

コーディング手法

－ コンパイルの効率化と処理把握のための厳密なコーディング －

2014 年 1 月 20 日 初版

板垣 衛

■ 改訂履歴

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

■ 目次

■ 概略	1
■ 目的	1
■ プログラミング Tips	1
▼ ヘッダーファイルのインクルードは必要最小限に	1
▼ プリコンパイル済みヘッダーを活用する	3
▼ 共通ライブラリをきちんと使用する／算術系処理は独自実装しない	6
▼ テンプレートクラスの意識的な分割と継承	6
▼ 「.inl」ファイルの活用と、テンプレートの明示的なインスタンス化	8
▼ コンストラクタの explicit 宣言はできるだけ付ける	9
▼ 暗黙的なコピーコンストラクタと代入演算子の禁止	10
▼ インクリメント・デクリメントは、特に理由がない限りは前置を使う	12
▼ 仮想クラス (virtual) を避ける	13

■ 概略

C++のコンパイルを効率化するためのちょっとしたプログラミング方法を解説。

■ 目的

本書は、大規模開発において、無造作なプログラミングで大勢のスタッフの作業効率に影響を及ぼすような問題が起きないようにすることを目的とする。

また、ごく簡単なコーディングルールにより、他のスタッフの処理を誤って使用するようなことを防ぐ。

■ プログラミング Tips

▼ ヘッダーファイルのインクルードは必要最小限に

面倒がって「何でもインクルード」ヘッダーファイルを作ってはいけない。

一見して一つのファイルしかインクルードしていないにもかかわらず、実際には連鎖的に大量のファイルを全.cpp ファイルがインクルードすることになる。

一つ一つの.cpp ファイルのコンパイル時に走査されるファイルが劇的に増えてコンパイル速度が低下することはもとより、ちょっとしたヘッダーファイルの修正で、本来無関係な多数の .cpp ファイルの再コンパイルが起こってしまい、作業効率の妨げとなる。

また、意図しない #define マクロや定数の競合が起きて面倒を起こしたりすることもある。

常に必要最小限のインクルードを心がける。

【NG】 common.h

```
#pragma once
#include <stdio.h>
#include <string.h>
#include "types.h"
#include "math.h"
#include "game.h"
#include "model.h"
#include "chara.h"
#include "map.h"
#include "effect.h"

#define XXX (10)
enum { YYY, ZZZ }
```

```
const i8 AAA = 20;
```

【NG】 chara.h

```
#pragma once
#include "common.h"
class CChara ...
```

【NG】 chara.cpp

```
#include "common.h"
class CChara ...
```

【NG】 other.cpp

```
#include "common.h"
CChara obj;
...
```

↓ (改善)

【OK】 common.h

```
#pragma once

//” common.h” のような共通ヘッダーの存在自体は否定しない。
//” types.h” のような「言語レベルの標準化」に近い設定を “common.h” にインクルードする。
//全てのファイルで確実に必須の要素だけをまとめる。
//<stdio.h>のような標準ライブラリのヘッダーは含めない。

#include "types.h"

//” math.h” のような算術系のライブラリヘッダーを含めるかどうかは判断が際どいところ。
//往々にして、含めたほうが効率的。

#include "math.h"

//また、” common.h” は、全てのファイルで確実にインクルードしてもらわなければならない困るレベルのものであるため、
//個別に #include するのではなく、コンパイラの設定で「強制インクルード」として設定しておく。
//プリコンパイル済みヘッダーにすることも有効。
```

