc");//OK：コンパイル時に評価
      #endif//USE_CRC32C
      #else//USE_STATIC_ASSERT
      #ifndef USE_CRC32C
            assert(crc == 0x261daee5u);//OK：ランタイム時に評価
      #else//USE_CRC32C
            assert(crc == 0xf3dbd4feu);//OK：ランタイム時に評価
      #endif//USE_CRC32C
      #endif//USE_STATIC_ASSERT
            printf("ユーザー定義リテラルによる CRC 計算結果=0x%08x¥n", crc);
      #ifdef ENABLE_MORE_TEST
            CONST crc32_t TEXT1_CRC = "1"_crc32;
            CONST crc32_t TEXT2_CRC = "12"_crc32;
            CONST crc32_t TEXT3_CRC = "123"_crc32;
            CONST crc32_t TEXT4_CRC = "1234"_crc32;
            CONST crc32_t TEXT5_CRC = "12345"_crc32;
            CONST crc32_t TEXT6_CRC = "123456"_crc32;
            CONST crc32_t TEXT7_CRC = "1234567"_crc32;
            CONST crc32_t TEXT8_CRC = "12345678"_crc32;
            CONST crc32_t TEXT9_CRC = "123456789"_crc32;
            CONST crc32_t TEXT10_CRC = "1234567890"_crc32;
            printf(" CRC32:TEXT1:¥"%s¥" = 0x%08x(%u)¥n", TEXT1, TEXT1_CRC, TEXT1_CRC);
            printf(" CRC32:TEXT2:¥"%s¥" = 0x%08x(%u)¥n", TEXT2, TEXT2_CRC, TEXT2_CRC);
            printf(" CRC32:TEXT3:¥"%s¥" = 0x%08x(%u)¥n", TEXT3, TEXT3_CRC, TEXT3_CRC);
            printf(" CRC32:TEXT4:¥"%s¥" = 0x%08x(%u)¥n", TEXT4, TEXT4_CRC, TEXT4_CRC);
            printf(" CRC32:TEXT5:¥"%s¥" = 0x%08x(%u)¥n", TEXT5, TEXT5_CRC, TEXT5_CRC);
            printf(" CRC32:TEXT6:¥"%s¥" = 0x%08x(%u)¥n", TEXT6, TEXT6_CRC, TEXT6_CRC);
            printf(" CRC32:TEXT7:¥"%s¥" = 0x%08x(%u)¥n", TEXT7, TEXT7_CRC, TEXT7_CRC);
            printf(" CRC32:TEXT8:¥"%s¥" = 0x%08x(%u)¥n", TEXT8, TEXT8_CRC, TEXT8_CRC);
            printf(" CRC32:TEXT9:¥"%s¥" = 0x%08x(%u)¥n", TEXT9, TEXT9_CRC, TEXT9_CRC);
            printf(" CRC32:TEXT10:¥"%s¥" = 0x%08x(%u)¥n", TEXT10, TEXT10_CRC, TEXT10_CRC);
      #endif//ENABLE_MORE_TEST
      }
#endif//ENABLE_USER_DEFINED_LITERALS_TEST
#ifdef ENABLE_RUNTIME_TEST
      {
            printf("¥n");
            NOCONST crc32_t crc = calcCRC32("1234567890");//通常関数でランタイム時に計算
      #ifndef USE_CRC32C
            assert(crc == 0x261daee5u);//OK：ランタイム時に評価
      #else//USE_CRC32C
            assert(crc == 0xf3dbd4feu);//OK：ランタイム時に評価
      #endif//USE_CRC32C
            printf("ランタイム関数による CRC 計算結果=0x%08x¥n", crc);
      #ifdef ENABLE_MORE_TEST
            NOCONST crc32_t TEXT1_CRC = calcCRC32(TEXT1);
            NOCONST crc32_t TEXT2_CRC = calcCRC32(TEXT2);
            NOCONST crc32_t TEXT3_CRC = calcCRC32(TEXT3);
            NOCONST crc32_t TEXT4_CRC = calcCRC32(TEXT4);
            NOCONST crc32_t TEXT5_CRC = calcCRC32(TEXT5);
            NOCONST crc32_t TEXT6_CRC = calcCRC32(TEXT6);
            NOCONST crc32_t TEXT7_CRC = calcCRC32(TEXT7);
            NOCONST crc32_t TEXT8_CRC = calcCRC32(TEXT8);
            NOCONST crc32_t TEXT9_CRC = calcCRC32(TEXT9);
            NOCONST crc32_t TEXT10_CRC = calcCRC32(TEXT10);
            printf(" CRC32:TEXT1:¥"%s¥" = 0x%08x(%u)¥n", TEXT1, TEXT1_CRC, TEXT1_CRC);
            printf(" CRC32:TEXT2:¥"%s¥" = 0x%08x(%u)¥n", TEXT2, TEXT2_CRC, TEXT2_CRC);
            printf(" CRC32:TEXT3:¥"%s¥" = 0x%08x(%u)¥n", TEXT3, TEXT3_CRC, TEXT3_CRC);
            printf(" CRC32:TEXT4:¥"%s¥" = 0x%08x(%u)¥n", TEXT4, TEXT4_CRC, TEXT4_CRC);
            printf(" CRC32:TEXT5:¥"%s¥" = 0x%08x(%u)¥n", TEXT5, TEXT5_CRC, TEXT5_CRC);
            printf(" CRC32:TEXT6:¥"%s¥" = 0x%08x(%u)¥n", TEXT6, TEXT6_CRC, TEXT6_CRC);
            printf(" CRC32:TEXT7:¥"%s¥" = 0x%08x(%u)¥n", TEXT7, TEXT7_CRC, TEXT7_CRC);
            printf(" CRC32:TEXT8:¥"%s¥" = 0x%08x(%u)¥n", TEXT8, TEXT8_CRC, TEXT8_CRC);
            printf(" CRC32:TEXT9:¥"%s¥" = 0x%08x(%u)¥n", TEXT9, TEXT9_CRC, TEXT9_CRC);
            printf(" CRC32:TEXT10:¥"%s¥" = 0x%08x(%u)¥n", TEXT10, TEXT10_CRC, TEXT10_CRC);
```

```
	#endif//ENABLE_MORE_TEST
	}
#endif//ENABLE_RUNTIME_TEST
#ifdef ENABLE_PERFORMANCE_TEST
	{
		printf("¥n");
		const auto begin_time = std::chrono::system_clock::now();
		const int repeat= 100000000;
		crc32_t crc = 0;
		static const char* str = "1234567890";
		for (int loop = 0; loop < repeat; ++loop)
		{
			extern crc32_t test_performance(const char* str, const int dummy);
			crc = test_performance(str, loop);
		}
		const auto end_time = std::chrono::system_clock::now();
		const auto duration_time = end_time - begin_time;
		const float elapsed_time = static_cast<float>(static_cast<double>(
				std::chrono::duration_cast<std::chrono::microseconds>(duration_time).count()) / 1000000.);
		printf("perfomance test(%d times calculate): str=¥"%s¥",
			crc=0x%08x, elapsed-time=%.06f sec¥n", repeat, str, crc, elapsed_time);
	}
#endif//ENABLE_PERFORMANCE_TEST
}

#ifdef ENABLE_MAIN
//テスト
int main(const int arcg, const char* argv[])
{
	test_constexpr();

	return EXIT_SUCCESS;
}
#endif
```

【constexpr_performance.cpp】

```
#include "constexpr_test.h"

#ifdef ENABLE_PERFORMANCE_TEST
//※コンパイラの最適化の影響を受けずに比較し易いように、
//  ダミー引数を受け取り、テスト専用関数として構成
crc32_t test_performance(const char* str, const int dummy)
{
	return calcCRC32(str);//通常関数でランタイム時に計算
}
#endif//ENABLE_PERFORMANCE_TEST
```

↓（実行結果）

```
constexpr による CRC 計算結果=0xf3dbd4fe

ユーザー定義リテラルによる CRC 計算結果=0xf3dbd4fe

ランタイム関数による CRC 計算結果=0xf3dbd4fe
```

↓（パフォーマンス比較）※Visual C++ 2013【Release ビルド】

```
↓SSE 命令使用時
perfomance test(100000000 times calculate): str="1234567890", crc=0xf3dbd4fe, elapsed-time=0.742044 sec

↓SSE 命令未使用時（生成多項式計算時）
perfomance test(100000000 times calculate): str="1234567890", crc=0xf3dbd4fe, elapsed-time=6.583373 sec

