効果的なデバッグログとアサーション

– 効果的なデバッグトレース手法 –

2014 年 2 月 24 日 初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 2 月 24 日	板垣 衛	（初稿）

■ 目次

■ 概略

より開発効率を向上させるための、一歩進んだデバッグログとアサーションのシステムを設計する。

■ 目的

本書は、問題発生時に迅速に的確な原因追跡を行うことを可能とするシステムを設計し、開発効率を向上させることを目的とする。

開発環境以外のランタイム時に発生する問題を捕捉し、原因追求を可能とすることも目的とする。

■ マルチスレッドについて

本書のデバッグロギングシステムは、マルチスレッドシステムとして最適化するため、以降の説明にはマルチスレッドプログラミングの用語を用いている。用語の説明については、別紙の「マルチスレッドプログラミングの基礎」を参照。

本書では、「アトミック型」（または「インターロック操作」）、「セマフォ」、「モニター」（「条件変数」）といった要素を扱っている。

■ 要件定義

▼ 基本要件

本書が扱うシステムの基本要件は下記のとおり。

- マルチスレッド環境で安全に利用可能なデバッグログシステムとする。
- デバッグログの出力処理がメイン処理の実行のパフォーマンスを極力阻害しないものとする。
- スレッドの先着処理順に的確に表示しつつ、スレッドの処理をブロックしないものとする。
- 共通処理のログは「何の処理に使われたものか」がすぐにわかるようなシステムとする。

・ ログレベル（ログ表示のマスク）はランタイム時に動的に変更できるものとする。
・ スタッフごとに「既定のログレベル」を設定できるものとする。
・ アサーション違反発生時はブレークポイント割り込みを発生させるものとする。
・ アサーション違反発生時はコールスタックを表示するものとする。
・ 重大なログは画面にも表示し、ランタイム時にも問題発生を気づかせるようにするものとする。
・ ランタイムでのアサーション違反発生時は、ゲームを自動的にポーズするものとし、さらに、その後の処理続行も可能なものとする。
・ メッセージ出力時は文字コードを指定する事で、シフト JIS コードと UTF-8 コードのどちらも表示できるものとする。

▼ 要求仕様／要件定義

以下、本書が扱うシステムの要件を定義する。なお、要件として不確定の要求仕様も併記する。

・ デバッグログは、メッセージをキューイングすることでゲーム本体のパフォーマンスに極力影響しないものとする。

・ キューイングされたメッセージは、ログ出力スレッドによって逐次出力する。
  ➢ ログ出力スレッドの優先度は、メインスレッドよりも低く設定する。

・ メッセージ出力時は、メッセージのレベル（重大度）とカテゴリを指定する。

・ メッセージ出力時は、（ランタイムの）ログレベルにより、出力するメッセージをレベルに応じて制限する。
  ➢ つまり、常にすべてのログがビルドされる。（プログラム上に実装される。）
  ➢ メッセージ出力のマスク判定はメッセージ出力関数内で行い、出力しない場合はメッセージのキューイングも行わない。

・ ランタイム時の「ログレベル」の変更は、デバッグメニューなどの外部のシステムから行えるものとする。

・ 規定の「ログレベル」は、「ユーザー設定ファイル」に記録される情報の一つとして扱い、ユーザーごとに規定を設定することを可能とする。
  ➢ 「ユーザー設定ファイル」は、ユーザーごとのゲームの挙動を設定したファイルで、ユーザー自身の PC 上に配置して扱う。ゲーム起動時のそのファイルが読み込まれる。
  ➢ 主に制作スタッフや QA スタッフが自身に関係のあるメッセージしか表示させないようにするために使用する。

・ ゲーム画面上への表示が必要なメッセージ（とくに重大度の高いものなど）については、ログ出力スレッドが画面表示用のキューにメッセージを再投入して扱う。
  - つまり、メッセージを「どう扱うか？」の判断は、やはりログ出力スレッドが行う。
  - 「どこまでのログを画面に表示させるか？」については、デバッグメニューやユーザー設定ファイルを通して、ランタイム時に設定できるものとする。

・ ゲーム画面にログを表示するシステムは、別途メインスレッド（メインループ）で稼働し、キューを拾って逐次処理する。
  - ログをピックアップしてキューから削除するシステムと、画面に表示するシステムは処理のレイヤーを分け、ゲームタイトルに応じた表示処理を行えるものとする。
  - 画面に大きくメッセージを表示するような場合は、メッセージを全文表示しないような工夫を行う。例えば、「エラー : XXX ¥t 理由…¥n…」といったメッセージの場合、画面上には「¥t」や「¥n」までしか表示しないといった対応を取る。または、40 文字で表示を打ち切るなど、タイトル固有の仕様で実装する。

・ メッセージのバッファはロギングシステムが用意し、（ゲーム側の）各スレッドが自身のスタックをほとんど使用しないものとする。
  - これにより、少ないスタックで動作するスレッドからもメッセージの長さをほとんど気にせずに、ログ出力できるものとする。
  - ロギングシステムはあらかじめ幾つかの固定バッファ（2KB×16 個ほど）を用意しておき、ログ出力の都度未使用のバッファを処理に割り当てる。
    - バッファを「16 個ほど」と多めに記したのは、それだけのスレッドが同時にプリントする可能性を考慮したものではなく、一気に大量のプリント文が列挙される可能性を考慮したもの。
  - ログ出力処理（ゲーム側がログを出力する処理）は、固定バッファへの書き込みが終了したら即座に処理が返る。
  - 固定バッファが全て使用中で割り当てができない時は、ゲーム側のスレッドがブロックして待機する。
    - これが起こらないように、十分な数のバッファを用意しておかなければならない。
    - バッファのサイズと数はゲームタイトルに応じて設定可能。
    - ブロック処理には、単純にはセマフォを活用。ゲーム終了時に確実に開放し、解放忘れのデッドロックを起こさないように注意。
  - メッセージをキューイングするためのバッファは別途用意されており、固定バッファからのキュー（バッファ）への移し替えが完了した時に、固定バッファが解放される。
    - このため、固定バッファの利用状況を監視し、キューに移し替えするだけのスレッドも用意する。
    - この固定バッファ監視スレッドの優先度はメインスレッドと同じか、それより高く設定する。（ゲーム側の処理をブロックすることにつながるため）

- このキューの実装は、メッセージ情報をプールアロケータからメモリを割り当てたうえで連結リスト化し、不定長のメッセージテキストをスマートスタックアロケータで管理する手法が簡単である。これらのメモリアロケータについては、別紙の「様々なメモリ管理手法と共通アロケータインターフェース」で詳しく説明する。

・ メッセージにはシーケンス番号を付け、マルチスレッドで確実に処理の到達順序を保証して出力するものとする。
  - ➢ アトミック型のシーケンス番号を用意し、ログ出力時には真っ先にシーケンス番号の発番を行う。
    - アトミック型もしくはインターロック操作を用いることにより、ロックフリーな発番を、各処理スレッド自身で行う。
  - ➢ 例えば、二つのスレッドがほぼ同時にログ出力処理を行った際、先着のスレッドの方が書き込みに時間がかかり、完了が後になる可能性がある。このような場合でも、ログ出力スレッドは、シーケンス番号どおりのメッセージが到着するのを待ってから出力する。
    - ただし、結局ログ出力処理によって、スレッドの本来の処理効率を損なう問題は残る。この対処として、シーケンス番号だけ先に発番して、後で（一通りの処理が完了してから）メッセージを書き込む処理にも対応する。
    - シーケンス番号の先行発番は複数行ってもよい。
    - 仮に「シーケンス番号を先行発番したが、対応するメッセージが出力されなかった」ということがあると、そこでログ出力が完全に停止してしまう。この問題の対処として、ログ出力スレッドは、一定時間待ってもキューイングされない場合はタイムアウトしてスキップするものとする。その待機時間はせいぜい 1～2 ミリ秒ほど。
    - タイムアウト時間を過ぎて書き込まれたメッセージもログ出力される。
    - ユーザーが挙動を完全に捕捉できるように、オプションにより、シーケンス番号付きでメッセージをログ出力するモードにも対応する。（デバッグメニューなどから ON/OFF できる）

・ 「コールポイントシステム」を用意し、共通処理のメッセージを呼び出し元に応じてマスクすることを可能とする。
  - ➢ 処理ブロック内（関数の先頭など）で、「コールポイント宣言」を行うことで、共通処理のログ出力時に「どのコールポイントを通過したか？」を判定できる。
  - ➢ コールポイントは階層的にスタックされ、「コールポイントに基づくログ出力マスク」のような処理は、直近のコールポイントに基づいて処理する。
  - ➢ コールポイントの情報は、ローカル変数の情報をそのまま扱う。
  - ➢ TLS（スレッドローカルストレージ）を活用し、スレッド間で干渉することなく、安全にコールポイントのスタックを扱う。

・ アサーションにもこのロギングシステムを活用する。
  - アサーション違反発生時は、ブレークポイント割り込みを発生させる。
    - Windows なら DebugBreak()関数の呼び出し、Unix 系なら SIGTRAP シグナルの発行など。
  - アサーション違反時は、ランタイムでも気がつけるように、重大なメッセージとして、ログ出力、画面表示も行う。
  - ログ出力では、コールスタックも表示する。
    - ランタイムでシンボル情報を読み取ってコールスタックを表示するライブラリがある。後述する。
  - ログ出力では、コールポイントのスタックも表示する。
    - 情報が冗長するようなら、コールスタックかコールポイントスタックのどちらかは表示しないように処理を変える。
  - 【オプション】アサーション違反発生時は、自動的にゲーム（ゲームループ）をポーズ状態にし、「START」ボタンなどで続行可能とする。
  - ブレークポイント割り込みを呼び出す直前に、キューイングされたログをフラッシュ（強制全出力）する。
    - メッセージを出力したあと、すぐにフラッシュし、フラッシュの完了を待つ。
    - コールポイントスタックの出力は、通常のメッセージ出力の仕組みを用いられるため、フラッシュを行う前に出力する。
    - コールスタックの出力は、Boost C++ ライブラリの backtrace を用いることを想定しており、「`std::ostream <<`」で出力する必要がある。このインターフェースには無理に対応せず、フラッシュの完了を待ってから、「`std:cerr << boost::trace(e);`」といった処理でそのまま出力する。
    - 処理順序としては、「メッセージ出力」→「コールポイントスタック出力」→「ログ出力フラッシュ」（完了待ち）→「コールスタック出力」→「ブレークポイント割り込み」（開発環境以外では無反応）→「ポーズ」となる。

・ アサーション以外にも、任意の「ウォッチポイント」、「ブレークポイント」を設定可能とする。
  - ウォッチポイントはアサーションと逆に「ブレークする条件」を式に記述する。
  - ウォッチポイント、ブレークポイントは、メッセージをログ出力してブレークポイント割り込みを発生させる。
    - この時のメッセージの重大度も指定する。
    - ログレベルによってマスクされたメッセージのウォッチポイント、ブレークポイントは、ブレークポイント割り込みも反応しない。

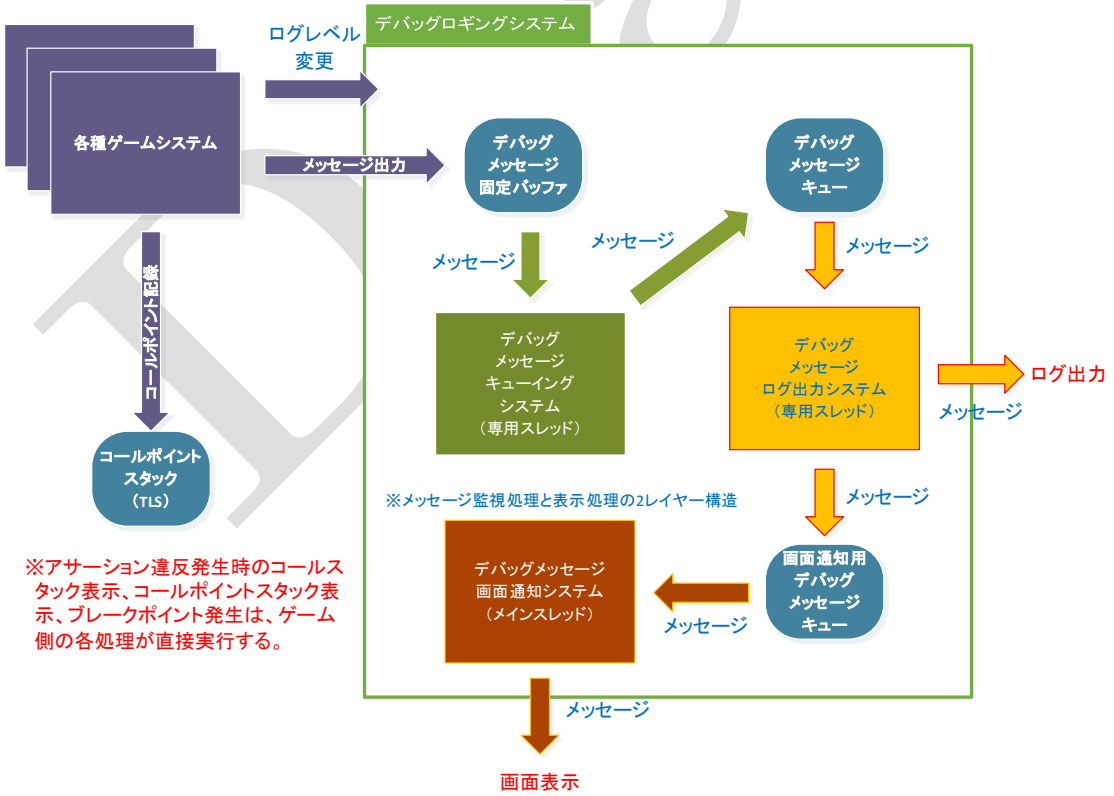
・ メッセージ出力時は、そのメッセージが「UTF-8」か「シフト JIS」か指定できるものとする。
  - ➢ 文字コード変換は、ログ出力スレッドがメッセージを出力する際に行う。
  - ➢ 一つのゲームシステム上で、両方の文字コードを意識的に使用するケースがあるため、この仕組みを用意する。
    - UTF-8 の方が欧州文字の扱いなど表現の問題が少ないが、シフト JIS の方がテキストのサイズが小さくなるため、ゲーム画面上で扱わないテキストやソースコード埋め込みのテキストはシフト JIS コードで扱われることも多い。