【OK】 chara.h

```
#pragma once

//ヘッダーファイルでは、不要なインクルードは一切しない。
//可能な限り、ヘッダーファイルに #include 文は書かない。「ほぼ禁止」レベル。
//common.h は強制インクルードされるので明示的には記述しない。

//特定の .cpp ファイルでしか必要としない #define マクロや定数も書かない。
//そのようなものは、むしろ .cpp 内に記述するか、別途専用のヘッダーファイルを設ける。
//「マクロや定数は.hに書くもの」というような固定観念を持つてはいけない。
//全ての「クラス」「関数」「マクロ」「定数」「変数」は、常に必要最小限の「スコープ」となるように一人一人が心がける。

//他のヘッダーで定義されているクラスや構造体の記述がどうしても必要な場合、
//それが「ポインター変数の型」にしか使用されていないなら、インクルードはせずに、
//クラス名／構造体名の宣言だけで済ませる。
class CDependant;

class CChara ...
```

【OK】 chara.cpp

```
//面倒でも .cpp 内で必要なファイルを一っずつインクルードする。
//むしろ、プログラマーなら、自分が作成する .cpp ファイルが依存するファイルを
//明確に把握しておくべき。

#include "chara.h"
#include "model.h"
```

```

// #define マクロや定数も、一つの .cpp 内の処理でしか必要としないものであれば、
// .h に記述せずに、.cpp に記述する。

```

```

#define XXX (10)
enum { YYY, ZZZ }

Class CChara ...

```

【OK】 other.cpp

```

// 標準ライブラリのヘッダーも含めて、必要最小限のインクルードを個々の .cpp ファイル内で行う。

```

```

#include <stdio.h>
#include "chara.h"
#include "map.h"

CChara obj;
...

```

▼ プリコンパイル済みヘッダーを活用する

コンパイルの効率を上げるためにも、プリコンパイル済みヘッダーの使用は最適である。

以下、Visual C++ でプリコンパイル済みヘッダーを作成・使用し、さらに、明示的にインクルードしなくても、全ソースにそれを強制インクルードする方法を解説。

なお、Visual C++ のウィザードが自動的に作成するプリコンパイル済みヘッダーの名前は「stdafx.h」とあるが、この名前であることは必須ではない。以下の説明では「common.h」という名前のファイルを用いる。

【手順 1】 プリコンパイル済みヘッダーとなる共通ヘッダーファイル（全ソースコード必須かつ必要最小限の内容）を定義

例： common.h

```

#pragma once
#ifdef __COMMON_H_

// 全ソースコード共通ヘッダー
// ※基本的には直接定数やマクロを定義せず、
// 必要なインクルードファイルを集めたものとする

// コンパイラスイッチ
// デバッグ/リリースビルドに最適なオプションなどが定義
#include "compile_switch.h"

// プラットフォームスイッチ
// プラットフォームを識別するマクロや定数が定義
#include "platform.h"

// 型定義
#include "types.h"

// 基本算術関数
#include "basic_math.h"

#endif // __COMMON_H_

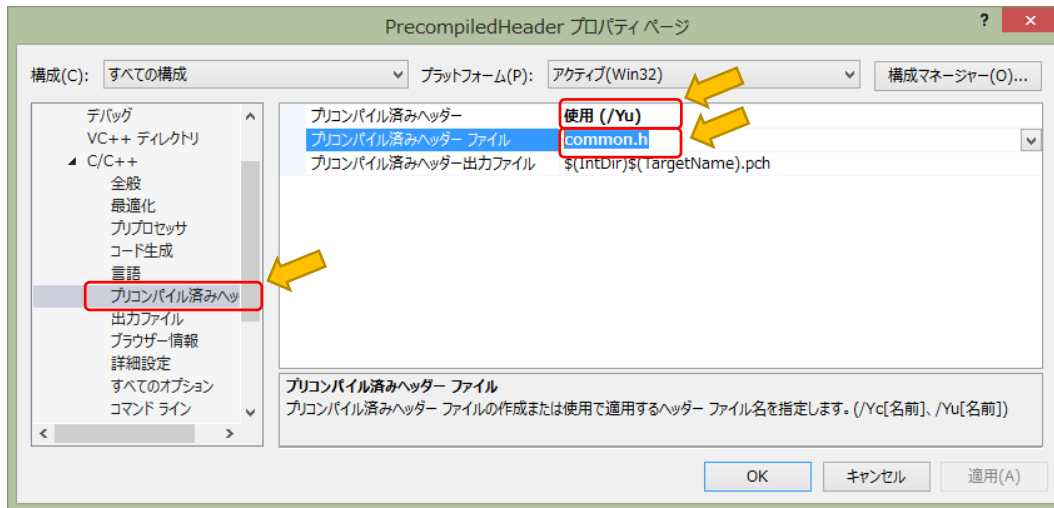
```

