本当にちょっとしたプログラミング Tips

– 共通認識にしたいプログラミングテクニック –

2014 年 4 月 13 日 第二稿

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 13 日	板垣 衛	「構造に関する Tips」「最適化に関する Tips」を追加。

■ 目次

■ 概略

プログラミングの際に知っておくと役に立つことを解説。

■ 目的

本書は、ゲームプログラミングを少しでも効率的・効果的にすることを目的とする。

このような基礎的なプロミングの方法論がチーム内での共通認識となることで、共同開発を少しでも円滑なものにする。

■ コーディングに関する Tips

▼ 配列の初期化

C 言語で配列を初期化する際は、通常下記のように記述する。

例：配列の初期化

```
int arr[5] = { 1, 2, 3, 4, 5 };
```

配列要素数を超える初期化を行うとコンパイルエラーになるが、少なかったときは、残りの要素が 0 で初期化されることを C 言語の仕様は保証している。

例：要素数と一致しない配列の初期化

```
int arr[5] = { 1, 2, 3, 4, 5, 6 }; //←初期化の数が要素数より多いのでコンパイルエラー
int arr[5] = { 1, 2, 3 }; //←初期値が指定されなかった要素は 0 になる　⇒ { 1, 2, 3, 0, 0 }
```

この仕様に基づき、ゼロクリアした配列を用意した場合は、初期値を一つも書かずに初期化を指定するだけでよい。

例：配列要素をすべて 0 で初期化

```
int arr[5] = {}; //←初期値を一つも指定しなければ、全ての要素が 0 になる　⇒ { 0, 0, 0, 0, 0 }
```

▼ #define マクロの活用

● 便利な「定義済みマクロ」

まず、Visual C++ で使用可能な定義済みマクロを紹介する。ゲームプログラミングでも特に有用なマクロを赤字で示す。

なお、説明内容は、Visual Stduio 2013 の Web ページからそのまま引用。(http://msdn.microsoft.com/ja-jp/library/b0084kay.aspx)

【ANSI C 準拠の定義済みマクロ】

- __DATE__ ..... 現在のソース ファイルのコンパイル日付。
  日付は Mmm dd yyyy 形式のリテラル文字列です。 月の名前 Mmm は、TIME.H で宣言された asctime ライブラリ関数によって生成される日付のものと同じです。
- __FILE__ ..... 現在のソース ファイルの名前。
  __FILE__ は二重引用符で囲まれた文字列に展開されます。 ファイルへの完全なパスが表示されるようにするには、/FC (診断時のソース コード ファイルの完全パス) を使用します。
- __LINE__ ..... 現在のソース ファイルの行番号。
  行番号は 10 進数の整数定数です。 これは #line ディレクティブを使用して変更できます。
- __STDC__ ..... ANSI C 標準への完全準拠。
  /Za コンパイラ オプションを指定している場合で、C++ コードをコンパイルしないときにのみ、整数定数 1 として定義されます。それ以外の場合は、定義されません。
- __TIME__ ..... 現在のソース ファイルの最新のコンパイル時間。
  時間は hh:mm:ss 形式の文字列リテラルです。
- __TIMESTAMP__ ..... 現在のソース ファイルの最終更新日時。
  日時は Ddd Mmm Date hh:mm:ss yyyy 形式の文字列リテラルとして表記します。ここで、Ddd は曜日の略称であり、Date は 1 ～ 31 の整数です。

【Microsoft 固有の定義済みマクロ】

- _ATL_VER ..... ATL バージョンを定義します。
  Visual Studio 2012 では、_ATL_VER は 0x0B00 として定義されます。
- _CHAR_UNSIGNED ..... 既定の char 型が unsigned になります。
  /J を指定している場合に定義されます。
- __CLR_VER ..... アプリケーションのコンパイル時に使用された共通言語ランタイムのバージョンを定義します。
  返される値は次の形式になります。
  「Mmmbbbbb 」
  指定項目 :
  「M」はランタイムのメジャー バージョンです。
  「mm」はランタイムのマイナー バージョンです。
  「bbbbb」はビルド番号です。

- __cplusplu_cli ..... /clr、/clr:pure、または /clr:safe を指定してコンパイルする場合に定義されます。
  __cplusplus_cli の値は 200406 です。__cplusplus_cli は翻訳単位全体で有効になります。
- __cplusplu_winnt ..... /ZW オプションを使用してコンパイルする場合に定義されます。
  __cplusplus_winrt の値は 201009 です。
- __COUNTER__ ..... 0 から始まる整数に展開されます。
  ソース ファイルで使用されるか、ソース ファイルのヘッダーに追加されるたびに 1 ずつ増えます。__COUNTER__ はプリコンパイル済みヘッダーを使用するときのその状態を記憶します。
- __cplusplus ..... C++ プログラムの場合にのみ定義されます。
- __CPPRTTI ..... /GR （ランタイムの型情報の有効化） を指定してコンパイルしたコードの場合に定義されます。
- __CPPUNWIND ..... /EH （例外処理モデル） フラグのいずれかを使用してコンパイルしたコードの場合に定義されます。
- _DEBUG ..... /LDd、/MDd、/MTd を使用してコンパイルする場合に定義されます。
- _DLL ..... /MD または /MDd （マルチスレッド DLL） を指定した場合に定義されます。
- __FUNCDNAME__ ..... 外側の関数の装飾名を文字列として定義します。
  関数でのみ有効です。/EP または /P コンパイラ オプションを使用する場合、__FUNCDNAME__ は展開されません。
- __FUNCSIG__ ..... 外側の関数のシグネチャを文字列として定義します。
  関数でのみ有効です。/EP または /P コンパイラ オプションを使用する場合、__FUNCSIG__ は展開されません。
  64 ビット オペレーティング システムでは、既定の呼び出し規約は __cdecl です。
- __FUNCTION__ ..... 外側の関数の非装飾名を文字列として定義します。
  関数でのみ有効です。/EP または /P コンパイラ オプションを使用する場合、__FUNCTION__ は展開されません。
- _INTEGRAL_MAX_BITS ..... 整数型の最大サイズ （ビット単位） をレポートします。
- _M_ALPHA ..... DEC ALPHA プラットフォームの場合に定義されます（サポート終了）。
- _M_AMD64 ..... x64 プロセッサの場合に定義されます。
- _M_CEE ..... /clr の任意の形式 （/clr:oldSyntax、/clr:safe など） を使用するコンパイルの場合に定義されます。
- _M_CEE_PURE ..... /clr:pure を使用するコンパイルの場合に定義されます。
- _M_CEE_SAFE ..... /clr:safe を使用するコンパイルの場合に定義されます。

- _M_IX86 ..... x86 プロセッサの場合に定義されます。x64 プロセッサの場合には定義しません。

  M_IX86 = 600 (既定)
  ビルドオプション = Blend (/GB)
  ※将来のコンパイラは、最も優先度の高いプロセッサを反映する別の値を規定に出力する予定です
  _M_IX86 = 500
  ビルドオプション = Pentium (/G5)
  _M_IX86 = 600
  ビルドオプション = Pentium Pro、Pentium II、Pentium III (/G6)
  _M_IX86 = 300
  ビルドオプション = 80386 (/G3)
  _M_IX86 = 400
  ビルドオプション = 80486 (G4)
- _M_IA64 ..... Itanium プロセッサ ファミリの 64 ビット プロセッサの場合に定義されます。
- _M_ARM_FP ..... どの /arch コンパイラ オプションが使用されたかを示す値に展開されます。

  /arch オプションが指定されていない場合は 30 ～ 39。ARM の既定のアーキテクチャ (VFPv3) が使用されたことを示します。
  /arch:VFPv4 が使用された場合は 40 ～ 49。
- _M_IX86_FP ..... どの /arch コンパイラ オプションが使用されたかを示す値に展開されます。

  /arch:IA32 が使用された場合は 0。
  /arch:SSE を使用した場合は 1。
  /arch:SSE2 が使用された場合は 2。
  /arch を指定しなかった場合は、これが既定値になります。
- _M_MPPC ..... Power Macintosh プラットフォームの場合に定義されます (サポート終了)。
- _M_MRX000 ..... MIPS プラットフォームの場合に定義されます (サポート終了)。
- _M_PPC ..... PowerPC プラットフォームの場合に定義されます (サポート終了)。
- _M_X64 ..... x64 プロセッサの場合に定義されます。
- _MANAGED ..... /clr を指定している場合に 1 として定義されます。
- _MFC_VER ..... MFC バージョンを定義します。

