

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 16 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 16 日	板垣 衛	「構造に関する Tips」「最適化に関する Tips」を追加。 「CPU キャッシュに関する最適化」と「命令パイプラインに関する最適化」は準備中。

■ 目次

■ 概略	1
■ 目的	1
■ コーディングに関する Tips	1
▼ 配列の初期化	1
▼ #define マクロの活用	2
● 便利な「定義済みマクロ」	2
● 定義済みマクロのデバッグ活用：アサーション	6
● 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	7
● コードの連結：トークン連結演算子	10
● プリプロセッサ	11
● メタプログラミング：意識すべき問題点	12
● メタプログラミング：マクロ以外の方法	13
▼ #pragma（プラグマ）の活用	13
■ 分析に関する Tips	14
▼ 分析例：パフォーマンスカウンター	14
▼ 集計の考え方①：ログ分析	14
▼ 集計の考え方②：合計値／平均値	14
▼ 集計の考え方③：最大値（最小値）	15
▼ 集計の考え方④：中央値（格差問題の対処）	15
▼ 集計の考え方⑤：しきい値（あるいは閾値）	16
▼ 集計の考え方⑥：効果的なログ分析	16
■ 処理に関する Tips	19
▼ 「時間」ベースの処理で汎用化（サンプル：フェード処理）	19
● フレームベースの処理	19
● 時間（秒）ベースの処理	20
● サンプル：フェード処理	20
■ 構造に関する Tips	23
▼ ゼロオーバーヘッドの原則	23
● C++言語における「ゼロオーバーヘッドの原則」	23
● 【実例】ゼロオーバーヘッドの原則：virtual メンバー関数	23

● ゲーム開発における「ゼロオーバーヘッドの原則」	25
<hr/>	
■ 最適化に関する Tips	26
▼ SIMD 演算の活用	26
● SIMD 演算とは	27
● 代表的な SIMD 拡張命令セット	27
● SIMD 演算サンプル	29
● 最適化（高速化）のポイント	58
▼ CPU キャッシュメモリに関する最適化	59
● 【基礎】 CPU キャッシュメモリの構成	59
● 【基礎】 メインメモリと L1 キャッシュ～L3 キャッシュメモリ	59
● 【基礎】 インストラクションキャッシュとデータキャッシュ	59
● 【基礎】 キャッシュライトバックとキャッシュライトスルー	59
● 【基礎】 キャッシュコヒーレンシとバススヌーピング	59
● 最適化①：インストラクションキャッシュに関する最適化	59
● 最適化②：データキャッシュに関する最適化	59
<hr/>	
■ 高度な最適化に関する Tips	59
▼ 命令パイプラインに関する最適化	60
● 【基礎】 CPU の命令とサイクル	60
● 【基礎】 命令パイプラインとスーパースカラー	60
● 【基礎】 アウト・オブ・オーダー実行	60
● 【基礎】 同時マルチスレッディング（SMT = Simultaneous MultiThreading） ...	60
● 【基礎】 ハザードとストール	60
● 最適化：ストールの回避策	60

■ 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グラ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 Studio 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。
「Mmmbbb」
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。
「bbb」はビルド番号です。

- `__cplusplus_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。
- `__cplusplus_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__` はブ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` `/ZI` 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("__FUNCNAME__   = %s\n", __FUNCNAME__);
    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 (__FUNCNAME__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
__FUNCNAME__   = "?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、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() getFileLine2(__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() getFileLine2(__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* getFileLine2(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() getFileLine2(__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

ここまでで説明した「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「++」演算子がある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っても効果的である。

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](#)」のようになるとうまくいくが、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いよい。

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

よりの確な分析を求めるなら、全ての情報をログ出力し、Excel にコピーして、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ないことを強調する。**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かまとう。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 使用率の分析にとっても効果的である。

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「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 にコピー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...
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。

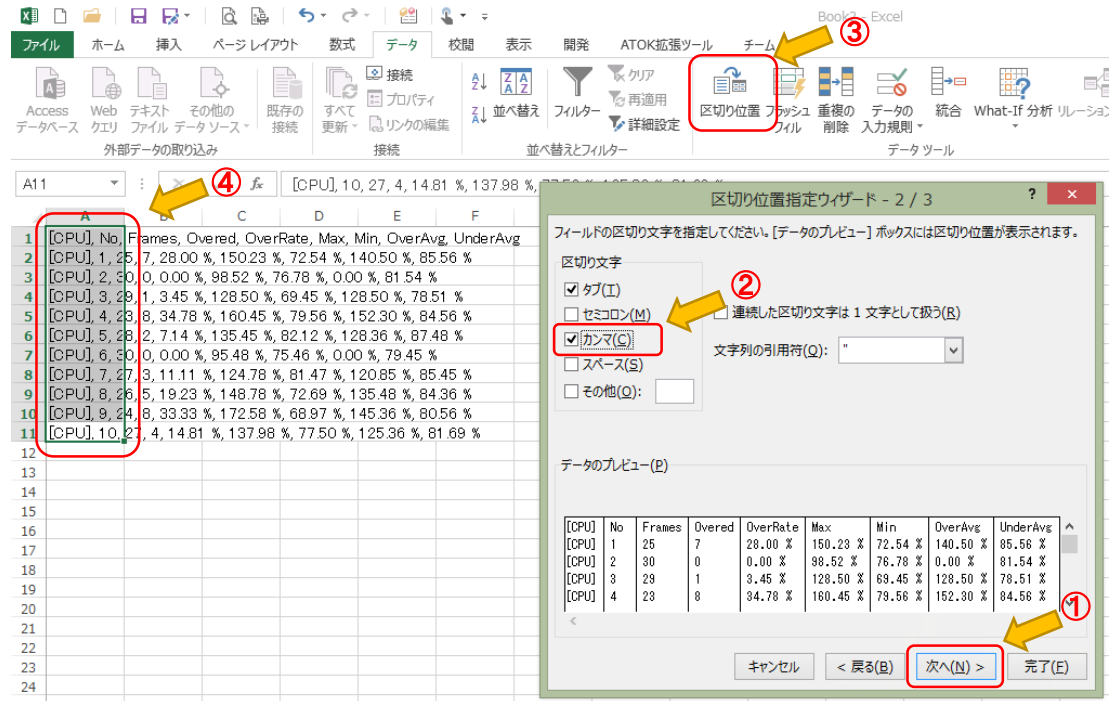
例：awk によるログ選別サンプル ※ログは log.txt に記録されているものとする

```
$ 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 にコピーしてグラフ化するサンプルを示す。

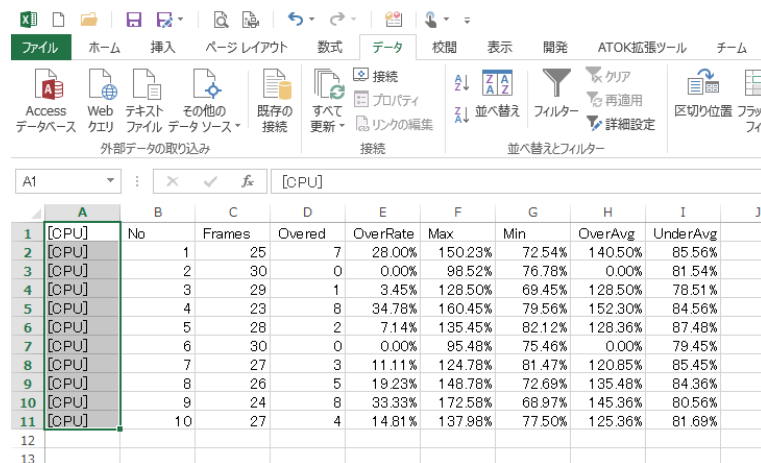
例：Excel にコピーし、カンマ区切りで各セルに情報を展開



The screenshot shows the Excel 'データ' (Data) tab with the '区切り位置指定ウィザード' (Text Import Wizard) dialog box open. The dialog is at step 2 of 3. The '区切り文字' (List separator) option is selected, and 'カンマ' (Comma) is chosen as the delimiter. The preview table shows the following data:

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

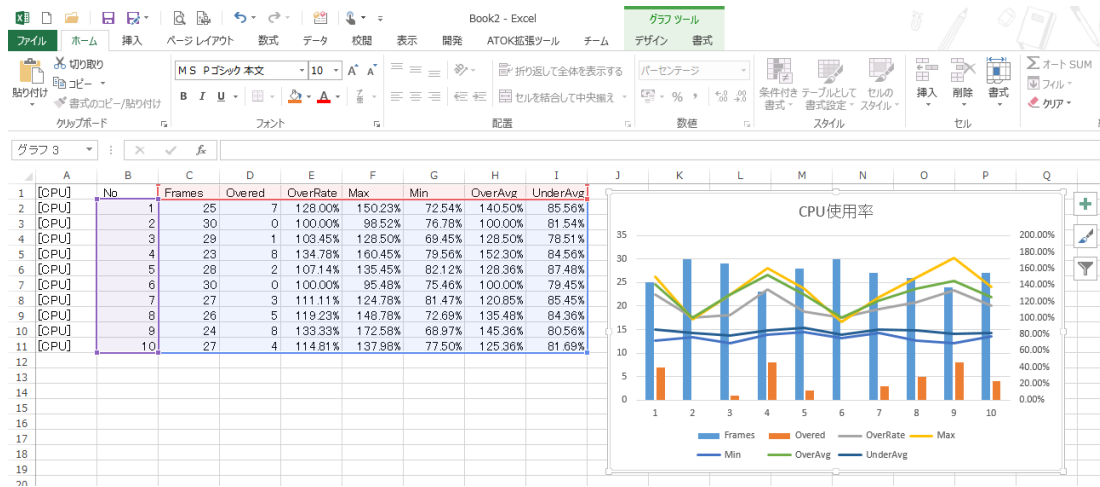
例：Excel にデータが展開できた状態



The screenshot shows the Excel spreadsheet with the data from the log file imported and expanded. The data is organized into columns: A (CPU), B (No), C (Frames), D (Overed), E (OverRate), F (Max), G (Min), H (OverAvg), and I (UnderAvg). The first column (A) contains the CPU identifiers, and the subsequent columns contain the corresponding performance metrics.

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

例：グラフ化



■ 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、構造体／クラスの派生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。

virtual メンバー関数を用いると、構造体のデータサイズと関数呼び出しにオーバーヘッドが生じるが、使わない限りは全く影響がない。

```

//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(const int 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(); //通常メンバー関数
    obj3->func2(); //通常メンバー関数
    obj3->func3(); //通常メンバー関数
    return EXIT_SUCCESS;
};

```

↓ (実行結果)

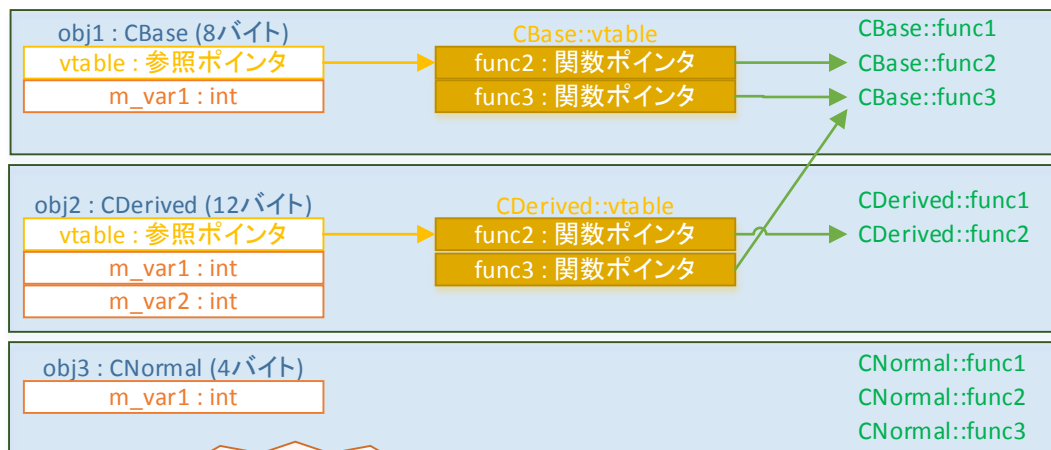
```

[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。



【ゼロオーバーヘッドの原則】
CBase::func1, CDerived::func1, CNormal::func*
はvirtual メンバー関数ではないため、
vtableを通さず直接呼び出す
(パフォーマンスの劣化がない)

【ゼロオーバーヘッドの原則】
CNormal には virtual メンバー関数が
ないため、vtableが存在しない
(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」、「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」などの一手間を加えて最適化する。