【手順 2】 プリコンパイル済みヘッダーファイルをインクルードするだけのソースファイルを用意

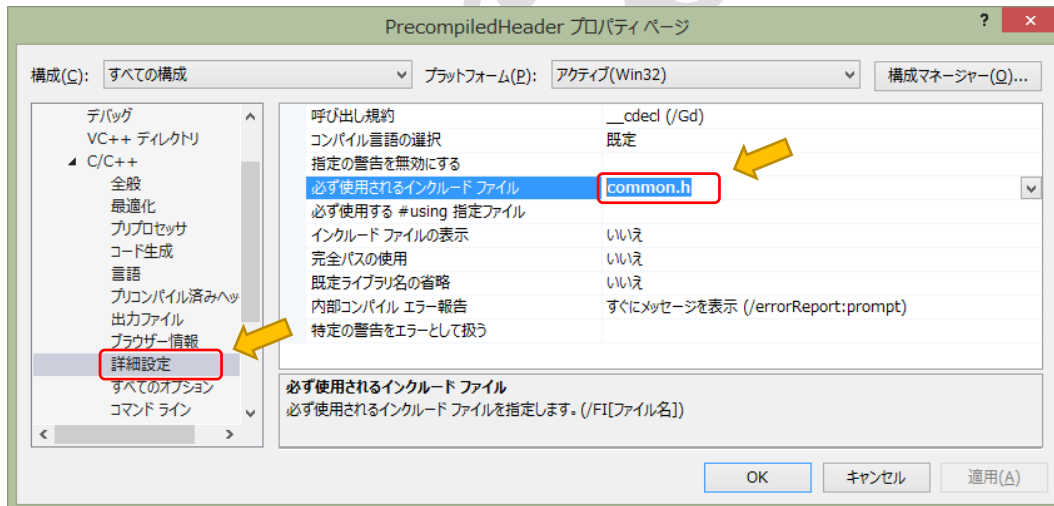
例： common.cpp

```
//プリコンパイル済みヘッダーファイル一つだけをインクルード
#include "common.h"
```