  たとえば、Visual Studio 2012 では、_MFC_VER は 0x0B00 として定義されます。
- _MSC_BUILD ..... コンパイラのバージョン番号のリビジョン番号部分に評価されます。

  リビジョン番号はピリオド区切りのバージョン番号の 4 番目の部分です。 たとえば、Visual C++ コンパイラのバージョン番号が 15.00.20706.01 ある場合、_MSC_BUILD マクロは 1 に評価されます。
- _MSC_EXTENSIONS ..... /Ze コンパイラ オプション (既定) を使用してコンパ

イルする場合に定義されます。
その値は定義されると 1 になります。

- _MSC_FULL_VER ..... コンパイラのバージョン番号のメジャー番号、マイナー番号、ビルド番号の部分に評価されます。
  メジャー番号はピリオド区切りのバージョン番号の最初の部分、
  マイナー番号は 2 番目の部分、
  ビルド番号は 3 番目の部分です。
  たとえば、Visual C++ コンパイラのバージョン番号が 15.00.20706.01 ある場合、_MSC_FULL_VER マクロは 150020706 に評価されます。
  コマンド ラインで「cl /?」と入力すると、コンパイラのバージョン番号が表示されます。
- _MSC_VER ..... コンパイラのバージョン番号のメジャー番号とマイナー番号の部分に評価されます。
  メジャー番号はピリオド区切りのバージョン番号の最初の部分、
  マイナー番号は 2 番目の部分です。
  たとえば、Visual C++ コンパイラのバージョン番号が 17.00.51106.1 である場合、_MSC_VER マクロは 1700 に評価されます。
- __MSC_RUNTIME_CHECKS ... /RTC コンパイラ オプションのいずれかを指定している場合に定義されます。
- _MT ..... /MD または /MDd (マルチスレッド DLL)、あるいは /MT または /MTd (マルチスレッド) を指定している場合に定義されます。
- _NATIVE_WCHAR_T_DEFINED
  ..... /Zc:wchar_t を使用している場合に定義されます。
- _OPENMP ..... /openmp を使用してコンパイルする場合に定義されます。
  Visual C++ に実装されている OpenMP 仕様の日付を表す整数が返されます。
- _VC_NODEFAULTLIB ..... /Zl を使用している場合に定義されます。
- _WCHAR_T_DEFINED ..... /Zc:wchar_t を使用している場合、またはプロジェクトのシステム ヘッダー ファイルで wchar_t 型を定義している場合に定義されます。
- _WIN32 ..... Win32 および Win64 用のアプリケーションの場合に定義されます。
  これは、常に定義されます。(Win32 か Win64 かに関わらず、Windows 用のビルドかどうかの判定に使用できます)
- _WIN64 ..... Win64 用のアプリケーションの場合に定義されます。
- _Wp64 ..... /Wp64 を指定している場合に定義されます。

【GCC 固有の定義済みマクロ】

- __FUNCTION__ ..... VC++の__FUNCTION__ と同じ。
  ただし、const char* を返すマクロのため、プリプロセッサで文字列

のトークン連結ができない点に注意。

- __PRETTY_FUNCTION__ .... VC++の__FUNCSIG__とほぼ同じ。
  ただし、const char* を返すマクロのため、プリプロセッサで文字列のトークン連結ができない点に注意。

幾つかのマクロの表示例を示す。

例：定義済みマクロ表示プログラムサンプル

```
//定義済みマクロ表示
int printDefinedMacro(const int argc, const char* argv[])
{
	printf("¥n【ANSI C準拠の定義済みマクロ】¥n");
	printf("__FILE__      = ¥"%s¥"¥n", __FILE__);
	printf("__LINE__      = %d¥n", __LINE__);
	printf("__DATE__      = ¥"%s¥"¥n", __DATE__);
	printf("__TIME__      = ¥"%s¥"¥n", __TIME__);
	printf("__TIMESTAMP__ = ¥"%s¥"¥n", __TIMESTAMP__);
	printf("¥n【Microsoft 固有の定義済みマクロ】¥n");
	printf("__COUNTER__         = %d¥n", __COUNTER__);	//←カウントアップを確認するために３回使用
	printf("__COUNTER__         = %d¥n", __COUNTER__);	//←
	printf("__COUNTER__         = %d¥n", __COUNTER__);	//←
	printf("__FUNCDNAME__       = ¥"%s¥"¥n", __FUNCDNAME__);
	printf("__FUNCSIG__         = ¥"%s¥"¥n", __FUNCSIG__);
	printf("__FUNCTION__        = ¥"%s¥"¥n", __FUNCTION__);
	printf("_INTEGRAL_MAX_BITS = %d¥n", _INTEGRAL_MAX_BITS);
	printf("_M_IX86_FP          = %d¥n", _M_IX86_FP);
	printf("_MSC_FULL_VER       = %d¥n", _MSC_FULL_VER);
	printf("_MSC_VER            = %d¥n", _MSC_VER);
	return 0;
}
//関数オーバーロード用のダミー関数（__FUNCDNAME__テスト用）
void printDefinedMacro()
{}
```

↓（実行結果）

```
【ANSI C準拠の定義済みマクロ】
__FILE__      = "e:¥work¥github¥public¥test¥program¥c++¥rtti¥src¥main.cpp"
__LINE__      = 431
__DATE__      = "Jan 21 2014"
__TIME__      = "10:04:38"
__TIMESTAMP__ = "Tue Jan 21 10:04:38 2014"

【Microsoft 固有の定義済みマクロ】
__COUNTER__         = 0
__COUNTER__         = 1
__COUNTER__         = 2
__FUNCDNAME__       = "?printDefinedMacro@@YAHHQAPBD@Z"
__FUNCSIG__         = "int __cdecl printDefinedMacro(const int,const char *[])"
__FUNCTION__        = "printDefinedMacro"
_INTEGRAL_MAX_BITS = 64
_M_IX86_FP          = 2
_MSC_FULL_VER       = 180021005
_MSC_VER            = 1800
```

● 定義済みマクロのデバッグ活用：アサーション

よく使われるテクニックとして、アサーション違反発生時などに __FILE__ や __LINE__ などのマクロの情報を表示すると、ランタイム時に問題発生箇所を確認し易くなる。

例：定義済みマクロの活用サンプル

```
#include <stdio.h>
#include <assert.h>

//Visual C++ 固有の定義済みマクロ「__FUNCSIG__」を標準化するために、
//「__FUNC__」として再定義（他のビルド環境でも __FUNC__ を用いればソースコードを共通利用できるようにする）
#define __FUNC__ __FUNCSIG__

//ファイル名取得関数
const char* getFileName(const char* path)
{
	const char* p = path + strlen(path) - 1;
	while (p > path)
	{
		if (*p == '¥¥' || *p == '/')
			return p + 1;
		--p;
	}
	return path;
}

//定義済みマクロを活用したアサーションマクロ
#define MY_ASSERT(expr, msg) ¥
	if (!(expr)) ¥
	{ ¥
		fprintf(stderr, "-----------------------------------------------------------¥n"); ¥
		fprintf(stderr, "Assertion failed!¥n"); ¥
		fprintf(stderr, "%s(%d): %s¥n", getFileName(__FILE__), __LINE__, __FUNC__); ¥
		fprintf(stderr, "%s¥n", #expr);	/* ←expr を # で文字列化 */ ¥
		fprintf(stderr, "%s¥n", msg); ¥
		fprintf(stderr, "-----------------------------------------------------------¥n"); ¥
		assert(expr); /* ←処理比較と abort のために、標準で用意されている assert()を実行 */ ¥
	}

//アサーションテスト
int testAssert(const int argc, const char* argv[])
{
	struct STRUCT
	{
		char memberA;
		short memberB;
		int memberC;
		char memberD;
	};
	MY_ASSERT(sizeof(STRUCT) == 8, "STRUCT 構造体のサイズが 8 バイトではありません！");
	return 0;
}
```

↓（実行結果）

```
-----------------------------------------------------------
Assertion failed!
main.cpp(498): int __cdecl testAssert(const int,const char *[])
sizeof(STRUCT) == 8
STRUCT 構造体が 8 バイトではありません！
-----------------------------------------------------------
Assertion failed: sizeof(STRUCT) == 8, file e:¥work¥github¥public¥test¥program¥c++¥rtti¥src¥main.cpp, line 498
```