↓SSE 命令未使用時（事前計算済み多項式テーブル使用時）
perfomance test(100000000 times calculate): str="1234567890", crc=0xf3dbd4fe, elapsed-time=2.287128 sec
```

▼ 活用例②：rank()関 ／ extentof() 関数

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

この関数は、配列の次元数と要素数を取得するために用いる。テンプレート関数を利用することで、配列変数の情報を正確に得ることができる。

【サンプル：基本形】

テンプレート関数によるサンプル：

```
//一次配列要素数を取得
template<typename T, std::size_t N1>
inline std::size_t extent1of(const T (&var)[N1])
{
     return N1;
}
//二次配列要素数を取得
template<typename T, std::size_t N1, std::size_t N2>
inline std::size_t extent2of(const T (&var)[N1][N2])
{
     return N2;
}
//三次配列要素数を取得
template<typename T, std::size_t N1, std::size_t N2, std::size_t N3>
inline std::size_t extent3of(const T (&var)[N1][N2][N3])
{
     return N3;
}
```

使用例：

```
int var1[1] = { 0 };
int var2[2][3] = { 0 };
int var3[4][5][6] = { 0 };
printf("var1[%d]¥n",          extent1of(var1));
printf("var2[%d][%d]¥n",      extent1of(var2), extent2of(var2));
printf("var3[%d][%d][%d]¥n", extent1of(var3), extent2of(var3), extent3of(var3));
```

↓実行結果

```
var1[1]
var2[2][3]
var3[4][5][6]
```

同様の処理を#define マクロで作成すると、下記のようになる。これらは全てテンプレート関数と同じ実行結果になる。

マクロによるサンプル：

```
//一次配列要素数を取得
#define extent1of(var) (sizeof(var) / sizeof(var[0]))
//二次配列要素数を取得
#define extent2of(var) (sizeof(var[0]) / sizeof(var[0][0]))
//三次配列要素数を取得
#define extent3of(var) (sizeof(var[0][0]) / sizeof(var[0][0][0]))
```

このテンプレート関数の仕組みを利用することで、マクロでは扱いきれないような、更に使い勝手の良い関数を作成することができる。

上記のサンプルを拡張したサンプルを示す。

【サンプル：拡張版】

テンプレート関数によるサンプル：

```
//非配列用
template<typename T>
inline std::size_t rankof(const T& data){ return 0; }//次元数
template<typename T>
inline std::size_t sizeofelemof(const T& data){ return sizeof(T); }//1 要素のサイズ
template<typename T>
inline std::size_t extentof(const T& data){ return 0; }//全要素数
template<typename T>
inline std::size_t extent1of(const T& data){ return 0; }//一次要素数
template<typename T>
inline std::size_t extent2of(const T& data){ return 0; }//二次要素数
template<typename T>
inline std::size_t extent3of(const T& data){ return 0; }//三次要素数

//一次配列用
template<typename T, std::size_t N1>
inline std::size_t rankof(const T(&data)[N1]){ return 1; }//次元数
template<typename T, std::size_t N1>
inline std::size_t sizeofelemof(const T(&data)[N1]){ return sizeof(T); }//1 要素のサイズ
template<typename T, std::size_t N1>
inline std::size_t extentof(const T(&data)[N1]){ return N1; }//全要素数
template<typename T, std::size_t N1>
inline std::size_t extent1of(const T(&data)[N1]){ return N1; }//一次要素数

//二次配列用
template<typename T, std::size_t N1, std::size_t N2>
inline std::size_t rankof(const T(&data)[N1][N2]){ return 2; }//次元数
template<typename T, std::size_t N1, std::size_t N2>
inline std::size_t sizeofelemof(const T(&data)[N1][N2]){ return sizeof(T); }//1 要素のサイズ
template<typename T, std::size_t N1, std::size_t N2>
inline std::size_t extentof(const T(&data)[N1][N2]){ return N1 * N2; }//全要素数
template<typename T, std::size_t N1, std::size_t N2>
inline std::size_t extent2of(const T(&data)[N1][N2]){ return N2; }//二次要素数

//三次配列用
template<typename T, std::size_t N1, std::size_t N2, std::size_t N3>
inline std::size_t rankof(const T(&data)[N1][N2][N3]){ return 3; }//次元数
template<typename T, std::size_t N1, std::size_t N2, std::size_t N3>
inline std::size_t sizeofelemof(const T(&data)[N1][N2][N3]){ return sizeof(T); }//1 要素のサイズ
template<typename T, std::size_t N1, std::size_t N2, std::size_t N3>
inline std::size_t extentof(const T(&data)[N1][N2][N3]){ return N1 * N2 * N3; }//全要素数
template<typename T, std::size_t N1, std::size_t N2, std::size_t N3>
inline std::size_t extent3of(const T(&data)[N1][N2][N3]){ return N3; }//三次要素数
```

使用例：

```
int var0 = 0;
int var1[1] = { 0 };
int var2[2][3] = { 0 };
int var3[4][5][6] = { 0 };
printf("var0[%d][%d][%d](Rank=%d, Total=%d, ElemtSize=%d)¥n",
                  extent1of(var0), extent2of(var0), extent3of(var0), rankof(var0), extentof(var0), sizeofelemof(var0));
printf("var1[%d][%d][%d](Rank=%d, Total=%d, ElemtSize=%d)¥n",
                  extent1of(var1), extent2of(var1), extent3of(var1), rankof(var1), extentof(var1), sizeofelemof(var1));
printf("var2[%d][%d][%d](Rank=%d, Total=%d, ElemtSize=%d)¥n",
                  extent1of(var2), extent2of(var2), extent3of(var2), rankof(var2), extentof(var2), sizeofelemof(var2));
printf("var3[%d][%d][%d](Rank=%d, Total=%d, ElemtSize=%d)¥n",
                  extent1of(var3), extent2of(var3), extent3of(var3), rankof(var3), extentof(var3), sizeofelemof(var3));
```

↓実行結果

```
var0[0][0][0](Rank=0, Total=0, ElemtSize=4)
var1[1][0][0](Rank=1, Total=1, ElemtSize=4)
var2[2][3][0](Rank=2, Total=6, ElemtSize=4)
var3[4][5][6](Rank=3, Total=120, ElemtSize=4)
```

上記の rankof()、extentof()、sizeofelemof()のような関数は、マクロでは作る事ができないので、テンプレート関数がとても重宝する。

なお、これらの関数名は、C++11 のライブラリの std::rank、std::extent に由来する。std::rank、std::extent は、型を指定して要素数を取得するための構造体。使用方法は下記のとおり。

std::rank、std::extent のサンプル：

```
typedef int arr0_t;
typedef int arr1_t[1];
typedef int arr2_t[2][3];
typedef int arr3_t[4][5][6];
printf("arr0_t[%d][%d][%d](Rank=%d)¥n", std::extent<arr0_t, 0>::value, std::extent<arr0_t, 1>::value,
                                        std::extent<arr0_t, 2>::value, std::rank<arr0_t>::value);
printf("arr1_t[%d][%d][%d](Rank=%d)¥n", std::extent<arr1_t, 0>::value, std::extent<arr1_t, 1>::value,
                                        std::extent<arr1_t, 2>::value, std::rank<arr1_t>::value);
printf("arr2_t[%d][%d][%d](Rank=%d)¥n", std::extent<arr2_t, 0>::value, std::extent<arr2_t, 1>::value,
                                        std::extent<arr2_t, 2>::value, std::rank<arr2_t>::value);
printf("arr3_t[%d][%d][%d](Rank=%d)¥n", std::extent<arr3_t, 0>::value, std::extent<arr3_t, 1>::value,
                                        std::extent<arr3_t, 2>::value, std::rank<arr3_t>::value);
```

↓実行結果

```
arr0_t[0][0][0](Rank=0)
arr1_t[1][0][0](Rank=1)
arr2_t[2][3][0](Rank=2)
arr3_t[4][5][6](Rank=3)
```

std::extent は次数をテンプレート引数に指定して使用する。次数 0 が一次配列、次数 1 が二次配列を指す。また、extentof()関数は、次数を指定しない場合に配列の合計要素数を返すものとしたが、std::extent の場合は次数 0 として扱われる。

▼ 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複雑な計算を、テンプレートの特殊化を活用して実現する方法を説明する。

サンプルとして、「べき乗」の計算結果コンパイル時に定数化する。まずはサンプルプログラムを示す。

【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（テンプレートクラス版）※整数の正の数専用
template<typename T, T N, int E>
struct static_pow{
    static const T value = N * static_pow<T, N, E - 1>::value;
```

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

使用例：

```
const int n0 = static_pow<int, 2, 0>::value;
const int n1 = static_pow<int, 2, 1>::value;
const int n2 = static_pow<int, 2, 2>::value;
const int n3 = static_pow<int, 2, 3>::value;
const int n4 = static_pow<int, 2, 4>::value;
printf("static_pow<2, e> = {%d, %d, %d, %d, %d}¥n", n0, n1, n2, n3, n4);
```

