プログラミング禁則事項

– 従うべきプロジェクトのルールに配慮する –

2014 年 2 月 18 日　初稿

板垣 衛

■ 改訂履歴

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

■ 目次

■ 概略

メモリなどの制約が厳しいゲーム機向けのプログラミングにおいて、とくに気をつけるべき要素を解説。

メモリ操作方法、コンパイルオプションによって使用可／不可が変わる機能、仮想クラス／STL／ファイルの扱いなど、開発プロジェクトの方針によって禁則もしくは用法が限定される事項を示す。

■ 目的

本書は、個人個人の意識が低いまま行われたプログラミングによって、全体の動作に影響を及ぼし、開発後期になって追跡困難な問題に発展するといった不測の事態に陥らないようにすることを目的とする。

標準ライブラリの関数やクラスを安易に使っていると、その時は大丈夫でも、後々になって深刻なメモリ不足やメモリリークなどのトラブルを招くことがある。

■ 禁則事項・注意事項

▼ malloc()／calloc()／realloc()／free()関数

標準関数のメモリ操作関数を使用してよいかどうかは、開発プロジェクトの方針に従うこと。

通常のゲーム開発では禁止されるていることが多い。というよりも、標準関数が使われていると、細かいメモリチューニングやバグ追跡の際に融通が利かないので、直接利用するべきではない。禁止されていない開発プロジェクトは危険視すべき。

なお、開発プロジェクトによっては malloc()関数／free()関数を独自関数に置き換えている場合もあるので、よく確認すること。

▼ ゲーム用 SDK が提供するメモリ操作関数

開発プロジェクトの方針に従うこと。

会社で独自のメモリマネージャを用意しているケースも多い。

▼ new／delete 演算子

開発プロジェクトの方針に従って用法を守ること。配置 new の使用や、クラスごとの new 演算子のオーバーロードといった規定があるはず。もしくは、new をラップした #define マクロの使用などが規定されていることもある。

▼ メモリ操作全般

利便性のために、マクロを使用して独自のメモリ操作命令を用意しているケースもある。

new 時にデバッグ情報（__FILE__ や __FUNCTION__）を自動的に記録したり、用途（グラフィックデータ向け、サウンドデータ向け、プログラムオブジェクト向けなど）に合わせた複数のメモリ管理ブロックを使い分けたりといったことが行われる。

また、スレッドにまたがって扱われるデータのメモリ確保には、さらに注意が必要である。

グラフィックデータや AI がアクセスする設定データなど、各プログラマーが思い思いにメモリ確保することを禁止し、必ずリソースマネージャを通すといった規約が設けらている可能性がある。スレッドセーフなメモリ管理の意識はとても重要である。

とにかく、メモリ操作については、プログラマー全員が最も気をつけるべきことである。開発プロジェクトのメモリ管理方針と用法を、最初にしっかりと把握することが重要。

▼ STL（各種コンテナ）／string

STL や string の使用は開発効率を向上させるが、問題も多い。利用の際は、開発プロジェクトの方針に従わなければならない。

● コンテナクラス

STL には配列のように多数の要素を格納するためのクラスが用意されている。vector, list, map などが代表的。これらのクラスは「コンテナ」と呼ばれる。

「コンテナ」クラスは内部で自動的にメモリを確保するので、その利用には十分に注意が必要である。

以下、STL のコンテナクラスを列挙する。

- vector ..... ベクター（動的配列）

- list ..... リスト（双方向リスト）
- queue ..... キュー（FIFO）
- deque ..... デキュー（両端キュー）
- priority_queue ..... プライオリティキュー（優先順位付キュー）
- stack ..... スタック（FILO）
- set ..... セット（ソート済みリスト：重複不可）
- multiset ..... マルチセット（ソート済みリスト：重複可）
- map ..... マップ（連想配列：キー重複不可）
- multimap ..... マルチマップ（連想配列：キー重複可）
- bitset ..... ビットセット（固定ビット配列）