● 定義済みマクロのデバッグ活用：文字列連結と文字列化のテクニック

前述のサンプルでもアサーションの条件として指定した「式」を表示するために、「#」（文字列化演算子）を使用して文字列化している。

これを利用して、前述の「main.cpp(498)： int __cdecl testAssert(const int,const char *[])」のように、__FILE__ + __LINE__ + __FUNC__ をつなげた一つの文字列を生成し、デバッグ情報のプールを効率化する。

例えば、独自のメモリマネージャを作成して、一つ一つのメモリ管理情報の中に、呼び出し元を識別するための情報を保存しておくと、後からメモリリークを追跡する際などに役立つ。この時、保存したい情報を一つの文字列に連結しておくことで、デバッグ情報のためのメモリを効率化できる。（プログラム中の文字列リテラルが増えて、プログラムサイズが大きくなる点には注意）

文字列を連結する際、__LINE__ のような整数を返すマクロを単純に文字列化できないので注意が必要である。以下にその問題点と解決策を示す。

例：文字列連結のサンプル①：NG バージョン

```
//ファイル名取得関数
const char* getFileName2(const char* path, const size_t path_len)
{
	const char* p = path + path_len - 1;
	while (p > path)
	{
		if (*p == '¥¥' || *p == '/')
			return p + 1;
		--p;
	}
	return path;
}

//処理行情報作成マクロ
#define MAKE_FUNC_LINE_INFO() getFileName2(__FILE__ "(" #__LINE__ "):" __FUNC__, strlen(__FILE__))

//文字列連結テスト
int testConcatenateStr(const int argc, const char* argv[])
{
	const char* func_line_info = MAKE_FUNC_LINE_INFO();
	printf("MAKE_FUNC_LINE_INFO() = ¥"%s¥"¥n", func_line_info);
	return 0;
}
```

↓（コンパイル結果）

```
error C2121: #' : 無効な文字です : マクロ展開が解決できません。
```

C/C++言語では、コンパイル時に文字列を連結したい場合、ただ単に文字列リテラルを並べるだけで良い。

単純に __LINE__ に # を付けて文字列化しようとするとコンパイルエラーになってしまう。

これを回避するために、文字列化のマクロを用意する。

例：文字列連結のサンプル②：NG バージョン　※サンプル①との差分のみ

```
//文字列化マクロ
#define TO_STRING(s) #s
```

```
//処理行情報作成マクロ
#define MAKE_FUNC_LINE_INFO() getFileName2(__FILE__ "(" TO_STRING(__LINE__) "):" __FUNC__ TO_STRING( [テスト!])
TO_STRING(123), strlen(__FILE__))  // ←TO_STRING の動作チェック用にテストコードも追加
```

↓（実行結果）

```
MAKE_FUNC_LINE_INFO() = "main.cpp(__LINE__):int __cdecl testConcatenate(const int,const char *[])[テスト!]123"
```

TO_STRING() マクロはおおむね意図どおりに動作しているが、__LINE__ マクロがそのまま文字列化されてしまっている。この問題を解決するために、一度 __LINE__ マクロを展開してから TO_STRING() マクロに渡すようにする。

例：文字列連結のサンプル③：OK バージョン　※サンプル①＋②との差分のみ

```
//マクロ展開付き文字列化マクロ
#define TO_STRING_EX(s) TO_STRING(s)

//処理行情報作成マクロ
#define MAKE_FUNC_LINE_INFO() getFileName2(__FILE__ "(" TO_STRING_EX(__LINE__) "):" __FUNC__ TO_STRING( [テス
ト!]) TO_STRING(123), strlen(__FILE__))  // ←TO_STRING の動作チェック用にテストコードも追加
```

↓（実行結果）

```
MAKE_FUNC_LINE_INFO() = "main.cpp(535):int __cdecl testConcatenate(const int,const char *[])[テスト!]123"
```

以上により問題を解決。

改めて、修正版のサンプルプログラム全文を示す。

例：文字列連結のサンプル：OK バージョン全文　※テスト部分は削除

```
//文字列化マクロ
#define TO_STRING(s) #s

//マクロ展開付き文字列化マクロ
#define TO_STRING_EX(s) TO_STRING(s)

//ファイル名取得関数
const char* getFileName2(const char* path, const size_t path_len)
{
	const char* p = path + path_len - 1;
	while (p > path)
	{
		if (*p == '¥¥' || *p == '/')
			return p + 1;
		--p;
	}
	return path;
}

//処理行情報作成マクロ
#define MAKE_FUNC_LINE_INFO() getFileName2(__FILE__ "(" TO_STRING_EX(__LINE__) "):" __FUNC__, strlen(__FILE__))

//文字列連結テスト
int testConcatenateStr(const int argc, const char* argv[])
{
	const char* func_line_info = MAKE_FUNC_LINE_INFO();
	printf("MAKE_FUNC_LINE_INFO() = ¥"%s¥"¥n", func_line_info);
	return 0;
}
```

↓（実行結果）

```
MAKE_FUNC_LINE_INFO() = "main.cpp(535):int __cdecl testConcatenate(const int,const char *[])
```

● コードの連結：トークン連結演算子

前述のサンプルにて、文字列の連結方法と「#」（文字列化演算子）による文字列化を説明しているが、C/C++言語には、もう一つ似たような演算子に「##」（トークン連結演算子）というものがあるので、違いを説明しておく。

トークン連結演算子「##」は、「プログラムコードとしての文字列」を連結するためのものである。これもマクロと組み合わせて使う事が多い。以下、具体的な使用例を示す。

例：トークン連結のサンプル

```
//幾つかの関数
void functionForTokenConcatenateTestAAA01(){ printf("TEST: AAA-01¥n"); }
void functionForTokenConcatenateTestAAA02(){ printf("TEST: AAA-02¥n"); }
void functionForTokenConcatenateTestBBB01(){ printf("TEST: BBB-01¥n"); }
void functionForTokenConcatenateTestBBB02(){ printf("TEST: BBB-02¥n"); }

//トークン連結演算子を使ったマクロ
#define TEST(ID, NO) functionForTokenConcatenateTest ## ID ## NO ()

//トークン連結テスト
int testConcatenateToken(const int argc, const char* argv[])
{
	TEST(AAA, 01);
	TEST(AAA, 02);
	TEST(BBB, 01);
	TEST(BBB, 02);
	return 0;
}
```

↓（実行結果）

```
TEST: AAA-01
TEST: AAA-02
TEST: BBB-01
TEST: BBB-02
```

この仕組みを活用して、関数を量産することもできる。上記のサンプルプログラムを修正し、関数定義もマクロを使用するようにしてみる。実行結果は同じ。

例：トークン連結で関数を定義するサンプル

```
//トークン連結演算子を使った関数定義マクロ
#define MAKE_TEST(ID, NO) ¥
	void functionForTokenConcatenateTest ## ID ## NO () ¥
	{ ¥
		printf("TEST: " #ID "-" #NO "¥n"); ¥
	}

//幾つかの関数
MAKE_TEST(AAA, 01)
MAKE_TEST(AAA, 02)
MAKE_TEST(BBB, 01)
MAKE_TEST(BBB, 02)

//トークン連結演算子を使った関数呼び出しマクロ
#define TEST(ID, NO) functionForTokenConcatenateTest ## ID ## NO ()

//トークン連結テスト
```

```
int testConcatenateToken(const int argc, const char* argv[])
{
	TEST(AAA, 01);
	TEST(AAA, 02);
	TEST(BBB, 01);
	TEST(BBB, 02);
	return 0;
}
```

ただし、この「関数を作るために使うマクロ」は、コーディングしにくくデバッグもしにくいのでできる限り行うべきではない。多くの場合はテンプレートで代用できる上、作業効率も良い。

あえてマクロで関数を定義するのは、主には下記のような場面である。

- __FILE__ や __LINE__ などの定義済みマクロを使用したい。
- コンパイル時に確実に定数化したい。（別紙の「効果的なテンプレートテクニック」にて詳述）

● プリプロセッサ

ここまでに説明した「文字列化」「トークン連結」は、「コメント除去」や「#include 展開」、「マクロ展開」などと同様に、「プリプロセッサ」で処理される。

「プリプロセッサ」とは「前処理」の事である。実際にコンパイルする前の前処理として、上記の処理が実行され、純粋なプログラムのテキストになったものをコンパイルするのである。

Visual C++ の場合、コンパイルの設定を変えて「プリプロセッサ」設定の「ファイルの前処理」を「はい」にすると、拡張子「.i」のファイルが出力されて、プリプロセッサの結果を確認することができる。