■ 仕様概要

▼ システム構成図

要件に基づくシステム構成図を示す。

デバッグロギングシステムのシステム構成図：



■ 処理仕様

処理仕様として、各処理の使用方法を示す。

▼ デバッグメッセージ

デバッグメッセージは基本的にプリント文である。重大度とカテゴリを指定してプリントする。

ログレベルの設定と画面通知レベルの設定が可能。ログレベル、画面通知レベルはカテゴリごとに設定する。全カテゴリの一括設定も可能。

また、スレッド間の処理到達順序を保証したメッセージ出力のための「遅延メッセージ」に対応する。

メッセージのシーケンス番号を表示に加えることも可能。

デバッグメッセージの使用イメージ：

```
//---------------------------------------
//デバッグメッセージ出力
//※メッセージレベルとカテゴリを指定

using dbg;//関数や定数を簡潔に使用するためのネームスペース using 宣言

//Lv.1：通常メッセージ
printAsNormal(forAny, "通常メッセージ by %s¥n", name);
//Lv.0：冗長メッセージ
printAsVerbose(forAny, "冗長メッセージ by %s¥n", name);
//Lv.0：詳細メッセージ
printAsDetail(forAny, "詳細メッセージ by %s¥n", name);
//Lv.2：重要メッセージ
printAsImportant(forAny, "重要メッセージ by %s¥n", name);
//Lv.3：警告メッセージ
printAsWarning(forAny, "警告メッセージ by %s¥n", name);
//LV.4：重大（問題）メッセージ
printAsCritical(forAny, "重大メッセージ by %s¥n", name);
//Lv.5：絶対（必須）メッセージ
printAsAbsolute(forAny, "絶対メッセージ by %s¥n", name);

//レベルを引数指定
print(asNormal, forAny, "通常メッセージ by %s¥n", name);

//---------------------------------------
//ログレベル

//ログレベル変更
//※指定の値以上のレベルのメッセージのみをログ出力する
setLogLevel(asWarning, forAny);//警告以上のメッセージのみログ出力

//ログレベル取得
getLogLevel(forAny);//戻り値としてログレベルが返る

//---------------------------------------
//画面通知レベル

//画面通知レベル変更
```

```
//※ログレベルより低いレベルを設定しても通知されない
setNoticeLevel(asCritical, forAny);//重大メッセージのみ画面通知
//※各レベルの推奨表示カラー
//  Lv.1：通常       ... 黒
//  Lv.0：冗長/詳細 ...  （表示されない）
//  Lv.2：重要       ... 青
//  Lv.3：警告       ... 紫
//  Lv.4：重大       ... 赤
//  Lv.5：絶対       ...  （表示されない）

//画面通知レベル取得
getNoticeLevel(forAny);//戻り値として画面通知レベルが返る

//---------------------------------------
//定数

//レベル定数
enum levelEnum : unsigned char
{
     asNormal = 1,//通常メッセージ
     asVerbose = 0,//冗長メッセージ
     asDetail = 0,//詳細メッセージ
     asImportant = 2,//重要メッセージ
     asWarning = 3,//警告メッセージ
     asCritical = 4,//重大メッセージ
     asAbsolute = 5,//絶対メッセージ（ログレベルに関係なく出力したいメッセージ）
     //以下、ログレベル／画面通知レベル変更用
     asSilent = 5,//静寂（絶対メッセージ以外出力しない）
     asSilentAbsolutely = 6,//絶対静寂（全てのメッセージを出力しない）
};

//カテゴリ定数
//※制作スタッフにとって分かり易く、
//  メッセージを仕込むプログラマーにとって
//  分かり易い程度の分類にする
//※forReserved**を任意の用途に割り当てて使用可
enum categoryEnum : unsigned char
{
     forAny = 0,//なんでも（カテゴリなし）
     forLogic = 1,//プログラム関係
     forResource = 2,//リソース関係
     for3D = 3,//3D グラフィックス関係
     for2D = 4,//2D グラフィックス関係
     forSound = 5,//サウンド関係
     forScript = 6,//スクリプト関係
     forGameData = 7,//ゲームデータ関係
     forReserved01 = 8,//(予約 01)
     //...（略）...
     forReserved56 = 63,//(予約 56) ※0～63 まで使用可
     //以下、ログレベル／画面通知レベル変更用
     forEvery = 64,//全部まとめて変更
     //以下、特殊なカテゴリ（プリント時専用）
     forCallPoint = 65,//直近のコールポイントのカテゴリに合わせる（なければ forAny 扱い）
     forCriticalCallPoint = 66,//直近の重大コールポイントのカテゴリに合わせる（なければ forAny 扱い）
};
```

一塊の複数のメッセージを扱う場合：

```
//---------------------------------------
//デバッグメッセージ出力
//※一塊の複数のメッセージを扱う場合

//画面通知をしないメッセージを明示する
//※log***()関数は、ログ出力専用
logAsCritical(forAny, "----------------------------------------¥n");
```

```
printAsCritical(forAny, "重大メッセージ：%s¥n", name);
logAsCritical(forAny, "...理由など...¥n");
logAsCritical(forAny, "----------------------------------------¥n");
//※一塊の複数のメッセージの内、一部しか画面通知に反応させたくない場合に使い分ける
//※逆に画面通知専用関数は、notice***()。
```

遅延デバッグメッセージの使用イメージ：

```
//---------------------------------------
//遅延デバッグメッセージ
//※スレッド間での処理到達の順序性を厳密に保証してプリントするための仕組み

//メッセージシーケンス番号予約
reservePrint reserved(asCritical, forAny, 3);//予約数を引数で指定する
//※reservePrint はクラスであり、内部で予約した番号（64bit）を保持する
//※予約の時点でレベルとカテゴリを指定し、ログレベルが適合せず出力されない場合は予約されない
//※2 つ以上の予約の際は、連番で予約されることを保証する
//   (std::atomic<long long>::fetch_add(n) などを使用して安全なロックフリー処理で確保する)

//...（処理）...

//遅延メッセージ出力
reserved.print("重大メッセージ１：%s", name);
reserved.print("重大メッセージ２：%s", name);
reserved.print("重大メッセージ３：%s", name);
//※メッセージレベルにより、予約が失敗している場合は、メッセージの出力は無視される
//※出力するごとに reserved の内部のカウンタが進み、予約数を超えたものは出力されない
//※この仕組みを利用し、一塊の複数のメッセージの途中で、他のスレッドのメッセージが
//  割り込むことがないようにすることができる

//メッセージ出力キャンセル
reserved.cancelPrint();
//※未使用の予約番号はキャンセルされ、欠番扱いになる
//※処理ブロックを抜ける際、reservePrint クラスのデストラクタにより、
//  キャンセル処理は自動実行される
```

デバッグメッセージ処理属性の使用イメージ：

```
//---------------------------------------
//デバッグメッセージ処理属性

//デバッグメッセージ処理属性をセット
setPrintAttr(withSeqNo);//属性追加：メッセージ出力時にシーケンス番号を付加
//表示例：
//  [23]重大メッセージ：func1
//  [24]通常メッセージ：func2

//デバッグメッセージ処理属性をリセット
resetPrintAttr(withSeqNo);//属性解除：メッセージ出力時のシーケンス番号の付加を取りやめ
```

▼ コールポイント

コールポイントは明示的なコールスタックのようなもので、アサーション違反のような問題発生時に呼び出し履歴を表示したり、通常のメッセージをログ出力する際に直前のコールポイントの情報を表示したりといった使い方をする。

また、「共通処理のアサーション判定を無効にする」といった、以降の処理でのメッセージやアサーションの挙動を制御することにも用いる。

● コールポイントの使用イメージ

まず、コールポイントの使用イメージを示す。

コールポイントの使用イメージ：

```
//関数１
void func1()
{
     callPoint call_point(asNormal, forAny, "func1");//コールポイント設定（レベルとカテゴリと名前を与える）
     func2();//関数２呼び出し
     func3();//関数３呼び出し
}
//関数２
void func2()
{
     commonFunc();//共通関数呼び出し
}
//関数３
void func3()
{
     callPoint call_point(asCritical, forLogic, "func3");//コールポイント設定（レベルとカテゴリと名前を与える）
     commonFunc();//共通関数呼び出し
}
//共通関数
void commonFunc()
{
     printCallPointStackAsCritical(forAny, "commonFunc");//コールポイントスタック表示（カテゴリと名前を与える）
}
```

↓（実行結果）

```
----------------------------------------
Call point stack at "commonFunc"
"func1" asNormal, forAny
----------------------------------------
----------------------------------------
Call point stack at "commonFunc"
"func3" asCritical, forLogic
"func1" asNormal, forAny
----------------------------------------
```

「printCallPointStack***()」を実行すると、それまでに通過したコールポイントの名前とカテゴリが逆順（スタック順）で表示される。ここで与える名前はログ出力に用いられ、出力の有無はメッセージ出力と同様にカテゴリとログレベルに従って判定される。

なお、この関数は通常のデバッグプリント文の仕組みを通してログ出力スレッドに出力されるが、どのログレベルであっても画面通知されることはない。

● コールポイントの仕組み

コールポイントは、TLS(スレッドローカルストレージ)を利用した連結リストで管理する。このため、他のスレッドが干渉することのない、そのスレッドだけのコールスタックを作る事ができる。

「callPoint」クラスのコンストラクタで連結リストにつなぎ、デストラクタで自動的に外す。自身のスレッドに限定された処理なので、とくに何らかのメモリ確保もせず、ローカ

ル変数をそのまま連結する。

TLS については別紙の「マルチスレッドプログラミングの基礎」を参照。

● コールポイントの高度な利用イメージ

コールポイントを利用することで、共通処理の挙動を変えることができる。

デバッグメッセージを出力する際は、前述のとおり「カテゴリ」を指定する。この時、「forCallpoint」カテゴリを指定すると、直近のコールポイントのカテゴリと同じ扱いになる。

また、コールポイントは一時的にログレベルを引き上げる機能にも対応する。既存の設定より引き上げるだけで、下げることはできない。

以上の機能を、例えば「メモリマネージャ内のメモリ確保失敗検出用のアサーション」に活用する。

メモリ確保失敗は重大な問題であるが、「メモリ確保失敗時は、次のフレームでリトライする」といった処理要件では、このアサーションが邪魔になる。通常の対応では、やむなくアサーションを外すか、アサーション実行判定のためのフラグを渡すなどの煩雑な対応になっていく。new 演算子を通すことを考えると、後者の対応はさらに難しい。

この対処として、コールポイントを利用する。

デバッグ機能だけで完結するので、通常の処理インターフェースを汚すこともない。

以下、このような利用例を具体的に示す。

コールポイントの高度な利用イメージ：

【memory_manager.cpp】

```
//メモリ確保
void* CMemoryManager::alloc(const std::size_tsize)
{
	//…（処理）…
	//メモリ確保成功アサーション
	assertAsCritical(forCallPoint, p != nullptr, "not enough memory!");
	//※「forCallpoint」カテゴリ指定により、コールポイント時のカテゴリに基づいてアサーションを判定する
	//…（処理）…
	return p;
}
//配置 new
void* operator new(const std::size_t size, CMemoryManager* mem_man){ return mem_man->alloc(size); }
```

【resource_manager.cpp】

```
//リソース構築
template<class T>
T* CResourceManager::createResource()
{
	//…（処理）…
	//インスタンス生成
	T* p = nullptr;
	{
		//コールポイントを設定
```

```
            callPoint call_point(asNormal, forResource, "CResourceManager::createResource");
            //このコールポイントの間だけ、forResource カテゴリのログレベルを asSilentAbsolutely に設定
            call_point.setLogLevel(asSilentAbsolutely, forResource);
            //インスタンス生成
            p = new(mem_man) T();
            //処理ブロックから抜けるとコールポイントが解除され、ログレベルも元に戻る
        }
        //... (処理) ...
        return p;
}
```