● 気を付けるべきコンテナクラスの挙動①：問題点

- メモリが動的に確保される。
  - vector では、要素数拡大時に realloc が行われる。処理速度にも影響する。
  - string は、16 バイトまではメモリ確保を行わず、string オブジェクト本体内に文字列を記録するが、それを超えると malloc が発生する。
- オブジェクトのコンテナの場合、コピーコンストラクタが呼び出される。

  例：

```
std::vector<MyClass> array;
array.push_back(MyClass());   //MyClass のインスタンス生成→vector に受け渡される
                              //→vector がコピーコンストラクタでインスタンスを生成して
                              //  受け渡されたインスタンスの内容をコピー
                              //→受け渡した MyClass のインスタンスは破棄（デストラクタが呼ばれる）
```

  - vector では、realloc が発生すると全要素のコピー（コピーコンストラクタでインスタンスを生成）と破棄（コピー元のデストラクタ呼び出しと破棄）が行われ、かなり重い処理になる。
- 要素を削除してもメモリが解放されない。
  - erase()で要素を削除しても、見た目上存在しなくなるだけで、予約領域として残り続ける。
  - string も同様に、長い文字列がセットされているところに短い文字列を代入しても、一番長かった文字列の領域がそのまま使われ続ける。
- bitset は動的なメモリ確保が行われないので、普通に使っても問題ない。

● 気を付けるべきコンテナクラスの挙動②：対策

STL を使用して良いかどうかの判断がまず先行するが、どうしても使用する場合は、前述の問題点に気を付け、以下の対策を行うこと。

- 最大要素数を予測して事前に reserve()し、動的な要素拡大が行われないようにする。
  - 予約を含めた全体の要素数は capacity()で、実際に有効な要素数は size() で確認できる。

例：

```
std::vector<MyClass*> array;                //vector 型変数宣言
array.reserve(10);                          //10 個の要素を予約
printf( “size=%d”, array.size());           //有効な要素数を取得
printf( “capacity=%d”, array.capacity()); //予約されている要素数を取得
```

- オブジェクトのコンテナは使わない。必要なら std::vector<MyClass*> のようにして、オブジェクトの参照を扱う。
  - ポインターで扱う限りは、コンストラクタ／デストラクタの呼び出しは発生しない。
- 最終的に要素を削除したい場合、swap を使用して空にする。
  - swap() は、vector の配列をまるごと入れ替えるためのメソッド。これを利用すると有効な要素数だけの領域が作られ、予約領域が削除される。

例：

```
std::vector<MyClass*>(array).swap(array);  //自分から自分への swap で有効サイズのみに縮小
std::vector<MyClass*>().swap(array);       //空の vector への swap で全削除
```

● STL を使ってよいかどうかの判断

STL が利用できるかどうかの判断基準は主に下記の２点。

- 暗黙的なメモリ確保の問題が解決されている場合。
- STL の使用によるプログラムサイズの増大が問題にならない場合。
  - STL は一見してコードを簡潔にするがテンプレートクラスによってプログラムインスタンスが増大する。

● カスタムアロケータ

STL のメモリ確保の問題を解決する手段として、STL は「カスタムアロケータ」を使用できるようにしている。

例えば vector なら、std::vector<MyClass*, MyAllocator> といった形で、独自のアロケータクラスをテンプレート引数に渡すことができる。

開発プロジェクトで STL 向けのカスタムアロケータが用意されている場合、それを使って STL を利用することが許可されている可能性がある。

● アルゴリズム／イテレータ

STL を利用する利点の一つは、アルゴリズム、イテレータを利用できることにある。

イテレータは STL の既存のコンテナクラスを利用するだけでなく、自作することもできる。std::iterator<category, type> を継承したクラスを自作すれば、for_each() などの STL が提供するアルゴリズムを使用できる。

それぞれ、#include <iterator>、#include <algorithm> で使用できる。

● 自作コンテナ

イテレータの自作とアルゴリズムの利用に関しては、勝手なメモリ確保は発生しないので、STL コンテナの利用が禁止されている場合でも、これだけは許可されている可能性がある。