なお、この設定を適用すると、「.obj」ファイルが作成されず、ビルドできなくなるので注意。

例：「.i」ファイルのサンプルの一部

```
#line 1 "c:¥¥work¥¥test¥¥program¥¥c++¥¥rtti¥¥src¥¥main.cpp"
#line 1 "c:¥¥program files (x86)¥¥microsoft visual studio 12.0¥¥vc¥¥include¥¥stdio.h"

#pragma once

#line 1 "d:¥¥program files (x86)¥¥microsoft visual studio 12.0¥¥vc¥¥include¥¥crtdefs.h"int

testConcatenateStr(const int argc, const char* argv[])
{
	const char* func_line_info = getFileName2("c:¥¥work¥¥test¥¥program¥¥c++¥¥rtti¥¥src¥¥main.cpp" "(" "538"
"):" __FUNCSIG__, strlen("c:¥¥work¥¥test¥¥program¥¥c++¥¥rtti¥¥src¥¥main.cpp"));
	printf("MAKE_FUNC_LINE_INFO() = ¥"%s¥"¥n", func_line_info);
	return 0;
}

void functionForTokenConcatenateTestAAA01(){ printf("TEST: AAA-01¥n"); }
```

```
void functionForTokenConcatenateTestAAA02(){ printf("TEST: AAA-02¥n"); }
void functionForTokenConcatenateTestBBB01(){ printf("TEST: BBB-01¥n"); }
void functionForTokenConcatenateTestBBB02(){ printf("TEST: BBB-02¥n"); }

int testConcatenateToken(const int argc, const char* argv[])
{
	functionForTokenConcatenateTestAAA01 ();
	functionForTokenConcatenateTestAAA02 ();
	functionForTokenConcatenateTestBBB01 ();
	functionForTokenConcatenateTestBBB02 ();
	return 0;
}
```

● メタプログラミング：意識すべき問題点

マクロは、「#define CALC(x) (x + 2)」のように定義して、「a = CALC(b);」のように、一種の関数のように使う事ができる。この処理は、プリプロセッサによって「a = (b + 2);」という式に置き換わる。

また、「a = CALC(3);」のようにリテラル値が指定された場合、プリプロセッサによって「a = (3 + 2);」という式に置き換わると、コンパイル時に計算を済ませることができ、「a = 5;」という定数値に置き換わる。

このように、コンパイルの時点で（プログラムが実行可能になる前に）、プログラムを生成し、ある程度処理を済ませてしまう手法を「メタプログラミング」と呼ぶ。

マクロは一見して関数のように便利に使うことができるが、プリプロセッサの時点で「文字列が展開されているだけに過ぎない」という点には十分に注意が必要。以下、それが問題になるケースを示す。

例：問題になるマクロの使い方

```
//最大値判定マクロ
#define MAX(a, b) (a > b ? a : b)

//問題になるマクロの使い方テスト
int test (const int argc, const char* argv[])
{
	int x = MAX(1, 2);	//結果：x=2
	int y = MAX(4, 3);	//結果：y=3
	int z = MAX(++x, ++y); //結果：z=5（期待値）⇒ z=6（実際）
	printf("x=%d, y=%d, z=%d¥n", x, y, z);
	return 0;
}
```

↓（実行結果）

```
x=3, y=6, z=6
```

これは、プリプロセッサを通した時に、下記のようなプログラムに展開されるためである。

例：問題になるマクロの使い方

```
int test (const int argc, const char* argv[])
{
```

```
    int x = (1 > 2 ? 1, 2);
    int y = (4 > 3 ? 4 : 3);
    int z = (++x > ++y ? ++x, ++y);
    printf("x=%d, y=%d, z=%d¥n", x, y, z);
    return 0;
}
```

「MAX()」マクロへの引数に指定した「++」演算子がそのまま展開されていることが問題の原因である。

このように、マクロを使用する際は、「引数の中にヘタに式を書かないように」などの配慮が必要になる。

● メタプログラミング：マクロ以外の方法

上記のような問題は、マクロではなくテンプレートを使用することで解決できる。テンプレートもメタプログラミングのための強力な手段である。

テンプレートを使用したメタプログラミングの手法は、別紙の「効果的なテンプレートテクニック」にて詳述する。

▼ #pragma （プラグマ）の活用

「#pragma」（プラグマディレクティブ）は、コンパイラに対して、コンパイルのための何らかの指示・指定に使用する。例えば、「#pragma optimize(“ts” , on)」は、コンパイラに最適化手法を指定するためのプラグラマである。インクルードファイルを一回しかインクルードしないようにするための「#pragma once」などが特に有名。

プラグマは、一時的なコンパイルオプションの変更にとても効果的である。
例えば、通常はプログラムサイズが大きくなりすぎないように、「インライン関数の展開」を「規定」や「inline のみ」にしていところを、特にパフォーマンスが要求される特定のコードでのみ「拡張可能な関数すべて」に変えるといったことができる。

プラグマの内容・用法はコンパイラによって異なるため、各プラットフォーム向けの共通コードで利用するのは若干難しい。その回避策として、プラグマを#define マクロを使って「#define OPTIMIZE_ON #pragma optimize(“ts” , on)」のようなことをしようとすると、「#」が文字列化演算子とみなされてしまい、うまくいかない。「#define OPTIMIZE_ON optimize(“ts” , on)」「#pragma OPTIMIZE_ON」のようにするとうまくいくが、プラグマの指定自体は省略できなくなる。省略したい場合は、「message」などの何の影響もない無難なプラグマを指定しておくとよい。
なお、存在しないプラグマが指定されてもエラーにはならないため（警告にはなる）、警

告前提で複数プラットフォーム向けのプラグマを列挙するようなことは可能。
　どのようなプラグマがあるかは各プラットフォームで調べること。

【重要】プラグマを適用する際は、個人の判断で自由に行わず、開発プロジェクトの方針に従う必要があるため、よく相談してから使用しなければならない。

■ 分析に関する Tips

▼ 分析例：パフォーマンスカウンター

　ゲーム開発では、CPU 使用率をバーグラフや数値で画面にリアルタイム表示することが多い。単純にマイフレームリアルタイム表示しているだけだと、目視で「だいたいの状態」を確認できるだけであり、また、めまぐるしく細かく表示が変わるため、正確な数字もつかみにくい。

　そこで、リアルタイム表示とは別に、「１秒間間隔」や「５秒間隔」などで情報を表示すると、落ち着いて正確な情報を確認できる。

　その場合、複数の情報をひとまとめにした情報として表示する必要がある（１秒間隔なら３０フレーム分の CPU 使用率をまとめる）。
　では、どのような集計を行うのが効果的か、という点について、幾つかの考え方を示しながら説明する。

▼ 集計の考え方①：ログ分析

　より的確な分析を求めるなら、全ての情報をログ出力し、Excel にコピペして、Excel の集計やグラフの機能を使って分析することである。
　ただし、CPU 使用率のように、瞬間的に膨大な情報が出力されるようなものに対しては、この方法は現実的ではない。以降は、実機の処理で集計する方法について言及していく。

▼ 集計の考え方②：合計値／平均値

　「多数の情報を集計する」という要件に対して、すぐに思い当たる方法は、「合計値」と「平均値」の算出である。これなら実機上でも簡単に計算できる。
　ただし、本節で強調したいのは「情報の分析は目的に見合ったものでなければならない」ということである。

「CPU 使用率」の場合、「合計値」はほとんど無意味であることはすぐに理解できる。わざわざ画面に表示する必要もない情報である。ただ、この「合計値」を求めることで「平均値」を算出できるので、内部で集計する意義はある。

「CPU 使用率」の「平均値」はどれほどの意義がある情報であるだろうか？無意味ではないが、「最も重要な情報」ではない。

▼ 集計の考え方③：最大値（最小値）

「CPU 使用率」の場合、「最も重要な情報」は「最大値」である。

強調するが、これは「CPU 使用率の場合」である。「CPU 使用率」の監視で最も注目すべきなのは「最悪」の状態である。情報の分析では「何の情報が必要か」ということを強く意識しなければならいことを強調する。それを意識していないと、分析用の処理を書く手間も、集計に要する処理時間も、集計を記録するメモリも、全て無駄である。