なお、いくら遠くのオブジェクトでも、「アニメーションの連鎖で地面全体を動かしている」や「遠方でもある程度の範囲内なら攻撃行動を実行出来なければならない」、「遠方なら可視状態でも SE が聞こえないが、近づいたらループ音が聞こえない」といった、シーンへの影響を十分考慮した処理を行う必要がある。例えば、近くにいるキャラはカメラの範囲外だからといってアニメーションを止めるような最適化を行う場合、「毎フレームバウンディングボックスを再計算」といった処理を伴わないとうまくいかず（唐突にキャラが消えたり出現したりする）、また、アニメーションのタイミングをみてヒット処理や弾の発射、エフェクト発生などの処理を行っている場合、アニメーションを止めるわけにはいかない。

■ 最適化に関する Tips

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 演算の活用

「SIMD 演算」の活用は、劇的なパフォーマンスの改善をもたらすことが期待できる。

● SIMD 演算とは

「SIMD」とは、「Single Instruction Multi Data」の略で、「一つの命令を複数のデータに並列で適用すること」を意味する。これに対して、通常の「一つの命令を一つのデータに適用すること」を「SISD」(Single Instruction Single Data)と呼ぶ。

SIMD は、SISD と比べて多くのデータを一括して処理できるため、結果として処理が早く完了する。

以下、SISD と SIMD の処理の対比を示す。

【イメージ】 SISD (Single Instruction Single Data) 処理

```
float a[4] = {1.f, 2.f, 3.f, 4.f};
float b[4] = {0.1f, 0.2f, 0.3f, 0.4f};
float c[4];
c[0] = a[0] + b[0]; // SISD 処理×4
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
c[3] = a[3] + b[3];
printf("c={%.1f, %.1f, %.1f, %.1f}\n", c[0], c[1], c[2], c[3]);
--結果--
c={1.1, 2.2, 3.3, 4.4}
```



【イメージ】 SIMD (Single Instruction Multi Data) 処理

```
float a[4] = {1.f, 2.f, 3.f, 4.f};
float b[4] = {0.1f, 0.2f, 0.3f, 0.4f};
float c[4];
c = a + b; // SIMD 処理×1 ※あくまでも処理のイメージであり、実際にこのように処理を記述できるわけではない
printf("c={%.1f, %.1f, %.1f, %.1f}\n", c[0], c[1], c[2], c[3]);
--結果--
c={1.1, 2.2, 3.3, 4.4}
```

上記の SIMD 処理のイメージでは、32bit の浮動小数点データを4つまとめて処理する。SISD に比べて、単純に4倍の処理効率となる。

どれだけの処理を同時実行できるかは、SIMD 演算装置のレジスタのサイズに依存する。128bit レジスタを使用する場合は、8bit 整数×16、16bit 整数×8、32bit 整数 or 浮動小数点×4 の同時処理が可能。

SIMD 演算ユニットは浮動小数点演算ユニットを兼ねることが多く、浮動小数点の演算を高速に行うことができる。

● 代表的な SIMD 拡張命令セット

SIMD 演算ユニットは、CPU や GPU に実装される。

インテル系の CPU に搭載される MMX, SSE, AVX 系の SIMD 拡張命令セットがとくに有名。以下、それらの命令セットを簡単に説明する。

➤ MMX (MultiMedia eXtensions)

インテルの Pentium プロセッサに搭載された SIMD 型拡張命令セット。

64bit の SIMD 演算に対応する。8bit×8, 16bit×4, 32bit×2 の整数演算が可能。56 個の命令を持つ。

浮動小数点演算ユニット (x87FPU) のレジスタを転用するため、浮動小数点演算と排他的に使用しなければならない。

後継の SSE の台頭により、MMX は非推奨となった。

➤ 3DNow!

AMD の K6-2 プロセッサに搭載された SIMD 型拡張命令セット。

MMX を基盤に、浮動小数点演算を含む 21 個の命令を追加。

インテルの SSE より先行し、後継の「Enhanced 3DNow!」「3DNow! Professional」で命令強化と SSE との互換性に対応してきたものの、結局普及しなかった。

➤ SSE (Streaming SIMD Extensions)

インテルが開発した SIMD 拡張命令セット。x86 / x64 系 CPU の SIMD 演算のデファクトスタンダード。

x86 アーキテクチャに対して、SIMD 演算用に 8 本の 128 ビットレジスタを新設。浮動小数点演算に対応し、単精度 (32bit) ×4 の浮動小数点演算が可能。70 個の命令を持つ。加えて、MMX の命令も実行可能。

普通に浮動小数点演算ユニットとしても扱われるため、命令セットの中には、単一の値を扱う「スカラー演算」にも対応する。例えば、単精度浮動小数点データ (32bit) のスカラー演算では、128bit レジスタの先頭 32bit のみを演算し、残りの 96bit は演算せずにコピーする。

➤ SSE2 (Streaming SIMD Extensions 2)

SSE に、倍精度 (64bit) ×2 の浮動小数点演算を含む 144 個の命令を追加。

Wikipedia によれば、「64 ビットの倍精度浮動小数点演算のサポート及び MMX を 128Bit 幅に拡張する整数演算命令の追加、キャッシュの制御機能の強化がなされた」とのこと。

➤ SSE3 (Streaming SIMD Extensions 3)

SSE2 に、13 個の命令を追加。

Wikipedia によれば「メモリアクセス及び複素数計算の高速化、仮想 CPU のスレッドの動作制御などの機能が搭載され、主に動画圧縮の処理が向上した」とのこと。

➤ SSSE3 (Supplemental Streaming SIMD Extensions 3)

SSSE3 に、32 個の命令を追加。

➤ SSE4.1 (Supplemental Streaming SIMD Extensions 4.1)

SSSE3 に、47 個の命令を追加。

➤ SSE4.2 (Supplemental Streaming SIMD Extensions 4.2)

SSE4.1 に、7 個の命令を追加。文字列命令に対応。

➤ SSE4a

AMD 独自の 4 命令。SSE4 とは互換性がない。

➤ AVX (Advanced Vector eXtensions)

MMX/SSE の後継命令セット。

AVX レジスタとして SSE レジスタを 256bit に拡張し、単精度 (32bit) ×8、倍精度 (64bit) ×4 の浮動小数点演算が可能。70 個の命令を持つ。

AMD の FX シリーズ (Bulldozer 世代) の場合、AVX をサポートしているものの、128bit 幅の演算を二つ使って実行しているため、処理効率率は向上していない。

➤ AVX2 (Advanced Vector eXtensions 2)

Wikipedia によれば「従来の SIMD 整数演算命令が 128 ビットから 256 ビットに拡張されるのが主な変更点であるが、要素ごとに独立したシフト量を設定できるシフト命令、非連続なデータを並べ替えながらロードが可能なギャザー命令等の新たな命令も実装される」とのこと。

● SIMD 演算サンプル

上記の SSE 系命令セットを用いた SIMD 演算処理のサンプルプログラムを示す。

ごく単純な三次元ベクトルクラスのサンプルとして、SSE 命令使用時と未使用時のそれぞれのサンプルとそのパフォーマンスを示す。

なお、このサンプルプログラムは Visual C++ 2013 と GCC の両方でコンパイル可能である。GCC でコンパイルする際は、一部ソースコードを改変し、vector.h の「USE_GCC」マクロを有効化した上で (コメントを外した上で)、コンパイルオプションに「-msse4.1」(SSE4.1 を有効化) を指定してコンパイルする。

なお、SIMD 演算関数の使用箇所を赤に着色して示す。

【vector.h】※SIMD 演算有効化状態

```
#ifndef __VECTOR_H_
#define __VECTOR_H_

// #define USE_GCC // GCC を使用
#define USE_SIMD // 明示的な SIMD 演算を使用する
// #define USE_LOAD_STORE // 明示的なロード/ストア処理を使用
// ※使わない方が速い
// #define USE_MM_EXTRACT_PS // _mm_extract_ps 関数 (SSE3) を使用
// ※使った方がソースコードの互換性が確実だが、使わない方が速い

// アラインメント属性
#ifdef USE_GCC
#define ATTR_ALIGN(n) __attribute__((aligned(n))) // for GCC
#else // USE_GCC
#define ATTR_ALIGN(n) __declspec(align(n)) // for Visual C++
#endif // USE_GCC
#define ATTR_ALIGN16 ATTR_ALIGN(16) // 16 バイトアラインメント指定
```

```

//SIMD 演算対象データ用属性
#ifdef USE_SIMD
#define ATTR_SIMD128 ATTR_ALIGN16//SIMD 演算対象データ用 128 ビットアラインメント指定
#else//USE_SIMD
#define ATTR_SIMD128
#endif//USE_SIMD

#ifdef USE_SIMD
//※このサンプルプログラムでは、SSE～SSE4.1 の命令を使用
//#include <x86intrin.h>//x86 instructions
//#include <mmintrin.h>//MMX (Pentium MMX!)
//#include <mm3dnow.h>//3dnow! (K6 - 2) (deprecated)
//#include <xmmintrin.h>//SSE + MMX (Pentium 3, Athlon XP)
//#include <emmintrin.h>//SSE2 + SSE + MMX (Pentium 4, Athlon 64)
//#include <pmmintrin.h>//SSE3 + SSE2 + SSE + MMX (Pentium 4 Prescott, Athlon 64 San Diego)
//#include <tmmintrin.h>//SSSE3 + SSE3 + SSE2 + SSE + MMX (Core 2, Bulldozer)
//#include <popcntintrin.h>//POPCNT (Core i7, Phenom subset of SSE4.2 and SSE4A)
//#include <ammintrin.h>//SSE4A + SSE3 + SSE2 + SSE + MMX (Phenom)
#include <smmintrin.h>//SSE4.1 + SSSE3 + SSE3 + SSE2 + SSE + MMX (Core i7, Bulldozer)
//#include <mmmintrin.h>//SSE4.2 + SSE4.1 + SSSE3 + SSE3 + SSE2 + SSE + MMX (Core i7, Bulldozer)
//#include <wmmintrin.h>//AES (Core i7 Westmere, Bulldozer)
//#include <immintrin.h>//AVX, SSE4_2 + SSE4_1 + SSSE3 + SSE3 + SSE2 + SSE + MMX (Core i7 Sandy Bridge, Bulldozer)
//#include <ivec.h>//MMX による整数ベクトルクラス
//#include <fvec.h>//SSE による浮動小数点ベクトルクラス
//#include <dvec.h>//SSE2 による整数ベクトルクラス
#else//USE_SIMD
#include <math.h>//sqrt
#endif//USE_SIMD

#ifdef USE_SIMD
//-----
//ベクトルクラス計算用関数
//二項演算子用関数と共通関数
//※SIMD 演算専用
inline float getScalar(const __m128 val)
{
#ifdef USE_MM_EXTRACT_PS
float f;
_MM_EXTRACT_FLOAT(f, val, 0);//(dest, src, ndx): *((int*)&(dest)) = _mm_extract_ps((src), (ndx))
return f;
#else//USE_MM_EXTRACT_PS
return *reinterpret_cast<const float*>(&val);
#endif//USE_MM_EXTRACT_PS
}
inline __m128 calcAdd(const __m128 lhs, const __m128 rhs)//二項の足し算
{
return _mm_add_ps(lhs, rhs);//ret ← {lhs[0] + rhs[0], lhs[1] + rhs[1], lhs[2] + rhs[2], lhs[3] + rhs[3]}
}
inline __m128 calcSub(const __m128 lhs, const __m128 rhs)//二項の引き算
{
return _mm_sub_ps(lhs, rhs);//ret ← {lhs[0] - rhs[0], lhs[1] - rhs[1], lhs[2] - rhs[2], lhs[3] - rhs[3]}
}
inline __m128 calcMul(const __m128 lhs, const float rhs)//二項の掛け算
{
return _mm_mul_ps(lhs, _mm_set1_ps(rhs));//ret ← {lhs[0] * rhs, lhs[1] * rhs, lhs[2] * rhs, lhs[3] * rhs}
}
inline __m128 calcDiv(const __m128 lhs, const float rhs)//二項の割り算
{
return _mm_div_ps(lhs, _mm_set1_ps(rhs));//ret ← {lhs[0] / rhs, lhs[1] / rhs, lhs[2] / rhs, lhs[3] / rhs}
}
inline float calcNorm2(const __m128 val)//ノルム^2 計算
{
return getScalar{//ret ← arg0[0]
_mm_dp_ps(val, val, 0x71)//arg0 ← {val[0] * val[0] + val[1] * val[1] + val[2] * val[2] + 0.f,