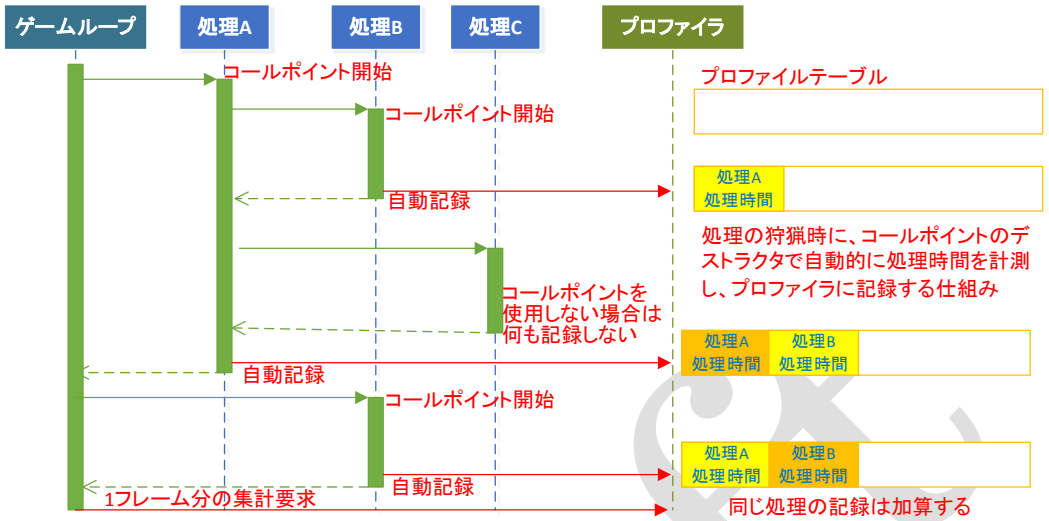
これにより、通常はメモリマネージャ内のアサーションが判定されるが、リソース構築処理の時だけ判定を行わないようにする。

● コールポイントの更なる活用：プロファイラとしての活用

コールポイントの仕組みをさらに拡張子、プロファイラとして活用する。

- コールポイントが開始と終了を自動的に処理することを利用し、処理時間を計測する。
- さらに、別途処理時間を集計する仕組みを設け、そこに記録していくようにする。
- 処理時間を記録する際は、別紙の「効率化と安全性のためのロック制御」で示すスレッド ID 管理の仕組みと組み合わせ、スレッド ID（スレッド名）＋処理名で記録する。
- コールポイントの親子関係も記録する。
- なお、処理時間を記録する際は、__FILE__ マクロと __LINE__ マクロを活用してユニークなキーとすると、集計しやすい。
- 「5 秒間隔」などの設定で集計を行えるようにし、重い処理をリアルタイムに画面表示するなどして活用する。
- 画面表示の際は、「カテゴリ」でマスクすることもできるため、特定のカテゴリに絞った確認も取りやすい。
- 以上の仕組みにより、例えば「描画スレッド」に限定した確認なども簡単に行える。
- リアルタイムのプロファイラであることが何より大きな特徴。プログラマーの開発環境以外でパフォーマンスの問題を確認した際にその場で調査できる。

コールポイントとプロファイラの連動イメージ：



▼ アサーション／ウォッチポイント／ブレークポイント

アサーションは普通によく使われるプログラミングの手段の一つであるが、標準関数のように（アサーション違反時に） abort() せず、ブレークポイント割り込みを発生させて、処理を継続できるものとする。

アサーションの処理は下記のとおり。

- 「評価式」を判定し、結果が「偽」なら、下記の処理を行う。「真」なら何もせず終了。
- 「評価式」、「任意のメッセージ」、「ソースファイル名」（__FILE__）、「ソースファイルのタイムスタンプ」（__TIMESTAMP__）、「関数名」（__FUNCSIG__など）、「行番号」（__LINE__）を表示。
- 「任意のメッセージ」のみを画面通知。
- コールポイントスタックを表示。
- コールスタックを表示。
- ブレークポイント割り込み。
- ゲームのポーズ要求。（ゲーム稼働中のみ）

アサーションも、デバッグメッセージと同様のレベルとカテゴリを扱い、時にはランタイム時に無効化させることができる。（本来推奨されるようなことではないが、前述のコールポイントのような利用に効果を発揮する）

アサーションと同様の機能で、もっとレベル（重大度）の低いチェックのために、「ウォッチポイント」の機能を設ける。「アサーション」は、「本当に起こってはいけないこと（誓約）」に用いるべきなので、きちんと使い分けて、アサーションの濫用を防ぐ。

「ウォッチポイント」の機能はほとんどアサーションと同じであるが、下記の点で異なる。

- 「評価式」が「真」の時に反応する。
- 画面通知は行わない。
- ゲームのポーズ要求は行わない。

ほか、コールポイント、コールスタック、レベル、カテゴリの扱いは同じ。

「ブレークポイント」は、「ウォッチポイント」の無条件版。

アサーション／ウォッチポイント／ブレークポイントの使用イメージ：

```
//アサーション
//※デバッグメッセージと同様に、レベルとカテゴリを指定する
//※続いて、評価式（偽なら停止）、メッセージ、メッセージのパラメータを指定する
assertAsCritical(forCallpoint, p != nullptr, "重大メッセージ:%s", name);
//※なお、アサーションは「重大」(asCritical) レベルにしか対応しない。

//ウォッチポイント
//※デバッグメッセージと同様に、レベルとカテゴリを指定する
//※続いて、評価式（真なら停止）、メッセージ、メッセージのパラメータを指定する
watchAsWarning(forCallpoint, p == nullptr, "警告メッセージ:%s", name);
//※ウォッチポイントは、「絶対」(asAbsolute) レベルを除く全てのレベル（0～4）に対応する

//ブレークポイント
//※デバッグメッセージと同様に、レベルとカテゴリを指定する
//※続いて、メッセージ、メッセージのパラメータを指定する
breakAsWarning(forImportant, "重要メッセージ:%s", name);
//※ブレークポイントは、「絶対」(asAbsolute) レベルを除く全てのレベル（0～4）に対応する
```

▼ コールスタック

開発環境以外のところでアサーション違反が発生した場合、コールスタックの表示が非常に役に立つ。

コールスタックは SDK の API などを用いて表示させることもできるが、Boost C++ のライセンスで公開されているソースコードがある。(Boost C++ のライブラリには組み込まれていない)

シンボル情報もたどって、きちんと名前で表示してくれるので便利。「例外」(try～catch) が必須であるほか、「std::ostream」「std::string」を内部で多用しているので、メモリには注意が必要。

backtrace.cpp と backtrace.hpp をそのままプロジェクトに組み込むだけですぐに使える。ゲーム機向けの SDK ではそのまま使えない可能性があるが、Windows や Linux には対応している。

ソースの配布元 URL は、ソースを入手した個人のブログしか見つけきれなかったので記載は省略。「boost backtrace」で検索すると見つかる。

boost::bactrace の使用イメージ：※例外のキャッチが必須

```
#include <boost/backtrace.hpp>

void debug::printInfo()
{
	//... (処理) ...
	//コールスタック表示
	try
	{
		throw boost::runtime_error("Test");//例外をスロー
	}
	catch (std::exception const &e)
	{
		std::cerr << boost::trace(e);//例外をキャッチしてバックトレースを std::ostream に出力
	}
	//... (処理) ...
}
```

↓（実行結果）

```
0132B88F: boost::stack_trace::trace +0x3f
0132E62A: boost::backtrace::backtrace +0x9a
0132E802: boost::runtime_error::runtime_error +0x62
013339B3: debug::printInfo +0x313
01333B90: debug::printInfo +0x40
01332487: debug::flushInfo +0x267
0133275E: debug::flushMessageBlock +0x3e
0132EDB2: debug::CMessageBlock::~CMessageBlock +0x42
01335549: sub +0x4e9
01335C2D: thread +0xed
01335F04: main +0x44
013369F9: __tmainCRTStartup +0x199
01336BED: mainCRTStartup +0xd
76FB495D: BaseThreadInitThunk +0xe
771F98EE: RtlInitializeExceptionChain +0x84
771F98C4: RtlInitializeExceptionChain +0x5a
```

▼ マルチスレッドでの状態監視について

システムの要件として、メッセージバッファの使用状態やキューイングの状態を監視する処理を行うものとしている。

このような処理は、ループでポーリングし続けるような処理ではなく、できるだけ「スリープ」することが望ましい。

「処理不要な時は可能な限り何もせずスリープ」し、「処理が必要になったら即座に反応する」という動作が望ましいので、ループしながらスリープを入れて反応が遅れるようなことも避けたい。

このような処理の対応としては、「モニター」の仕組みを用いると良い。POSIX スレッドライブラリや C++11 では「条件変数」が、WIN32 では「イベント」が使える。どちらも扱いに少しクセがあるが、「条件変数」の方が何かと柔軟に扱えてよい。

詳しくは別紙の「マルチスレッドプログラミングの基礎」を参照。

■ 処理実装サンプル

コールポイントの有効性を示すためのサンプルプログラムを示す。

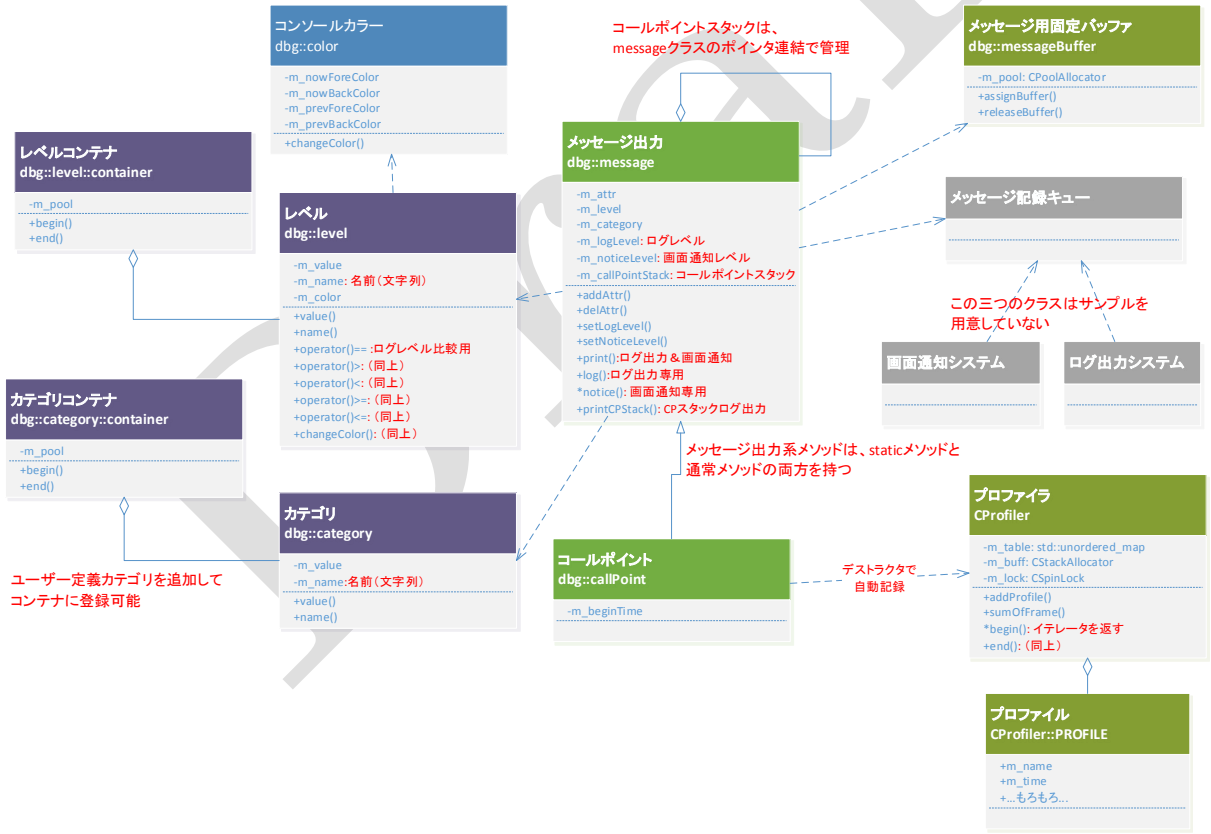
コールポイントは、デバッグメッセージ、プロファイラと組み合わせることで使い勝手のよいシステムとなる。これらが実際に機能する状態のサンプルを示す。

コールポイントを使用した処理の説明を主体にするが、システム本体のコードも参考までに掲載する。

▼ デバッグロギングシステムのクラス図

まず、デバッグロギングシステムのクラス図を示す。

デバッグロギングシステムのクラス図：



▼ サンプルプログラムについて

サンプルプログラムは、非常に便利な C++11 の新しい構文やライブラリを随所で使用し

ている。C++11 は、（アトミック操作以外は）どうしても必須ということはないが、生産性に大きな差が出る。

　また、別紙に掲載したサンプルコードも流用している。別紙から引用する必要のあるコードについては後述する。

▼ デバッグロギングシステムの使用サンプル

　デバッグロギングシステムの使用サンプルを示す。

　システムの使用箇所を赤字で示す。

【ネームスペース使用宣言】

```
//--------------------
//ネームスペース使用宣言
using namespace dbg;
using namespace dbg::ext;//ユーザー定義拡張（カテゴリ追加）
```