「平均値」も無意味ではないが、一定間隔で CPU 使用率を表示するなら、まず「最大値」を表示することである。

毎フレームの処理の格差が激しいような場合は、「最小値」も算出して、その差を出してみるのも参考になる。

▼ 集計の考え方④：中央値（格差問題の対処）

「平均値」ではなく「中央値」という考え方がある。

例えば、「ある地域に住む人の一般的な所得」などには「平均値」ではなく「中央値」が使われる。対象が 100 人の場合、「平均値」は「全員の所得の合計」÷「人数」で算出されるが、「中央値」は「50 番目に多い人の所得」で算出される。これは、圧倒的格差の資産家が地域に一人二人いるだけで平均所得が引き上げられてしまうためである。

「CPU 使用率」の場合も格差問題がつきまとう。例えば、「仲間の AI キャラがターゲットを倒した瞬間、次のターゲットを索敵する処理がとんでもなく遅い」といったことがある。この場合、普段は落ち着いているのに、一瞬だけ CPU 使用率が跳ね上がる。

CPU 使用率を一定間隔で表示するのは、このような瞬間的な問題発生を見逃さないようにすることにも意義がある（あまりにひどい処理落ちは画面を見ていれば気がつけるが）。

「平均値」は、「最悪」の状態をわからなくさせてしまうだけではなく、「最悪以外の普段の状態」としてもあまりあてにならない。時折発生する格差の大きな結果に引っ張られずに普段の状態を計測するには、この「中央値」が適している。

しかし、中央値を算出するには、全ての状態を記憶し、かつ、ソートも必要であるため、

リアルタイムな処理に適用するのははばかられる。

▼ 集計の考え方⑤：しきい値（あるいは閾値）

「中央値」に近い分析を、少ないメモリと処理量で実現するために、「しきい値」を活用すると良い。（ちなみに「閾値」の本来の読み方は「いきち」で、「しきいち」と読むのは慣用読みとの事）

仮に「CPU 使用率」の「しきい値」を「100%」として、「しきい値未満の平均」を「中央値」の代わりとして扱うといった用法である。また、「しきい値未満時の平均」と「しきい値以上時の平均」の差を取ると、平均的な処理の格差を確認することができる。

しきい値の活用は、CPU 使用率の分析にとても効果的である。

例えば、「1 秒間に処理できたのが 25 フレームで、しきい値（100%）を超えた回数が 7 回」という計測結果から、「7 / 25 = 28%の割合で処理落ちが発生」ということができる（この例の場合、7 回中 2 回は処理落ちの次のフレームの処理が軽く、遅れを取り戻している）。ちなみに、1 秒間のフレームの達成率は 25 / 30 = 83% である。

先の「28%」は、問題発生の「頻度」として見ることができ、処理落ちの原因となる処理のヒントになりえる。

また、時にはこの「しきい値」を「90～95%」ぐらいに下げて、「普段どれほどぎりぎりの状態で動作しているか」といったことを確認するのも有効。

更に、こうした分析を、いつかの処理系ごとに分けて行えるようにしておくと、問題の範囲を絞るのに役立つ。例えば、AI 系の処理だけの CPU 使用率を計測し、しきい値を 3% にして問題の発生頻度を確認するといった具合である。

こうした問題発生箇所を絞り込みやすい仕組みを作っておくと、「プロファイラ」（パフォーマンスアナライザー）を活用する際にも、範囲を特定できて扱い易い。

▼ 集計の考え方⑥：効果的なログ分析

「CPU 使用率」も、上記のような集計結果であれば、ログ出力して長期的な状態の推移を分析するのも悪くない。例えば、「10 秒ごとの集計を出力」であれば十分に現実的である。

このような対応を行うにあたっては、ログ出力のために、以下のような準備をしておくと良い。

- 特定のログ出力の ON/OFF をゲーム中に切り替えできる仕組みの実装。
- 集計したいログは、行頭に特定のタグをつけるなどして選別しやすくする。

・ 出力されるログの数値部をカンマやタブで区切り、Excel にコピペしやすいようにしておく。

以下、ログのサンプルを示す。

例：ログのサンプル：[CPU] のタグが付いたログが CPU 使用率のログ

```
[CPU], No, Frames, Overed, OverRate, Max, Min, OverAvg, UnderAvg
[CPU], 1, 25, 7, 128.00 %, 150.23 %, 72.54 %, 140.50 %, 85.56 %
他の処理のログ...
[CPU], 2, 30, 0, 100.00 %, 98.52 %, 76.78 %, 100.00 %, 81.54 %
他の処理のログ...
他の処理のログ...
他の処理のログ...
[CPU], 3, 29, 1, 103.45 %, 128.50 %, 69.45 %, 128.50 %, 78.51 %
[CPU], 4, 23, 8, 134.78 %, 160.45 %, 79.56 %, 152.30 %, 84.56 %
他の処理のログ...
[CPU], 5, 28, 2, 107.14 %, 135.45 %, 82.12 %, 128.36 %, 87.48 %
他の処理のログ...
他の処理のログ...
他の処理のログ...
他の処理のログ...
[CPU], 6, 30, 0, 100.00 %, 95.48 %, 75.46 %, 100.00 %, 79.45 %
[CPU], 7, 27, 3, 111.11 %, 124.78 %, 81.47 %, 120.85 %, 85.45 %
[CPU], 8, 26, 5, 119.23 %, 148.78 %, 72.69 %, 135.48 %, 84.36 %
他の処理のログ...
他の処理のログ...
他の処理のログ...
[CPU], 9, 24, 8, 133.33 %, 172.58 %, 68.97 %, 145.36 %, 80.56 %
他の処理のログ...
[CPU], 10, 27, 4, 114.81 %, 137.98 %, 77.50 %, 125.36 %, 81.69 %
他の処理のログ...
他の処理のログ...
他の処理のログ...
```

ログを選別する際には、「awk」（オーク）のような古典的なツールも役立つ。以下、上記のログを「awk」で選別するサンプルを示す。

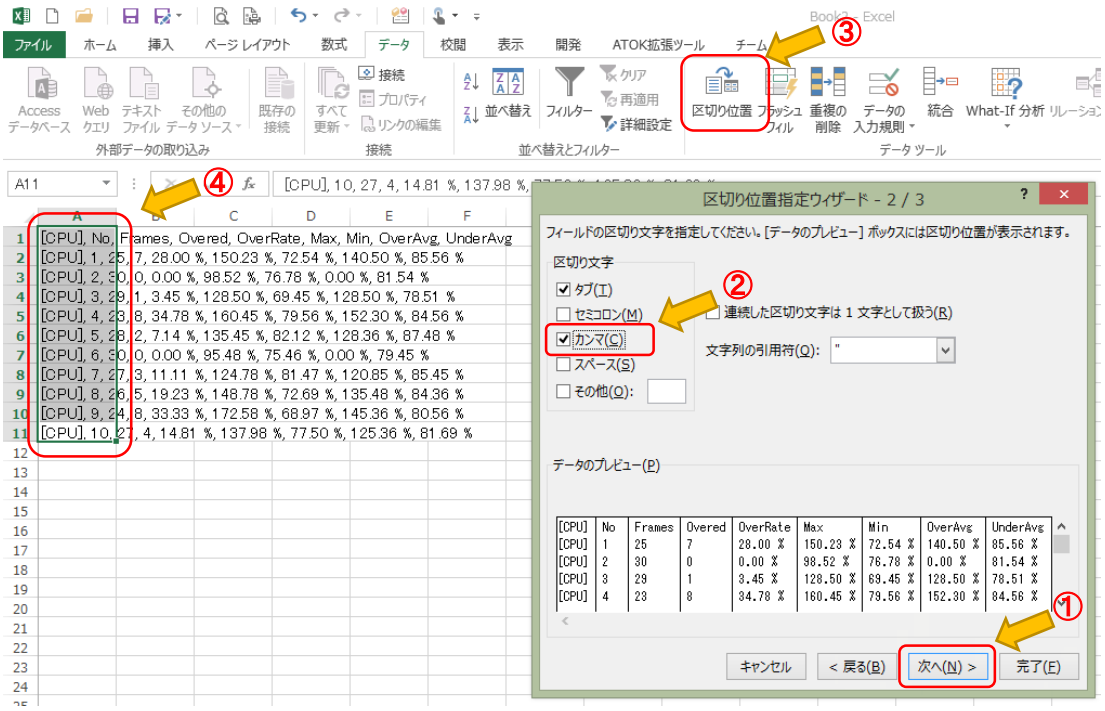
例：awk によるログ選別サンプル　※ログは log.txt に記録されているものとする

```
$ cat log.txt | awk '/^¥[CPU¥]/{print $0;}'  ←パイプでログを渡して、正規表現でマッチした行を出力
[CPU], No, Frames, Overed, OverRate, Max, Min, OverAvg, UnderAvg
[CPU], 1, 25, 7, 128.00 %, 150.23 %, 72.54 %, 140.50 %, 85.56 %
[CPU], 2, 30, 0, 100.00 %, 98.52 %, 76.78 %, 100.00 %, 81.54 %
[CPU], 3, 29, 1, 103.45 %, 128.50 %, 69.45 %, 128.50 %, 78.51 %
[CPU], 4, 23, 8, 134.78 %, 160.45 %, 79.56 %, 152.30 %, 84.56 %
[CPU], 5, 28, 2, 107.14 %, 135.45 %, 82.12 %, 128.36 %, 87.48 %
[CPU], 6, 30, 0, 100.00 %, 95.48 %, 75.46 %, 100.00 %, 79.45 %
[CPU], 7, 27, 3, 111.11 %, 124.78 %, 81.47 %, 120.85 %, 85.45 %
[CPU], 8, 26, 5, 119.23 %, 148.78 %, 72.69 %, 135.48 %, 84.36 %
[CPU], 9, 24, 8, 133.33 %, 172.58 %, 68.97 %, 145.36 %, 80.56 %
[CPU], 10, 27, 4, 114.81 %, 137.98 %, 77.50 %, 125.36 %, 81.69 %
```