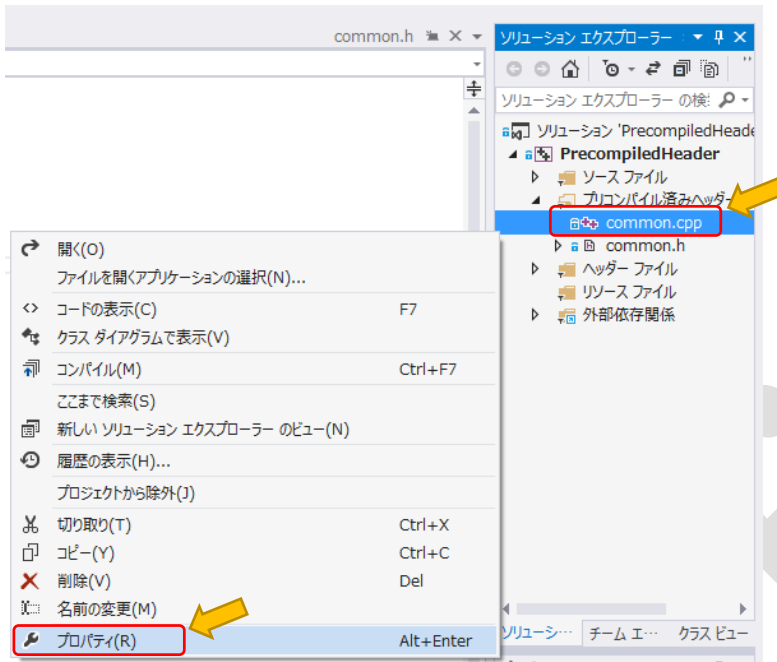
【手順 3】 コンパイラオプションで「プリコンパイル済みヘッダー」を指定



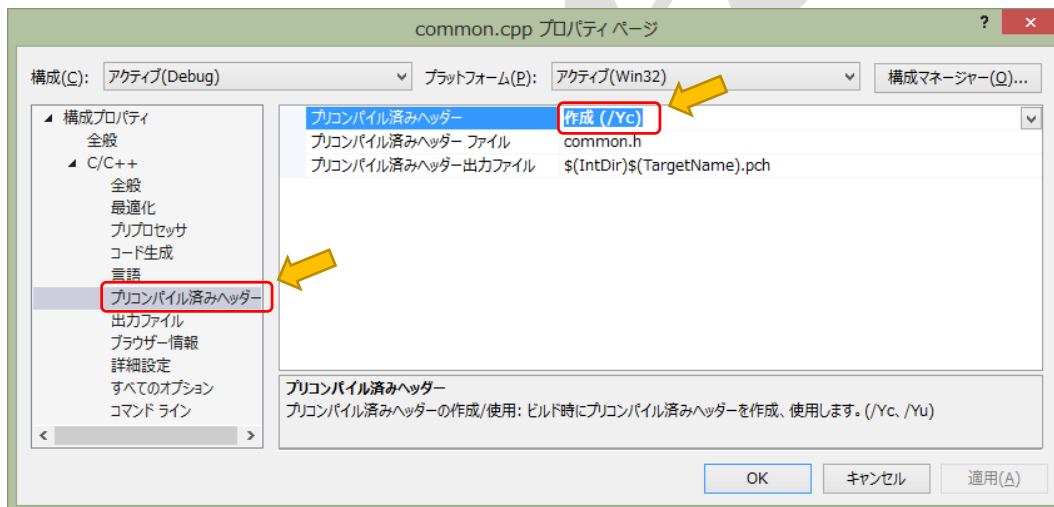
【手順 4】 コンパイラオプションで「必ず使用されるインクルードファイル」を指定



【手順5】 common.cpp を選択して、ファイル固有のコンパイル設定を変更



【手順6】 コンパイラオプションで「プリコンパイル済みヘッダー」の「作成」を指定



以上により、共通ヘッダーファイル `common.h` がプリコンパイル済みヘッダーになる上、強制インクルードファイルに指定される。

これにより、明示的に `common.h` をインクルードすることなく、全てのソースファイルに `common.h` が自動的にインクルードされ(つまり必ず `common.h` に基づかなければ成らない状態になる)、かつ、コンパイルの効率も最適化された状態となる。

▼ 共通ライブラリをきちんと使用する／算術系処理は独自実装しない

例えば、自作のプログラムをゲームに流用する際に、自前のベクトル演算クラスや行列演算クラスを使いたくなることがある。

汎用的な処理を自前で組み込むと、同じような処理が乱立してごちゃごちゃするだけでなく、プログラムが不当に肥大化してしまい、メモリの厳しいゲーム機では致命的な問題に発展することがある。

そのため、**勝手な判断で汎用処理の独自実装をしてはいけない。**

これは、**一般的なライブラリを使用する場合であっても同様**である。STL や標準入出力 (printf 文や cout, string など) の使用にあたっては、開発プロジェクトの方針に確実に従わなければならない。