【テスト（共通関数）】

```
#include <random>//C++11 乱数
#include <thread>//C++11 スレッド
#include <chrono>//C++11 時間

//--------------------
//テスト（共通関数）
void printCommon(const char* name)
{
	printf("---------- printCommon(%s) ----------¥n", name);//C 言語標準のプリント文
	printAsNormal(forAny, "通常メッセージ(%s)¥n", name);
	printAsVerbose(forAny, "冗長メッセージ¥n");
	printAsDetail(forAny, "詳細メッセージ¥n");
	printAsImportant(forAny, "重要メッセージ¥n");
	printAsWarning(forAny, "警告メッセージ¥n");
	printAsCritical(forAny, "重大メッセージ¥n");
	printAsAbsolute(forAny, "絶対メッセージ¥n");
	printAsNormal(for3D, "【3D】通常メッセージ¥n");
	printAsNormal(for2D, "【2D】通常メッセージ¥n");
	printAsNormal(forLogic, "【Logic】通常メッセージ¥n");
	printAsCritical(forCallPoint, "【CP のカテゴリ】重大メッセージ¥n");
	printAsCritical(forCriticalCallPoint, "【クリティカル CP のカテゴリ】重大メッセージ¥n");
	print(asNormal, forAny, "【レベルを引数指定】通常メッセージ¥n");
	logAsCritical(forAny, "【ログ出力専用関数】重大メッセージ¥n");
	noticeAsCritical(forAny, "【画面通知専用関数】重大メッセージ¥n");
	message msg(asWarning, forAny);
	msg.delAttr(withLevel | withCategory);//msg オブジェクトで print した時だけレベルとカテゴリの表示を OFF
	msg.print("message クラスを使って、");
	msg.print("同じレベルとカテゴリのメッセージを¥n");
	msg.print("連続で出力¥n");
	printCPStackAsNormal(forAny, "CP(コールポイント)スタック表示");
}
//ランダム時間ウェイトする
static std::mt19937 s_rndEngine;
void initRnadom()//乱数初期化
{
	//乱数初期化
	std::random_device rd;
	s_rndEngine.seed(rd());
}
void randomWait()//ウェイト
{
```

```
	std::uniform_int_distribution<int> time(10, 100);
	const int sleep_time = time(s_rndEngine);
	std::this_thread::sleep_for(std::chrono::milliseconds(sleep_time));
}
```

【議事ゲームループ処理】※コールポイントスタックの確認とプロファイルの集計結果確認用

```
//--------------------
//テスト（擬似ゲームループ）
void gameLoop()
{
	for (int frame = 0; frame < 10; ++frame)
	{
		//フレームの処理開始
		//※デストラクタで処理するため、処理ブロックで囲む
		{
			callPointAsCritical cp(forAny, "ゲームループ", recProfile, __CPARGS());
			extern void subProc1();
			extern void subProc2();
			extern void subProc3();
			extern void subProc4();
			extern void subProc5();
			extern void subProc6();
			extern void subProc7(const char* thread_name);
			std::thread th1 = std::thread(subProc7, "スレッドA");//スレッド処理
			subProc1();//処理１
			subProc2();//処理２
			subProc3();//処理３
			std::thread th2 = std::thread(subProc7, "スレッドB");//スレッド処理
			subProc4();//処理４
			subProc5();//処理５
			subProc6();//処理６
			std::thread th3 = std::thread(subProc7, "スレッドC");//スレッド処理
			printCommon(cp.getName());//共通処理
			th1.join();
			th2.join();
			th3.join();
		}

		//フレームの処理時間を集計
		{
			CProfiler profiler;
			profiler.sumOnFrame();
		}
	}
	extern void subProc8();
	subProc8();//処理８
}
//処理１
void subProc1()
{
	callPointAsNormal cp(for3D, "処理１：表示属性変更", recProfile, __CPARGS());//コールポイント
                                                                                //※__CPARGS()は必ず付ける
	cp.delAttr(withLevel);//レベル表示なし
	//cp.delAttr(withCategory);//カテゴリ表示なし
	randomWait();//ウェイト
	printCommon(cp.getName());//共通処理
}
//処理２
void subProc2()
{
	callPointAsNormal cp(for2D, "処理２：静寂", recProfile, __CPARGS());
	cp.setLogLevel(asSilent, forEvery);//以降のログ表示なし（画面通知はあり）
	randomWait();//ウェイト
	printCommon(cp.getName());//共通処理
}
```

```
//処理３
void subProc3()
{
	callPointAsNormal cp(forLogic, "処理３：カラー非表示", recProfile, __CPARGS());
	cp.addAttr(withoutColor);//カラー表示なし（ログにのみ影響）
	randomWait();//ウェイト
	printCommon(cp.getName());//共通処理
}
//処理４
void subProc4()
{
	callPointAsNormal cp(forGameData, "処理４：絶対静寂", recProfile, __CPARGS());
	cp.setLogLevel(asSilentAbsolutely, forAny);//以降の forAny のログ表示を完全停止
	cp.setNoticeLevel(asSilentAbsolutely, forEvery);//以降の全ての画面通知表示を完全停止
	randomWait();//ウェイト
	printCommon(cp.getName());//共通処理
}
//処理５
void subProc5()
{
	callPointAsNormal cp(forSound, "処理５：変更なし", recProfile, __CPARGS());
	randomWait();//ウェイト
	printCommon(cp.getName());//共通処理
}
//処理６
void subProc6()
{
	//コールポイントがクリティカル＆プロファイル集計無し
	callPointAsCritical cp(forTaro, "処理６：クリティカル CP", noProfile, __CPARGS());
	randomWait();//ウェイト
	subProc5();//処理５を呼び出し
}
//処理７
void subProc7(const char* thread_name)
{
	callPointAsCritical cp(forThread, "処理７：スレッド", recProfile, __CPARGS());
	CThreadID thread_id(thread_name);//スレッドに名前を付ける
	//スレッド用（ウェイトのみ）
	randomWait();//ウェイト
}
//処理８
void subProc8()
{
	callPointAsNormal cp(forGameData, "処理８：基本ログレベルを変更", recProfile, __CPARGS());
	setLogLevel(asDetail, forEvery);//全てのカテゴリのログレベルを asDetail に
	setNoticeLevel(asDetail, forEvery);//全てのカテゴリの画面通知レベルを asDetail に
	randomWait();//ウェイト
	printCommon(cp.getName());//共通処理
	//【補足】
	//コールポイントに対する操作ではなく、同名の関数でログレベル／画面通知レベルを変更
	//この場合、永続的な設定の変更のため、コールポイントを抜けても元に戻らない
	//デバッグメニューなどを用いて、ユーザーが任意のタイミングで変更することを想定
	//つまり、この基本設定の変更処理は通常使用禁止（デバッグメニュー用処理など、一部の処理でのみ行う）
	//基本設定は「ユーザーのための設定」である都合から、コールポイントでそれより低いログレベルを
	//設定しても無効となる（以下、そのテスト）
	setLogLevel(asSilent, forEvery);//基本設定を「静寂」に
	setNoticeLevel(asSilent, forEvery);//（同上）
	cp.setLogLevel(asDetail, forEvery);//コールポイントに対しては「詳細」を設定
	cp.setNoticeLevel(asDetail, forEvery);//（同上）
	printCommon(cp.getName());//共通処理
}
```

【プロファイル表示】

```
#include <vector>//STL vector
//※CStackAllocatorWithBuff を使用
```

```
//※CStackAllocAdp を使用
//※CTempPolyStaclAllocator を使用

//--------------------
//テスト（プロファイルを集計して表示）
void printProfile()
{
	printf("--- PROFILE ---¥n");
	CProfiler profiler;//プロファイラ取得
	printf("size()=%d, Buffer:Total=%d,Used=%d,Remain=%d¥n", profiler.size(), profiler.getBuffTotal(),
		profiler.getBuffUsed(), profiler.getBuffRemain());
	CStackAllocatorWithBuff<2048> work;//ワーク領域確保
	CTempPolyStackAllocator alloc(work);//new 演算子でワーク領域を使うように設定（処理ブロックを抜けたら自動解除）
	//ソートのために vector に移し替え
	typedef std::vector<const CProfiler::PROFILE*> vec_t;
	vec_t vec;
	for (auto& ite : profiler)//プロファイラ自身がイテレータなので、そのままループ処理で計測結果を取り出す
	{
		const CProfiler::PROFILE& profile = ite.second;
		vec.push_back(&profile);
	}
	//ソート
	std::sort(vec.begin(), vec.end(),
		[](const CProfiler::PROFILE* lhs, const CProfiler::PROFILE* rhs) -> bool
		{
			return lhs->m_measure.m_total.m_sum > rhs-> m_measure.m_total.m_sum;//トータル処理時間が大きい順
		}
	);
	//表示
	for (auto& ite : vec)
	{
		const CProfiler::PROFILE& profile = *ite;
		printf("%8.6f sec (%d cnt):¥"%s¥" %s¥n", profile.m_measure.m_total.m_sum, profile.m_measure.m_total.m_count,
			profile.m_name.c_str(), profile.m_funcName.c_str());

		//スレッド情報表示
		{
			//スレッドの処理が終わった時に、スタックのマーカーを自動的に元の位置に戻す設定
			CStackAllocAdp work_tmp(work,CIStackAllocAdp::AUTO_REWIND);

			//ソートのために vector に移し替え
			typedef std::vector<const CProfiler::PROFILE::THREAD_INFO*> vec_th_t;
			vec_th_t vec_th;
			for (auto& ite : profile.m_threadList)
			{
				const CProfiler::PROFILE::THREAD_INFO& thread_info = ite.second;
				vec_th.push_back(&thread_info);
			}
			//ソート
			std::sort(vec_th.begin(), vec_th.end(),
				[](const CProfiler::PROFILE::THREAD_INFO* lhs, const CProfiler::PROFILE::THREAD_INFO* rhs)
					-> bool
				{
					return lhs-> m_measure.m_total.m_sum > rhs-> m_measure.m_total.m_sum;
						//トータル処理時間が大きい順
				}
			);
			for (auto& ite_th : vec_th)
			{
				const CProfiler::PROFILE::THREAD_INFO& thread_info = *ite_th;
				printf("                        %8.6f sec (%d cnt):¥"%s¥"¥n",
					thread_info.m_measure.m_total.m_sum, thread_info.m_measure.m_total.m_count,
					thread_info.m_name.c_str());
			}
		}
```

```
	}
	printf("work:total=%d,used=%d,remain=%d¥n", work.getTotal(), work.getUsed(), work.getRemain());
}
```

【テスト（メイン）】

```
//--------------------
//テスト（メイン）
int main(const int argc, const char* argv[])
{
	//スレッドに名前を付ける
	CThreadID("メインスレッド");

	//レベルを列挙
	printf("---------- レベルを列挙 ----------¥n");
	printLevelAll();

	//カテゴリを列挙
	printf("---------- カテゴリを列挙 ----------¥n");
	printCategoryAll();

	//初期状態でプリント
	printCommon("初期状態");

	//擬似ゲームループ
	gameLoop();

	//プロファイルを表示
	printProfile();

	return EXIT_SUCCESS;
}
```

【ユーザー定義カテゴリ】

```
//--------------------------------------
//デバッグメッセージのユーザー拡張
namespace dbg
{
	//ユーザー拡張
	namespace ext
	{
		//--------------------------------------
		//カテゴリ定数
		enum categoryEnum : category::value_t
		{
			//拡張用のカテゴリ番号を、ユーザー定義カテゴリに割り当て
			forThread = define_reservedCategory(0),//スレッド
			forTaro = define_reservedCategory(1),//太郎専用
		};
		//--------------------------------------
		//カテゴリ定義
		//※カテゴリ定数に対するカテゴリの実体（名前など）を作成（以下はテンプレートクラスを実体化するマクロ）
		//※ヘッダーで公開する必要なし
		declare_reservedCategory(forThread, true, false);//スレッド
		declare_reservedCategory(forTaro, true, true);//太郎専用
		//--------------------------------------
		//ユーザー定義カテゴリをカテゴリコンテナに登録
		//※コンストラクタ＋静的変数で main 関数の前に自動実行
		//※一度クラスのインタンスを生成すれば、自動的に登録される
		struct categoryContainerExt
		{
			//コンストラクタ
			categoryContainerExt()
			{
				category::container::initializeOnce();//念のため、先に組み込みカテゴリの初期化を実行
```

```
                    category_forThread();//スレッド
                    category_forTaro();//太郎専用
                }
            };
            //----------------------------------------
            //変数
            static categoryContainerExt s_categoryForInitialize;//初期化処理実行のためのインスタンス
    }//namespace ext
}//namespace dbg
```