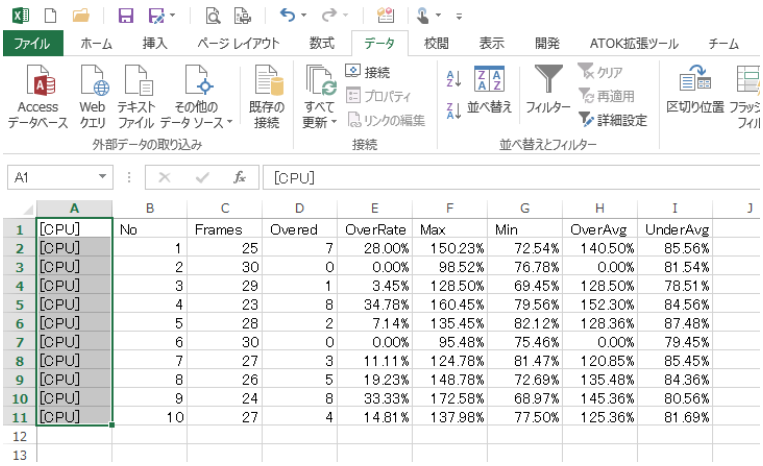
awk は元々Unix 系 OS で一般的なコマンドである。Windows 用のコマンドも探せばあるようだが、Cygwin のような Unix エミュレータ環境があればすぐに使うことができる。

このようなログを Excel にコピペしてグラフ化するサンプルを示す。

例：Excel にコピペし、カンマ区切りで各セルに情報を展開

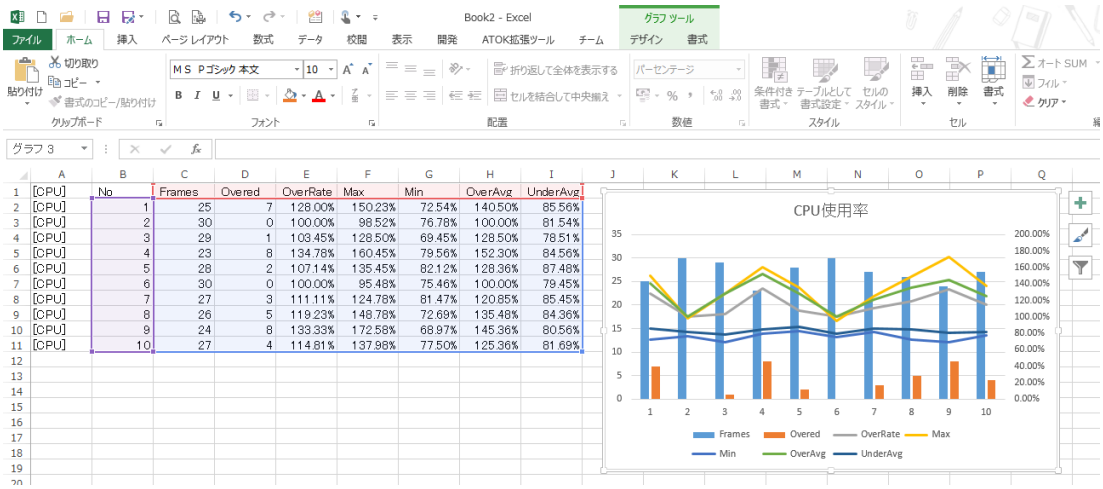


例：Excel にデータが展開できた状態



	A	B	C	D	E	F	G	H	I
1	[CPU]	No	Frames	Overed	OverRate	Max	Min	OverAvg	UnderAvg
2	[CPU]	1	25	7	28.00%	150.23%	72.54%	140.50%	85.56%
3	[CPU]	2	30	0	0.00%	98.52%	76.78%	0.00%	81.54%
4	[CPU]	3	29	1	3.45%	128.50%	69.45%	128.50%	78.51%
5	[CPU]	4	23	8	34.78%	160.45%	79.56%	152.30%	84.56%
6	[CPU]	5	28	2	7.14%	135.45%	82.12%	128.36%	87.48%
7	[CPU]	6	30	0	0.00%	95.48%	75.46%	0.00%	79.45%
8	[CPU]	7	27	3	11.11%	124.78%	81.47%	120.85%	85.45%
9	[CPU]	8	26	5	19.23%	148.78%	72.69%	135.48%	84.36%
10	[CPU]	9	24	8	33.33%	172.58%	68.97%	145.36%	80.56%
11	[CPU]	10	27	4	14.81%	137.98%	77.50%	125.36%	81.69%

例：グラフ化



No	Frames	Overed	OverRate	Max	Min	OverAvg	UnderAvg
[CPU] 1	25	7	128.00%	150.23%	72.54%	140.50%	85.56%
[CPU] 2	30	0	100.00%	98.52%	76.78%	100.00%	81.54%
[CPU] 3	29	1	103.45%	128.50%	69.45%	128.50%	78.51%
[CPU] 4	23	8	134.78%	160.45%	79.56%	152.30%	84.56%
[CPU] 5	28	2	107.14%	135.45%	82.12%	128.36%	87.48%
[CPU] 6	30	0	100.00%	95.48%	75.46%	100.00%	79.45%
[CPU] 7	27	3	111.11%	124.78%	81.47%	120.85%	85.45%
[CPU] 8	26	5	119.23%	148.78%	72.69%	135.48%	84.36%
[CPU] 9	24	8	133.33%	172.58%	68.97%	145.36%	80.56%
[CPU] 10	27	4	114.81%	137.98%	77.50%	125.36%	81.69%

■ 処理に関する Tips

▼ 「時間」ベースの処理で汎用化（サンプル：フェード処理）

「フェード処理」は、ゲームプログラミングにおいてとても一般的な処理である。画面暗転のフェードイン／アウト、BGM のフェードイン／アウトなど、頻繁に使用される。

このフェード処理をサンプルにして、「モーション」や「移動」などのアニメーション全般を含む、「時間変化のある処理」に対する考え方を提示する。

こうした「考え方」の統一は、開発プロジェクトを無難に進行するためにも、とても重要なことである。

● フレームベースの処理

よくあるフェード処理は、「Fadeout(30);」（30 フレームでフェードアウト）、「Fadein(15)」（15 フレームでフェードイン）といった「フレームベース」の書き方である。更にサウンド系の処理では「BGMFade(15, 1.0, 0.5);」（15 フレームで BGM の音量を最大から半分に）といった、完全にフェードイン／アウトする以外の指定のしかたもする。

「フレームベース」はプログラマーにとって計算が直感的で楽である。しかし、開発プロジェクトによって「フレーム」の扱いはまちまちで、スクリプトやデータを作成する側からしたら、「フレームベース」は開発プロジェクトの「ローカルルール」として扱いにくいものとなる。

このことは、「フェード」に限らず、「移動」や「モーション」なども含む「アニメーション全般」に言えることである。

● 時間（秒）ベースの処理

本節では、処理を分かり易く解説するために「フェード」を取り上げているが、「時間を指定する要素全般に対して、『フレーム』ではなく『秒』で指定することを規定する」ということを提唱する。