↓実行結果

```
static_pow<2, e> = {1, 2, 4, 8, 16}
```

↓※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 n0 = static_pow<int, 2, 0>::value;
     const int n1 = static_pow<int, 2, 1>::value;
     const int n2 = static_pow<int, 2, 2>::value;
     const int n3 = static_pow<int, 2, 3>::value;
     const int n4 = static_pow<int, 2, 4>::value;
     printf("static_pow<2, e> = {%d, %d, %d, %d, %d}¥n", n0, n1, n2, n3, n4);
00292230  push        10h     ←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       ←（同上）
00292237  push        4       ←（同上）
00292239  push        2       ←（同上）
0029223B  push        1       ←（同上）
0029223D  push        2B9E80h
00292242  call        printf (0297787h)
```

この手法を用いると、確実にコンパイル時に定数化することができる。ただし、再帰のネストが深すぎるとコンパイルエラーになるので注意。

コンパイラがどのように定数化するのかを、「static_pow<int,2,3>」が指定された場合を例に説明する。

まず、コンパイラがテンプレートクラスをインスタンス化（実体化）すると、「struct static_pow<int,2,3>」という構造体になる。その構造体内では「static_pow<T,N,E-1>」が使用されるているため、再帰的に「struct static_pow<int,2,2>」をイスタンス化する。更に再帰を繰り返し「struc static_pow<int,2,1>」→「struct static_pow<int,2,0>」とインスタンス化していく。

最後の「struct static_pow<int,2,0>」では、「テンプレートクラスの特殊化」が作用して「template<typename T, int N> struct static_pow<T, N, 0>{ static const T value = 1;}」の方に適合する。このテンプレートクラスは定数「1」を返すだけなので、ここで再帰が終了し、全ての構造体のインスタンス化が完了する。

続いて、インスタンス化されたそれぞれの構造体の定数 value に値を展開していく。

末端が「value = 1」で、その上のクラスでは「value = 2 * 1」、更にその上では「value = 2 * 2」→「value = 2 * 4」と展開されていき、最終的に「value = 8」が確定する。

このコンパイルの過程でインスタンス化された多数のクラスは、実行時に使用されるク

ラスではないため、コンパイルが完了すると共に消滅し、実行コードには含まれない。

注意点として、このクラスは整数に限定されたものであり、指数には正の数しか指定できない。

整数に限定される理由は、クラス／構造体の static メンバー定数には、整数型しか初期値を与えることができないためである。浮動小数点型の値は扱えない。ただし、整数であれば char 型や long long 型なども使用可能。

指数が正の数に限定されるのは、負の数を指定するといくら減算しても 0 にならず、特殊化が成立しないため、無限の再帰に陥りコンパイルエラーが発生するためである。

補足として、このサンプルでクラス（class）ではなく構造体（struct）を使用しているのは、単にデフォルトのスコープが public スコープだからという理由だけである。クラスを使う場合に必要な、明示的な「public:」宣言を省くことを目的としている。

同様の処理をテンプレート関数で作成することもできるが、その場合、確実な定数化は保証されず、引数に直地を指定してもランタイム時の関数呼び出しになる可能性がある。また、テンプレート関数では部分特殊化が使用できない点にも注意。以下、テンプレート関数によるべき乗のサンプルプログラムを示す。テンプレートクラスでは扱えなかった浮動小数点のべき乗、負の数の指数を扱う方法も合わせて示す。

【サンプル】

テンプレート関数によるべき乗算出のサンプル：

```
//べき乗（テンプレート関数版）※指数に負の数を指定するとコンパイルエラー
template<typename T, int E>
T power(const T n){ return n * power<T, E - 1>(n); }
//べき乗：再帰終点のための特殊化
//template<typename T>
//T power<T, 0>(const T n){ return static_cast<T>(1); }
//※【NG】テンプレート関数では部分特殊化を使用しようとするとコンパイルエラーとなる
//  そのため、下記のように各型で展開した特殊化関数を用意する必要がある。
template<>
char power<char, 0>(const char n){ return 1; }
template<>
unsigned char power<unsigned char, 0>(const unsigned char n){ return 1; }
template<>
short power<short, 0>(const short n){ return 1; }
template<>
unsigned short power<unsigned short, 0>(const unsigned short n){ return 1; }
template<>
int power<int, 0>(const int n){ return 1; }
template<>
unsigned int power<unsigned int, 0>(const unsigned int n){ return 1; }
template<>
long power<long, 0>(const long n){ return 1; }
template<>
unsigned long power<unsigned long, 0>(const unsigned long n){ return 1; }
template<>
long long power<long long, 0>(const long long n){ return 1; }
template<>
unsigned long long power<unsigned long long, 0>(const unsigned long long n){ return 1; }
template<>
```

```
float power<float, 0>(const float n){ return 1.f; }
template<>
double power<double, 0>(const double n){ return 1.; }
//負の数の指数用のべき乗（テンプレート関数版）※指数に正の数を指定するとコンパイルエラー
template<typename T, int E>
T minus_power(const T n){ return static_cast<T>(1) / power<T, -E>(n); }
```

使用例：

```
const int ni0 = power<int, 0>(2);
const int ni1 = power<int, 1>(2);
const int ni2 = power<int, 2>(2);
const int ni3 = power<int, 3>(2);
const int ni4 = power<int, 4>(2);
const float nf0 = power<float, 0>(2.f);
const float nf1 = power<float, 1>(2.f);
const float nf2 = power<float, 2>(2.f);
const float nf3 = power<float, 3>(2.f);
const float nf4 = power<float, 4>(2.f);
const float nfm0 = minus_power<float, 0>(2.f);
const float nfm1 = minus_power<float, -1>(2.f);
const float nfm2 = minus_power<float, -2>(2.f);
const float nfm3 = minus_power<float, -3>(2.f);
const float nfm4 = minus_power<float, -4>(2.f);
printf("power<int, e>(2) = {%d, %d, %d, %d, %d}¥n", ni0, ni1, ni2, ni3, ni4);
printf("power<float, e>(2.f) = {%.1f, %.1f, %.1f, %.1f, %.1f}¥n", nf0, nf1, nf2, nf3, nf4);
printf("minus_power<float, -e>(2.f) = {%.4f, %.4f, %.4f, %.4f, %.4f}¥n", nfm0, nfm1, nfm2, nfm3, nfm4);
```

↓実行結果

```
power<int, e>(2) = {1, 2, 4, 8, 16}
power<float, e>(2.f) = {1.0, 2.0, 4.0, 8.0, 16.0}
minus_power<float, -e>(2.f) = {1.0000, 0.5000, 0.2500, 0.1250, 0.0625}
```

↓※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 ni0 = power<int, 0>(2);
    const int ni1 = power<int, 1>(2);
    const int ni2 = power<int, 2>(2);
    const int ni3 = power<int, 3>(2);
    const int ni4 = power<int, 4>(2);
    const float nf0 = power<float, 0>(2.f);
    const float nf1 = power<float, 1>(2.f);
    const float nf2 = power<float, 2>(2.f);
    const float nf3 = power<float, 3>(2.f);
    const float nf4 = power<float, 4>(2.f);
    const float nfm0 = minus_power<float, 0>(2.f);
    const float nfm1 = minus_power<float, -1>(2.f);
    const float nfm2 = minus_power<float, -2>(2.f);
    const float nfm3 = minus_power<float, -3>(2.f);
    const float nfm4 = minus_power<float, -4>(2.f);
    printf("power<int, e>(2) = {%d, %d, %d, %d, %d}¥n", ni0, ni1, ni2, ni3, ni4);
0036313A 6A 10          push    10h   ←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）
0036313C 6A 08          push    8     ←（同上）
0036313E 6A 04          push    4     ←（同上）
00363140 6A 02          push    2     ←（同上）
00363142 6A 01          push    1     ←（同上）
00363144 68 2C AF 38 00 push    38AF2Ch
00363149 E8 A9 56 00 00 call    printf (03687F7h)
    printf("power<float, e>(2.f) = {%.1f, %.1f, %.1f, %.1f, %.1f}¥n", nf0, nf1, nf2, nf3, nf4);
0036314E 0F 28 05 40 B1 38 00 movaps  xmm0,xmmword ptr ds:[38B140h]   ※分かりにくいが、定数化された浮動小数点
00363155 83 C4 08       add     esp,8                                   データをメモリから読み出してスタックに
00363158 0F 11 44 24 18 movups  xmmword ptr [esp+18h],xmm0              積んで printf 文を呼び出している。
0036315D 0F 28 05 30 B1 38 00 movaps  xmm0,xmmword ptr ds:[38B130h]    power()関数は呼び出していない。
00363164 0F 11 44 24 08 movups  xmmword ptr [esp+8],xmm0
00363169 F2 0F 10 05 98 B0 38 00 movsd    xmm0,mmword ptr ds:[38B098h]
00363171 F2 0F 11 04 24 movsd   mmword ptr [esp],xmm0
00363176 68 58 AF 38 00 push    38AF58h
```

```
0036317B E8 77 56 00 00       call        printf (03687F7h)
     printf("minus_power<float, -e>(2.f) = {%.4f, %.4f, %.4f, %.4f, %.4f}¥n", nfm0, nfm1, nfm2, nfm3, nfm4);
00363180 0F 28 05 00 B1 38 00 movaps      xmm0,xmmword ptr ds:[38B100h]      ※分かりにくいが、定数化された浮動小数点
00363187 83 C4 04             add         esp,4                                データをメモリから読み出してスタックに
0036318A 0F 11 44 24 18       movups      xmmword ptr [esp+18h],xmm0           積んで printf 文を呼び出している。
0036318F 0F 28 05 10 B1 38 00 movaps      xmm0,xmmword ptr ds:[38B110h]      minus_power()関数は呼び出していない。
00363196 0F 11 44 24 08       movups      xmmword ptr [esp+8],xmm0
0036319B F2 0F 10 05 98 B0 38 00 movsd       xmm0,mmword ptr ds:[38B098h]
003631A3 F2 0F 11 04 24       movsd       mmword ptr [esp],xmm0
003631A8 68 90 AF 38 00       push        38AF90h
003631AD E8 45 56 00 00       call        printf (03687F7h)
003631B2 83 C4 2C             add         esp,2Ch
```