なお、ここで上げた STL, printf, cout, string は、いずれも、往々にしてゲームプログラムには使用するべきではない。この点については別紙の「プログラミング禁則事項」にて詳述する。本書は「最適化」に焦点を当てるものであり、特に「算術系処理」に関して、このようなプログラミング規約の徹底を強調する。

ベクトル演算や行列演算などの算術系処理は、とくに最適化を意識すべきである。ゲーム機向けのライブラリが提供する算術系処理は、「SIMD 演算」に最適化されているなど、恩恵が大きい。「自前の処理クラスの方が多機能で便利」といった考えは捨てて、パフォーマンスを最大化することを心がける。

余談になるが、座標と無関係の浮動小数点変数 3～4 個を四則演算するような時に、ベクトル演算としてまとめて処理すると、SIMD 演算が活用されて高速化されることもある。

例：

```
var1 *= 2.f;
var2 *= 2.f;
var3 *= 2.f;
var4 *= 2.f;
var5 = var1 + var2 + var3 + var4;
```

↓ (最適化) ※これぐらい単純な処理だとかえって非効率かもしれないが、あくまで例として示す。

```
Math::vec4 tmp(var1, var2, var3, var4); //四つの浮動小数点型変数をベクトル演算型に代入。
tmp *= 2.f; //一回の計算で四つの変数の計算を行う。実際にいっぺん計算されるので速い。
var5 = tmp[0] + tmp[1] + tmp[2] + tmp[3]; //以後、元の変数に値を戻す必要が無い限りは、ベクトル型変数を使い続ける。
```

▼ テンプレートクラスの意識的な分割と継承

クラスの「継承」は、往々にしてプログラムを分かりにくいものにさせてしまい、クラスの使用者の混乱を招くことも少なくない。そのため、クラス設計の際は、「できるだけ継承しない」「多重継承は原則禁止」「インターフェースの実装」を目的とした場合も含む) という方針のもとに、「本当に継承した方が良いのか？」をしっかりと検討した上で使用する

るべきである。

しかし、テンプレートクラスの場合は「継承」に対する意識が少し異なる。一見して「無意味」に見える継承が、大きな効果を発揮する場合がある。

テンプレートクラスは、「型」を与えた時にプログラムのインスタンスが生成される。これにより、与えられた「型」の数だけプログラムが増殖し、結果としてプログラムサイズが肥大化する。(テンプレートには「メタプログラミング」を意識的に活用して、実行時のコードサイズを逆に小さくしてしまうテクニックもあるが、ここでは言及しない。)

例：(コーディング時)

```
template<class T> T max(T a, T b) {return a > b ? a : b;}

int x = max(xx, 10);
float y = max(yy, 20.f);
```

↓ (コンパイル後のイメージ)

```
//コーディング時は一つだったプログラムが、コンパイル後には二つになる。
//この関数に関しては、プログラムサイズが単純に二倍になってしまう。
int max_i(int a, int b) {return a > b ? a : b;}
float max_f(float a, float b) {return a > b ? a : b;}

int x = max_i(xx, 10);
float y = max_f(yy, 20.f);
```

この問題は、テンプレートクラスのサイズが大きければ大きいほど顕著となる。

そこで、意識的にプログラムを分割し、「テンプレート化する必要がない部分」を親クラスにし、それを継承してテンプレートクラスを作ると、無駄なインスタンスの増殖を減らすことができる。

本来の「継承」が意図する「多数の派生クラスを作ること」を目的とするものではなく、たった一つの派生クラスしか定義されないため、一見して無意味な継承に見えるが、その効果は大きい。メモリの厳しいゲームプログラムでは、テンプレートによるプログラミングの効率化とメモリの最適化の両立を考えた場合、無視できないテクニックとなる。

例：(元のコーディング)

```
template<class T>
class CSample
{
public:
    int calcVar1() {...}
    T calcVar2() {...}
private:
    int m_var1;
    T m_var2;
};
```