このような取り決めをプログラマー全員はもとより、制作スタッフ全員が周知することにより、開発が少しでもスムーズになる。

【補足】前述の「フレームの扱いが開発プロジェクトでまちまち」と説明した件について補足する。
これは当然ゲームのフレームレートが 30fps か 60fps かに起因する話である。また、ゲームによっては、状況に応じて 30fps と 60fps を切り替えるものもある。この場合、60fps 稼働時は 1 フレームのカウントを 0.5fps として 30fps 基準に合わせるなどして混乱を回避することになる。
いずれにせよ、内部処理はともかく、少なくとも外部から与えられる時間の指定方法を「秒」に統一すると、統一的で、スクリプトやデータを作成する上での混乱がない。

● サンプル：フェード処理

フェード処理の処理要件としては、「フェードの途中状態からの変更」を考慮し、「変化の割合が常に一定となる」ように処理を組むべきである。特にサウンド系の処理では意識した方が良い。

以下、具体例を示す。

処理①：主人公キャラが屋内に移動。この時、「FadeBGM(1.0, 1.0, 0.5);」が指定され、BGM の音量を 1.0 秒かけて最大（1.0）から半分（0.5）までフェードする。

処理②：主人公キャラはわずか 0.5 秒ですぐに屋外に移動。音量が 0.75 までフェードしている段階だった。この時、「FadeBGM(1.0, 0.5, 1.0);」の指定により、BGM の音量を 1.0 秒かけて半分（0.5）から最大（1.0）までフェード。

このフェードが指定された時点で、実際の音量は 0.75 であるため、指定の「変化量」に基づき、0.5 秒で最大音量にフェードする。

【注】この時にやっては行けない処理：

・フェード開始時に、強制的に開始値（0.5）にしてしまうこと。

・現在地の 0.75 から 1.0 までのフェードを、1.0 秒かけて行うこと。

以上のように、フェード処理に与えられたパラーメータは、「ゴール」と「ゴールへの時間当たりの変化量」と解釈して処理する。

具体的な処理例を示す。

例：フェード処理のサンプル

```
//符号判定処理
int fsign(float val)
{
	return val < 0.f ? -1 : val > 0.f ? 1 : 0;
}

//フェード処理
class CFade
{
public:
	//ゴールに達しているか？
	bool isGoal() const { return m_now == m_goal; }
	//更新処理
	void update(float elapsed_time)
	{
		if (!isGoal())
		{
			int sign_before = fsign(m_now - m_goal);
			m_now += (m_gradual * elapsed_time);
			int sign_after = fsign(m_now - m_goal);
			if (sign_before != sign_after)
				m_now = m_goal;
		}
	}
	//フェード開始
	void start(float time, float begin, float end)
	{
		m_goal = end;
		m_gradual = (end - begin) / time;
	}
	//現在の状態を表示
	void debugPrint(int flame)
	{
		printf("%2d: now=%.3f, goal=%.3f, gradual=%.3f¥n", flame, m_now, m_goal, m_gradual);
	}
public:
	//コンストラクタ
	CFade(float now) :
		m_now(now),
		m_goal(now),
		m_gradual(0.f)
	{}
private:
	//フィールド
	float m_now;	//現在の状態
	float m_goal;	//ゴール
	float m_gradual;//1 秒あたりの変化量
};

//フェードテスト
void test()
```

```
{
	//30 フレーム想定で、1 フレーム当たりの経過時間
	//※このサンプルでは固定するが、本来は処理落ちの状況に応じて変動する（処理落ちがなければ固定される）
	const float elapsed_time = 1.f / 30.f;
	printf("elapsed_time=%.3f¥n", elapsed_time);
	int flame = 0;

	//フェードオブジェクトを初期化
	CFade fade(1.f);

	//フェード開始
	fade.start(1.5f, 1.f, 0.5f);//1.5 秒かけて、最大(1.0) から半分 (0.5) までフェード
	fade.debugPrint(flame++);

	//20 フレームだけ更新
	for (int i = 0; i < 20; ++i)
	{
		fade.update(elapsed_time);//更新処理
		fade.debugPrint(flame++);
	}

	//フェードを途中変更
	fade.start(1.5f, 0.5f, 1.f);//1.5 秒かけて、半分 (0.5) から最大(1.0) までフェード

	//フェード完了まで更新
	while (!fade.isGoal())
	{
		fade.update(elapsed_time);//更新処理
		fade.debugPrint(flame++);
	}
}
```

↓（実行結果）

```
elapsed_time=0.033
 0: now=1.000, goal=0.500, gradual=-0.333, isGoal()=0
 1: now=0.989, goal=0.500, gradual=-0.333, isGoal()=0
 2: now=0.978, goal=0.500, gradual=-0.333, isGoal()=0
 3: now=0.967, goal=0.500, gradual=-0.333, isGoal()=0
 4: now=0.956, goal=0.500, gradual=-0.333, isGoal()=0
 5: now=0.944, goal=0.500, gradual=-0.333, isGoal()=0
 6: now=0.933, goal=0.500, gradual=-0.333, isGoal()=0
 7: now=0.922, goal=0.500, gradual=-0.333, isGoal()=0
 8: now=0.911, goal=0.500, gradual=-0.333, isGoal()=0
 9: now=0.900, goal=0.500, gradual=-0.333, isGoal()=0
10: now=0.889, goal=0.500, gradual=-0.333, isGoal()=0
11: now=0.878, goal=0.500, gradual=-0.333, isGoal()=0
12: now=0.867, goal=0.500, gradual=-0.333, isGoal()=0
13: now=0.856, goal=0.500, gradual=-0.333, isGoal()=0
14: now=0.844, goal=0.500, gradual=-0.333, isGoal()=0
15: now=0.833, goal=0.500, gradual=-0.333, isGoal()=0
16: now=0.822, goal=0.500, gradual=-0.333, isGoal()=0
17: now=0.811, goal=0.500, gradual=-0.333, isGoal()=0
18: now=0.800, goal=0.500, gradual=-0.333, isGoal()=0
19: now=0.789, goal=0.500, gradual=-0.333, isGoal()=0
20: now=0.778, goal=0.500, gradual=-0.333, isGoal()=0
21: now=0.789, goal=1.000, gradual=0.333, isGoal()=0  ←ここでフェードの指定が途中変更されている
22: now=0.800, goal=1.000, gradual=0.333, isGoal()=0
23: now=0.811, goal=1.000, gradual=0.333, isGoal()=0
24: now=0.822, goal=1.000, gradual=0.333, isGoal()=0
25: now=0.833, goal=1.000, gradual=0.333, isGoal()=0
26: now=0.844, goal=1.000, gradual=0.333, isGoal()=0
27: now=0.856, goal=1.000, gradual=0.333, isGoal()=0
28: now=0.867, goal=1.000, gradual=0.333, isGoal()=0
29: now=0.878, goal=1.000, gradual=0.333, isGoal()=0
30: now=0.889, goal=1.000, gradual=0.333, isGoal()=0
```

```
31: now=0.900, goal=1.000, gradual=0.333, isGoal()=0
32: now=0.911, goal=1.000, gradual=0.333, isGoal()=0
33: now=0.922, goal=1.000, gradual=0.333, isGoal()=0
34: now=0.933, goal=1.000, gradual=0.333, isGoal()=0
35: now=0.944, goal=1.000, gradual=0.333, isGoal()=0
36: now=0.956, goal=1.000, gradual=0.333, isGoal()=0
37: now=0.967, goal=1.000, gradual=0.333, isGoal()=0
38: now=0.978, goal=1.000, gradual=0.333, isGoal()=0
39: now=0.989, goal=1.000, gradual=0.333, isGoal()=0
40: now=1.000, goal=1.000, gradual=0.333, isGoal()=1  ←同じ時間をかけてゴールに到達したので、
                                                      途中スタートしたフェードの時間進行が一定であったことが
                                                      わかる
```

■ 構造に関する Tips

▼ ゼロオーバーヘッドの原則

C++言語仕様の設計ルールには「ゼロオーバーヘッドの原則」というものがある。これは、多数の仕様を包括するゲームシステムにおいても重視すべきことである。

● C++言語における「ゼロオーバーヘッドの原則」

ゼロオーバーヘッドの原則:「C++言語は利用しない機能についてはオーバーヘッドが生じない」

要するに、機能の追加・変更が行われても、それを使用しない限りは、他の機能のパフォーマンスとメモリ使用量は保たれることを意味する。

一見当たり前のようなことではあるが、これは極めて意識すべきことである。

● 【実例】ゼロオーバーヘッドの原則：virtual メンバー関数

実例として、「virtual メンバー関数」の挙動を示す。

virtual メンバー関数は、解説するまでもなく、サブクラスでオーバーライドしたメンバー関数を、親クラスとして実行する仕組みである。

```
//親クラス
class CBase
{
public:
	void func1() { printf("[func1] This is CBase.¥n"); }
	virtual void func2() { printf("[func2] This is CBase.¥n"); }
	virtual void func3() { printf("[func3] This is CBase.¥n"); }
private:
	int m_var1;
};
//サブクラス
class CDerived : public CBase
```