▼ 活用例④：より高度なメタプログラミング

より高度なメタプログラミングの手法として、特殊化だけでは判定できない条件判定を伴うメタプログラミングの方法を説明する。

サンプルとして、素数計算を行う。

素数計算をコンパイル時に行うメリットはある。例えば、開番地法ハッシュテーブルの配列要素数は素数でなければならない。あらかじめ与えられた静的な配列要数に対して、コンパイル時にそれが素数か判定してエラーを出すか、もしくは、その値より大きい最初の素数に自動調整できると便利である。このように、プログラムの安全性を高める用途にもメタプログラミングは活用できる。

なお、実装には C++11 のクラスライブラリを使用するが、必須ではない。同様のクラスを作成すれば済むので、その内容も含めて説明する。

まずはランタイム版（通常関数版）のサンプルプログラムを示し、続いてそれと同等のメタプログラミング版を示す。

【ランタイプ版サンプル】

ラインタイム版：素数判定

```
//---------------------------------------
//【ランタイム版】素数判定
bool isPrime(const std::size_t n)
{
	if (n < 2)//2 未満は素数ではない
		return false;
	else if (n == 2)//2 は素数
		return true;
	else if ((n & 1) == 0)//偶数は素数ではない
		return false;
	for (std::size_t div = 3; div <= n / div; div += 2)//div = 3～n/div の範囲で割り切れる値があるか判定
	{
		if (n % div == 0)//割り切れる値がみつかったら素数ではない
			return false;
	}
	return true;//素数と判定
}
```

ラインタイム版：指定の値以下の素数を算出

```
//---------------------------------------
//【ランタイム版】指定の値より小さい最初の素数を算出
std::size_t makePrimeLT(const std::size_t n)
{
	if (n <= 2)//2 より小さい素数はない
		return 0;
	else if (n == 3)//3 の次に小さい素数は 2
		return 2;
	for (std::size_t nn = n - ((n & 1) == 0 ? 1 : 2);; nn -= 2)//素数がみつかるまでループ ※偶数は判定しない
	{
		if (isPrime(nn))//素数判定
			return nn;
	}
	return 0;//dummy
}
//---------------------------------------
//【ランタイム版】指定の値と同じか、それより小さい最初の素数を算出
std::size_t makePrimeLE(const std::size_t n)
{
	return isPrime(n) ? n : makePrimeLT(n);
}
```

ラインタイム版：指定の値以上の素数を算出

```
//---------------------------------------
//【ランタイム版】指定の値より大きい最初の素数を算出
std::size_t makePrimeGT(const std::size_t n)
{
	if (n < 2)//2 未満の値より大きい最初の素数は 2
		return 2;
	for (std::size_t nn = n + ((n & 1) == 0 ? 1 : 2);; nn += 2)//素数がみつかるまでループ ※偶数は判定しない
	{
		if (isPrime(nn))//素数判定
			return nn;
	}
	return 0;//dummy
}
//---------------------------------------
//【ランタイム版】指定の値と同じか、それより大きい最初の素数を算出
std::size_t makePrimeGE(const std::size_t n)
{
	return isPrime(n) ? n : makePrimeGT(n);
}
```

【メタプログラミング版サンプル】

メタプログラミング版：インクルード

```
#include <type_traits>//C++11 std::conditional, std:integral_constant 用
```

メタプログラミング版：素数判定

```
//---------------------------------------
//【メタプログラミング版】静的素数判定
//※偶数の判定を避けるために階層化する
//静的素数判定用の再帰クラス（直接使用しない）
template <std::size_t N, std::size_t DIV>
struct _isStaticPrime{
	typedef
		typename std::conditional<
			(DIV > N / DIV),//DIV = ～N/DIV の範囲で割り切れる値があるか判定
			std::integral_constant<bool, true>,//範囲を超えたので素数と判定
			typename std::conditional<
				(N % DIV == 0),//割り切れる値か判定
				std::integral_constant<bool, false>,//割り切れたので素数ではない
				_isStaticPrime<N, DIV + 2>//再帰で次の値が割り切れるか探索 ※偶数は判定しない
```

```
                >::type
            >::type
            type;
    static const bool value = type::value;
};
//静的素数判定クラス
template <std::size_t N>
struct isStaticPrime{
    typedef
        typename std::conditional<
            (N & 1) == 0,//偶数判定
            std::integral_constant<bool, false>,//偶数は素数ではない
            typename _isStaticPrime<N, 3>::type//素数判定ループ（再帰処理）呼び出し
        >::type
        type;
    static const bool value = type::value;
};
//特殊化：0 は素数ではない
template <>
struct isStaticPrime<0>{
    static const bool value = false;
};
//特殊化：1 は素数ではない
template <>
struct isStaticPrime<1>{
    static const bool value = false;
};
//特殊化：2 は素数
template <>
struct isStaticPrime<2>{
    static const bool value = true;
};
```

メタプログラミング版：指定の値以下の素数を算出

```
//----------------------------------------
//【メタプログラミング版】指定の値より小さい最初の素数を静的に算出
//※偶数の判定を避けるために階層化する
//静的素数算出用の再帰クラス（直接使用しない）
template<std::size_t N>
struct _makeStaticPrimeLT{
    typedef
        typename std::conditional<
            isStaticPrime<N>::value,//素数判定
            std::integral_constant<std::size_t, N>,//素数が見つかった
            _makeStaticPrimeLT<N - 2>//再帰で次に小さい値を探索 ※偶数は判定しない
        >::type
        type;
    static const std::size_t value = type::value;
};
//静的素数算出クラス
template<std::size_t N>
struct makeStaticPrimeLT{
    typedef
        typename std::conditional<
            (N & 1) == 0,//素数判定ループの初期値を奇数にするための判定
            _makeStaticPrimeLT<N - 1>,//素数判定ループ（再帰処理）呼び出し
            _makeStaticPrimeLT<N - 2>//素数判定ループ（再帰処理）呼び出し
        >::type
        type;
    static const std::size_t value = type::value;
};
//特殊化：0 より小さい素数はなし
template<>
struct makeStaticPrimeLT<0>{
    static const std::size_t value = 0;
```

```
};
//特殊化：1 より小さい素数はなし
template<>
struct makeStaticPrimeLT<1>{
	static const std::size_t value = 0;
};
//特殊化：2 より小さい素数はなし
template<>
struct makeStaticPrimeLT<2>{
	static const std::size_t value = 0;
};
//特殊化：3 より小さい素数は 2
template<>
struct makeStaticPrimeLT<3>{
	static const std::size_t value = 2;
};
//----------------------------------------
//【メタプログラミング版】指定の値と同じか、それより小さい最初の素数を静的に算出
//静的素数算出クラス
template<std::size_t N>
struct makeStaticPrimeLE{
	typedef
		typename std::conditional<
			isStaticPrime<N>::value,//指定の値が素数か？
			std::integral_constant<std::size_t, N>,//素数が見つかった
			makeStaticPrimeLT<N>//次に小さい値を探索
		>::type
	type;
	static const std::size_t value = type::value;
};
```