↓（処理結果）

```
---------- レベルを列挙 ----------
level=0, name="asVerbose", valueAsOutput=0, forLog=1, forNotice=0, forMask=1        ←valueForOutput の大小比較で
          prev=asVerbose(0)                                                          ログ／画面出力のマスクを行う
          next=asNormal(2)
level=1, name="asDetail", valueAsOutput=0, forLog=1, forNotice=0, forMask=1
          prev=asDetail(1)
          next=asNormal(2)
level=2, name="asNormal", valueAsOutput=1, forLog=1, forNotice=1, forMask=1
          prev=asVerbose(0)
          next=asImportant(4)
level=3, name="(undefined)", valueAsOutput=1, forLog=0, forNotice=0, forMask=0
          prev=(undefined)(3)
          next=(undefined)(3)
level=4, name="asImportant", valueAsOutput=2, forLog=1, forNotice=1, forMask=1
          prev=asNormal(2)
          next=asWarning(6)
level=5, name="(undefined)", valueAsOutput=2, forLog=0, forNotice=0, forMask=0
          prev=(undefined)(5)
          next=(undefined)(5)
level=6, name="asWarning", valueAsOutput=3, forLog=1, forNotice=1, forMask=1
          prev=asImportant(4)
          next=asCritical(8)
level=7, name="(undefined)", valueAsOutput=3, forLog=0, forNotice=0, forMask=0
          prev=(undefined)(7)
          next=(undefined)(7)
level=8, name="asCritical", valueAsOutput=4, forLog=1, forNotice=1, forMask=1
          prev=asWarning(6)
          next=asAbsolute(10)
level=9, name="(undefined)", valueAsOutput=4, forLog=0, forNotice=0, forMask=0
          prev=(undefined)(9)
          next=(undefined)(9)
level=10, name="asAbsolute", valueAsOutput=5, forLog=1, forNotice=0, forMask=1
          prev=asCritical(8)
          next=asAbsolute(10)
level=11, name="asSilent", valueAsOutput=5, forLog=0, forNotice=0, forMask=1
          prev=asSilent(11)
          next=asSilent(11)
level=12, name="asSilentAbsolutely", valueAsOutput=6, forLog=0, forNotice=0, forMask=1
          prev=asSilentAbsolutely(12)
          next=asSilentAbsolutely(12)
---------- カテゴリを列挙 ----------
category=0, name="forAny", isAssigned=1, isReserved=0, forLog=1, forNotice=1, forMask=1
category=1, name="forLogic", isAssigned=1, isReserved=0, forLog=1, forNotice=1,forMask=1
category=2, name="forResource", isAssigned=1, isReserved=0, forLog=1, forNotice=1, forMask=1
category=3, name="for3D", isAssigned=1, isReserved=0, forLog=1, forNotice=1, forMask=1
category=4, name="for2D", isAssigned=1, isReserved=0, forLog=1, forNotice=1, forMask=1
category=5, name="forSound", isAssigned=1, isReserved=0, forLog=1, forNotice=1,forMask=1
category=6, name="forScript", isAssigned=1, isReserved=0, forLog=1, forNotice=1, forMask=1
category=7, name="forGameData", isAssigned=1, isReserved=0, forLog=1, forNotice=1, forMask=1
category=8, name="forThread", isAssigned=0, isReserved=1, forLog=1, forNotice=0, forMask=1      ←ユーザー定義カテゴリ
category=9, name="forTaro", isAssigned=0, isReserved=1, forLog=1, forNotice=1, forMask=1        ←も正しく列挙される
category=10, name="(undefined)", isAssigned=0, isReserved=1, forLog=0, forNotice=0, forMask=0
category=11, name="(undefined)", isAssigned=0, isReserved=1, forLog=0, forNotice=0, forMask=0
...(略)...
```

```
category=62, name="(undefined)", isAssigned=0, isReserved=1, forLog=0, forNotice=0, forMask=0
category=63, name="(undefined)", isAssigned=0, isReserved=1, forLog=0, forNotice=0, forMask=0
category=64, name="forEvery", isAssigned=1, isReserved=0, forLog=0, forNotice=0, forMask=1
category=65, name="forCallPoint", isAssigned=1, isReserved=0, forLog=1, forNotice=1, forMask=0
category=66, name="forCriticalCallPoint", isAssigned=1, isReserved=0, forLog=1,
forNotice=1, forMask=0
---------- printCommon(初期状態) ----------
[asNormal,forAny] 通常メッセージ(初期状態)        ←デフォルトのログレベルは「通常」以上（冗長／詳細以外全て出力）
[asImportant,forAny] 重要メッセージ
[asWarning,forAny] 警告メッセージ
[asCritical,forAny] 重大メッセージ
 重大メッセージ                                   ←「画面通知」の代わりに、標準エラーに背景色付きで表示
[asAbsolute,forAny] 絶対メッセージ                  デフォルトの画面通知レベルは「重大」以上
[asNormal,for3D] 【3D】通常メッセージ
[asNormal,for2D] 【2D】通常メッセージ
[asNormal,forLogic] 【Logic】通常メッセージ
[asCritical,forAny] 【CP のカテゴリ】重大メッセージ          ←カテゴリが forCallPoint から forAny に変わっている
 【CP のカテゴリ】重大メッセージ                              （直前のコールポイントのカテゴリになる）
[asCritical,forAny] 【クリティカル CP のカテゴリ】重大メッセージ  ←カテゴリが forCriticalCallPoint から forAny に
 【クリティカル CP のカテゴリ】重大メッセージ                    （直近の重大コールポイントのカテゴリになる）
[asNormal,forAny] 【レベルを引数指定】通常メッセージ
[asCritical,forAny] 【ログ出力専用関数】重大メッセージ
【画面通知専用関数】重大メッセージ
message クラスを使って、同じレベルとカテゴリのメッセージを
連続で出力
=======================================================
[asNormal,forAny] Call point stack at "CP(コールポイント)スタック表示"
---------------------------------------
=======================================================      ←コールポイントが一つもなし
---------- printCommon(処理１：表示属性変更) ----------
[forAny] 通常メッセージ(処理１：表示属性変更)                ←属性の変更により「レベル」を非表示
[forAny] 重要メッセージ
[forAny] 警告メッセージ
[forAny] 重大メッセージ
 重大メッセージ
[forAny] 絶対メッセージ
[for3D] 【3D】通常メッセージ
[for2D] 【2D】通常メッセージ
[forLogic] 【Logic】通常メッセージ
[for3D] 【CP のカテゴリ】重大メッセージ
 【CP のカテゴリ】重大メッセージ
[forAny] 【クリティカル CP のカテゴリ】重大メッセージ
 【クリティカル CP のカテゴリ】重大メッセージ
[forAny] 【レベルを引数指定】通常メッセージ
[forAny] 【ログ出力専用関数】重大メッセージ
【画面通知専用関数】重大メッセージ
message クラスを使って、同じレベルとカテゴリのメッセージを
連続で出力
=======================================================
[forAny] Call point stack at "CP(コールポイント)スタック表示"
---------------------------------------
  "処理１：表示属性変更" [for3D]                          ←コールポイントスタックが二つ表示
  "ゲームループ" [forAny]                                 ←（同上）
=======================================================
---------- printCommon(処理２：静寂) ----------
重大メッセージ                                           ←「ログレベル」のみを「静寂」に設定（画面通知は変わらず）
[asAbsolute,forAny] 絶対メッセージ                       ←「静寂」時も「絶対」メッセージは出力される
【CP のカテゴリ】重大メッセージ
【クリティカル CP のカテゴリ】重大メッセージ
【画面通知専用関数】重大メッセージ                       ←コールポイントも、ログレベルの影響を受けて非表示
---------- printCommon(処理３：カラー非表示) ----------
[asNormal,forAny] 通常メッセージ(処理３：カラー非表示)
[asImportant,forAny] 重要メッセージ                      ←属性の変更により「カラー」を非表示
[asWarning,forAny] 警告メッセージ
[asCritical,forAny] 重大メッセージ
```

```
 重大メッセージ                                          ←「画面通知」は変わらず
[asAbsolute,forAny] 絶対メッセージ
[asNormal,for3D] 【3D】通常メッセージ
[asNormal,for2D] 【2D】通常メッセージ
[asNormal,forLogic] 【Logic】通常メッセージ
[asCritical,forLogic] 【CP のカテゴリ】重大メッセージ
 【CP のカテゴリ】重大メッセージ
[asCritical,forAny] 【クリティカル CP のカテゴリ】重大メッセージ
 【クリティカル CP のカテゴリ】重大メッセージ
[asNormal,forAny] 【レベルを引数指定】通常メッセージ
[asCritical,forAny] 【ログ出力専用関数】重大メッセージ
【画面通知専用関数】重大メッセージ
message クラスを使って、同じレベルとカテゴリのメッセージを
連続で出力
============================================================
[asNormal,forAny] Call point stack at "CP(コールポイント)スタック表示"
----------------------------------------
  "処理３：カラー非表示" [asNormal,forLogic]
  "ゲームループ" [asCritical,forAny]
============================================================
---------- printCommon(処理４：絶対静寂) ----------
[asNormal,for3D] 【3D】通常メッセージ                    ←「forAny」カテゴリのみ「絶対静寂」に設定
[asNormal,for2D] 【2D】通常メッセージ                    ←画面通知側も設定したため、いっしょに消えている
[asNormal,forLogic] 【Logic】通常メッセージ              ←「絶対」メッセージも消えている
[asCritical,forGameData] 【CP のカテゴリ】重大メッセージ  ←他のカテゴリはそのまま
---------- printCommon(処理５：変更なし) ----------
[asNormal,forAny] 通常メッセージ(処理５：変更なし)
[asImportant,forAny] 重要メッセージ                      ←元の状態に戻っている
[asWarning,forAny] 警告メッセージ                          ※明示的に戻したわけではなく、それぞれの
[asCritical,forAny] 重大メッセージ                         コールポイント終了時に一時設定が破棄されたため
 重大メッセージ
[asAbsolute,forAny] 絶対メッセージ
[asNormal,for3D] 【3D】通常メッセージ
[asNormal,for2D] 【2D】通常メッセージ
[asNormal,forLogic] 【Logic】通常メッセージ
[asCritical,forSound] 【CP のカテゴリ】重大メッセージ    ←カテゴリが forCallPoint から forSound に変わっている
 【CP のカテゴリ】重大メッセージ
[asCritical,forAny] 【クリティカル CP のカテゴリ】重大メッセージ
 【クリティカル CP のカテゴリ】重大メッセージ
[asNormal,forAny] 【レベルを引数指定】通常メッセージ
[asCritical,forAny] 【ログ出力専用関数】重大メッセージ
【画面通知専用関数】重大メッセージ
message クラスを使って、同じレベルとカテゴリのメッセージを
連続で出力
============================================================
[asNormal,forAny] Call point stack at "CP(コールポイント)スタック表示"
----------------------------------------
  "処理５：変更なし" [asNormal,forSound]
  "ゲームループ" [asCritical,forAny]
============================================================
---------- printCommon(処理５：変更なし) ----------      ←「処理６」から再度「処理５」が呼び出されている
[asNormal,forAny] 通常メッセージ(処理５：変更なし)
[asImportant,forAny] 重要メッセージ
[asWarning,forAny] 警告メッセージ
[asCritical,forAny] 重大メッセージ
 重大メッセージ
[asAbsolute,forAny] 絶対メッセージ
[asNormal,for3D] 【3D】通常メッセージ
[asNormal,for2D] 【2D】通常メッセージ
[asNormal,forLogic] 【Logic】通常メッセージ
[asCritical,forSound] 【CP のカテゴリ】重大メッセージ
 【CP のカテゴリ】重大メッセージ
[asCritical,forTaro] 【クリティカル CP のカテゴリ】重大メッセージ  ←カテゴリが forCriticalCallPoint から forTaro に
 【クリティカル CP のカテゴリ】重大メッセージ                        (「処理６」の影響)
[asNormal,forAny] 【レベルを引数指定】通常メッセージ
```

```
[asCritical,forAny] 【ログ出力専用関数】重大メッセージ
【画面通知専用関数】重大メッセージ
message クラスを使って、同じレベルとカテゴリのメッセージを
連続で出力
=====================================================
[asNormal,forAny] Call point stack at "CP(コールポイント)スタック表示"
-----------------------------------------
  "処理５：変更なし" [asNormal,forSound]                ←コールポイントスタックが三つ表示
  "処理６：クリティカル CP" [asCritical,forTaro]        ←（同上）
  "ゲームループ" [asCritical,forAny]                    ←（同上）
=====================================================
---------- printCommon(ゲームループ) ----------
[asNormal,forAny] 通常メッセージ(ゲームループ)          ←元の状態に戻っている
[asImportant,forAny] 重要メッセージ
[asWarning,forAny] 警告メッセージ
[asCritical,forAny] 重大メッセージ
重大メッセージ
[asAbsolute,forAny] 絶対メッセージ
[asNormal,for3D] 【3D】通常メッセージ
[asNormal,for2D] 【2D】通常メッセージ
[asNormal,forLogic] 【Logic】通常メッセージ
[asCritical,forAny] 【CP のカテゴリ】重大メッセージ
【CP のカテゴリ】重大メッセージ
[asCritical,forAny] 【クリティカル CP のカテゴリ】重大メッセージ
【クリティカル CP のカテゴリ】重大メッセージ
[asNormal,forAny] 【レベルを引数指定】通常メッセージ
[asCritical,forAny] 【ログ出力専用関数】重大メッセージ
【画面通知専用関数】重大メッセージ
message クラスを使って、同じレベルとカテゴリのメッセージを
連続で出力
=====================================================
[asNormal,forAny] Call point stack at "CP(コールポイント)スタック表示"
-----------------------------------------
  "ゲームループ" [asCritical,forAny]
=====================================================
…（略）…　※10 回繰り返し
---------- printCommon(処理８：基本ログレベルを変更) ----------
[asNormal,forAny] 通常メッセージ(処理８：基本ログレベルを変更)
通常メッセージ(処理８：基本ログレベルを変更)          ←基本画面通知レベルの変更によって表示
[asVerbose,forAny] 冗長メッセージ                     ←基本ログレベルの変更によって表示
[asDetail,forAny] 詳細メッセージ                      ←（同上）
[asImportant,forAny] 重要メッセージ                     ※「詳細」と「絶対」は画面通知が行われない
重要メッセージ
[asWarning,forAny] 警告メッセージ
警告メッセージ
[asCritical,forAny] 重大メッセージ
重大メッセージ
[asAbsolute,forAny] 絶対メッセージ
[asNormal,for3D] 【3D】通常メッセージ
【3D】通常メッセージ
[asNormal,for2D] 【2D】通常メッセージ
【2D】通常メッセージ
[asNormal,forLogic] 【Logic】通常メッセージ
【Logic】通常メッセージ
[asCritical,forGameData] 【CP のカテゴリ】重大メッセージ
【CP のカテゴリ】重大メッセージ
[asCritical,forAny] 【クリティカル CP のカテゴリ】重大メッセージ
【クリティカル CP のカテゴリ】重大メッセージ
[asNormal,forAny] 【レベルを引数指定】通常メッセージ
【レベルを引数指定】通常メッセージ
[asCritical,forAny] 【ログ出力専用関数】重大メッセージ
【画面通知専用関数】重大メッセージ
message クラスを使って、message クラスを使って、同じレベルとカテゴリのメッセージを   ←一つの文を分けて出力しているため
同じレベルとカテゴリのメッセージを                                                      ログと画面通知が交互に
連続で出力
```

```
連続で出力
=========================================================
[asNormal,forAny] Call point stack at "CP(コールポイント)スタック表示"
-----------------------------------------
  "処理８：基本ログレベルを変更" [asNormal,forGameData]
=========================================================
---------- printCommon(処理８：基本ログレベルを変更) ----------
[asAbsolute,forAny] 絶対メッセージ                              ←基本ログレベルを厳しくした後でコールポイントの
                                                                ログレベルを緩くしても反映されない
                                                                （基本ログレベル以下の設定にはできない）
--- PROFILE ---
size()=8, Buffer:Total=65536,Used=14536,Remain=51000
4.750269 sec (10 cnt):"ゲームループ" void __cdecl gameLoop(void)        ←コールポイントで集計した各処理の処理時間の
                        4.750269 sec (10 cnt):"メインスレッド"            集計（時間がかかっている順）
1.597638 sec (30 cnt):"処理７：スレッド" void __cdecl subProc7(const char *)
                        0.606996 sec (10 cnt):"スレッドC"               ←スレッド別にも集計
                        0.517123 sec (10 cnt):"スレッドA"
                        0.473519 sec (10 cnt):"スレッドB"
1.353264 sec (20 cnt):"処理５：変更なし" void __cdecl subProc5(void)
                        1.353264 sec (20 cnt):"メインスレッド"
0.665267 sec (10 cnt):"処理３：カラー非表示" void __cdecl subProc3(void)
                        0.665267 sec (10 cnt):"メインスレッド"
0.609403 sec (10 cnt):"処理１：表示属性変更" void __cdecl subProc1(void)
                        0.609403 sec (10 cnt):"メインスレッド"
0.484056 sec (10 cnt):"処理４：絶対静寂" void __cdecl subProc4(void)
                        0.484056 sec (10 cnt):"メインスレッド"
0.482811 sec (10 cnt):"処理２：静寂" void __cdecl subProc2(void)
                        0.482811 sec (10 cnt):"メインスレッド"
0.000000 sec (0 cnt):"処理８：基本ログレベルを変更" void __cdecl subProc8(void)
                        0.000000 sec (0 cnt):"メインスレッド"
work:total=2048,used=108,remain=1940
```