```

```

        //      0.f, 0.f, 0.f
    );
}
inline float calcNorm(const __m128 val)//ノルム計算
{
    return getScalar>//ret ← arg1[0]
        _mm_sqrt_ss>//arg1 ← {sqrt(arg0[0]), arg0[1], arg0[2], arg0[3]}
        _mm_dp_ps(val, val, 0x71)//arg0 ← {val[0] * val[0] + val[1] * val[1] + val[2] * val[2] + 0.f,
        //      0.f, 0.f, 0.f}
    );
}
inline __m128 calcNormalize(const __m128 val)//正規化
{
    return _mm_div_ps>//ret ← {arg4[0] / arg3[0], arg4[1] / arg3[1], arg4[2] / arg3[2], arg4[3] / arg3[3]}
        val, //arg4: val
        _mm_set1_ps>//arg3 ← {arg2, arg2, arg2, arg2}
        getScalar>//arg2 ← arg1[0]
        _mm_sqrt_ss>//arg1 ← {sqrt(arg0[0]), arg0[1], arg0[2], arg0[3]}
        _mm_dp_ps(val, val, 0x71)//arg0 ← {val[0] * val[0] + val[1] * val[1] + val[2] * val[2] + 0.f,
        //      0.f, 0.f, 0.f}
    );
}
inline float calcDot(const __m128 lhs, const __m128 rhs)//内積計算
{
    return getScalar>//ret ← arg4[0]
        _mm_dp_ps(lhs, rhs, 0x71)//arg0 ← {lhs[0] * rhs[0] + lhs[1] * rhs[1] + lhs[2] * rhs[2] + 0.f,
        //      0.f, 0.f, 0.f}
    );
}
inline float calcNormalizeDot(const __m128 lhs, const __m128 rhs)//正規化ベクトルで内積計算
{
    return calcDot(calcNormalize(lhs), calcNormalize(rhs));//ret ← calcDot(calcNormalize(lhs),
        //      calcNormalize(rhs))
}
inline __m128 calcCross(const __m128 lhs, const __m128 rhs)//外積計算
{
    return _mm_sub_ps>//ret ← {arg2[0] - arg5[0], arg2[1] - arg5[1], arg2[2] - arg5[2], arg2[3] - arg5[3]}
        _mm_mul_ps>//arg2 ← {arg0[0] * arg1[0], arg0[1] * arg1[1], arg0[2] * arg1[2], arg0[3] * arg1[3]}
        _mm_shuffle_ps(lhs, lhs, _MM_SHUFFLE(3, 0, 2, 1)),//arg0 ← {lhs[1], lhs[2], lhs[0], lhs[3]}
        _mm_shuffle_ps(rhs, rhs, _MM_SHUFFLE(3, 1, 0, 2)) //arg1 ← {rhs[2], rhs[0], rhs[1], rhs[3]}
    ),
    _mm_mul_ps>//arg5 ← {arg3[0] * arg4[0], arg3[1] * arg4[1], arg3[2] * arg4[2], arg3[3] * arg4[3]}
        _mm_shuffle_ps(lhs, lhs, _MM_SHUFFLE(3, 1, 0, 2)),//arg3 ← {lhs[2], lhs[0], lhs[1], lhs[3]}
        _mm_shuffle_ps(rhs, rhs, _MM_SHUFFLE(3, 0, 2, 1)) //arg4 ← {rhs[1], rhs[2], rhs[0], rhs[3]}
    );
    return lhs;
}
inline float calcDistance2(const __m128 from, const __m128 to)//距離^2 計算
{
    const __m128 val_diff = _mm_sub_ps(to, from);//ret ← {to[0] - from[0], to[1] - from[1], to[2] - from[2],
        //      to[3] - from[3]}
    return getScalar>//ret ← arg0[0]
        _mm_dp_ps(val_diff, val_diff, 0x71)//arg0 ← {val_diff[0] * val_diff[0] +
        //      val_diff[1] * val_diff[1] +
        //      val_diff[2] * val_diff[2] + 0.f, 0.f, 0.f, 0.f}
    );
}
inline float calcDistance(const __m128 from, const __m128 to)//距離計算
{
    const __m128 val_diff = _mm_sub_ps(to, from);//ret ← {to[0] - from[0], to[1] - from[1], to[2] - from[2],

```

```

        // to[3] - from[3]}
return getScalar(//ret ← arg1[0]
    __mm_sqrt_ss(//arg1 ← {sqrt(arg0[0]), arg0[1], arg0[2], arg0[3]}
    __mm_dp_ps(val_diff, val_diff, 0x71)//arg0 ← {val_diff[0] * val_diff[0] +
        // val_diff[1] * val_diff[1] +
        // val_diff[2] * val_diff[2] + 0.f, 0.f, 0.f, 0.f}
    )
);
}
inline __m128 calcAdd(const __m128 lhs, const float rhs)//二項の足し算
{
    return calcAdd(lhs, calcMul(calcNormalize(lhs), rhs));//ret ← lhs + normalize(lhs) * rhs
}
inline __m128 calcSub(const __m128 lhs, const float rhs)//二項の引き算
{
    return calcSub(lhs, calcMul(calcNormalize(lhs), rhs));//ret ← lhs - normalize(lhs) * rhs
}
#endif//USE_SIMD

#ifdef USE_SIMD
//-----
//ベクトルクラス計算用構造体
#ifdef USE_GCC
#define DECLARE_M128_S//__m128_s 構造体を使用
#endif//USE_GCC
#ifndef DECLARE_M128_S
//Visual C++ では、__m128 は元々構造体なので、__m128_s は__m128 の別名とする
typedef __m128 __m128_s;
#else//DECLARE_M128_S
//二項演算子 operator に __m128 が使えるように、__m128 をラップした構造体 __m128_s を用意
struct __m128_s
{
    __m128 m_val;
    inline operator __m128() const { return m_val; }
    inline __m128_s& operator=(const __m128 val)
    {
        m_val = val;
        return *this;
    }
    inline __m128_s(const __m128 val) :
        m_val(val)
    {}
};
#endif//DECLARE_M128_S
#endif//USE_SIMD

//-----
//ベクトルクラス
struct vec3
{
    //フィールド
#ifdef USE_SIMD
#ifdef USE_LOAD_STORE
//SIMD 演算使用、かつ、union で最適化したデータ構造
union
{
    __m128 m_val128;
    float m_val[4];
};
#else//USE_LOAD_STORE
//SIMD 演算使用、かつ、明示的なロード/ストア使用時のデータ構造
float ATTR_SIMD128 m_val[4];
#endif//USE_LOAD_STORE
#else//USE_SIMD
//SIMD 非使用時のデータ構造

```

```

float m_val[3];
#endif//USE_SIMD
//アクセッサ
float& x() { return m_val[0]; }
float& y() { return m_val[1]; }
float& z() { return m_val[2]; }
float x() const { return m_val[0]; }
float y() const { return m_val[1]; }
float z() const { return m_val[2]; }
void x(const float val){ m_val[0] = val; }
void y(const float val){ m_val[1] = val; }
void z(const float val){ m_val[2] = val; }
float& operator[] (const int index){ return m_val[index]; }
const float& operator[] (const int index) const { return m_val[index]; }

#ifdef USE_SIMD
//ロード/ストアメソッド
inline __m128 load() const//ロード
{
#ifdef USE_LOAD_STORE
    return m_val128;//ret ← m_val128
#else//USE_LOAD_STORE
    return _mm_load_ps(m_val);//ret ← {m_val[0], m_val[1], m_val[2], m_val[3]}
#endif//USE_LOAD_STORE
}

inline __m128 loadI() const//ロード
{
#ifdef USE_LOAD_STORE
    //NG
    //return _mm_set1_ps(m_val[0]);//ret ← {m_val[0], m_val[0], m_val[0], m_val[0]}
    //OK
    return _mm_set_ps(0.f, m_val[0], m_val[0], m_val[0]);//ret ← {m_val[0], m_val[0], m_val[0], 0.f}
#else//USE_LOAD_STORE
    //NG
    //return _mm_load1_ps(m_val);//ret ← {m_val[0], m_val[0], m_val[0], m_val[0]}
    //OK
    return _mm_blend_ps(//ret ← {arg0[0], arg0[1], arg0[2], arg1[3]}
        _mm_load1_ps(m_val), //arg0 ← {m_val[0], m_val[0], m_val[0], m_val[0]}
        _mm_setzero_ps(), //arg1 ← {0.f, 0.f, 0.f, 0.f}
        0x08
    );
#endif//USE_LOAD_STORE
}

inline __m128 loadS() const//ロード
{
#ifdef USE_LOAD_STORE
    return _mm_set_ss(m_val[0]);//ret ← {m_val[0], 0.f, 0.f, 0.f}
#else//USE_LOAD_STORE
    return _mm_load_ss(m_val);//ret ← {m_val[0], 0.f, 0.f, 0.f}
#endif//USE_LOAD_STORE
}

inline float loadSF() const//ロード
{
#ifdef USE_LOAD_STORE
    return m_val[0]//ret ← m_val[0]
#else//USE_LOAD_STORE
    return getScalar(//ret ← arg0[0]
        _mm_load_ss(m_val)//arg0 ← {m_val[0], 0.f, 0.f, 0.f}
    );
#endif//USE_LOAD_STORE
}

inline void store(const __m128 val)//ストア
{
#ifdef USE_LOAD_STORE
    m_val128 = val;//val → m_val128
#else//USE_LOAD_STORE

```

```

    _mm_store_ps(m_val, val); //{val[0], val[1], val[2], val[3]} → m_val
#endif//USE_LOAD_STORE
}
inline void store1(const __m128 val)//ストア
{
#ifdef USE_LOAD_STORE
    // [NG]
    // m_val128 = _mm_set1_ps(//{arg0, arg0, arg0, arg0} → m_val128
    // getScalar(val)//arg0 ← val[0]
    // );
    // [OK]
    m_val128 = _mm_blend_ps(//arg1[0], arg1[1], arg1[2], arg2[3]} → m_val128
        _mm_set1_ps( //arg1 ← {arg0, arg0, arg0, arg0}
            getScalar(val)//arg0 ← val[0]
        ),
        _mm_setzero_ps(), //arg2 ← {0.f, 0.f, 0.f, 0.f}
        0x08
    );
#else//USE_LOAD_STORE
    // [NG]
    // _mm_store1_ps(m_val, //{arg0, arg0, arg0, arg0} → m_val
    // getScalar(val)//arg0 ← val[0]
    // );
    // [OK]
    _mm_store_ps(m_val, //arg3 → m_val
        _mm_blend_ps(//arg3 ← {arg1[0], arg1[1], arg1[2], arg2[3]}
            _mm_set1_ps( //arg1 ← {arg0, arg0, arg0, arg0}
                getScalar(val)//arg0 ← val[0]
            ),
            _mm_setzero_ps(), //arg2 ← {0.f, 0.f, 0.f, 0.f}
            0x08
        )
    );
#endif//USE_LOAD_STORE
}
inline void store1(const float val)//ストア
{
#ifdef USE_LOAD_STORE
    // [NG]
    // m_val128 = _mm_set1_ps(val); //{val, val, val, val} → m_val128
    // [OK]
    m_val128 = _mm_set_ps(0.f, val, val, val); //{val, val, val, 0.f} → m_val128
#else//USE_LOAD_STORE
    // [NG]
    // _mm_store1_ps(m_val, val); //{val[0], val[0], val[0], val[0]} → m_val
    // [OK]
    _mm_store_ps(m_val, //arg0 → m_val128
        _mm_set_ps(0.f, val, val, val)//arg0 ← {val, val, val, 0.f}
    );
#endif//USE_LOAD_STORE
}
inline void storeS(const __m128 val)//ストア
{
#ifdef USE_LOAD_STORE
    m_val[0] = getScalar(val); //val[0] → m_val[0]
#else//USE_LOAD_STORE
    _mm_store_ss(m_val, val); //val[0] → m_val[0]
#endif//USE_LOAD_STORE
}
inline void storeSF(const float val)//ストア
{
#ifdef USE_LOAD_STORE
    m_val[0] = val; //val → m_val[0]
#else//USE_LOAD_STORE
    _mm_store_ss(m_val, //arg0[0] → m_val[0]