自作クラス内の配列要素にアクセスするためのイテレータを用意するのも、可読性の良いプログラムを作成する有効な手段である。

例：

```
//=============================================================================
//自作イテレータテスト

#include <iterator>
#include <algorithm>
#include <stdio.h>

//テストクラス
class MyClass
{
public:
	//アクセッサ
	const char* getName() const { return m_name; }//名前
public:
	//コンストラクタ
	MyClass(const char* name) :
		m_name(name)
	{}
private:
	//フィールド
	const char* m_name;//名前
};

//自作コンテナ用配置 new
class MyContainer;
void* operator new(size_t size, MyContainer& con, void* buff){ return buff; }
void operator delete(void* p, MyContainer& con, void* buff){}

//自作コンテナ
class MyContainer
{
public:
	//自作イテレータ
	class Iterator : public std::iterator<std::forward_iterator_tag, MyClass>
	{
	public:
		//コンテナ要素アクセスオペレータ
		MyClass& operator*() { return *m_obj; }
```

```
            MyClass* operator->() { return m_obj; }
            const MyClass& operator*() const { return *m_obj; }
            const MyClass* operator->() const { return m_obj; }
            //イテレータのインクリメント
            Iterator& operator++() { m_obj = m_cont[++m_index]; return *this; }
            Iterator operator++(int) { Iterator prev = *this; m_obj = m_cont[++m_index]; return prev; }
            //イテレータの比較
            bool operator==(const Iterator& ite) const { return m_obj == ite.m_obj; }
            bool operator!=(const Iterator& ite) const { return m_obj != ite.m_obj; }
            //コンストラクタ
            Iterator(MyContainer& coll, int index) :
                  m_cont(coll),
                  m_index(index),
                  m_obj(nullptr)
            {
                  m_obj = m_cont[m_index];
            }
      private:
            //フィールド
            MyContainer& m_cont;//コンテナ
            int m_index;        //参照要素インデックス
            MyClass* m_obj;     //コンテナ要素の参照
      };
public:
      //イテレータ取得
      Iterator begin(){ return Iterator(*this, 0); }  //先頭
      Iterator end(){ return Iterator(*this, m_num); }//末尾
      //要素情報取得
      bool empty() const { return m_num == 0; }//要素が空？
      int size() const { return m_num; }       //要素数
      //要素取得
      MyClass* operator[](const int index)
      {
            if (index < 0 || index >= m_num)
                  return nullptr;
            return reinterpret_cast<MyClass*>(m_buff[index]);
      }
      //要素追加
      MyClass* add(const char* name)
      {
            if (m_num >= ELEM_MAX)
                  return nullptr;
            return new(*this, m_buff[m_num++])MyClass(name);//既存のバッファにオブジェクト構築（解放不要）
      }
public:
      //コンストラクタ
      MyContainer() :
            m_num(0)
      {}
      //デストラクタ
      ~MyContainer()
      {}
private:
      //フィールド
      static const int ELEM_MAX = 8;         //最大要素数
      char m_buff[ELEM_MAX][sizeof(MyClass)];//要素用のバッファ
      int m_num;                             //要素数
};

//自作イテレータテスト
void testMyIterator()
{
      //コンテナ作成
      MyContainer cont;
      cont.add("太郎");
```

```
        cont.add("次郎");
        cont.add("三郎");

        //for_each 用関数オブジェクト（ファンクタ）
        struct MyFunctor{
              void operator()(MyClass& obj){ printf("name=¥"%s¥"¥n", obj.getName()); }
        };

        //全コンテナ要素処理
        std::for_each(cont.begin(), cont.end(), MyFunctor());
}
```