メタプログラミング版：指定の値以上の素数を算出

```
//----------------------------------------
//【メタプログラミング版】指定の値より大きい最初の素数を静的に算出
//※偶数の判定を避けるために階層化する
//静的素数算出用の再帰クラス（直接使用しない）
template<std::size_t N>
struct _makeStaticPrimeGT{
	typedef
		typename std::conditional<
			isStaticPrime<N>::value,//素数判定
			std::integral_constant<std::size_t, N>,//素数が見つかった
			_makeStaticPrimeGT<N + 2>//再帰で次に大きい値を探索 ※偶数は判定しない
		>::type
		type;
	static const std::size_t value = type::value;
};
//静的素数算出クラス
template<std::size_t N>
struct makeStaticPrimeGT{
	typedef
		typename std::conditional<
			(N & 1) == 0,//素数判定ループの初期値を奇数にするための判定
			_makeStaticPrimeGT<N + 1>,//素数判定ループ（再帰処理）呼び出し
			_makeStaticPrimeGT<N + 2>//素数判定ループ（再帰処理）呼び出し
		>::type
		type;
	static const std::size_t value = type::value;
};
//特殊化：0 より大きい素数は 2
template<>
struct makeStaticPrimeGT<0>{
	static const std::size_t value = 2;
};
//特殊化：1 より大きい素数は 2
```

```
template<>
struct makeStaticPrimeGT<1>{
	static const std::size_t value = 2;
};
//----------------------------------------
//【メタプログラミング版】指定の値と同じか、それより大きい最初の素数を静的に算出
//静的素数算出クラス
template<std::size_t N>
struct makeStaticPrimeGE{
	typedef
		typename std::conditional<
			isStaticPrime<N>::value,//指定の値が素数か？
			std::integral_constant<std::size_t, N>,//素数が見つかった
			makeStaticPrimeGT<N>//次に小さい値を探索
		>::type
		type;
	static const std::size_t value = type::value;
};
```

【参考】C++11 ライブラリ std::conditional と std::integral_constant の内容

（同じ内容のクラスを定義すれば、C++11 を使用せずとも上記の素数計算が可能）

```
//----------------------------------------
//std::conditional
//TEST の結果に応じて型 type を定義するクラス（デフォルトは型 T を採用）
template<bool TEST, class T, class F>
struct conditional
{
	typedef T type;//型
};
//特殊化：TEST の結果が false なら型 F を採用
template<class T, class F>
struct conditional<false, T, F>
{
	typedef F type;//型
};
//----------------------------------------
//std::integral_constant
//value メンバーに定数を設定し、型 type（自身の型）を定義するクラス
template<class T, T V>
struct integral_constant
{
	static const T value = V;//定数
	typedef integral_constant type;//型（自身の型）
	//以下、必要に応じて実装（必須ではない）
	typedef T value_type;//値の型
	constexpr operator value_type() const { return value; }//キャストオペレータ
	constexpr value_type operator()() const { return value; }//関数呼び出し演算子
};

//※様々なテンプレートメタプログラミングを汎用化するために、
//  特定の名前で定数（value）と型（type）を定義する。
//※同様に使えるテンプレートクラスは多数あり、
//  std::enable_if、std::is_same などもよく用いられる。
```

【テスト用プログラム】

ランタイム版テスト用プログラム：

```
//----------------------------------------
//【ランタイム版】素数判定／算出テスト
//素数計算結果表示関数（再帰関数）
void printPrime(const std::size_t min, const std::size_t max)
{
	if (max > min)
		printPrime(min, max - 1);
```

```
	printf("%6d is %s [prev=%6d(%6d), next=%6d(%6d)]¥n", max, isPrime(max) ? "PRIME.    " : "NOT prime.",
                                        makePrimeLT(max), makePrimeLE(max), makePrimeGT(max), makePrimeGE(max));
}
```

メタプログラミング版テスト用プログラム：

```
//---------------------------------------
//【メタプログラミング版】静的素数判定／算出テスト
//静的素数計算結果表示クラス用関数（直接使用しない）
template<std::size_t N>
void _printPrimeCommon()
{
	printf("%6d is %s [prev=%6d(%6d), next=%6d(%6d)]¥n", N, isStaticPrime<N>::value ? "PRIME.    " : "NOT prime.",
		makeStaticPrimeLT<N>::value, makeStaticPrimeLE<N>::value, makeStaticPrimeGT<N>::value,
                                                                                makeStaticPrimeGE<N>::value);
}
//静的素数計算結果表示クラス
template<std::size_t MIN, std::size_t MAX>
struct printStaticPrime{
	void operator()()
	{
		printStaticPrime<MIN, MAX - 1>()();
		_printPrimeCommon<MAX>();
	}
};
//特殊化：最小値の処理
template<std::size_t MIN>
struct printStaticPrime<MIN, MIN>{
	void operator()()
	{
		_printPrimeCommon<MIN>();
	}
};
```

テスト：

```
//--------------------
//素数計算のテスト
static const std::size_t MIN = 0;
static const std::size_t MAX = 10;

//素数計算結果表示
printf("----- Check and Make Prime for Runtime -----¥n");
printPrime(MIN, MAX);

//静的素数計算結果表示
printf("----- Check and Make Prime for Meta-Programming -----¥n");
printStaticPrime<MIN, MAX>()();
```

↓実行結果

```
----- Check and Make Prime for Runtime -----
     0 is NOT prime. [prev=     0(     0), next=     2(     2)]
     1 is NOT prime. [prev=     0(     0), next=     2(     2)]
     2 is PRIME.     [prev=     0(     2), next=     3(     2)]
     3 is PRIME.     [prev=     2(     3), next=     5(     3)]
     4 is NOT prime. [prev=     3(     3), next=     5(     5)]
     5 is PRIME.     [prev=     3(     5), next=     7(     5)]
     6 is NOT prime. [prev=     5(     5), next=     7(     7)]
     7 is PRIME.     [prev=     5(     7), next=    11(     7)]
     8 is NOT prime. [prev=     7(     7), next=    11(    11)]
     9 is NOT prime. [prev=     7(     7), next=    11(    11)]
    10 is NOT prime. [prev=     7(     7), next=    11(    11)]
----- Check and Make Prime for Meta-Programming -----
     0 is NOT prime. [prev=     0(     0), next=     2(     2)]
     1 is NOT prime. [prev=     0(     0), next=     2(     2)]
     2 is PRIME.     [prev=     0(     2), next=     3(     2)]
     3 is PRIME.     [prev=     2(     3), next=     5(     3)]
```

```
     4 is NOT prime. [prev=     3(     3), next=     5(     5)]
     5 is PRIME.     [prev=     3(     5), next=     7(     5)]
     6 is NOT prime. [prev=     5(     5), next=     7(     7)]
     7 is PRIME.     [prev=     5(     7), next=    11(     7)]
     8 is NOT prime. [prev=     7(     7), next=    11(    11)]
     9 is NOT prime. [prev=     7(     7), next=    11(    11)]
    10 is NOT prime. [prev=     7(     7), next=    11(    11)]
```

サンプルプログラムを見てわかるとおり、テンプレートクラスの条件判定はかなりややこしい。これは、条件判定によって値を得る通常のプログラミング手法とは異なり、条件判定によって適切な「型」を選択・生成する手法のためである。それは最終的に一つの型に収束し、そこから「値」を取り出すことで定数を得る。

メタプログラミングの特徴の一つは、変数が使えないことにある。通常のループ処理は、変数の値を書き換えながらループの終了を判定するが、それができないため、再帰的に型を生成していき、特殊化もしくは条件判定により、再帰のない型を返すことで収束する。このようなプログラミングの手法は、関数型言語と同様のものである。

なお、テンプレートクラス内に定義された型（typedef）にアクセスする際は、typename キーワードを明示する必要がある点に注意。上記のサンプルでも、「typename std::conditional<…>::type」のように使用している。

実際の挙動を確認したところでは、VC++では typename の指定を省略してもコンパイルが通ることがあったが、GCC では許されなかった。

また、メタプログラミングの注意点として、テンプレートの再帰深度の限界を超えない範囲でしか処理できないという点がある。

C++11 が推奨する深度限界は 1024 とのことだが、実測すると、Visual C++ 2013 では 499、GCC(4.8.2)では 900 であった。

素数計算のサンプルプログラムの場合、Visual C++ 2013 による実測では、isStaticPrime<126477>, makeStaticPrimeGT<952788> が限界で、次の素数 isStaticPrime<1262479>,makeStaticPrimeGT<952789>はコンパイルエラーとなった。