```

```

        _mm_set_ss(val)//arg0 ← {val, 0.f, 0.f, 0.f}
    );
#endif//USE_LOAD_STORE
}
inline void storeZero()//ストア
{
#ifdef USE_LOAD_STORE
    m_val128 = _mm_setzero_ps();//{0.f, 0.f, 0.f, 0.f} → m_val128
#else//USE_LOAD_STORE
    _mm_store_ss(m_val, //arg0 → m_val
        _mm_setzero_ps())//arg0 ← {0.f, 0.f, 0.f, 0.f}
    );
#endif//USE_LOAD_STORE
}
#endif//USE_SIMD
//オペレータ
inline vec3& operator+=(const vec3& rhs)
{
#ifdef USE_SIMD
    store>//arg2 → m_val
    calcAdd>//arg2 ← {arg0[0] + arg1[0], arg0[1] + arg1[1], arg0[2] + arg1[2], arg0[3] + arg1[3]}
    load(), //arg0 ← m_val
    rhs.load()//arg1 ← rhs.m_val
    )
    );
#else//USE_SIMD
    for (int index = 0; index < 3; ++index)
        m_val[index] += rhs.m_val[index];
#endif//USE_SIMD
    return *this;
}
inline vec3& operator+=(const float rhs)
{
#ifdef USE_SIMD
    store>//arg2 → m_val
    calcAdd>//arg2 ← arg0 + normalize(arg0) * arg1
    load(), //arg0 ← m_val
    rhs //arg1: rhs
    )
    );
#else//USE_SIMD
    vec3 rhs_tmp(*this);
    rhs_tmp.normalize();
    rhs_tmp *= rhs;
    *this += rhs_tmp;
#endif//USE_SIMD
    return *this;
}
inline vec3& operator-=(const vec3& rhs)
{
#ifdef USE_SIMD
    store>//arg2 → m_val
    calcSub>//arg2 ← {arg0[0] - arg1[0], arg0[1] - arg1[1], arg0[2] - arg1[2], arg0[3] - arg1[3]}
    load(), //arg0 ← m_val
    rhs.load()//arg1 ← rhs.m_val
    )
    );
#else//USE_SIMD
    for (int index = 0; index < 3; ++index)
        m_val[index] -= rhs.m_val[index];
#endif//USE_SIMD
    return *this;
}
inline vec3& operator-=(const float rhs)
{

```

```

#ifdef USE_SIMD
    store(arg2 → m_val
    calcSub(arg2 ← arg0 - normalize(arg0) * arg1
    load(), arg0 ← m_val
    rhs //arg1 ← rhs
    )
);
#else//USE_SIMD
    vec3 rhs_tmp(*this);
    rhs_tmp.normalize();
    rhs_tmp *= rhs;
    *this -= rhs_tmp;
#endif//USE_SIMD
    return *this;
}

inline vec3& operator*=(const float rhs)
{
#ifdef USE_SIMD
    store(arg2 → m_val
    calcMul(arg2 ← {arg0[0] * arg1, arg0[1] * arg1, arg0[2] * arg1, arg0[3] * arg1}
    load(), arg0 ← m_val
    rhs //arg1 ← rhs
    )
);
#else//USE_SIMD
    for (int index = 0; index < 3; ++index)
        m_val[index] *= rhs;
#endif//USE_SIMD
    return *this;
}

inline vec3& operator/=(const float rhs)
{
#ifdef USE_SIMD
    store(arg2 → m_val
    calcDiv(arg2 ← {arg0[0] / arg1, arg0[1] / arg1, arg0[2] / arg1, arg0[3] / arg1}
    load(), arg0 ← m_val
    rhs //arg1 ← rhs
    )
);
#else//USE_SIMD
    for (int index = 0; index < 3; ++index)
        m_val[index] /= rhs;
#endif//USE_SIMD
    return *this;
}

//キャストオペレータ
inline operator float() const
{
    return norm();
}

#ifdef USE_SIMD
    inline operator __m128() const
    {
        return load();//ret ← m_val
    }
#endif

#ifdef DECLARE_M128_S
    inline operator __m128_s() const
    {
        return load();//ret ← m_val
    }
#endif//DECLARE_M128_S
#endif//USE_SIMD
//コピーオペレータ
inline vec3& operator=(const vec3& rhs)
{

```



```

#ifdef USE_SIMD
    store(rhs.load())://rhs → m_val
#else//USE_SIMD
    for (int index = 0; index < 3; ++index)
        m_val[index] = rhs.m_val[index];
#endif//USE_SIMD
    return *this;
}

#ifdef USE_SIMD
inline vec3& operator=(const __m128 rhs)
{
    store(rhs);//rhs → m_val
    return *this;
}
#endif

#ifdef DECLARE_M128_S
inline vec3& operator=(const __m128_s& rhs)
{
    store(rhs);
    return *this;
}
#endif//DECLARE_M128_S
#endif//USE_SIMD

//メソッド
//内積
inline float dot(const vec3& target) const
{
#ifdef USE_SIMD
    return calcDot();//ret ← calcDot(arg0, arg1)
        load(),      //arg0 ← m_val
        target.load();//arg1 ← target.m_val
    );
#else//USE_SIMD
    return m_val[0] * target.m_val[0] +
        m_val[1] * target.m_val[1] +
        m_val[2] * target.m_val[2];
#endif//USE_SIMD
}

//正規化内積
inline float normalizeDot(const vec3& target) const
{
#ifdef USE_SIMD
    return calcNormalizeDot();//ret ← calcNormalizeDot(arg0, arg1)
        load(),      //arg0 ← m_val
        target.load();//arg1 ← target.m_val
    );
#else//USE_SIMD
    vec3 tmp1(*this);
    vec3 tmp2(target);
    tmp1.normalize();
    tmp2.normalize();
    return tmp1.m_val[0] * tmp2.m_val[0] +
        tmp1.m_val[1] * tmp2.m_val[1] +
        tmp1.m_val[2] * tmp2.m_val[2];
#endif//USE_SIMD
}

//外積
#ifdef USE_SIMD
inline __m128_s cross(const vec3& target) const
{
    return calcCross();//ret ← calcCross(arg0, arg1)
        load(),      //arg0 ← m_val
        target.load();//arg1 ← target.m_val
    );
}
#else//USE_SIMD

```

```

inline vec3 cross(const vec3& target) const
{
    return vec3(
        m_val[1] * target.m_val[2] - m_val[2] * target.m_val[1],
        m_val[2] * target.m_val[0] - m_val[0] * target.m_val[2],
        m_val[0] * target.m_val[1] - m_val[1] * target.m_val[0]
    );
}
#endif//USE_SIMD
//ノルム^2 取得
inline float norm2() const
{
#ifdef USE_SIMD
    return calcNorm2(ret ← calcNorm2(arg0)
        load()//arg0 ← m_val
    );
#else//USE_SIMD
    return m_val[0] * m_val[0] +
        m_val[1] * m_val[1] +
        m_val[2] * m_val[2];
#endif//USE_SIMD
}
//ノルム取得
inline float norm() const
{
#ifdef USE_SIMD
    return calcNorm(ret ← calcNorm(arg0)
        load()//arg0 ← m_val128
    );
#else//USE_SIMD
    return sqrt(norm2());
#endif//USE_SIMD
}
//長さ^2 取得
inline float length2() const
{
    return norm2();
}
//長さ取得
inline float length() const
{
    return norm();
}
//正規化
inline vec3& normalize()
{
#ifdef USE_SIMD
    store(ret ← m_val
        calcNormalize(ret ← calcNormalize(arg0)
            load()//arg0 ← m_val
        )
    );
#else//USE_SIMD
    const float val_norm = norm();
    for (int index = 0; index < 3; ++index)
        m_val[index] /= val_norm;
#endif//USE_SIMD
    return *this;
}
#ifdef USE_SIMD
inline __m128_s normalize() const
{
    return calcNormalize(ret ← calcNormalize(arg0)
        load()//arg0 ← m_val
    );
}

```

```

    }
#else//USE_SIMD
    inline vec3 normalize() const
    {
        const float val_norm = norm();
        return vec3(m_val[0] / val_norm, m_val[1] / val_norm, m_val[2] / val_norm);
    }
#endif//USE_SIMD
//距離^2 計算
inline float distance2(const vec3& from)
{
#ifdef USE_SIMD
    return calcDistance2(ret ← calcDist2(arg0, arg1)
        from.load(), //arg0 ← from.m_val
        load()        //arg1 ← m_val
    );
#else//USE_SIMD
    vec3 tmp(*this);
    tmp -= from;
    return tmp.norm2();
#endif//USE_SIMD
}
//距離計算
inline float distance(const vec3& from)
{
#ifdef USE_SIMD
    return calcDistance(ret ← calcDist(arg0, arg1)
        from.load(), //arg0 ← from.m_val
        load()        //arg1 ← m_val
    );
#else//USE_SIMD
    vec3 tmp(*this);
    tmp -= from;
    return tmp.norm();
#endif//USE_SIMD
}
//コピーコンストラクタ
inline vec3(const vec3& obj)
#ifdef USE_SIMD
#ifdef USE_LOAD_STORE
    : m_val128(obj.m_val128) //obj.m_val128 → m_val128
    {}
#else//USE_LOAD_STORE
    {
        store(obj.load()) //obj.m_val → m_val
    }
#endif//USE_LOAD_STORE
#else//USE_SIMD
    {
        for (int index = 0; index < 3; ++index)
            m_val[index] = obj.m_val[index];
    }
#endif//USE_SIMD
//コンストラクタ
inline vec3(const float x, const float y, const float z)
#ifdef USE_SIMD
#ifdef USE_LOAD_STORE
    : m_val128(mm_set_ps(0.f, z, y, x)) // {x, y, z, 0.f} → m_val128
    {}
#else//USE_LOAD_STORE
    {
        store(mm_set_ps(0.f, z, y, x)) // {x, y, z, 0.f} → m_val
    }
#endif//USE_LOAD_STORE
#else//USE_SIMD

```

```

{
    m_val[0] = x;
    m_val[1] = y;
    m_val[2] = z;
}
#endif//USE_SIMD
inline vec3(const float val)
#ifdef USE_SIMD
#ifndef USE_LOAD_STORE
// [NG]
//:m_val128(_mm_set1_ps(val))//[val, val, val, val] → m_val128
// [OK]
:m_val128(_mm_set_ps(0.f, val, val, val))//[val, val, val, 0.f] → m_val128
{}
#else//USE_LOAD_STORE
{
    // [NG]
    //:store(_mm_set1_ps(val))//[val, val, val, val] → m_val
    // [OK]
    store(val) //[val, val, val, 0.f] → m_val
}
#endif//USE_LOAD_STORE
#else//USE_SIMD
{
    for (int index = 0; index < 3; ++index)
        m_val[index] = val;
}
#endif//USE_SIMD
#ifdef USE_SIMD
inline vec3(const __m128 val)
#ifndef USE_LOAD_STORE
:m_val128(val)//val → m_val128
{}
#else//USE_LOAD_STORE
{
    store(val)//val → m_val
}
#endif//USE_LOAD_STORE
#endif//DECLARE_M128_S
inline vec3(const __m128_s& val)
#ifndef USE_LOAD_STORE
:m_val128(val)//val → m_val128
{}
#else//USE_LOAD_STORE
{
    store(val)//val → m_val
}
#endif//USE_LOAD_STORE
#endif//DECLARE_M128_S
#endif//USE_SIMD
inline vec3()
#ifdef USE_SIMD
#ifndef USE_LOAD_STORE
:m_val128(_mm_setzero_ps())//[0.f, 0.f, 0.f, 0.f] → m_val128
{}
#else//USE_LOAD_STORE
{
    storeZero()//[0.f, 0.f, 0.f, 0.f] → m_val128
}
#endif//USE_LOAD_STORE
#else//USE_SIMD
{
    for (int index = 0; index < 3; ++index)
        m_val[0] = 0.f;
}