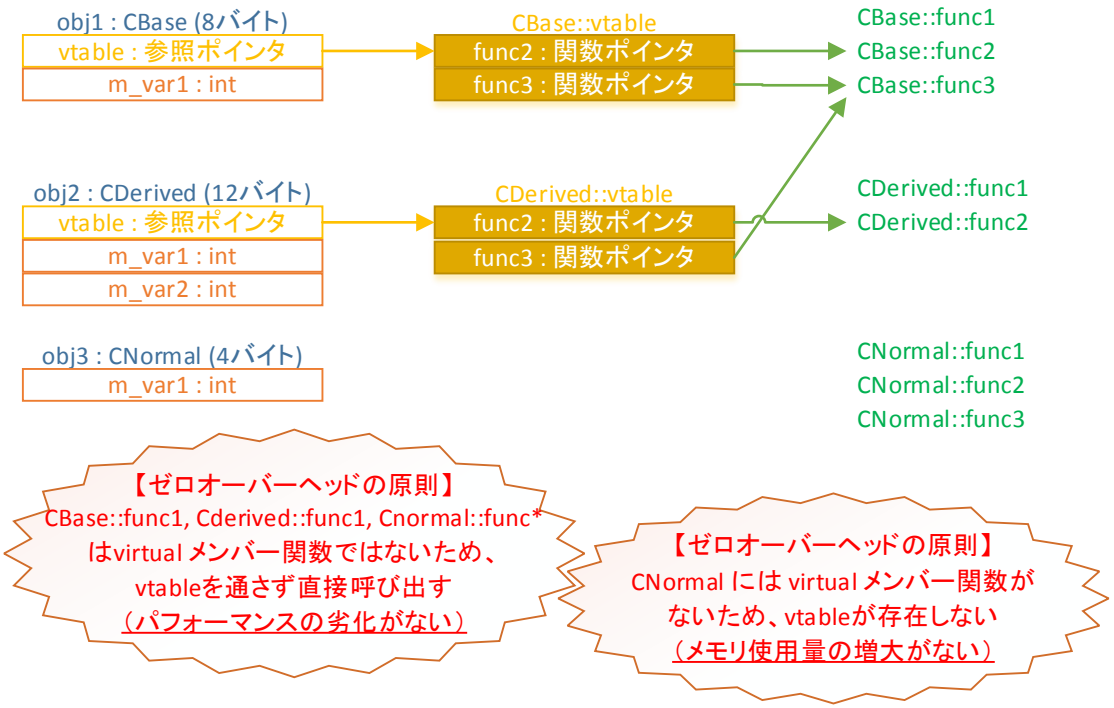
```
{
public:
	void func1() { printf("[func1] This is CDerived.¥n"); }
	void func2() override { printf("[func2] This is CDerived.¥n"); }
private:
	int m_var2;
};
//通常クラス
class CNormal
{
public:
	void func1() { printf("[func1] This is CNormal.¥n"); }
	void func2() { printf("[func2] This is CNormal.¥n"); }
	void func3() { printf("[func3] This is CNormal.¥n"); }
private:
	int m_var1;
};
//実行
int main(cons tint argc, const char* argv[])
{
	CBase obj1 = new CBase;
	CBase obj2 = new CDerived;
	CNormal obj3 = new CNormal;
	obj1->func1();//通常メンバー関数
	obj1->func2();//仮想メンバー関数
	obj1->func3();//仮想メンバー関数
	obj2->func1();//通常メンバー関数
	obj2->func2();//仮想メンバー関数
	obj2->func3();//仮想メンバー関数
	obj3->func1();
	obj3->func2();
	obj3->func3();
	return EXIT _SUCCESS;
};
```

↓（実行結果）

```
[func1] This is CBase.
[func2] This is CBase.
[func3] This is CBase.
[func1] This is CBase.
[func2] This is CDerived.
[func3] This is CBase.
[func1] This is CNormal.
[func2] This is CNormal.
[func3] This is CNormal.
```

これは、vtable という関数ポインタのテーブルを通して実行することで実現している。メンバー関数を呼び出す際に、直接関数を呼び出すのではなく、最初にインスタンスの vtable を参照して関数ポインタを取得してから呼び出す仕組みである。これにより、実行速度が低下する。仕組み上、インライン展開もできない。

しかし、virtual を使用していない関数の呼び出しは vtable を通さず直接呼び出しをおこなうため、パフォーマンスの劣化がない。また、virtual が一つもないクラスは vtable 自体存在しないため、メモリ使用量の増大がない。



● ゲーム開発における「ゼロオーバーヘッドの原則」

ゲームでの機能追加・変更においても、極力この「ゼロオーバーヘッドの原則」を意識すべきである。具体的には下記のようなことを常日頃心がけるようにする。

- **シーンに無関係の処理は極力実行しない**

  例えば、ミニゲームの追加実装により、通常ゲームのパフォーマンスが劣化するようなことがあってはならない。

  ミニゲームのような処理は完全に局所化すべきものではあるが、ミニゲームのために必要な処理を汎用処理に追加する必要性も生じる。そのような処理が通常ゲームに影響しないように注意しなければならない。

- **汎用構造体／クラスへの安易なメンバー追加を行わない**

  例えば、ミニゲームのような局所的な用途に必要な情報・処理を、汎用構造体／クラスに追加することはできるだけ避けるべきである。

  また、この対処として「継承」を用いるのも安易である。プールメモリのサイズ変更や広範囲な共通処理の変更といった対応が必要になる可能性がある。

  局所的な情報や処理は、できるだけ独立した構造体／クラスで構成し、汎用構造体／クラスはなるべくそのままの形で使用するようにする。このため、一つのゲームオブジェクトのための情報が複数のオブジェクトで構成される形になる可能性があるが、結果的にその方が整然として問題を起こしにくい。

- **virtual メンバー関数の追加は可能な限り行わない**

  とくにそれまで virtual メンバー関数が一つもない構造体／クラスに追加する時は慎重に対応すべきである。vtable の追加によるサイズの増大も発生するためである。
  virtual メンバー関数の追加はパフォーマンスの劣化も招くため、本当に多態性が必要なのかどうか、それが静的な多態性（コンパイル時の多態性）によって実現可能かなどもよく考えて対応する。
  静的な多態性については、別紙の「効果的なテンプレートテクニック」の「CRTP（テンプレートメソッドパターン）」を参照。

- **シーンに登場しないオブジェクトは極力処理しない**

  「ゼロオーバーヘッドの原則」とはやや趣向が異なるが、シーンに登場しないオブジェクトは極力処理すべきではない。
  例えば、シーン上のアクティブなオブジェクトが 0 なら一切処理しないことや、十分に遠いオブジェクトはアンメーションやサウンドを処理しないなどである。この時、「地面を動かす」などのシーンに影響するアニメーションでないことや、近づいたらループ音が聞こえなければならないといったことを考慮した上で、不要な処理を省くようにする。

■ 最適化に関する Tips

最後に、本書標題が示す「本当にちょっとした」という枠を超えた Tips を示す。
やや高度な知識に基づくが、やるべきことは「ちょっとした」留意で実践できることである。

▼ SIMD

▼ 命令パイプライン

▼ CPU キャッシュ効率を考慮した処理：キャッシュの概要

「インストラクションキャッシュ」は、「実行コード」（プログラム自体）のキャッシュ

である。CPU のキャッシュメモリ（L1 キャッシュ）は、「インストラクションキャッシュ」と「データキャッシュ」を別々に管理する。

▼ CPU キャッシュ効率を考慮した処理①：インストラクションキャッシュ

「インストラクションキャッシュ」は、「実行コード」（プログラム自体）のキャッシュである。CPU のキャッシュメモリ（L1 キャッシュ）は、「インストラクションキャッシュ」と「データキャッシュ」を別々に管理する。

【参考】

▼ CPU キャッシュ効率を考慮した処理②：データキャッシュ

■■以上■■

■ 索引

索引項目が見つかりません。

本当にちょっとしたプログラミング Tips

以　上