↓ (最適化)

```
//テンプレート化不要な部分を親クラスとして分割する
class CSampleBase
{
public:
    int calcVar1() {...}
private:
    int m_var1;
```

```
};
//親クラスを継承して、必要最小限のテンプレートクラスにする
template<class T>
class CSample : public CSampleBase
{
public:
    T calcVar2() {...}
private:
    T m_var2;
};
```

▼ 「.inl」 ファイルの活用と、テンプレートの明示的なインスタンス化

テンプレートクラスは、コンパイル時に処理をインスタンス化するために、ヘッダーファイルに処理を記述しておかなければならない。

これによって、クラスのメンバーが分かりにくくなることは元より、コンパイルやリンクの速度も遅くなってしまう。

この対処として、普通のクラスの .h と .cpp の関係のように、宣言部を .h に、実際のコード部を .inl に記述した上で、テンプレートクラスのインスタンス化を明示的に行うようにする。

【サンプル】

template.h

```
#pragma once
Template<typename T>
class CSample
{
public:
    //アクセッサ ※この程度は普通にインライン化する
    inline T getMember() const { return m_member; }
    inline void setMember(T var) { m_member = var; }

    //処理 ※プロトタイプ宣言のみ
    void method1();
    T method2(T var);
private:
    T m_member;
};
```

template.inl

```
#include "template.h"
//処理 ※実体部
Template<typename T>
void CSample::calComplex()
{
    ...
}
Template<typename T>
T CSample::method2(T var)
{
    ...
}
```

normal.cpp

```
#include "template.h" // .cpp の上部では、普通に .h のみインクルード

Void func()
```

```
{
    CSample<int> x;
    CSample<float> y;
    CSample<C0ther> z;
    ...
}

//テンプレートクラスの明示的なインスタンス化 ※.cpp の末尾に書く。もしくは、別の .cpp にまとめても良い
#include "template.inl"
template class CSample<int>;
template class CSample<float>;
template class CSample<C0ther>;
```

なお、この「テンプレートクラスの明示的なインスタンス化」は絶対のルールではない。入り組んだテンプレートクラスの場合、明示的なインスタンス化が非常に面倒なことがある。このような場合、明示的なインスタンス化は、その時代表されるクラスに対してだけ行うようにすれば良い。

例：

```
#include "template1.h"
#include "template2.h"
...

//テンプレートクラスの明示的なインスタンス化
//※CSample2<T> の中で、CSample1<T> が使用されている場合：
// 必要な .inl をまとめてインクルードして CSample2 だけ
// インスタンス化すれば、CSample1<int> の明示的なインスタンス化は不要。
#include "template2.inl"
#include "template1.inl"
template class CSample2<int>;
```

テンプレートが複雑すぎて、明示的なインスタンス化がうまくいかない場合は、最終手段として、.h い並べて .inl をインクルードしてしまっても良い。

例：

```
#include "template.h"
#include "template.inl" // .h と同時にインクルード
...

//テンプレートクラスの明示的なインスタンス化
//※ (なし)
```

▼ コンストラクタの explicit 宣言はできるだけ付ける

コンストラクタの explicit 宣言は、暗黙的なキャストを禁止するためのものである。

経験的に、オブジェクト（クラス）に対して暗黙的なキャストが必要になった場面はほぼない。

ゲームプログラミングでは、型制約のないスクリプトのような「深く気にしなくても動作する」ことよりも、「動作原理をきちんと把握する」という事に努めるべきである。

そのため、「オブジェクトに対する暗黙的なキャストの発生 = コーディングミス」とみなし、コンパイルエラーになることが望ましい。explicit 宣言はそのためのものである。

explicit 宣言の有無による挙動の違いを示す。

【サンプル】

コード：

```
//クラス
class CSample
{
public:
    CSample(float x) : m_var(x) { printf("constructor! : m_var = %.1f\n", m_var); }
    void print() const { printf("print : m_var = %.1f\n", m_var); }
private:
    float m_var;
};

//整数を返す関数
int func() {return 123;}

//テスト
void test()
{
    CSample obj(1.2f);
    obj = func(); //←コーディングを誤って代入（コンパイルエラーにならない）
}
```