```

```

#endif//USE_SIMD
//デストラクタ
inline ~vec3()
{}
};

//二項演算子用オペレータ
#ifdef USE_SIMD
//※SIMD 演算時は vec3 型のままではなく、__m128 型として演算し、__m128_s 型を返す
inline __m128_s operator+(const vec3& lhs, const vec3& rhs) {
    return calcAdd(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator+(const vec3& lhs, const __m128_s& rhs) {
    return calcAdd(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator+(const __m128_s& lhs, const vec3& rhs) {
    return calcAdd(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator+(const __m128_s& lhs, const __m128_s& rhs) {
    return calcAdd(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator+(const vec3& lhs, const float rhs) {
    return calcAdd(static_cast<__m128>(lhs), rhs); }
inline __m128_s operator+(const __m128_s& lhs, const float rhs) {
    return calcAdd(static_cast<__m128>(lhs), rhs); }

inline __m128_s operator-(const vec3& lhs, const vec3& rhs) {
    return calcSub(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator-(const vec3& lhs, const __m128_s& rhs) {
    return calcSub(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator-(const __m128_s& lhs, const vec3& rhs) {
    return calcSub(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator-(const __m128_s& lhs, const __m128_s& rhs) {
    return calcSub(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator-(const vec3& lhs, const float rhs) {
    return calcSub(static_cast<__m128>(lhs), rhs); }
inline __m128_s operator-(const __m128_s& lhs, const float rhs) {
    return calcSub(static_cast<__m128>(lhs), rhs); }

inline __m128_s operator*(const vec3& lhs, const float rhs) {
    return calcMul(static_cast<__m128>(lhs), rhs); }
inline __m128_s operator*(const __m128_s& lhs, const float rhs) {
    return calcMul(static_cast<__m128>(lhs), rhs); }

inline __m128_s operator/(const vec3& lhs, const float rhs) {
    return calcDiv(static_cast<__m128>(lhs), rhs); }
inline __m128_s operator/(const __m128_s& lhs, const float rhs) {
    return calcDiv(static_cast<__m128>(lhs), rhs); }

inline float dot(const vec3& lhs, const vec3& rhs) {
    return calcDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline float dot(const vec3& lhs, const __m128_s& rhs) {
    return calcDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline float dot(const __m128_s& lhs, const vec3& rhs) {
    return calcDot(lhs, static_cast<__m128>(rhs)); }
inline float dot(const __m128_s& lhs, const __m128_s& rhs) {
    return calcDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }

inline float normalizeDot(const vec3& lhs, const vec3& rhs) {
    return calcNormalizeDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline float normalizeDot(const vec3& lhs, const __m128_s& rhs) {
    return calcNormalizeDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline float normalizeDot(const __m128_s& lhs, const vec3& rhs) {
    return calcNormalizeDot(lhs, static_cast<__m128>(rhs)); }
inline float normalizeDot(const __m128_s& lhs, const __m128_s& rhs) {
    return calcNormalizeDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }

inline __m128_s cross(const vec3& lhs, const vec3& rhs) {
    return calcCross(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }

```

```

inline __m128_s cross(const vec3& lhs, const __m128_s& rhs) {
    return calcCross(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s cross(const __m128_s& lhs, const vec3& rhs) {
    return calcCross(lhs, static_cast<__m128>(rhs)); }
inline __m128_s cross(const __m128_s& lhs, const __m128_s& rhs) {
    return calcCross(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }

inline float norm2(const vec3& val) {
    return calcNorm2(static_cast<__m128>(val)); }
inline float norm2(const __m128_s& val) {
    return calcNorm2(static_cast<__m128>(val)); }
inline float norm(const vec3& val) {
    return calcNorm(static_cast<__m128>(val)); }
inline float norm(const __m128_s& val) {
    return calcNorm(static_cast<__m128>(val)); }

inline float length2(const vec3& val) {
    return calcNorm2(static_cast<__m128>(val)); }
inline float length2(const __m128_s& val) {
    return calcNorm2(static_cast<__m128>(val)); }
inline float length(const vec3& val) {
    return calcNorm(static_cast<__m128>(val)); }
inline float length(const __m128_s& val) {
    return calcNorm(static_cast<__m128>(val)); }

inline __m128_s normalize(const vec3& val) {
    return calcNormalize(static_cast<__m128>(val)); }
inline __m128_s normalize(const __m128_s& val) {
    return calcNormalize(static_cast<__m128>(val)); }

inline float distance2(const vec3& from, const vec3& to) {
    return calcDistance2(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance2(const vec3& from, const __m128_s& to) {
    return calcDistance2(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance2(const __m128_s& from, const vec3& to) {
    return calcDistance2(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance2(const __m128_s& from, const __m128_s& to) {
    return calcDistance2(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance(const vec3& from, const vec3& to) {
    return calcDistance(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance(const vec3& from, const __m128_s& to) {
    return calcDistance(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance(const __m128_s& from, const vec3& to) {
    return calcDistance(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance(const __m128_s& from, const __m128_s& to) {
    return calcDistance(static_cast<__m128>(from), static_cast<__m128>(to)); }

#else//USE_SIMD
//※非 SIMD 演算時は vec3 型のまま演算する
inline vec3 operator+(const vec3& lhs, const vec3& rhs) { vec3 tmp(lhs); return (tmp += rhs); }
inline vec3 operator+(const vec3& lhs, const float rhs) { vec3 tmp(lhs); return (tmp += rhs); }

inline vec3 operator-(const vec3& lhs, const vec3& rhs) { vec3 tmp(lhs); return (tmp -= rhs); }
inline vec3 operator-(const vec3& lhs, const float rhs) { vec3 tmp(lhs); return (tmp -= rhs); }

inline vec3 operator*(const vec3& lhs, const float rhs) { vec3 tmp(lhs); return (tmp *= rhs); }

inline vec3 operator/(const vec3& lhs, const float rhs) { vec3 tmp(lhs); return (tmp /= rhs); }

inline float dot(const vec3& lhs, const vec3& rhs) { return lhs.dot(rhs); }
inline float normalizeDot(const vec3& lhs, const vec3& rhs) { return lhs.normalizeDot(rhs); }

inline vec3 cross(const vec3& lhs, const vec3& rhs) { return lhs.cross(rhs); }

inline float norm2(const vec3& val) { return val.norm2(); }

```

```

inline float norm(const vec3& val){ return val.norm(); }

inline float length2(const vec3& val){ return val.length2(); }
inline float length(const vec3& val){ return val.length(); }

inline vec3 normalize(const vec3& val){ vec3 tmp(val); return tmp.normalize(); }

inline float distance2(const vec3& from, const vec3& to){ vec3 tmp(to); return tmp.distance2(from); }
inline float distance(const vec3& from, const vec3& to){ vec3 tmp(to); return tmp.distance(from); }
#endif//USE_SIMD

#endif//__VECTOR_H_

```

【vector.h】※SIMD 演算無効化状態

```

#ifndef __VECTOR_H_
#define __VECTOR_H_

// #define USE_GCC // GCC を使用
// #define USE_SIMD // 明示的な SIMD 演算を使用する
// #define USE_LOAD_STORE // 明示的なロード／ストア処理を使用
// ※使わない方が速い
// #define USE_MM_EXTRACT_PS // _mm_extract_ps 関数 (SSE3) を使用
// ※使った方がソースコードの互換性が確実だが、使わない方が速い

// アラインメント属性
#ifdef USE_GCC
#define ATTR_ALIGN(n) __attribute__((aligned(n))) // for GCC
#else//USE_GCC
#define ATTR_ALIGN(n) __declspec(align(n)) // for Visual C++
#endif//USE_GCC
#define ATTR_ALIGN16 ATTR_ALIGN(16) // 16 バイトアラインメント指定

// SIMD 演算対象データ用属性
#ifdef USE_SIMD
#define ATTR_SIMD128 ATTR_ALIGN16 // SIMD 演算対象データ用 128 ビットアラインメント指定
#else//USE_SIMD
#define ATTR_SIMD128
#endif//USE_SIMD

#ifdef USE_SIMD
// ※このサンプルプログラムでは、SSE～SSE4.1 の命令を使用
// #include <x86intrin.h> // x86 instructions
// #include <mmintrin.h> // MMX (Pentium MMX!)
// #include <nm3dnow.h> // 3dnow! (K6 - 2) (deprecated)
// #include <xmmintrin.h> // SSE + MMX (Pentium 3, Athlon XP)
// #include <emmintrin.h> // SSE2 + SSE + MMX (Pentium 4, Athlon 64)
// #include <pmmintrin.h> // SSE3 + SSE2 + SSE + MMX (Pentium 4 Prescott, Athlon 64 San Diego)
// #include <tmmmintrin.h> // SSSE3 + SSE3 + SSE2 + SSE + MMX (Core 2, Bulldozer)
// #include <popcntintrin.h> // POPCNT (Core i7, Phenom subset of SSE4.2 and SSE4A)
// #include <ammintrin.h> // SSE4A + SSE3 + SSE2 + SSE + MMX (Phenom)
// #include <smmintrin.h> // SSE4.1 + SSSE3 + SSE3 + SSE2 + SSE + MMX (Core i7, Bulldozer)
// #include <nmmintrin.h> // SSE4.2 + SSE4.1 + SSSE3 + SSE3 + SSE2 + SSE + MMX (Core i7, Bulldozer)
// #include <wmmintrin.h> // AES (Core i7 Westmere, Bulldozer)
// #include <immintrin.h> // AVX, SSE4.2 + SSE4.1 + SSSE3 + SSE3 + SSE2 + SSE + MMX (Core i7 Sandy Bridge, Bulldozer)
// #include <ivec.h> // MMX による整数ベクトルクラス
// #include <fvec.h> // SSE による浮動小数点ベクトルクラス
// #include <dvec.h> // SSE2 による整数ベクトルクラス
#else//USE_SIMD
#include <math.h> // sqrt
#endif//USE_SIMD

#ifdef USE_SIMD
// -----
// ベクトルクラス計算用関数
// 二項演算子用関数と共通関数

```

```

//※SIMD 演算専用
inline float getScalar(const __m128 val)
{
#ifdef USE_MM_EXTRACT_PS
    float f;
    _MM_EXTRACT_FLOAT(f, val, 0); //(dest, src, ndx): *((int*)&(dest)) = _mm_extract_ps((src), (ndx))
    return f;
#else//USE_MM_EXTRACT_PS
    return *reinterpret_cast<const float*>(&val);
#endif//USE_MM_EXTRACT_PS
}

inline __m128 calcAdd(const __m128 lhs, const __m128 rhs)//二項の足し算
{
    return _mm_add_ps(lhs, rhs); //ret ← {lhs[0] + rhs[0], lhs[1] + rhs[1], lhs[2] + rhs[2], lhs[3] + rhs[3]}
}

inline __m128 calcSub(const __m128 lhs, const __m128 rhs)//二項の引き算
{
    return _mm_sub_ps(lhs, rhs); //ret ← {lhs[0] - rhs[0], lhs[1] - rhs[1], lhs[2] - rhs[2], lhs[3] - rhs[3]}
}

inline __m128 calcMul(const __m128 lhs, const float rhs)//二項の掛け算
{
    return _mm_mul_ps(lhs, _mm_set1_ps(rhs)); //ret ← {lhs[0] * rhs, lhs[1] * rhs, lhs[2] * rhs, lhs[3] * rhs}
}

inline __m128 calcDiv(const __m128 lhs, const float rhs)//二項の割り算
{
    return _mm_div_ps(lhs, _mm_set1_ps(rhs)); //ret ← {lhs[0] / rhs, lhs[1] / rhs, lhs[2] / rhs, lhs[3] / rhs}
}

inline float calcNorm2(const __m128 val)//ノルム^2 計算
{
    return getScalar(//ret ← arg0[0]
        _mm_dp_ps(val, val, 0x71) //arg0 ← {val[0] * val[0] + val[1] * val[1] + val[2] * val[2] + 0.f,
        // 0.f, 0.f, 0.f}
    );
}

inline float calcNorm(const __m128 val)//ノルム計算
{
    return getScalar(//ret ← arg1[0]
        _mm_sqrt_ss(//arg1 ← {sqrt(arg0[0]), arg0[1], arg0[2], arg0[3]}
        _mm_dp_ps(val, val, 0x71) //arg0 ← {val[0] * val[0] + val[1] * val[1] + val[2] * val[2] + 0.f,
        // 0.f, 0.f, 0.f}
    )
    );
}

inline __m128 calcNormalize(const __m128 val)//正規化
{
    return _mm_div_ps(//ret ← {arg4[0] / arg3[0], arg4[1] / arg3[1], arg4[2] / arg3[2], arg4[3] / arg3[3]}
        val, //arg4: val
        _mm_set1_ps(//arg3 ← {arg2, arg2, arg2, arg2}
            getScalar(//arg2 ← arg1[0]
                _mm_sqrt_ss(//arg1 ← {sqrt(arg0[0]), arg0[1], arg0[2], arg0[3]}
                _mm_dp_ps(val, val, 0x71) //arg0 ← {val[0] * val[0] + val[1] * val[1] + val[2] * val[2] + 0.f,
                // 0.f, 0.f, 0.f}
            )
        )
    );
}

inline float calcDot(const __m128 lhs, const __m128 rhs)//内積計算
{
    return getScalar(//ret ← arg4[0]
        _mm_dp_ps(lhs, rhs, 0x71) //arg0 ← {lhs[0] * rhs[0] + lhs[1] * rhs[1] + lhs[2] * rhs[2] + 0.f,
        // 0.f, 0.f, 0.f}
    );
}

inline float calcNormalizeDot(const __m128 lhs, const __m128 rhs)//正規化ベクトルで内積計算