▼ デバッグロギングシステムのサンプル

デバッグロギングシステム本体のサンプルを示す。

【ネームスペース】

```
namespace dbg
```

【コンソールカラークラス定義】

```
#define USE_WINDOWS_CONSOLE_COLOR//【MS 固有仕様】Windows スタイルのコンソールカラーを使用

//Windows スタイルカラー用
#ifdef USE_WINDOWS_CONSOLE_COLOR
#include <windows.h>
#include <conio.h>
#endif//USE_WINDOWS_CONSOLE_COLOR

//---------------------------------------
//コンソールカラークラス
class color
{
public:
	//型
	//ハンドル型
#ifdef USE_WINDOWS_CONSOLE_COLOR//Windows 用
	typedef HANDLE handle_t;
#else//USE_WINDOWS_CONSOLE_COLOR//エスケープシーケンス用
	typedef FILE* handle_t;
#endif//USE_WINDOWS_CONSOLE_COLOR
public:
	//定数
	//カラー値
```

```
	enum color_t : unsigned char
	{
		reset = 0x10,//リセット（基本状態に戻す）※「前のカラー」に戻すのではない
		through = 0x20,//スルー（現状維持：何もしない）

	#ifdef USE_WINDOWS_CONSOLE_COLOR//Windows 用
		R = 0x04,//Red:赤
		G = 0x02,//Green:緑
		B = 0x01,//Blue:青
		I = 0x08,//Intensity:鮮やか
		BG_SHIFT = 4,//背景色指定時のビットシフト数
	#else//USE_WINDOWS_CONSOLE_COLOR//エスケープシーケンス用
		R = 0x01,//Red:赤
		G = 0x02,//Green:緑
		B = 0x04,//Blue:青
		I = 0x08,//high Intensity:鮮やか
	#endif//USE_WINDOWS_CONSOLE_COLOR

		RGB = 0x07,//RGB マスク
		RGBI = 0x0f,//RGB+I マスク

		//標準カラー
		black = 0,//黒
		blue = B,//青
		red = R,//赤
		magenta = R | B,//紫
		green = G,//緑
		cyan = G | B,//水
		yellow = G | R,//黄
		white = G | R | B,//白

		//鮮やかなカラー
		iBlack = I | black,//黒
		iBlue = I | blue,//青
		iRed = I | red,//赤
		iMagenta = I | magenta,//紫
		iGreen = I | green,//緑
		iCyan = I | cyan,//水
		iYellow = I | yellow ,//黄
		iWhite = I | white,//白
	};
	//対象
	enum target_t : unsigned char
	{
		stdOut = 0,//対象：標準出力
		stdErr = 1,//対象：標準エラー
	};
	static const int targetNum = 2;
public:
	//アクセッサ
	target_t getTarget() const { return m_target; }//対象
	color_t getForeColor() const { return m_foreColor; }//カラー：前面
	color_t getBackColor() const { return m_backColor; }//カラー：背面
	color_t getPrevForeColor() const { return m_prevForeColor; }//前のカラー:前面
	color_t getPrevBackColor() const { return m_prevBackColor; }//前のカラー:背面
	bool isAutoRestore() const { return m_isAutoRestore; }//自動カラー復元実行指定
public:
	//キャストオペレータ
	operator int() const { return static_cast<int>(m_target); }//対象
	operator target_t() const { return m_target; }//対象
public:
	//メソッド
	//カラー変更
	//※実際にコンソールに反映させる
	void changeColor()
```

```
	{
		if (m_foreColor == through)
			return;
		changeColor(m_target, m_foreColor, m_backColor);
	}
	void changeColor(const target_t target, const color_t fore_color, const color_t back_color = black)
	{
		//対象ハンドル取得
		const handle_t target_h = m_handle[target];

		//カラー変更
		if (fore_color == reset)
		{
		#ifdef USE_WINDOWS_CONSOLE_COLOR//Windows 用
			SetConsoleTextAttribute(target_h, m_ConsoleScreenBufferInfo[target].wAttributes);
		#else//USE_WINDOWS_CONSOLE_COLOR//エスケープシーケンス用
			fprintf(target_h, "¥x1b[39m¥x1b[49m");//"¥x1b[0m"
		#endif//USE_WINDOWS_CONSOLE_COLOR
		}
		else
		{
		#ifdef USE_WINDOWS_CONSOLE_COLOR//Windows 用
			SetConsoleTextAttribute(target_h, ((back_color & RGBI) << BG_SHIFT) | (fore_color & RGBI));
		#else//USE_WINDOWS_CONSOLE_COLOR//エスケープシーケンス用
			fprintf(target_h, "¥x1b[%dm¥x1b[%dm", (fore_color & I ? 90 : 30) + (fore_color & RGB),
													(back_color & I ? 100 : 40) + (back_color & RGB));
		#endif//USE_WINDOWS_CONSOLE_COLOR
		}
	}
public:
	//カラー初期化（一回限り）
	static void initializeOnce();
public:
	//デフォルトコンストラクタ
	color() :
		m_target(stdOut),//対象
		m_foreColor(reset),//カラー：前面
		m_backColor(reset),//カラー：背面
		m_prevForeColor(reset),//前のカラー：前面を記憶
		m_prevBackColor(reset),//前のカラー：背面を記憶
		m_isAutoRestore(false)//自動カラー復元実行指定
	{
		initializeOnce();
	}
	//ムーブコンストラクタ
	color(color&& src) :
		m_target(src.m_target),//対象
		m_foreColor(src.m_foreColor),//カラー：前面
		m_backColor(src.m_backColor),//カラー：背面
		m_prevForeColor(src.m_prevForeColor),//前のカラー：前面を記憶
		m_prevBackColor(src.m_prevBackColor),//前のカラー：背面を記憶
		m_isAutoRestore(src.m_isAutoRestore)//自動カラー復元実行指定
	{
		//ムーブコンストラクタにつき、移動元の内容を改変し、デストラクタが機能しないようにする
		*const_cast<bool*>(&src.m_isAutoRestore) = false;//自動カラー復元実行指定を無効化
	}
	//コンストラクタ
	color(const target_t target, const color_t fore_color, const color_t back_color = black,
		const bool is_auto_restore = true) :
		m_target(target),//対象
		m_foreColor(fore_color),//カラー：前面
		m_backColor(back_color),//カラー：背面
		m_prevForeColor(m_nowForeColor[m_target]),//前のカラー：前面を記憶
		m_prevBackColor(m_nowBackColor[m_target]),//前のカラー：背面を記憶
		m_isAutoRestore(is_auto_restore)//自動カラー復元実行指定
```

```
	{
		if (m_foreColor == through || !m_isAutoRestore)
			return;
		m_nowForeColor[m_target] = m_foreColor;//現在のカラー：前面（記憶用）を更新
		m_nowBackColor[m_target] = m_backColor;//現在のカラー：背面（記憶用）を更新
		changeColor();
	}
	//デストラクタ
	~color()
	{
		if (m_foreColor == through || !m_isAutoRestore)
			return;
		m_nowForeColor[m_target] = m_prevForeColor;//現在のカラー：前面を前のカラーに戻す
		m_nowBackColor[m_target] = m_prevBackColor;//現在のカラー：背面を前のカラーに戻す
		changeColor(m_target, m_prevForeColor, m_prevBackColor);
	}
private:
	//フィールド
	const target_t m_target;//対象
	const color_t m_foreColor;//カラー：前面
	const color_t m_backColor;//カラー：背面
	const color_t m_prevForeColor;//前のカラー：前面
	const color_t m_prevBackColor;//前のカラー：背面
	const bool m_isAutoRestore;//自動カラー復元実行指定
	static bool m_isInitialized;//初期化済み
	static color_t m_nowForeColor[color::targetNum];//現在のカラー：前面（記憶用）
	static color_t m_nowBackColor[color::targetNum];//現在のカラー：背面（記憶用）
#ifdef USE_WINDOWS_CONSOLE_COLOR//Windows 用
	static handle_t m_handle[color::targetNum];//ハンドル
	static CONSOLE_SCREEN_BUFFER_INFO m_ConsoleScreenBufferInfo[color::targetNum];
                                                    //コンソールスクリーンバッファ（リセット用）
#else//USE_WINDOWS_CONSOLE_COLOR//エスケープシーケンス用
	static const handle_t m_handle[color::targetNum];//ハンドル
#endif//USE_WINDOWS_CONSOLE_COLOR
};
//----------------------------------------
//カラーの静的変数をインスタンス化
bool color::m_isInitialized = false;
color::color_t color::m_nowForeColor[color::targetNum] =//現在のカラー：前面（記憶用）
{
	color::reset,//標準出力用
	color::reset,//標準エラー用
};
color::color_t color::m_nowBackColor[color::targetNum] =//現在のカラー：背面（記憶用）
{
	color::reset,//標準出力用
	color::reset,//標準エラー用
};
#ifdef USE_WINDOWS_CONSOLE_COLOR//Windows 用
color::handle_t color::m_handle[color::targetNum];
CONSOLE_SCREEN_BUFFER_INFO color::m_ConsoleScreenBufferInfo[color::targetNum];//Windows 用のリセット用カラー情報
#else//USE_WINDOWS_CONSOLE_COLOR//エスケープシーケンス用
const color::handle_t color::m_handle[] =
{
	stdout,//標準出力ハンドル
	stderr,//標準エラーハンドル
};
#endif//USE_WINDOWS_CONSOLE_COLOR
//----------------------------------------
//コンソールカラー初期化（一回限り）
void color::initializeOnce()
{
	//初期化済みチェック
	if (m_isInitialized)
		return;
```

```
	//静的変数を初期化
	m_isInitialized = true;
#ifdef USE_WINDOWS_CONSOLE_COLOR//Windows 用
	//ハンドル
	m_handle[stdOut] = GetStdHandle(STD_OUTPUT_HANDLE);//標準出力ハンドル
	m_handle[stdErr] = GetStdHandle(STD_ERROR_HANDLE);//標準エラーハンドル
	//カラー変更開始
	GetConsoleScreenBufferInfo(m_handle[stdOut], &m_ConsoleScreenBufferInfo[stdOut]);
	GetConsoleScreenBufferInfo(m_handle[stdErr], &m_ConsoleScreenBufferInfo[stdErr]);
#endif//USE_WINDOWS_CONSOLE_COLOR
}
//---------------------------------------
//コンソールカラー用変数
static color s_colorForInitialize;//初期化処理実行のためのインスタンス
```