↓（処理結果）

```
name="太郎"
name="次郎"
name="三郎"
```

● 自作汎用コンテナ

STL のような自動的にメモリ確保して拡張するコンテナの使用を禁止し、社内ライブラリとしてコンテナクラスを自作するのが有効な対策である。

STL 互換の挙動をしつつ動的なメモリ確保をしないコンテナを用意できると、安全性と生産性の両方を確保できる。

▼ Boost C++ などの外部ライブラリ

Boost C++ など、有用な外部ライブラリも多い。

STL と同様のコンテナクラスや boost::smart_ptr など、暗黙的にメモリ確保を行っているものもあるので、使用の際には注意が必要である。

外部ライブラリの利用は、個人の判断で導入するようなことはせず、開発プロジェクトの責任者とよく相談した上で、方針を固めてから利用すること。

▼ ファイル操作

ファイル操作には何かと制約が多い。ファイルデスクリプタの同時オープン数が規制されていたり、ファイル操作の処理によってゲーム全体が長時間ブロックされたりといったことである。

個人の判断で、fopen などの標準ライブラリの関数を安易に使用することは、基本的に禁止されているのが通例である。

SDK が提供する API を使えばよいかという事でもなく、ファイル操作は開発プロジェクトごとに用意された仕組みを使用する。ミドルウェアをそのまま使用するケースもあれば、独自のファイルマネージャーを通すケースもある。

やはりファイル操作についてもメモリ操作と同様に、開発プロジェクトの方針にきちん

と従う必要がある。

なお、ゲームでは非同期読み込みが必要とされる場面が多いので、読み込み要求～完了待ち～読み込み／キャンセルの作法を、開発プロジェクトで規定していることが多い。

▼ 仮想クラス（virtual）

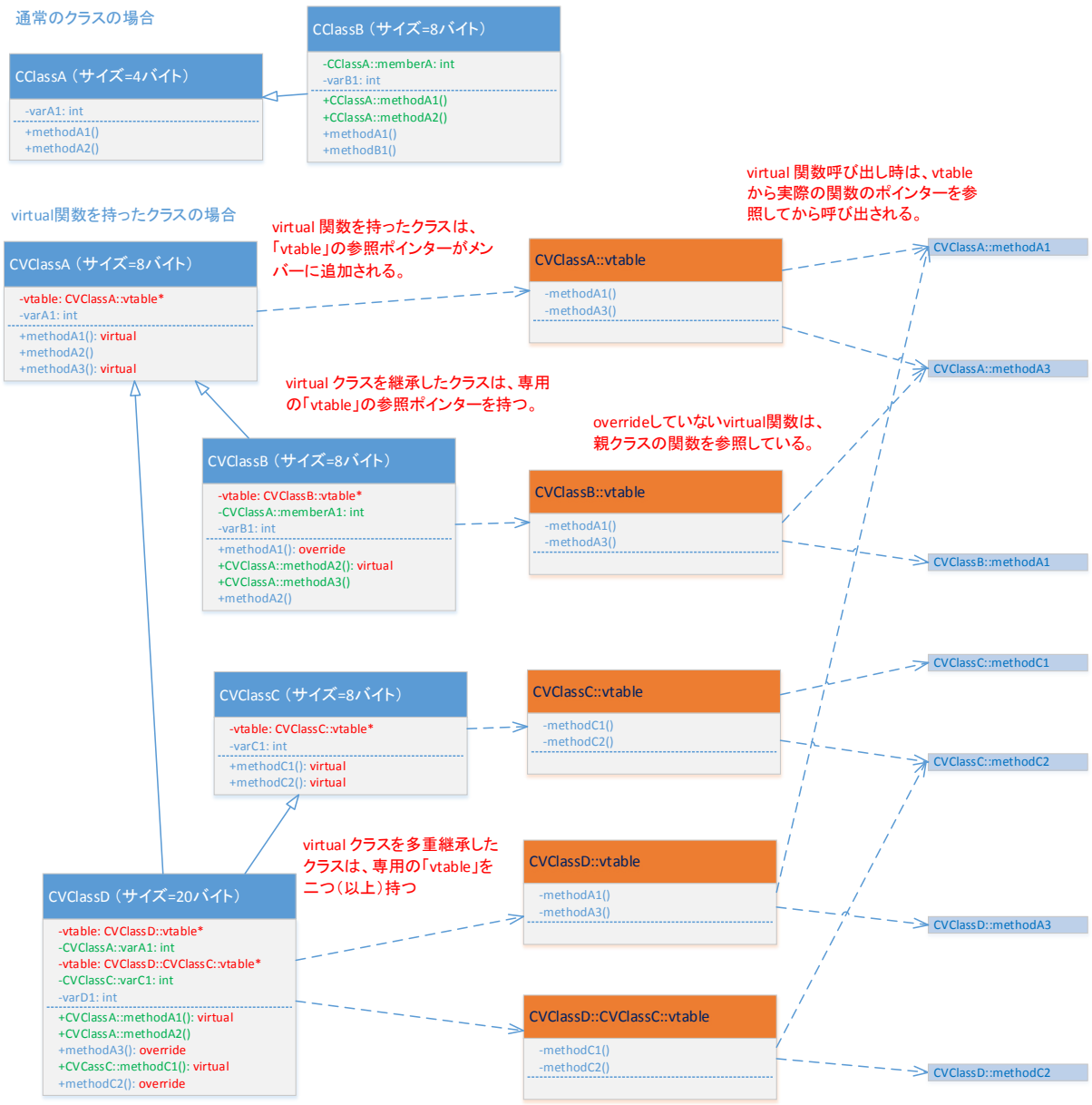
まず「クラスのデストラクタには virtual を付けるもの」という固定観念は捨てるべきである。