```



```

{
    return calcDot(calcNormalize(lhs), calcNormalize(rhs)); //ret ← calcDot(calcNormalize(lhs),
                                                             //                               calcNormalize(rhs))
}
inline __m128 calcCross(const __m128 lhs, const __m128 rhs) //外積計算
{
    return _mm_sub_ps(//ret ← {arg2[0] - arg5[0], arg2[1] - arg5[1], arg2[2] - arg5[2], arg2[3] - arg5[3]}
        _mm_mul_ps(//arg2 ← {arg0[0] * arg1[0], arg0[1] * arg1[1], arg0[2] * arg1[2], arg0[3] * arg1[3]}
            _mm_shuffle_ps(lhs, lhs, _MM_SHUFFLE(3, 0, 2, 1)), //arg0 ← {lhs[1], lhs[2], lhs[0], lhs[3]}
            _mm_shuffle_ps(rhs, rhs, _MM_SHUFFLE(3, 1, 0, 2)) //arg1 ← {rhs[2], rhs[0], rhs[1], rhs[3]}
        ),
        _mm_mul_ps(//arg5 ← {arg3[0] * arg4[0], arg3[1] * arg4[1], arg3[2] * arg4[2], arg3[3] * arg4[3]}
            _mm_shuffle_ps(lhs, lhs, _MM_SHUFFLE(3, 1, 0, 2)), //arg3 ← {lhs[2], lhs[0], lhs[1], lhs[3]}
            _mm_shuffle_ps(rhs, rhs, _MM_SHUFFLE(3, 0, 2, 1)) //arg4 ← {rhs[1], rhs[2], rhs[0], rhs[3]}
        )
    );
    return lhs;
}
inline float calcDistance2(const __m128 from, const __m128 to) //距離^2 計算
{
    const __m128 val_diff = _mm_sub_ps(to, from); //ret ← {to[0] - from[0], to[1] - from[1], to[2] - from[2],
                                                       //                               to[3] - from[3]}
    return getScalar(//ret ← arg0[0]
        _mm_dp_ps(val_diff, val_diff, 0x71) //arg0 ← {val_diff[0] * val_diff[0] +
                                                       //                               val_diff[1] * val_diff[1] +
                                                       //                               val_diff[2] * val_diff[2] + 0.f, 0.f, 0.f, 0.f}
    );
}
inline float calcDistance(const __m128 from, const __m128 to) //距離計算
{
    const __m128 val_diff = _mm_sub_ps(to, from); //ret ← {to[0] - from[0], to[1] - from[1], to[2] - from[2],
                                                       //                               to[3] - from[3]}
    return getScalar(//ret ← arg1[0]
        _mm_sqrt_ss(//arg1 ← {sqrt(arg0[0]), arg0[1], arg0[2], arg0[3]}
            _mm_dp_ps(val_diff, val_diff, 0x71) //arg0 ← {val_diff[0] * val_diff[0] +
                                                       //                               val_diff[1] * val_diff[1] +
                                                       //                               val_diff[2] * val_diff[2] + 0.f, 0.f, 0.f, 0.f}
        )
    );
}
inline __m128 calcAdd(const __m128 lhs, const float rhs) //二項の足し算
{
    return calcAdd(lhs, calcMul(calcNormalize(lhs), rhs)); //ret ← lhs + normalize(lhs) * rhs
}
inline __m128 calcSub(const __m128 lhs, const float rhs) //二項の引き算
{
    return calcSub(lhs, calcMul(calcNormalize(lhs), rhs)); //ret ← lhs - normalize(lhs) * rhs
}
#endif //USE_SIMD

#ifdef USE_SIMD
//-----
//ベクトルクラス計算用構造体
#ifdef USE_GCC
#define DECLARE_M128_S//__m128_s 構造体を使用
#else //USE_GCC
#ifdef DECLARE_M128_S
//Visual C++ では、__m128 は元々構造体なので、__m128_s は__m128 の別名とする
typedef __m128 __m128_s;
#else //DECLARE_M128_S
//二項演算子 operator に __m128 が使えるように、__m128 をラップした構造体 __m128_s を用意
struct __m128_s
{
    __m128 m_val;
    inline operator __m128() const { return m_val; }
}

```

```

inline __m128_s& operator=(const __m128 val)
{
    m_val = val;
    return *this;
}
inline __m128_s(const __m128 val) :
    m_val(val)
{}
};
#endif//DECLARE_M128_S
#endif//USE_SIMD

//-----
//ベクトルクラス
struct vec3
{
    //フィールド
#ifdef USE_SIMD
#ifndef USE_LOAD_STORE
//SIMD 演算使用、かつ、union で最適化したデータ構造
union
{
    __m128 m_val128;
    float m_val[4];
};
#else//USE_LOAD_STORE
//SIMD 演算使用、かつ、明示的なロード／ストア使用時のデータ構造
float ATTR_SIMD128 m_val[4];
#endif//USE_LOAD_STORE
#else//USE_SIMD
//SIMD 非使用時のデータ構造
float m_val[3];
#endif//USE_SIMD
//アクセッサ
float& x() { return m_val[0]; }
float& y() { return m_val[1]; }
float& z() { return m_val[2]; }
float x() const { return m_val[0]; }
float y() const { return m_val[1]; }
float z() const { return m_val[2]; }
void x(const float val){ m_val[0] = val; }
void y(const float val){ m_val[1] = val; }
void z(const float val){ m_val[2] = val; }
float& operator[] (const int index){ return m_val[index]; }
const float& operator[] (const int index) const { return m_val[index]; }
#ifdef USE_SIMD
//ロード／ストアメソッド
inline __m128 load() const//ロード
{
    #ifndef USE_LOAD_STORE
        return m_val128;//ret ← m_val128
    #else//USE_LOAD_STORE
        return _mm_load_ps(m_val);//ret ← {m_val[0], m_val[1], m_val[2], m_val[3]}
    #endif//USE_LOAD_STORE
}
inline __m128 load1() const//ロード
{
    #ifndef USE_LOAD_STORE
        // [NG]
        //return _mm_set1_ps(m_val[0]);//ret ← {m_val[0], m_val[0], m_val[0], m_val[0]}
        // [OK]
        return _mm_set_ps(0.f, m_val[0], m_val[0], m_val[0]);//ret ← {m_val[0], m_val[0], m_val[0], 0.f}
    #else//USE_LOAD_STORE
        // [NG]
        //return _mm_load1_ps(m_val);//ret ← {m_val[0], m_val[0], m_val[0], m_val[0]}
    #endif//USE_LOAD_STORE

```

```

        //[OK]
        return _mm_blend_ps(//ret ← {arg0[0], arg0[1], arg0[2], arg1[3]}
            _mm_load1_ps(m_val), //arg0 ← {m_val[0], m_val[0], m_val[0], m_val[0]}
            _mm_setzero_ps(), //arg1 ← {0.f, 0.f, 0.f, 0.f}
            0x08
        );
#endif//USE_LOAD_STORE
}
inline __m128 loadS() const//ロード
{
#ifdef USE_LOAD_STORE
    return _mm_set_ss(m_val[0]); //ret ← {m_val[0], 0.f, 0.f, 0.f}
#else//USE_LOAD_STORE
    return _mm_load_ss(m_val); //ret ← {m_val[0], 0.f, 0.f, 0.f}
#endif//USE_LOAD_STORE
}
inline float loadSF() const//ロード
{
#ifdef USE_LOAD_STORE
    return m_val[0]; //ret ← m_val[0]
#else//USE_LOAD_STORE
    return getScalar(//ret ← arg0[0]
        _mm_load_ss(m_val) //arg0 ← {m_val[0], 0.f, 0.f, 0.f}
    );
#endif//USE_LOAD_STORE
}
inline void store(const __m128 val)//ストア
{
#ifdef USE_LOAD_STORE
    m_val128 = val; //val → m_val128
#else//USE_LOAD_STORE
    _mm_store_ps(m_val, val); //val[0], val[1], val[2], val[3] → m_val
#endif//USE_LOAD_STORE
}
inline void store1(const __m128 val)//ストア
{
#ifdef USE_LOAD_STORE
    //[NG]
    //m_val128 = _mm_set1_ps(//{arg0, arg0, arg0, arg0} → m_val128
    //    getScalar(val) //arg0 ← val[0]
    //);
    //[OK]
    m_val128 = _mm_blend_ps(//{arg1[0], arg1[1], arg1[2], arg2[3]} → m_val128
        _mm_set1_ps( //arg1 ← {arg0, arg0, arg0, arg0}
            getScalar(val) //arg0 ← val[0]
        ),
        _mm_setzero_ps(), //arg2 ← {0.f, 0.f, 0.f, 0.f}
        0x08
    );
#else//USE_LOAD_STORE
    //[NG]
    //mm_store1_ps(m_val, //{arg0, arg0, arg0, arg0} → m_val
    //    getScalar(val) //arg0 ← val[0]
    //);
    //[OK]
    _mm_store_ps(m_val, //arg3 → m_val
        _mm_blend_ps(//arg3 ← {arg1[0], arg1[1], arg1[2], arg2[3]}
            _mm_set1_ps( //arg1 ← {arg0, arg0, arg0, arg0}
                getScalar(val) //arg0 ← val[0]
            ),
            _mm_setzero_ps(), //arg2 ← {0.f, 0.f, 0.f, 0.f}
            0x08
        )
    );
#endif//USE_LOAD_STORE
}

```

```

}
inline void store1(const float val)//ストア
{
#ifdef USE_LOAD_STORE
    // [NG]
    // m_val128 = _mm_set1_ps(val); // {val, val, val, val} → m_val128
    // [OK]
    m_val128 = _mm_set_ps(0.f, val, val, val); // {val, val, val, 0.f} → m_val128
#else//USE_LOAD_STORE
    // [NG]
    // _mm_store1_ps(m_val, val); // {val[0], val[0], val[0], val[0]} → m_val
    // [OK]
    _mm_store_ps(m_val, //arg0 → m_val128
        _mm_set_ps(0.f, val, val, val) //arg0 ← {val, val, val, 0.f}
    );
#endif//USE_LOAD_STORE
}
inline void storeS(const __m128 val)//ストア
{
#ifdef USE_LOAD_STORE
    m_val[0] = getScalar(val); // val[0] → m_val[0]
#else//USE_LOAD_STORE
    _mm_store_ss(m_val, val); // val[0] → m_val[0]
#endif//USE_LOAD_STORE
}
inline void storeSF(const float val)//ストア
{
#ifdef USE_LOAD_STORE
    m_val[0] = val; // val → m_val[0]
#else//USE_LOAD_STORE
    _mm_store_ss(m_val, //arg0[0] → m_val[0]
        _mm_set_ss(val) //arg0 ← {val, 0.f, 0.f, 0.f}
    );
#endif//USE_LOAD_STORE
}
inline void storeZero()//ストア
{
#ifdef USE_LOAD_STORE
    m_val128 = _mm_setzero_ps(); // {0.f, 0.f, 0.f, 0.f} → m_val128
#else//USE_LOAD_STORE
    _mm_store_ss(m_val, //arg0 → m_val
        _mm_setzero_ps() //arg0 ← {0.f, 0.f, 0.f, 0.f}
    );
#endif//USE_LOAD_STORE
}
#endif//USE_SIMD
//オペレータ
inline vec3& operator+=(const vec3& rhs)
{
#ifdef USE_SIMD
    store //arg2 → m_val
    calcAdd //arg2 ← {arg0[0] + arg1[0], arg0[1] + arg1[1], arg0[2] + arg1[2], arg0[3] + arg1[3]}
    load(), //arg0 ← m_val
    rhs.load() //arg1 ← rhs.m_val
    )
    );
#else//USE_SIMD
    for (int index = 0; index < 3; ++index)
        m_val[index] += rhs.m_val[index];
#endif//USE_SIMD
    return *this;
}
inline vec3& operator+=(const float rhs)
{
#ifdef USE_SIMD