【レベルクラス定義】

```
#include <bitset>//std::bitset 用
#include <iterator>//std::iterator 用

//---------------------------------------
//レベルクラス
class level
{
public:
	//型
	typedef unsigned char value_t;//値（レベル）
	typedef unsigned char byte;//バッファ用
private:
	//定数（計算用）
	#define calcAsOutput(value) ((value) >> 1)//値を出力レベルに変換
	#define calcAsValue(value) ((value) << 1)//出力レベルを値に変換
public:
	//定数
	static const value_t NORMAL_NUM = 11;//通常レベル数
	static const value_t SPECIAL_NUM = 2;//特殊レベル数
	static const value_t NUM = NORMAL_NUM + SPECIAL_NUM;//レベル総数
	static const value_t MIN = 0;//レベル最小値
	static const value_t MAX = NUM - 1;//レベル最大値
	static const value_t NORMAL_MIN = MIN;//通常レベル最小値
	static const value_t NORMAL_MAX = NORMAL_MIN + NORMAL_NUM - 1;//通常レベル最大値
	static const value_t SPECIAL_MIN = NORMAL_MAX + 1;//特殊レベル最小値
	static const value_t SPECIAL_MAX = SPECIAL_MIN + SPECIAL_NUM - 1;//特殊レベル最大値
	static const value_t BEGIN = MIN;//レベル開始値（イテレータ用）
	static const value_t END = NUM;//レベル終端値（イテレータ用）
	static const value_t POOL_NUM = NUM + 1;//レベル記録数
	static const value_t OUTPUT_LEVEL_MIN = calcAsOutput(NORMAL_MIN);//出力レベル最小値
	static const value_t OUTPUT_LEVEL_MAX = calcAsOutput(NORMAL_MAX);//出力レベル最大値
	static_assert(SPECIAL_MAX == MAX, "invalid category numbers.");//定数チェック
public:
	//型
	//--------------------
	//イテレータ
	class iterator : public std::iterator<std::bidirectional_iterator_tag, level>//双方向イテレータとして実装
	{
	public:
		//オペレータ
		const level* operator->() const { return &container::get(m_value); }
		const level& operator*() const { return container::get(m_value); }
		bool operator==(const iterator rhs) const { return m_value == rhs.m_value; }
		bool operator!=(const iterator rhs) const { return m_value != rhs.m_value; }
		const level& operator++() const { return m_value >= container::endValue() ? container::getEnd() :
                                                                                     container::get(++m_value); }
		const level& operator++(int) const { return m_value >= container::endValue() ? container::getEnd() :
                                                                                     container::get(m_value++); }
		const level& operator--() const {
```

```
                if (m_value == container::beginValue()){
                      m_value = container::endValue(); return container::getEnd(); }
                return container::get(--m_value); }
        const level& operator--(int) const {
                if (m_value == container::beginValue()){
                      m_value = container::endValue(); return container::getBegin(); }
                return container::get(m_value--); }
public:
        //キャストオペレータ
        operator const level&() const { return container::get(m_value); }//値（レベル）
public:
        //コンストラクタ
        iterator(const value_t value) :
                m_value(value)
        {}
        iterator(const value_t value, int) :
                m_value(container::endValue())
        {}
        iterator(const level& obj) :
                m_value(obj.value())
        {}
        iterator(const level& obj, int) :
                m_value(container::endValue())
        {}
        iterator() :
                m_value(container::endValue())
        {}
        //デストラクタ
        ~iterator()
        {}
private:
        //フィールド
        value_t mutable m_value;//値（レベル）
};
//--------------------
//const イテレータ
typedef const iterator const_iterator;
//--------------------
//コンテナ（イテレータ用）
class container
{
        friend class level;
public:
        //メソッド
        static const value_t beginValue(){ return BEGIN; }//開始値取得
        static const value_t endValue(){ return END; }//終端値取得
        static const level& get(const value_t value){ return m_poolPtr[value]; }//要素を取得
        static const level& getBegin(){ return m_poolPtr[beginValue()]; }//開始要素を取得
        static const level& getEnd(){ return m_poolPtr[endValue()]; }//終端要素を取得
private:
        static void set(const value_t value, const level& obj)//要素を更新
        {
                if (!container::m_isAlreadyPool[value])
                {
                      container::m_poolPtr[value] = obj;
                      container::m_isAlreadyPool[value] = true;
                }
        }
public:
        static const iterator begin(){ return iterator(beginValue()); }//開始イテレータを取得
        static const iterator end() { return iterator(endValue()); }//終端イテレータを取得
        static const_iterator cbegin(){ return const_iterator(beginValue()); }//開始 const イテレータを取得
        static const_iterator cend(){ return const_iterator(endValue()); }//終端 const イテレータを取得
        //※reverse_iterator 非対応
        //メソッド
```

```
            //初期化（一回限り）
            static void initializeOnce();
      private:
            //フィールド
            static bool m_isInitialized;//初期化済み
            static level* m_poolPtr;//要素のプール（ポインタ）
            static byte m_pool[];//要素のプール（バッファ）
                        //※バッファとポインタを分けているのは、コンストラクタの実行を防止するため
            static std::bitset<POOL_NUM> m_isAlreadyPool;//要素の初期化済みフラグ
      };
public:
      //オペレータ
      //※出力レベルで比較する
      bool operator ==(const level& rhs) const { return valueAsOutput() == rhs.valueAsOutput(); }
      bool operator !=(const level& rhs) const { return valueAsOutput() != rhs.valueAsOutput(); }
      bool operator >(const level& rhs) const { return valueAsOutput() > rhs.valueAsOutput(); }
      bool operator >=(const level& rhs) const { return valueAsOutput() >= rhs.valueAsOutput(); }
      bool operator <(const level& rhs) const { return valueAsOutput() < rhs.valueAsOutput(); }
      bool operator <=(const level& rhs) const { return valueAsOutput() <= rhs.valueAsOutput(); }
private:
      //コピーオペレータ
      level& operator=(const level& rhs)
      {
            memcpy(this, &rhs, sizeof(*this));
            return *this;
      }
public:
      //キャストオペレータ
      operator int() const { return static_cast<int>(m_value); }//値（レベル）
      operator value_t() const { return m_value; }//値（レベル）
      operator const char*() const { return m_name; }//名前
public:
      //アクセッサ
      value_t value() const { return m_value; }//値（レベル）取得
      const char* name() const { return m_name; }//名前取得
      static value_t calcValueAsOutput(const value_t value){ return calcAsOutput(value); }//出力レベル計算
      value_t valueAsOutput() const { return calcAsOutput(m_value); }//出力レベル取得
      bool forLog() const { return m_forLog; }//ログレベルとして使用可能か？
      bool forNotice() const { return m_forNotice; }//画面通知レベルとして使用可能か？
      bool forMask() const { return m_forMask; }//出力レベルマスクとして使用可能か？
      color getColor() const { return std::move(changeColor(true)); }//カラー取得
      color getColorForNotice() const { return std::move(changeColorForNotice(true)); }//カラー取得（画面通知用）
public:
      //メソッド
      //カラー変更
      //※戻り値受け取り必須
      //  戻り値を受け取らないとすぐにデストラクタが実行されて元のカラーに戻るので実際に反映されない
      color changeColor(const bool is_only_get = false) const
      {
            return std::move(color(color::stdOut, m_foreColor, m_backColor, !is_only_get));
      }
      //カラー変更（画面通知用）
      //※戻り値受け取り必須
      //  戻り値を受け取らないとすぐにデストラクタが実行されて元のカラーに戻るので実際に反映されない
      color changeColorForNotice(const bool is_only_get = false) const
      {
            return std::move(color(color::stdErr, m_foreColorForNotice, m_backColorForNotice, !is_only_get));
      }
      //コンテナ要素を取得（ショートカット用）
      static const level& get(const value_t value){ return container::get(value); }
      //前のレベルを取得
      const level& prev() const
      {
            if (valueAsOutput() <= OUTPUT_LEVEL_MIN || valueAsOutput() > OUTPUT_LEVEL_MAX || !(m_forLog || m_forNotice))
                  return container::m_poolPtr[m_value];
```

```
            return container::m_poolPtr[calcAsValue(calcAsOutput(m_value) - 1)];
      }
      //次のレベルを取得
      const level& next() const
      {
            if (valueAsOutput() < OUTPUT_LEVEL_MIN || valueAsOutput() >= OUTPUT_LEVEL_MAX || !(m_forLog || m_forNotice))
                  return container::m_poolPtr[m_value];
            return container::m_poolPtr[calcAsValue(calcAsOutput(m_value) + 1)];
      }
public:
      //デフォルトコンストラクタ
      level() :
            m_name(nullptr),
            m_value(UCHAR_MAX),
            m_forLog(false),
            m_forNotice(false),
            m_forMask(false),
            m_foreColor(color::through),
            m_backColor(color::through),
            m_foreColorForNotice(color::through),
            m_backColorForNotice(color::through)
      {
            container::initializeOnce();//コンテナ初期化（一回限り）
      }
      //コピーコンストラクタ
      level(const level& src) :
            m_name(src.m_name),
            m_value(src.m_value),
            m_forLog(src.m_forLog),
            m_forNotice(src.m_forNotice),
            m_forMask(src.m_forMask),
            m_foreColor(src.m_foreColor),
            m_backColor(src.m_backColor),
            m_foreColorForNotice(src.m_foreColorForNotice),
            m_backColorForNotice(src.m_backColorForNotice)
      {
      }
      //コンストラクタ
      level(const value_t value, const char* name, const bool for_log, const bool for_notice, const bool for_mask,
                  const color::color_t fore_color, const color::color_t back_color,
                  const color::color_t fore_color_for_notice, const color::color_t back_color_for_notice) :
            m_name(name),
            m_value(value),
            m_forLog(for_log),
            m_forNotice(for_notice),
            m_forMask(for_mask),
            m_foreColor(fore_color),
            m_backColor(back_color),
            m_foreColorForNotice(fore_color_for_notice),
            m_backColorForNotice(back_color_for_notice)
      {
            assert(value >= BEGIN && value <= END);
            container::set(m_value, *this);//コンテナに登録
      }
      level(const value_t value) :
            m_name(nullptr),
            m_value(value),
            m_forLog(false),
            m_forNotice(false),
            m_forMask(false),
            m_foreColor(color::through),
            m_backColor(color::through),
            m_foreColorForNotice(color::through),
            m_backColorForNotice(color::through)
      {
```

```
            assert(value >= BEGIN && value <= END);
            *this = container::get(m_value);//コンテナから取得して自身にコピー
      }
      //デストラクタ
      ~level()
      {}
private:
      //フィールド
      const char* m_name;//名前
      const value_t m_value;//値（レベル）
      const bool m_forLog;//ログ出力レベルとして使用可能か？
      const bool m_forNotice;//画面通知レベルとして使用可能か？
      const bool m_forMask;//出力レベルマスクとして使用可能か？
      const color::color_t m_foreColor;//カラー：前面
      const color::color_t m_backColor;//カラー：背面
      const color::color_t m_foreColorForNotice;//カラー：前面（画面通知用）
      const color::color_t m_backColorForNotice;//カラー：背面（画面通知用）
private:
      //マクロ消去
      #undef calcAsOutput
      #undef calcAsValue
};
//---------------------------------------
//レベル定義用テンプレートクラス：通常レベル用
template<unsigned char V, bool for_log, bool for_notice, color::color_t fore_color, color::color_t back_color,
                                        color::color_t fore_color_for_notice, color::color_t back_color_for_notice>
class level_normal : public level
{
public:
      //定数
      static const value_t VALUE = V;//値（レベル）
      static_assert(VALUE >= NORMAL_MIN && VALUE <= NORMAL_MAX, "out of range of level");//値の範囲チェック
      static const bool FOR_LOG = for_log;//ログレベルとして使用可能か？
      static const bool FOR_NOTICE = for_notice;//画面通知レベルとして使用可能か？
      static const bool FOR_MASK = true;//出力レベルマスクとして使用可能か？
      static const color::color_t FORE_COLOR = fore_color;//カラー：前面
      static const color::color_t BACK_COLOR = back_color;//カラー：背面
      static const color::color_t FORE_COLOR_FOR_NOTICE = fore_color_for_notice;//カラー：前面（画面通知用）
      static const color::color_t BACK_COLOR_FOR_NOTICE = back_color_for_notice;//カラー：背面（画面通知用）
public:
      //コンストラクタ
      level_normal(const char* name) :
            level(VALUE, name, FOR_LOG, FOR_NOTICE, FOR_MASK, FORE_COLOR, BACK_COLOR, FORE_COLOR_FOR_NOTICE,
                                                                                        BACK_COLOR_FOR_NOTICE)
      {}
};
//---------------------------------------
//レベル定義用テンプレートクラス：特殊レベル用
template<unsigned char V>
class level_special : public level
{
public:
      //定数
      static const value_t VALUE = V;//値（レベル）
      static_assert(VALUE >= SPECIAL_MIN && VALUE <= SPECIAL_MAX, "out of range of level");//値の範囲チェック
      static const bool FOR_LOG = false;//ログレベルとして使用可能か？
      static const bool FOR_NOTICE = false;//画面通知レベルとして使用可能か？
      static const bool FOR_MASK = true;//出力レベルマスクとして使用可能か？
public:
      //コンストラクタ
      level_special(const char* name) :
            level(VALUE, name, FOR_LOG, FOR_NOTICE, FOR_MASK, color::through, color::through, color::through,
                                                                                        color::through)
      {}
};
```

```
//---------------------------------------
//レベル定義クラス：終端用
class level_end : public level
{
public:
	//定数
	static const value_t VALUE = END;//値（レベル）
	static const bool FOR_LOG = false;//ログレベルとして使用可能か？
	static const bool FOR_NOTICE = false;//画面通知レベルとして使用可能か？
	static const bool FOR_MASK = false;//出力レベルマスクとして使用可能か？
public:
	//コンストラクタ
	level_end() :
		level(VALUE, "(END)", FOR_LOG, FOR_NOTICE, FOR_MASK, color::through, color::through, color::through,
		                                                                                    color::through)
	{}
};
//---------------------------------------
//レベル定数
#define define_normalLevel(print_level, sub) (level::NORMAL_MIN + print_level * 2 + sub)
#define define_specialLevel(value) (level::SPECIAL_MIN + value)
enum levelEnum : level::value_t
{
	asNormal = define_normalLevel(1, 0),//通常メッセージ
	asVerbose = define_normalLevel(0, 0),//冗長メッセージ
	asDetail = define_normalLevel(0, 1),//詳細メッセージ
	asImportant = define_normalLevel(2, 0),//重要メッセージ
	asWarning = define_normalLevel(3, 0),//警告メッセージ
	asCritical = define_normalLevel(4, 0),//重大メッセージ
	asAbsolute = define_normalLevel(5, 0),//絶対メッセージ（ログレベルに関係なく出力したいメッセージ）
	//以下、ログレベル／画面通知レベル変更用
	asSilent = define_specialLevel(0),//絶対メッセージ（ログレベルに関係なく出力したいメッセージ）
	asSilentAbsolutely = define_specialLevel(1),//絶対静寂（全てのメッセージを出力しない）
};
//---------------------------------------
//レベル定義
#define declare_normalLevel(value, for_log, for_notice, fore_color, back_color, fore_color_for_notice,
                                                                                    back_color_for_notice) ¥
	struct level_##value : public level_normal<value, for_log, for_notice, fore_color, ¥
	                                           back_color, fore_color_for_notice, back_color_for_notice> ¥
	{ ¥
		level_##value () : ¥
			level_normal<value, for_log, for_notice, fore_color, back_color, fore_color_for_notice, ¥
				back_color_for_notice>(#value) ¥
		{} ¥
	}
#define declare_specialLevel(value) ¥
	struct level_##value : public level_special<value> ¥
	{ ¥
		level_##value () : ¥
			level_special<value>(#value) ¥
		{} ¥
	}
//※以下、ヘッダーで公開する必要なし
declare_normalLevel(asNormal, true, true, color::reset, color::reset, color::black, color::iWhite);
                                                                                    //通常メッセージ
declare_normalLevel(asVerbose, true, false, color::iBlack, color::black, color::iBlack, color::iWhite);
                                                                                    //冗長メッセージ
declare_normalLevel(asDetail, true, false, color::iBlack, color::black, color::iBlack, color::iWhite);//詳細メッセージ
declare_normalLevel(asImportant, true, true, color::iBlue, color::black, color::iBlue, color::iWhite);//重要メッセージ
declare_normalLevel(asWarning, true, true, color::iMagenta, color::black, color::black, color::iMagenta);
                                                                                    //警告メッセージ
declare_normalLevel(asCritical, true, true, color::iRed, color::black, color::iYellow, color::iRed);//重大メッセージ
declare_normalLevel(asAbsolute, true, false, color::through, color::through, color::through, color::through);
                                                        //絶対メッセージ(ログレベルに関係なく出力したいメッセージ)
```

```
//以下、ログレベル／画面通知レベル変更用
declare_specialLevel(asSilent);//静寂（絶対メッセージ以外出力しない）
declare_specialLevel(asSilentAbsolutely);//絶対静寂（全てのメッセージを出力しない）
//----------------------------------------
//レベルコンテナの静的変数をインスタンス化
bool level::container::m_isInitialized = false;
level* level::container::m_poolPtr = nullptr;
level::byte level::container::m_pool[(POOL_NUM)* sizeof(level)];
std::bitset<level::POOL_NUM> level::container::m_isAlreadyPool;
//----------------------------------------
//レベルコンテナ初期化（一回限り）
void level::container::initializeOnce()
{
	//初期化済みチェック
	if (m_isInitialized)
		return;
	//静的変数を初期化
	m_isInitialized = true;
	m_isAlreadyPool.reset();
	memset(m_pool, 0, sizeof(m_pool));
	m_poolPtr = reinterpret_cast<level*>(m_pool);
	//要素を初期化
	for (level::value_t value = 0; value < level::NUM; ++value)
	{
		level(value, "(undefined)", false, false, false, color::through, color::through, color::through,
		                                                                                  color::through);
		m_isAlreadyPool[value] = false;
	}
	//割り当て済みレベルを設定（コンストラクタで要素を登録）
	level_asNormal();//通常メッセージ
	level_asVerbose();//冗長メッセージ
	level_asDetail();//詳細メッセージ
	level_asImportant();//重要メッセージ
	level_asWarning();//警告メッセージ
	level_asCritical();//重大メッセージ
	level_asAbsolute();//絶対メッセージ（ログレベルに関係なく出力したいメッセージ）
	level_asSilent();//静寂（絶対メッセージ以外出力しない）
	level_asSilentAbsolutely();//絶対静寂（全てのメッセージを出力しない）
	level_end();//終端
}
//----------------------------------------
//レベル用変数
static level s_levelForInitialize;//初期化処理実行のためのインスタンス
//----------------------------------------
//レベルコンテナ列挙
void printLevelAll()
{
	for (auto& obj : level::container())//C++11 スタイル
	//for (auto ite = level::container::begin(); ite != level::container::end(); ++ite)//旧来のスタイル
	//for (auto ite = level::container::cbegin(); ite != level::container::cend(); ++ite)//旧来のスタイル
	{
		//const level& obj = ite;//イテレータを変換（イテレータのままでもアロー演算子で直接値操作可能）
		color col(obj.changeColor());
		fprintf(stdout, "level=%d, name=¥"%s¥", valueAsOutput=%d, forLog=%d, forNotice=%d, forMask=%d¥n",
			obj.value(), obj.name(), obj.valueAsOutput(), obj.forLog(), obj.forNotice(), obj.forMask());
		auto& prev = obj.prev();
		auto& next = obj.next();
		fprintf(stdout, "          prev=%s(%d)¥n", prev.name(), prev.value());
		fprintf(stdout, "          next=%s(%d)¥n", next.name(), next.value());
	}
}
```