確かに、有無を言わさずデストラクタに virtual を付けたほうが、デストラクタ呼び出しの確実性が増して安心な面がある。しかし、ゲームプログラミングはもっとメモリに厳しくあるべきである。virtual を一つでも使ったクラス／構造体は、わずかにサイズが大きくなる上、new 演算子等を使ってコンストラクタを確実に呼び出す必要がある。

また、virtual を使用していると、パフォーマンスへの影響もあるので、virtual の仕組みを理解した上で、本当に必要な所だけで使うようにするべきである。

以下、virtual の仕組みについて解説する。

- **virtual の仕組み：vtable（仮想関数テーブル）**



virtual 関数は、オブジェクトを親クラスにキャストした状態で呼び出しても、vtable に基づいて、インスタンスのメンバー関数が呼び出される。この図の例で説明すると、CVClassB のインスタンスを CVClassA* の変数に代入して methodA1() を呼び出した場合でも、CVClassB::methodA1() が呼び出される。

virtual 関数でない場合は、たとえ同名のメンバー関数であっても、インスタンスのメンバー関数ではなく、その時の変数のメンバー関数が呼び出される。この図の例で説明すると、CVClassB のインスタンスを CVClassA* の変数に代入して methodA2()

を呼び出した場合、CVClassB にも methodA2() があるが、それではなく CVClassA::methodA1() が呼び出される。

このように、virtual 関数の呼び出しは、vtalbe を参照してから実際に呼び出されるため、通常の関数呼び出しよりも少しだけ遅くなる。これがわずかなら問題ないが、大量のアクセッサを virtual にしている場合など、顕著に遅くなることがある。さらに、vtable の参照ポインターの分、わずかにクラスのサイズが大きくなる点も注意が必要である。vtable の参照ポインターは、コンストラクタ呼び出し時にセットされる。

深く考えずになんでもかんでも virtual 関数にしてしまえば融通が利くが、無駄が大きくなるので、よく考えて適用すべきである。「本当に継承（多態性）が必要なクラスか？」「本当に virtual 関数が必要か？」「virtual 関数は必要最小限に抑えてあるか？」といったことを深く考えて使用すること。

なお、共通の構造を持った複数のクラスに対して共通処理を適用したい場合、継承ではなくテンプレートを使う方法がある。この場合、コードサイズは増えるが、vtable の参照がないので高速に処理できる。別紙の「効果的なテンプレートテクニック」にて、「CRTP」というテンプレートのテクニックを紹介する。

▼ 実行時型情報（RTTI：typeid／dynamic_cast）

実行時型情報（RTTI = RunTime Type Identification）は、コンパイルオプションで有効／無効を切り替えできる。実行時型情報を使ってよいかどうかは開発プロジェクトの方針によるので、きちんと確認してから使用すること。