```

```

        store(arg2 → m_val
        calcAdd(arg2 ← arg0 + normalize(arg0) * arg1
        load(), arg0 ← m_val
        rhs //arg1: rhs
    )
);
#else//USE_SIMD
    vec3 rhs_tmp(*this);
    rhs_tmp.normalize();
    rhs_tmp *= rhs;
    *this += rhs_tmp;
#endif//USE_SIMD
    return *this;
}

inline vec3& operator+=(const vec3& rhs)
{
#ifdef USE_SIMD
    store(arg2 → m_val
    calcSub(arg2 ← {arg0[0] - arg1[0], arg0[1] - arg1[1], arg0[2] - arg1[2], arg0[3] - arg1[3]}
    load(), //arg0 ← m_val
    rhs.load() //arg1 ← rhs.m_val
    )
);
#else//USE_SIMD
    for (int index = 0; index < 3; ++index)
        m_val[index] -= rhs.m_val[index];
#endif//USE_SIMD
    return *this;
}

inline vec3& operator+=(const float rhs)
{
#ifdef USE_SIMD
    store(arg2 → m_val
    calcSub(arg2 ← arg0 - normalize(arg0) * arg1
    load(), arg0 ← m_val
    rhs //arg1 ← rhs
    )
);
#else//USE_SIMD
    vec3 rhs_tmp(*this);
    rhs_tmp.normalize();
    rhs_tmp *= rhs;
    *this -= rhs_tmp;
#endif//USE_SIMD
    return *this;
}

inline vec3& operator*=(const float rhs)
{
#ifdef USE_SIMD
    store(arg2 → m_val
    calcMul(arg2 ← {arg0[0] * arg1, arg0[1] * arg1, arg0[2] * arg1, arg0[3] * arg1}
    load(), arg0 ← m_val
    rhs //arg1 ← rhs
    )
);
#else//USE_SIMD
    for (int index = 0; index < 3; ++index)
        m_val[index] *= rhs;
#endif//USE_SIMD
    return *this;
}

inline vec3& operator/=(const float rhs)
{
#ifdef USE_SIMD
    store(arg2 → m_val

```

```

        calcDiv(//arg2 ← {arg0[0] / arg1, arg0[1] / arg1, arg0[2] / arg1, arg0[3] / arg1}
        load(), //arg0 ← m_val
        rhs      //arg1 ← rhs
    );
};
#else//USE_SIMD
    for (int index = 0; index < 3; ++index)
        m_val[index] /= rhs;
#endif//USE_SIMD
    return *this;
}
//キャストオペレータ
inline operator float() const
{
    return norm();
}
#ifdef USE_SIMD
    inline operator __m128() const
    {
        return load(); //ret ← m_val
    }
#endif
#ifdef DECLARE_M128_S
    inline operator __m128_s() const
    {
        return load(); //ret ← m_val
    }
#endif//DECLARE_M128_S
#endif//USE_SIMD
//コピーオペレータ
inline vec3& operator=(const vec3& rhs)
{
#ifdef USE_SIMD
    store(rhs.load()); //rhs → m_val
#else//USE_SIMD
    for (int index = 0; index < 3; ++index)
        m_val[index] = rhs.m_val[index];
#endif//USE_SIMD
    return *this;
}
#ifdef USE_SIMD
    inline vec3& operator=(const __m128 rhs)
    {
        store(rhs); //rhs → m_val
        return *this;
    }
#endif
#ifdef DECLARE_M128_S
    inline vec3& operator=(const __m128_s& rhs)
    {
        store(rhs);
        return *this;
    }
#endif//DECLARE_M128_S
#endif//USE_SIMD
//メソッド
//内積
inline float dot(const vec3& target) const
{
#ifdef USE_SIMD
    return calcDot(//ret ← calcDot(arg0, arg1)
        load(),      //arg0 ← m_val
        target.load() //arg1 ← target.m_val
    );
#else//USE_SIMD
    return m_val[0] * target.m_val[0] +
        m_val[1] * target.m_val[1] +

```

```

        m_val[2] * target.m_val[2];
#endif//USE_SIMD
}
//正規化内積
inline float normalizeDot(const vec3& target) const
{
#ifdef USE_SIMD
    return calcNormalizeDot(//ret ← calcNormalizeDot(arg0, arg1)
        load(),           //arg0 ← m_val
        target.load()//arg1 ← target.m_val
    );
#else//USE_SIMD
    vec3 tmp1(*this);
    vec3 tmp2(target);
    tmp1.normalize();
    tmp2.normalize();
    return tmp1.m_val[0] * tmp2.m_val[0] +
        tmp1.m_val[1] * tmp2.m_val[1] +
        tmp1.m_val[2] * tmp2.m_val[2];
#endif//USE_SIMD
}
//外積
#ifdef USE_SIMD
inline __m128_s cross(const vec3& target) const
{
    return calcCross(//ret ← calcCross(arg0, arg1)
        load(),       //arg0 ← m_val
        target.load()//arg1 ← target.m_val
    );
}
#else//USE_SIMD
inline vec3 cross(const vec3& target) const
{
    return vec3(
        m_val[1] * target.m_val[2] - m_val[2] * target.m_val[1],
        m_val[2] * target.m_val[0] - m_val[0] * target.m_val[2],
        m_val[0] * target.m_val[1] - m_val[1] * target.m_val[0]
    );
}
#endif//USE_SIMD
//ノルム^2 取得
inline float norm2() const
{
#ifdef USE_SIMD
    return calcNorm2(//ret ← calcNorm2(arg0)
        load()//arg0 ← m_val
    );
#else//USE_SIMD
    return m_val[0] * m_val[0] +
        m_val[1] * m_val[1] +
        m_val[2] * m_val[2];
#endif//USE_SIMD
}
//ノルム取得
inline float norm() const
{
#ifdef USE_SIMD
    return calcNorm(//ret ← calcNorm(arg0)
        load()//arg0 ← m_val128
    );
#else//USE_SIMD
    return sqrt(norm2());
#endif//USE_SIMD
}
//長さ^2 取得

```

```

inline float length2() const
{
    return norm2();
}
//長さ取得
inline float length() const
{
    return norm();
}
//正規化
inline vec3& normalize()
{
#ifdef USE_SIMD
    store(&arg1 → m_val
        calcNormalize(&arg1 ← calcNormalize(arg0)
            load(&arg0 ← m_val
        )
    );
#else//USE_SIMD
    const float val_norm = norm();
    for (int index = 0; index < 3; ++index)
        m_val[index] /= val_norm;
#endif//USE_SIMD
    return *this;
}
#ifdef USE_SIMD
inline __m128_s normalize() const
{
    return calcNormalize(&ret ← calcNormalize(arg0)
        load(&arg0 ← m_val
    );
}
#else//USE_SIMD
inline vec3 normalize() const
{
    const float val_norm = norm();
    return vec3(m_val[0] / val_norm, m_val[1] / val_norm, m_val[2] / val_norm);
}
#endif//USE_SIMD
//距離^2 計算
inline float distance2(const vec3& from)
{
#ifdef USE_SIMD
    return calcDistance2(&ret ← calcDist2(arg0, arg1)
        from. load(), &arg0 ← from.m_val
        load() //arg1 ← m_val
    );
#else//USE_SIMD
    vec3 tmp(*this);
    tmp -= from;
    return tmp.norm2();
#endif//USE_SIMD
}
//距離計算
inline float distance(const vec3& from)
{
#ifdef USE_SIMD
    return calcDistance(&ret ← calcDist(arg0, arg1)
        from. load(), &arg0 ← from.m_val
        load() //arg1 ← m_val
    );
#else//USE_SIMD
    vec3 tmp(*this);
    tmp -= from;
    return tmp.norm();
}

```



```

#endif//USE_SIMD
}
//コピーコンストラクタ
inline vec3(const vec3& obj)
#ifdef USE_SIMD
#ifndef USE_LOAD_STORE
: m_val128(obj.m_val128)//obj.m_val128 → m_val128
{}
#else//USE_LOAD_STORE
{
store(obj.load())://obj.m_val → m_val
}
#endif//USE_LOAD_STORE
#else//USE_SIMD
{
for (int index = 0; index < 3; ++index)
m_val[index] = obj.m_val[index];
}
#endif//USE_SIMD
//コンストラクタ
inline vec3(const float x, const float y, const float z)
#ifdef USE_SIMD
#ifndef USE_LOAD_STORE
:m_val128(_mm_set_ps(0.f, z, y, x))//[x, y, z, 0.f] → m_val128
{}
#else//USE_LOAD_STORE
{
store(_mm_set_ps(0.f, z, y, x))//[x, y, z, 0.f] → m_val
}
#endif//USE_LOAD_STORE
#else//USE_SIMD
{
m_val[0] = x;
m_val[1] = y;
m_val[2] = z;
}
#endif//USE_SIMD
inline vec3(const float val)
#ifdef USE_SIMD
#ifndef USE_LOAD_STORE
//[NG]
//:m_val128(_mm_set1_ps(val))//[val, val, val, val] → m_val128
//[OK]
:m_val128(_mm_set_ps(0.f, val, val, val))//[val, val, val, 0.f] → m_val128
{}
#else//USE_LOAD_STORE
{
//[NG]
//store(_mm_set1_ps(val))//[val, val, val, val] → m_val
//[OK]
store1(val)//[val, val, val, 0.f] → m_val
}
#endif//USE_LOAD_STORE
#else//USE_SIMD
{
for (int index = 0; index < 3; ++index)
m_val[index] = val;
}
#endif//USE_SIMD
#ifdef USE_SIMD
inline vec3(const __m128 val)
#ifndef USE_LOAD_STORE
:m_val128(val)//val → m_val128
{}
#else//USE_LOAD_STORE

```

```

    {
        store(val)://val → m_val
    }
#endif//USE_LOAD_STORE
#ifdef DECLARE_M128_S
    inline vec3(const __m128_s& val)
#endif USE_LOAD_STORE
    :m_val128(val)//val → m_val128
    {}
#else//USE_LOAD_STORE
    {
        store(val)://val → m_val
    }
#endif//USE_LOAD_STORE
#endif//DECLARE_M128_S
#endif//USE_SIMD
    inline vec3()
#ifdef USE_SIMD
#endif USE_LOAD_STORE
    :m_val128(_mm_setzero_ps())//{0.f, 0.f, 0.f, 0.f} → m_val128
    {}
#else//USE_LOAD_STORE
    {
        storeZero()//{0.f, 0.f, 0.f, 0.f} → m_val128
    }
#endif//USE_LOAD_STORE
#else//USE_SIMD
    {
        for (int index = 0; index < 3; ++index)
            m_val[0] = 0.f;
    }
#endif//USE_SIMD
    //デストラクタ
    inline ~vec3()
    {}
};

//二項演算子用オペレータ
#ifdef USE_SIMD
//※SIMD 演算時は vec3 型のままではなく、__m128 型として演算し、__m128_s 型を返す
inline __m128_s operator+(const vec3& lhs, const vec3& rhs){
    return calcAdd(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator+(const vec3& lhs, const __m128_s& rhs){
    return calcAdd(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator+(const __m128_s& lhs, const vec3& rhs){
    return calcAdd(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator+(const __m128_s& lhs, const __m128_s& rhs){
    return calcAdd(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator+(const vec3& lhs, const float rhs){
    return calcAdd(static_cast<__m128>(lhs), rhs); }
inline __m128_s operator+(const __m128_s& lhs, const float rhs){
    return calcAdd(static_cast<__m128>(lhs), rhs); }

inline __m128_s operator-(const vec3& lhs, const vec3& rhs){
    return calcSub(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator-(const vec3& lhs, const __m128_s& rhs){
    return calcSub(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator-(const __m128_s& lhs, const vec3& rhs){
    return calcSub(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator-(const __m128_s& lhs, const __m128_s& rhs){
    return calcSub(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s operator-(const vec3& lhs, const float rhs){
    return calcSub(static_cast<__m128>(lhs), rhs); }
inline __m128_s operator-(const __m128_s& lhs, const float rhs){
    return calcSub(static_cast<__m128>(lhs), rhs); }

```

```

inline __m128_s operator*(const vec3& lhs, const float rhs){
    return calcMul(static_cast<__m128>(lhs), rhs); }
inline __m128_s operator*(const __m128_s& lhs, const float rhs){
    return calcMul(static_cast<__m128>(lhs), rhs); }

inline __m128_s operator/(const vec3& lhs, const float rhs){
    return calcDiv(static_cast<__m128>(lhs), rhs); }
inline __m128_s operator/(const __m128_s& lhs, const float rhs){
    return calcDiv(static_cast<__m128>(lhs), rhs); }

inline float dot(const vec3& lhs, const vec3& rhs){
    return calcDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline float dot(const vec3& lhs, const __m128_s& rhs){
    return calcDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline float dot(const __m128_s& lhs, const vec3& rhs){
    return calcDot(lhs, static_cast<__m128>(rhs)); }
inline float dot(const __m128_s& lhs, const __m128_s& rhs){
    return calcDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }

inline float normalizeDot(const vec3& lhs, const vec3& rhs){
    return calcNormalizeDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline float normalizeDot(const vec3& lhs, const __m128_s& rhs){
    return calcNormalizeDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline float normalizeDot(const __m128_s& lhs, const vec3& rhs){
    return calcNormalizeDot(lhs, static_cast<__m128>(rhs)); }
inline float normalizeDot(const __m128_s& lhs, const __m128_s& rhs){
    return calcNormalizeDot(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }

inline __m128_s cross(const vec3& lhs, const vec3& rhs){
    return calcCross(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s cross(const vec3& lhs, const __m128_s& rhs){
    return calcCross(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }
inline __m128_s cross(const __m128_s& lhs, const vec3& rhs){
    return calcCross(lhs, static_cast<__m128>(rhs)); }
inline __m128_s cross(const __m128_s& lhs, const __m128_s& rhs){
    return calcCross(static_cast<__m128>(lhs), static_cast<__m128>(rhs)); }

inline float norm2(const vec3& val){
    return calcNorm2(static_cast<__m128>(val)); }
inline float norm2(const __m128_s& val){
    return calcNorm2(static_cast<__m128>(val)); }
inline float norm(const vec3& val){
    return calcNorm(static_cast<__m128>(val)); }
inline float norm(const __m128_s& val){
    return calcNorm(static_cast<__m128>(val)); }

inline float length2(const vec3& val){
    return calcNorm2(static_cast<__m128>(val)); }
inline float length2(const __m128_s& val){
    return calcNorm2(static_cast<__m128>(val)); }
inline float length(const vec3& val){
    return calcNorm(static_cast<__m128>(val)); }
inline float length(const __m128_s& val){
    return calcNorm(static_cast<__m128>(val)); }

inline __m128_s normalize(const vec3& val){
    return calcNormalize(static_cast<__m128>(val)); }
inline __m128_s normalize(const __m128_s& val){
    return calcNormalize(static_cast<__m128>(val)); }

inline float distance2(const vec3& from, const vec3& to){
    return calcDistance2(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance2(const vec3& from, const __m128_s& to){
    return calcDistance2(static_cast<__m128>(from), static_cast<__m128>(to)); }