テンプレート深度限界の実測例：

```
template<int N> struct recursive{    static const int value = recursive<N - 1>::value; };//再帰テンプレートクラス
template<>      struct recursive<0>{ static const int value = 1; };//特殊化
static const int _n = recursive<499>::value;//VC++2013 では限界 OK ※次の recursive<500> はコンパイルエラー
static const int _n = recursive<900>::value;//GCC4.8.2 では限界 OK ※次の recursive<901> はコンパイルエラー
```

▼ 活用例⑤：STATIC_ASSERT（静的アサーション）

メタプログラミングは、前述の素数計算でも説明したとおり、プログラムの安全性を高める目的でも活用できる。

プログラムがその動作要件を満たしていることを判定するために、assert() を用いることが多い。この判定を実行時ではなく、コンパイル時に行うことで、プログラムの安全性を

高める。

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、静的アサーションの実装方法を幾つか説明する。

【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構造化手法

「高階関数」とは、「関数を引数や戻り値として扱う関数」のことである。

C++では、高階関数の実現に「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。

高階関数にテンプレートは必須ではないが、テンプレートが用いられることは多い。例えば、STL のソートアルゴリズム `std::sort()` は、並び順序判定用の関数を受け渡すことができるテンプレート関数である。（余談だが、高階関数に受け渡す「boot 値を返す関数」のことを「プレディケート」と呼ぶ）

本節では、テンプレートによる高階関数の使用方法を示す。

高階関数の実用性を示すために、具体的な処理要件をサンプルにする。高階関数が必須な処理ではないが、それを使用することでメンテナンスし易くなるという様子を段階的に説明する。

● 【サンプル】処理要件

- 一時的な状況分析のための処理を作成する。（結果がわかったらすぐに不要になる）
- 分析処理は、int 型の配列を二つ受け取る。
- 配列の値をチェックし、10～100 の範囲に丸めて返す。
- 丸めが発生した場合、丸め前と後の値をログ出力する。
- 処理の最後に、配列の合計値、平均値をログ出力する。

● 処理要件の補足

サンプルのためのやや強引な要件ではあるが、原因が判明していない問題を追及する場合など、あの手この手で疑わしい情報を視覚化することはよくあることである。

この時「ちょっと確認したいだけだから」、「今だけ」、「すぐに消すから」と考え、単純にコピペも多用して一気に作ってしまいがちだが、いざやり始めると、意外と長々とその処理の改訂を繰り返すことも少なくない。「丸めの範囲を変えて再確認してみよう」、「ちゃんと分析したいから Excel にコピペしやすい書式に調整して再出力しよう」などといった要件が次々発生する。その際、最適化されていないコードの修正はけっこうな手間となる。

このような背景を踏まえて、最適化のクセをつけておくことで、作業効率の向上を図る。

● 処理の実行イメージ

処理要件の実行イメージを示す。

呼び出し側：

```
int data1[] = {1, 2, 3, 39, 200, 53, 8, 74, 12};//何らかの処理で収集したデータ①
int data2[] = {13, 6, 76, 43, 23, 125, 1 };//何らかの処理で収集したデータ②
#define lengthOfArray(arr) (sizeof(arr) / sizeof(arr[0]))//配列要素数を取得
//データ分析
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

まず、安易にコピペも使って一気に実装したコードを示す。

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

最初の最適化として、同じ内容の処理を共通化する。まずは単純に、共通関数化する。

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関数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、平均算出処理を共通化する。

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」を使用する。

「関数オブジェクト」は、関数呼び出し演算子「operator()」をオーバーロードしたク

ラス／構造体である。

　ループ処理は汎用高階関数としてテンプレート関数で定義し、関数オブジェクトを受け取って処理する。

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、だいぶ構造化された状態になる。処理効率、プログラムサイズの面でも特に劣化はない。

汎用ループ処理用テンプレート関数「for_each_array()」は、汎用的な高階関数であり、この処理要件に特化したものではない。その第 3 引数には、T 型の引数（もしくは T&型の引数）を受け取る関数を指定する。それにより、汎用処理の中から、特定の処理要件に特化した処理を実行することができる。

第 3 引数に受け渡す関数は、普通の関数のほか、関数オブジェクトやラムダ式も指定することができる。関数オブジェクトを使用するのは、それが関数スコープ内で定義できるためである。

● 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「std::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。

● 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「std::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 srd::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 srd::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。

なお、ここまで来ると、（本書の趣旨に反して）テンプレートは全く関係なくなる。

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：固定長配列の直接指定（例えば「1, 5, 13 のケースで処理したい」といった場合に便利）
    printf("data=");
    for (int  elem : {1, 5, 13})   // 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 型を使って簡略化してもよい
        {
            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");
    }
```

「範囲に基づく for ループ」は、固定長配列以外にも、「begin()」「end()」メソッドでイテレータを返すコンテナは全般的に使用できる。STL コンテナは全般的に利用可能。

■ 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

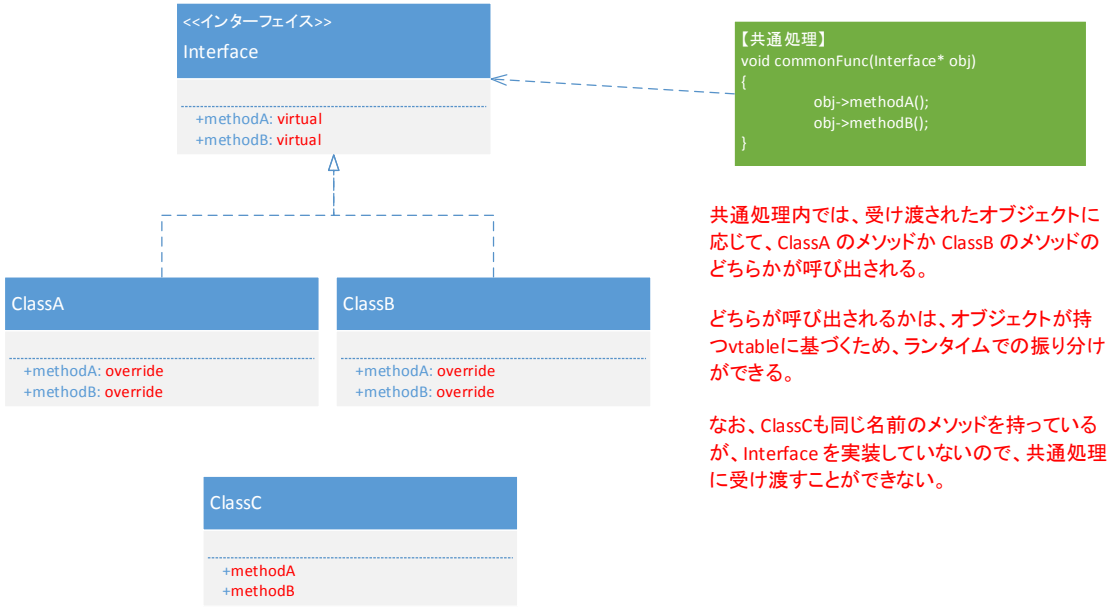
まず、仮想クラスを用いる方法が「動的な多態性」であるのに対して、テンプレートクラスで実現出来るのは「静的な多態性」であることを強調する。

「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。



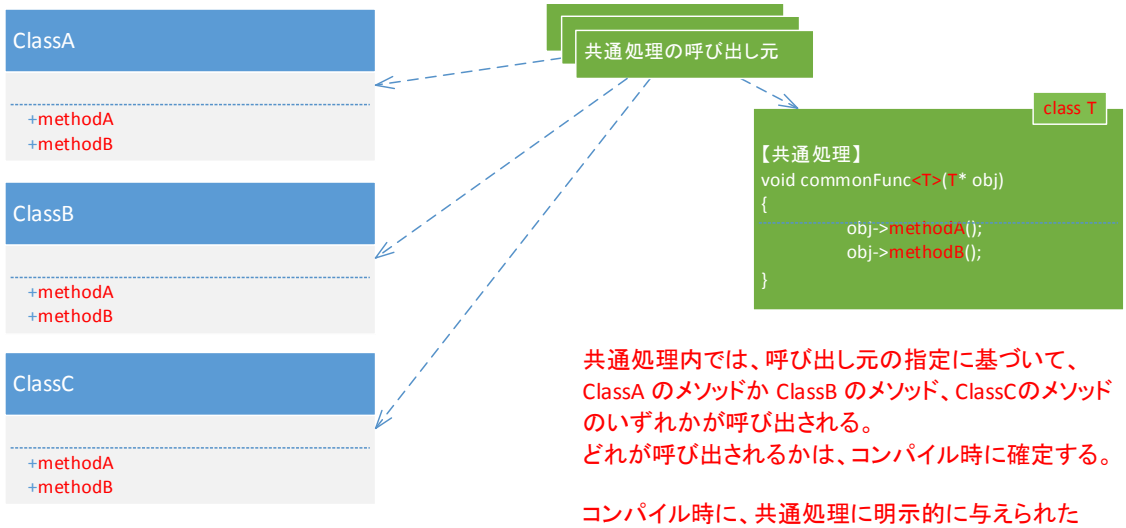
共通処理内では、受け渡されたオブジェクトに応じて、ClassA のメソッドか ClassB のメソッドのどちらかが呼び出される。