実行時型情報が必要になる処理は、主に下記の２つである。これらの処理が必須の場合は、実行時型情報を有効にしなければならない。

・ dynamic_cast を使用し、型安全なダウンキャストを行いたい。

```
CBase* base = getObject();
CDerived* derived = dynamic_cast<CDerived*>(base);    //base のインスタンスが、CDerive 型、もしくは、
                                                      //更にその子の型であった場合だけ、dynamic_cast は
                                                      //適切なポインターを返す。
                                                      //それ以外の型の場合は nullptr を返す。
```

・ typeid を使用し、親クラスの変数からインスタンスの型を判定したい。

```
CBase* base = getObject();
if(typeid(*base) == typeid(CDerived))     //base のインスタンスが、CDerive 型出会った場合だけ、
    return true;                          //この判定がマッチする
```

● dynamic_cast と typeid の注意点

dynamic_cast も typeid も、virtual 関数を持たないクラスに対しては意図通り動

作しない点に注意。実行時型情報は vtable に付随するため、virtual 関数を持たないクラスは実行時にインスタンスの型を判別することができない。

virtual 関数を持たないクラスに対して dynamic_cast や typeid を使ってもコンパイルエラーにはならないが、インスタンスの型ではなく変数の型で処理される。なお、この処理はコンパイル時に型を特定してしまうため、実行時型情報が無効であっても動作する。

● アップキャスト

「アップキャスト」とは、子クラスのポインター型を親クラスのポインター型にキャストすること。

実行時型情報の有効／無効、virtual 関数の有無に関係なく、static_cast と dynamic_cast が使用可能。アップキャストはコンパイル時に型が特定されても問題なく動作する。

なお、多重継承している子クラスの場合、キャスト時にポインターの位置が変わる場合がある点に注意。（適切な型の vtable の位置にポインターが移動する）

もう一つ注意点として、reinterpret_cast をアップキャストに使用してはいけない。reinterpret_cast はポインター位置を変えないキャストなので、多重継承している子クラスのキャストが正常に動作しない。

● ダウンキャスト

「ダウンキャスト」とは、親クラスのポインター型を子クラスのポインター型にキャストすること。

実行時型情報が有効な時に、dynamic_cast でしか行うことができない。static_cast ではコンパイルエラーになる。実行時型情報が無効な時は、dynamic_cast はコンパイル時にワーニングとなり、実行時には例外をスローする。

virtual 関数の無いクラスに対するダウンキャストは、static_cast も dynamic_cast もコンパイルエラーになる。

なお、reinterpret_cast はアップキャスト時と同様の理由で使用してはいけない。

● 実行時型情報の有無による違い

実行時型情報が有効だと、型情報がプログラムに組み込まれる分、プログラムのサイズがやや大きくなる。クラスのサイズが増えるようなことはない。また、処理速度にはほとんど影響がない。

よほどメモリが逼迫しているような状況でも無い限り、実行時型情報は有効にして、

dynamic_cast を使えるようにしておいた方が良い。

▼ 例外（exception：try ～ catch）

例外のスロー、キャッチは、コンパイルオプションで有効／無効を切り替えできる。開発プロジェクトの方針によって、「完全に禁止」「特定箇所のみで使用」「共通処理でスローしているのできちんとキャッチしなければならない」など決まっているはずである。

例外処理は融通の利いたエラー処理を書けるが、キャッチ処理はコーディング量が増えて煩わしい面がある。また、意識して細かくキャッチ処理を書いていないと、想定外に上位でキャッチされて処理の過程を見失ってしまうこともある。処理速度への影響も出やすいので、ゲーム開発では例外はあまり使わない方が得策である。

▼ 乱数

乱数の使用は意外と気を付けなければならない。

rand() 関数を使用することが禁止されている場合もあるので、安易に使用せず、よく確認を取らなくてはならない。

乱数については別紙の「プレイヤーに不満を感じさせないための乱数制御」に詳しくまとめている。

■■以上■■

■ 索引

索引項目が見つかりません。

プログラミング禁則事項

以　上