```

```

inline float distance2(const __m128_s& from, const vec3& to) {
    return calcDistance2(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance2(const __m128_s& from, const __m128_s& to) {
    return calcDistance2(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance(const vec3& from, const vec3& to) {
    return calcDistance(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance(const vec3& from, const __m128_s& to) {
    return calcDistance(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance(const __m128_s& from, const vec3& to) {
    return calcDistance(static_cast<__m128>(from), static_cast<__m128>(to)); }
inline float distance(const __m128_s& from, const __m128_s& to) {
    return calcDistance(static_cast<__m128>(from), static_cast<__m128>(to)); }

#else//USE_SIMD
//※非 SIMD 演算時は vec3 型のまま演算する
inline vec3 operator+(const vec3& lhs, const vec3& rhs) { vec3 tmp(lhs); return (tmp += rhs); }
inline vec3 operator+(const vec3& lhs, const float rhs) { vec3 tmp(lhs); return (tmp += rhs); }

inline vec3 operator-(const vec3& lhs, const vec3& rhs) { vec3 tmp(lhs); return (tmp -= rhs); }
inline vec3 operator-(const vec3& lhs, const float rhs) { vec3 tmp(lhs); return (tmp -= rhs); }

inline vec3 operator*(const vec3& lhs, const float rhs) { vec3 tmp(lhs); return (tmp *= rhs); }

inline vec3 operator/(const vec3& lhs, const float rhs) { vec3 tmp(lhs); return (tmp /= rhs); }

inline float dot(const vec3& lhs, const vec3& rhs) { return lhs.dot(rhs); }
inline float normalizeDot(const vec3& lhs, const vec3& rhs) { return lhs.normalizeDot(rhs); }

inline vec3 cross(const vec3& lhs, const vec3& rhs) { return lhs.cross(rhs); }

inline float norm2(const vec3& val) { return val.norm2(); }
inline float norm(const vec3& val) { return val.norm(); }

inline float length2(const vec3& val) { return val.length2(); }
inline float length(const vec3& val) { return val.length(); }

inline vec3 normalize(const vec3& val) { vec3 tmp(val); return tmp.normalize(); }

inline float distance2(const vec3& from, const vec3& to) { vec3 tmp(to); return tmp.distance2(from); }
inline float distance(const vec3& from, const vec3& to) { vec3 tmp(to); return tmp.distance(from); }
#endif//USE_SIMD

#endif//__VECTOR_H_

```

【sub.cpp】

```

#include "vector.h"

//静的変数
static float s_val1[3] = { 1.f, 2.f, 3.f };
static float s_val2[3] = { -4.f, -5.f, -6.f };

//パフォーマンステスト用関数
//※コンパイラの最適化の影響が起らないように、ソースファイルを分けて、かつ、ダミーパラメータを受け渡し
void sub(vec3& v1, vec3& v1_n, vec3& v2, vec3& v2_n, vec3& vdist, vec3& vdist_n, float& dist_val,
        float& dot_val, float& dot_val_n, vec3& vcross, vec3& vcross_n, vec3& v3, vec3& v3_n,
        const float add, const float mul, const float div, const float sub, const int dummy)
{
    vec3 tmp1(s_val1[0], s_val1[1], s_val1[2]);
    vec3 tmp2(s_val2[0], s_val2[1], s_val2[2]);
    v1 = tmp1;
    v1_n = normalize(tmp1);
    v2 = tmp2;
    v2_n = normalize(tmp2);
    vdist = tmp2 - tmp1;
    vdist_n = normalize(vdist);
}

```

```

    dist_val = distance(tmp1, tmp2);
    dot_val = dot(tmp1, tmp2);
    dot_val_n = normalizeDot(tmp1, tmp2);
    vcross = cross(tmp1, tmp2);
    vcross_n = normalize(vcross);
    vec3 tmp3 = (vcross + add) * mul / div - sub;
    v3 = tmp3;
    v3_n = normalize(tmp3);
}

```

【main.cpp】

```

#include "vector.h"

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

#include <chrono> // 処理時間計測用

extern void sub(vec3& v1, vec3& v1_n, vec3& v2, vec3& v2_n, vec3& vdist, vec3& vdist_n, float& dist_val,
               float& dot_val, float& dot_val_n, vec3& vcross, vec3& vcross_n, vec3& v3, vec3& v3_n,
               const float add, const float mul, const float div, const float sub, const int dummy);

// テストメイン
int main(const int argc, const char* argv[])
{
    const auto start = std::chrono::system_clock::now();
    vec3 v1;
    vec3 v1_n;
    vec3 v2;
    vec3 v2_n;
    vec3 vdist;
    vec3 vdist_n;
    float dist_val;
    float dot_val;
    float dot_val_n;
    vec3 vcross;
    vec3 vcross_n;
    vec3 v3;
    vec3 v3_n;
    const int repeat = 10000000;
    for (int loop = 0; loop < repeat; ++loop)
    {
        sub(v1, v1_n, v2, v2_n, vdist, vdist_n, dist_val, dot_val, dot_val_n, vcross, vcross_n, v3, v3_n,
            1.f, 2.f, 3.f, 4.f, loop);
    }
    const auto end = std::chrono::system_clock::now();
    const auto duration = end - start;
    const float sec = static_cast<float>(static_cast<double>(
        std::chrono::duration_cast<std::chrono::microseconds>(duration).count()) / 1000000.);

#ifdef USE_SIMD
    printf("<use simd>%n");
#else // USE_SIMD
    printf("<not use simd>%n");
#endif // USE_SIMD
    printf("repeat = %d times%n", repeat);
    printf("elapsed time = %.6f sec%n", sec);
    printf("v1      =[%.2f, %.2f, %.2f] (%.4f)%n", v1.x(), v1.y(), v1.z(), v1.length());
    printf("v1_n    =[%.2f, %.2f, %.2f] (%.4f)%n", v1_n.x(), v1_n.y(), v1_n.z(), v1_n.length());
    printf("v2      =[%.2f, %.2f, %.2f] (%.4f)%n", v2.x(), v2.y(), v2.z(), v2.length());
    printf("v2_n    =[%.2f, %.2f, %.2f] (%.4f)%n", v2_n.x(), v2_n.y(), v2_n.z(), v2_n.length());
    printf("vdist   =[%.2f, %.2f, %.2f] (%.4f)%n", vdist.x(), vdist.y(), vdist.z(), vdist.length());
    printf("vdist_n =[%.2f, %.2f, %.2f] (%.4f)%n", vdist_n.x(), vdist_n.y(), vdist_n.z(), vdist_n.length());
    printf("dist_val =%.4f%n", dist_val);
    printf("dot_val  =%.4f%n", dot_val);
    printf("dot_val_n=%.4f%n", dot_val_n);
}

```

```

printf("vcross  =[% .2f, % .2f, % .2f] (%.4f)¥n", vcross.x(), vcross.y(), vcross.z(), vcross.length());
printf("vcross_n=[% .2f, % .2f, % .2f] (%.4f)¥n", vcross_n.x(), vcross_n.y(), vcross_n.z(), vcross_n.length());
printf("v3      =[% .2f, % .2f, % .2f] (%.4f)¥n", v3.x(), v3.y(), v3.z(), v3.length());
printf("v3_n    =[% .2f, % .2f, % .2f] (%.4f)¥n", v3_n.x(), v3_n.y(), v3_n.z(), v3_n.length());
return EXIT_SUCCESS;
}

```

↓（処理結果）

```

↓ SIMD 演算有効化時
<use simd>
repeat = 10000000 times
elapsed time = 0.098004 sec
v1      =[1.00, 2.00, 3.00] (3.7417)
v1_n    =[0.27, 0.53, 0.80] (1.0000)
v2      =[-4.00, -5.00, -6.00] (8.7750)
v2_n    =[-0.46, -0.57, -0.68] (1.0000)
vdist   =[-5.00, -7.00, -9.00] (12.4499)
vdist_n =[-0.40, -0.56, -0.72] (1.0000)
dist_val=12.4499
dot_val =-32.0000
dot_val_n=-0.9746
vcross  =[3.00, -6.00, 3.00] (7.3485)
vcross_n=[0.41, -0.82, 0.41] (1.0000)
v3      =[0.64, -1.28, 0.64] (1.5656)
v3_n    =[0.41, -0.82, 0.41] (1.0000)

↓ SIMD 演算無効化時
<not use simd>
repeat = 10000000 times
elapsed time = 1.533085 sec          ←約 15 倍の性能差が確認できる
v1      =[1.00, 2.00, 3.00] (3.7417)
v1_n    =[0.27, 0.53, 0.80] (1.0000)
v2      =[-4.00, -5.00, -6.00] (8.7750)
v2_n    =[-0.46, -0.57, -0.68] (1.0000)
vdist   =[-5.00, -7.00, -9.00] (12.4499)
vdist_n =[-0.40, -0.56, -0.72] (1.0000)
dist_val=12.4499
dot_val =-32.0000
dot_val_n=-0.9746
vcross  =[3.00, -6.00, 3.00] (7.3485)
vcross_n=[0.41, -0.82, 0.41] (1.0000)
v3      =[0.64, -1.28, 0.64] (1.5656)
v3_n    =[0.41, -0.82, 0.41] (1.0000)

```

● 最適化（高速化）のポイント

上記のサンプルは、単純に計算処理を SIMD 演算に対応しただけではなく、更なる最適化も行っている。以下、主な最適化のポイントを示す。

➤ 極力メモリアクセスを避ける

ポインタアクセスやクラスのメンバー変数へのアクセスを極力避ける。などもアクセスが必要な変数は一旦ローカル変数に代入し、最後にメンバー変数等に戻すようにする。

➤ なるべく計算式をつなぐ

計算結果の代入をできるだけ減らし、計算式を可能な限り一本化する

➤ 短い関数は積極的にインライン展開する

分岐・ループを含まない数行の処理は積極的にインライン関数化する

▼ CPU キャッシュメモリに関する最適化

(準備中)

- 【基礎】CPU キャッシュメモリの構成

(準備中)

- 【基礎】メインメモリとL1 キャッシュ～L3 キャッシュメモリ

(準備中)

- 【基礎】インストラクションキャッシュとデータキャッシュ

(準備中)

- 【基礎】キャッシュライトバックとキャッシュライトスルー

(準備中)

- 【基礎】キャッシュコヒーレンシとバススヌーピング

(準備中)

- 最適化①：インストラクションキャッシュに関する最適化

(準備中)

- 最適化②：データキャッシュに関する最適化

(準備中)

■ 高度な最適化に関する Tips

本書標題が示す「本当にちょっとした」という枠を大きく超えて、最適化手法の延長として、高度な知識に基づく最適化手法を示す。

▼ 命令パイプラインに関する最適化

(準備中)

● 【基礎】 CPU の命令とサイクル

(準備中)

● 【基礎】 命令パイプラインとスーパースカラー

(準備中)

● 【基礎】 アウト・オブ・オーダー実行

(準備中)

● 【基礎】 同時マルチスレッディング (SMT = Simultaneous MultiThreading)

(準備中)

● 【基礎】 ハザードとストール

(準備中)

● 最適化 : ストールの回避策

(準備中)

■■■ 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