結果：

```
constructor! : m_var = 1.2
constructor! : m_var = 123.0 ← 「=」 演算子による整数の代入が、CSample(float) コンストラクタを暗黙的に呼び出している
```

↓ (explicit 宣言を付ける)

コード：(修正後)

```
//クラス
class CSample
{
public:
    //コンストラクタに explicit 宣言を付ける
    explicit CSample(float x) : m_var(x) { printf("constructor! : m_var=%.1f\n", m_var); }
private:
    float m_var;
};

//整数を返す関数
int func() {return 123;}

//テスト
void test()
{
    CSample obj(1.2f);
    obj = func(); //←コンパイルエラーになる
}
```

このように、explicit 宣言は「引数が一つのコンストラクタ」に対して機能するものである。しかし、実際にはそれ以外のコンストラクタに付けても問題はないので、「全てのコンストラクタに explicit 宣言を付ける」という単純なルールにしても構わない。

▼ 暗黙的なコピーコンストラクタと代入演算子の禁止

先の explicit 宣言と同様に、ゲームプログラミングでは、可能な限り暗黙的な動作を排除したほうが安全である。これは、テンプレートクラスの活用などによって、コーディング量を減らすための努力と相反するものでは決していない。

以下、明示的に書いていないコード「暗黙的なコピーコンストラクタ」や「暗黙的な代入演算子」が勝手に作られてしまうケースを示す。

【サンプル】

コード：

```
//クラス
class CSample
{
public:
    CSample(float x) : m_var(x) { printf( "constructor! : m_var=%.1f\n", m_var); }
private:
    float m_var;
};

//テスト
void test()
{
    CSample obj1(1.2f);
    CSample obj2(2.3f);
    CSample obj3(obj1); //←暗黙的なコピーコンストラクタの呼び出し（コンパイルエラーにならない）
    obj1 = obj2; //←暗黙的な代入演算子の呼び出し（コンパイルエラーにならない）
    obj1.print();
    obj2.print();
    obj3.print();
}
```

結果：

```
constructor! : m_var = 1.2
constructor! : m_var = 2.3
print : m_var = 2.3 //obj1 = obj2 の結果、obj1 に obj2 がコピーされている
print : m_var = 2.3
print : m_var = 1.2 //CSample obj3(obj1) の結果、obj3 に obj1 がコピーされている
```

クラスのオブジェクトは、単純に値がコピーされると問題になるケースがある。それを禁止したい場合、コピーコンストラクタと代入演算子を明示的に作成して、private にしておくが良い。

↓（明示的に private 化）

コード：（修正後）

```
//クラス
class CSample
{
private:
    CSample(CSample&) {} //←コピーコンストラクタを private 化（中身は空でよい）
    CSample& operator=(CSample&) {return *this;} //←代入演算子を private 化（中身は空でよい）
public:
    CSample(float x) : m_var(x) { printf( "constructor! : m_var=%.1f\n", m_var); }
private:
    float m_var;
};

//テスト
void test()
{
    CSample obj1(1.2f);
    CSample obj2(2.3f);
    CSample obj3(obj1); //←コンパイルエラーになる
    obj1 = obj2; //←コンパイルエラーになる
    obj1.print();
    obj2.print();
    obj3.print();
}
```

同様のテクニックで、デフォルトコンストラクタ（引数のないコンストラクタ）を private 化しておく、インスタンス生成時に、必ずパラメータを必要とするものにできる。