どちらが呼び出されるかは、オブジェクトが持つvtableに基づくため、ランタイムでの振り分けができる。

なお、ClassCも同じ名前のメソッドを持っているが、Interface を実装していないので、共通処理に受け渡すことができない。

▼ 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

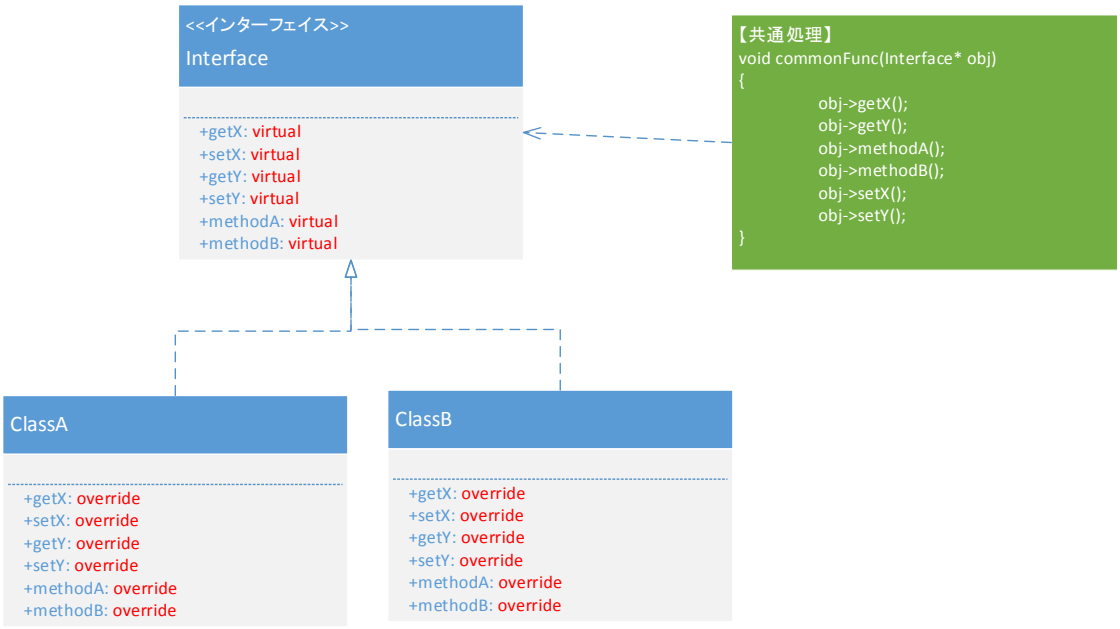
数にして各オブジェクトが update をオーバーライドしているのが良い。

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。

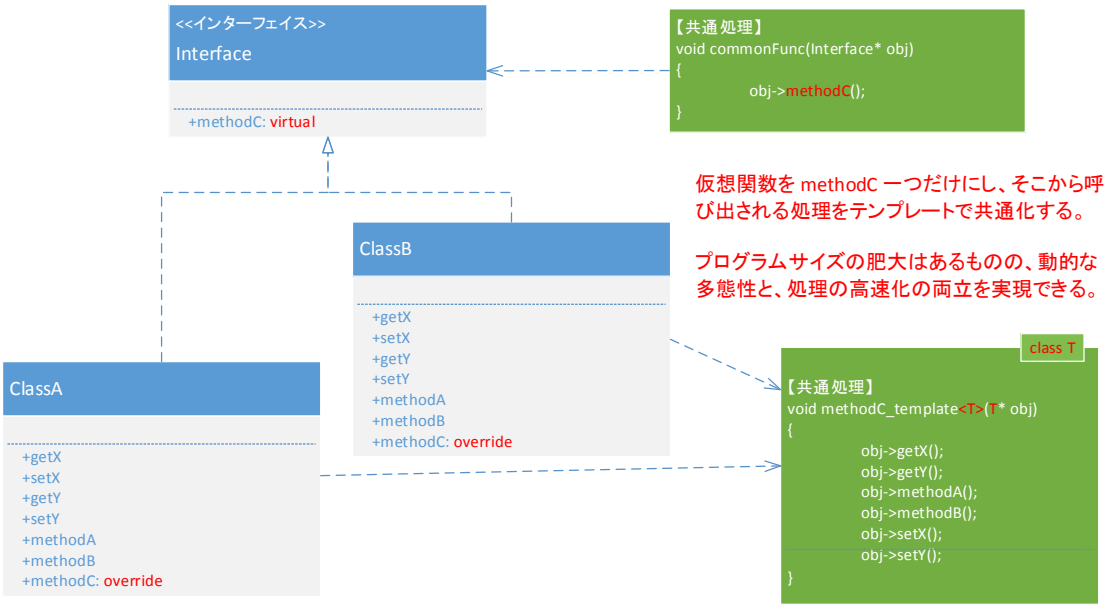
しかし、安易な対応はパフォーマンスの劣化を招くため、仮想関数は必要最小限のまとまった処理に対してのみ適用すべきである。その上で、処理の共通化にテンプレートを組み合わせると効果的である。