例：

```
//クラス
class CSample
{
private:
    CSample() {} //←デフォルトコンストラクタを private 化（中身は空でよい）
public:
    CSample(float x) : m_var(x) {} //明示的なパラメータが必要なコンストラクタのみ public に
private:
    float m_var;
};

//テスト
void test()
{
    CSample obj; //←コンパイルエラーになる
    CSample* obj = new CSample(); //←コンパイルエラーになる
    CSample obj(1.2f); //←OK
    CSample* obj = new CSample(1.2f); //←OK
}

//テストクラス
Class CTest
{
private:
    CSample m_member;
public:
    CTest() {} //←m_member を正しく初期化しないとコンパイルエラーになる
    CTest() : m_member(1.2f) {} //←OK
}
```

▼ インクリメント・デクリメントは、特に理由がない限りは前置を使う

前置インクリメント（デクリメント）と後置インクリメント（デクリメント）の違いは、前置が戻り値として計算後の値を返し、後置が計算前の値を返すことにある。

【サンプル】

コード：（前置インクリメントと後置デクリメントの結果を確認）

```
//テスト
void test()
{
    {
        int x = 1;
        printf("前置インクリメント : ++x = %d\n", ++x);
    }
    {
        int x = 1;
        printf("後置インクリメント : x++ = %d\n", x++);
    }
}
```

結果：

```
前置インクリメント : ++x = 2
後置インクリメント : x++ = 1
```

これは、ごくごく分かり切った結果である。

前置と後置のどちらを使うかは状況によるが、「どちらでも良い場合」というのも多い。そのような場合、極力「前置」を使用することを強く推奨する。

その理由は、前置インクリメント（デクリメント）と後置インクリメント（デクリメント）を独自のクラスに実装してみると明らかである。

【サンプル】

コード：（前置インクリメントと後置デクリメントを独自に実装）

```
//前置インクリメントと後置インクリメントを実装した独自クラス
class CSample
{
public:
    CSample(int var) : m_var(var) {}           //コンストラクタ
    operator int() const { return m_var; }      //キャストオペレータ

    CSample& operator++() { ++m_var; return *this; } //前置インクリメント
                                                    //（計算後の自分自身の参照を返す）

    CSample operator++(int) { CSample prev = *this; ++m_var; return prev; } //後置インクリメント
                                                    //（計算前の自分のコピーの実体を返す）

private:
    int m_var;
};
//テスト
void test()
{
    {
        CSample x = 1;
        printf( "前置インクリメント : ++x = %d\n", ++x );
    }
    {
        CSample x = 1;
        printf( "後置インクリメント : x++ = %d\n", x++ );
    }
}
```

「前置インクリメント」は、インクリメント計算のみで完了するが、「後置インクリメント」は計算前の値の退避とその実体の return という「コピー操作」が2回も発生しており、けっこうな処理の劣化を招く。

int 型などは「前置」でも「後置」でも大した差がないが、オブジェクトの場合の挙動の差は大きい。このため、もうクセとして「なるべく前置インクリメント（デクリメント）を使用する」としておく方が良い。

▼ 仮想クラス（virtual）を避ける

仮想クラスは、vtable を参照してメソッドを呼び出すため、パフォーマンスが悪くなる。

また、オーバーライドしたつもりで書いたメソッドの引数が間違っていると、正しくオーバーライドされない上、コンパイルエラーも出ないため、問題発見に非常に手間取ることがある。このような問題は、親クラスの設計者が安易にメソッド名やメソッドの引数を変えた時にも起こりえることで、その影響による問題の発覚が、意外と遅れてしまう。

このような問題の解消方法を、別紙の「効果的なテンプレートテクニック」にて詳述する。

■■以上■■

■ 索引

C

cout 6

E

explicit 宣言 9

P

printf 6

S

SIMD 演算 6

STL 6

string 6

V

virtual 13

vtable 13

い

インクリメント

後置インクリメント 12

前置インクリメント 12

か

仮想クラス 13

こ

コピーコンストラクタ 11

て

デフォルトコンストラクタ 12

テンプレートクラス 7

コーディング手法

以 上