以下、具体的な改善例を示す。

【改善前】



【改善後】



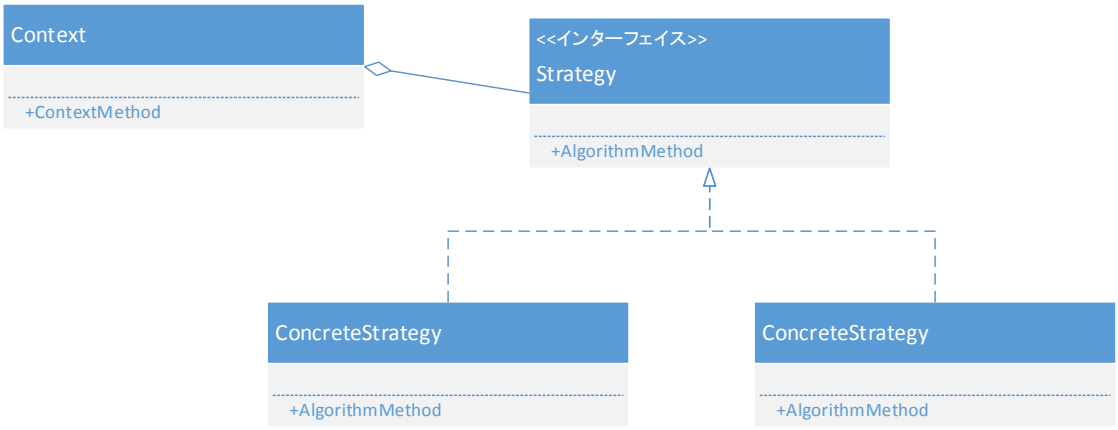
▼ ポリシー（ストラテジーパターン）

「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。

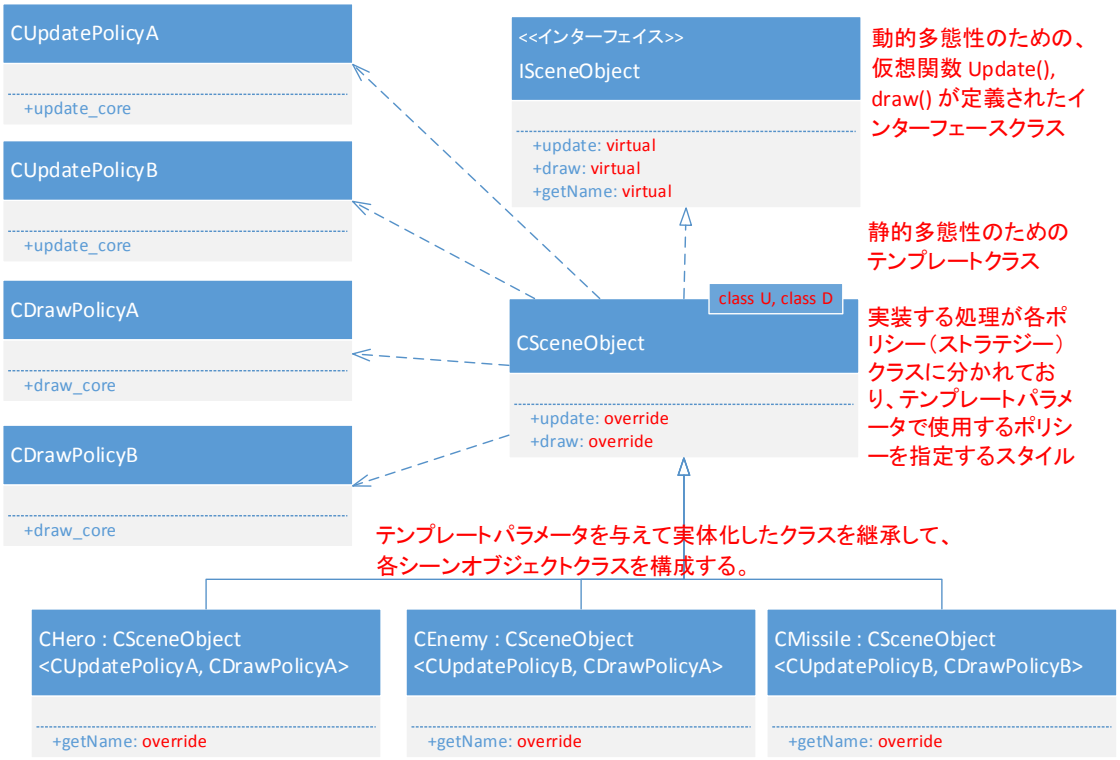
STL の各種コンテナが受け取る Allocator クラスは、ポリシー（ストラテジー）の実例の一つである。

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

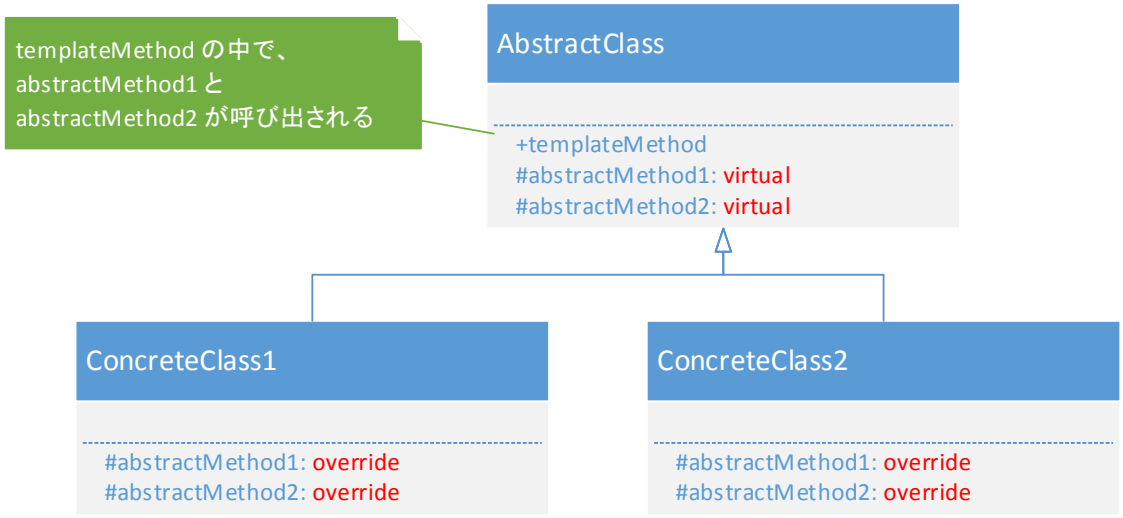
```
//親クラス（テンプレート）
template<class D>
class CBase
{
public:
	void foo() const
	{
		printf("CBase<D>::foo()¥n");
		static_cast<const D&>(*this).bar(); //子クラスのメンバーを呼び出し
	}
};

//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()
```

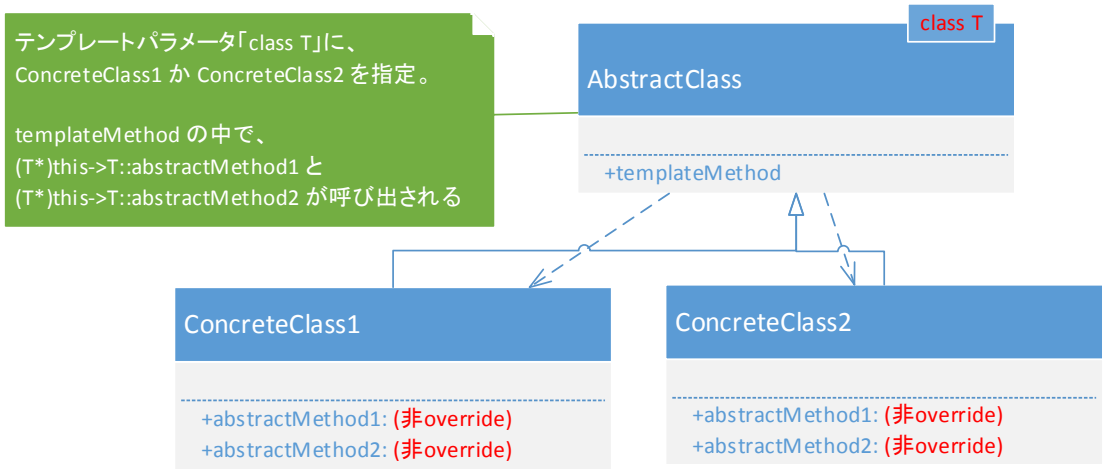
「CRTP」の主な用途は、「テンプレートメソッドパターン」の静的な実装である。「テンプレートメソッドパターン」とは、クラスのあるメソッド（「テンプレートメソッド」と呼ぶ）の中の一部のロジックを、同クラス内の抽象メソッド（仮想メソッド）に依存する、プログラミング手法である。つまり、メソッドの中の一部の処理を抽象化して、クラスを継承した子クラスの方でその一部のロジックを実装・置き換えする。

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。

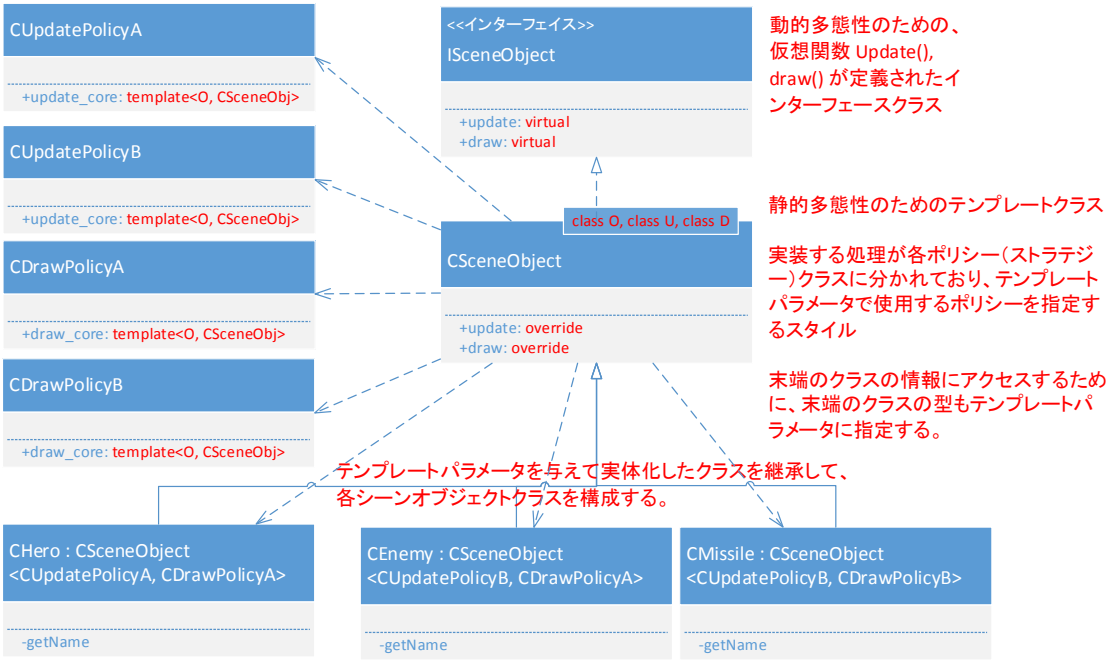
CRTP によるテンプレートメソッドパターンのクラス図：



二つのクラス図にも示しているとおり、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。vtable の独自実装である。

しかし、結論から言えば、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 Not 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、深く考えずに使える」処理を作成する事ができる。

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

v

い

お

か

き

こ

さ

す

せ

た

て

と

は

ほ

ま

め

ゆ

ら

効果的なテンプレートテクニック

以　上