【カテゴリクラス定義】

```
#include <bitset>//std::bitset 用
#include <iterator>//std::iterator 用
```

```
//---------------------------------------
//カテゴリクラス
class category
{
public:
	//型
	typedef unsigned char value_t;//値（カテゴリ）
	typedef unsigned char byte;//バッファ用
public:
	//定数
	static const value_t NORMAL_NUM = 64;//通常カテゴリ数
	static const value_t ASSIGNED_NUM = 8;//割り当て済みカテゴリ数
	static const value_t RESERVED_NUM = NORMAL_NUM - ASSIGNED_NUM;//予約カテゴリ数
	static const value_t SPECIAL_NUM = 3;//特殊カテゴリ数
	static const value_t NUM = NORMAL_NUM + SPECIAL_NUM;//カテゴリ総数
	static const value_t MIN = 0;//カテゴリ最小値
	static const value_t MAX = NUM - 1;//カテゴリ最大値
	static const value_t NORMAL_MIN = MIN;//通常カテゴリ最小値
	static const value_t NORMAL_MAX = NORMAL_MIN + NORMAL_NUM - 1;//通常カテゴリ最大値
	static const value_t ASSIGNED_MIN = NORMAL_MIN;//割り当て済みカテゴリ最小値
	static const value_t ASSIGNED_MAX = ASSIGNED_MIN + ASSIGNED_NUM - 1;//割り当て済みカテゴリ最大値
	static const value_t RESERVED_MIN = ASSIGNED_MAX + 1;//予約カテゴリ最小値
	static const value_t RESERVED_MAX = RESERVED_MIN + RESERVED_NUM - 1;//予約カテゴリ最大値
	static const value_t SPECIAL_MIN = NORMAL_MAX + 1;//特殊カテゴリ最小値
	static const value_t SPECIAL_MAX = SPECIAL_MIN + SPECIAL_NUM - 1;//特殊カテゴリ最大値
	static const value_t BEGIN = MIN;//カテゴリ開始値（イテレータ用）
	static const value_t END = NUM;//カテゴリ終端値（イテレータ用）
	static const value_t POOL_NUM = NUM + 1;//カテゴリ記録数
	static_assert(NORMAL_MAX == RESERVED_MAX, "invalid category numbers.");//定数チェック
	static_assert(SPECIAL_MAX == MAX, "invalid category numbers.");//定数チェック
public:
	//型
	//--------------------
	//イテレータ
	class iterator : public std::iterator<std::bidirectional_iterator_tag, category>//双方向イテレータとして実装
	{
	public:
		//オペレータ
		const category* operator->() const { return &container::get(m_value); }
		const category& operator*() const { return container::get(m_value); }
		bool operator==(const iterator rhs) const { return m_value == rhs.m_value; }
		bool operator!=(const iterator rhs) const { return m_value != rhs.m_value; }
		const category& operator++() const { return m_value >= container::endValue() ? container::getEnd() :
                                                                                        container::get(++m_value); }
		const category& operator++(int) const { return m_value >= container::endValue() ? container::getEnd() :
                                                                                        container::get(m_value++); }
		const category& operator--() const {
			if (m_value == container::beginValue()){
				m_value = container::endValue(); return container::getEnd(); }
			return container::get(--m_value); }
		const category& operator--(int) const {
			if (m_value == container::beginValue()){
				m_value = container::endValue(); return container::getBegin(); }
			return container::get(m_value--); }
	public:
		//キャストオペレータ
		operator const category&() const { return container::get(m_value); }//値（カテゴリ）
	public:
		//コンストラクタ
		iterator(const value_t value) :
			m_value(value)
		{}
		iterator(const value_t value, int) :
			m_value(container::endValue())
```

```
            {}
            iterator(const category& obj) :
                  m_value(obj.value())
            {}
            iterator(const category& obj, int) :
                  m_value(container::endValue())
            {}
            iterator() :
                  m_value(container::endValue())
            {}
            //デストラクタ
            ~iterator()
            {}
      private:
            //フィールド
            value_t mutable m_value;//値（カテゴリ）
      };
      //--------------------
      //const イテレータ
      typedef const iterator const_iterator;
      //--------------------
      //コンテナ（イテレータ用）
      class container
      {
            friend class category;
      public:
            //メソッド
            static const value_t beginValue(){ return BEGIN; }//開始値取得
            static const value_t endValue(){ return END; }//終端値取得
            static const category& get(const value_t value){ return m_poolPtr[value]; }//要素を取得
            static const category& getBegin(){ return m_poolPtr[beginValue()]; }//開始要素を取得
            static const category& getEnd(){ return m_poolPtr[endValue()]; }//終端要素を取得
      private:
            static void set(const value_t value, const category& obj)//要素を更新
            {
                  if (!container::m_isAlreadyPool[value])
                  {
                        container::m_poolPtr[value] = obj;
                        container::m_isAlreadyPool[value] = true;
                  }
            }
      public:
            static const iterator begin(){ return iterator(beginValue()); }//開始イテレータを取得
            static const iterator end()   { return iterator(endValue()); }//終端イテレータを取得
            static const_iterator cbegin(){ return const_iterator(beginValue()); }//開始 const イテレータを取得
            static const_iterator cend(){ return const_iterator(endValue()); }//終端 const イテレータを取得
            //※reverse_iterator 非対応
            //メソッド
            //初期化（一回限り）
            static void initializeOnce();
      private:
            //フィールド
            static bool m_isInitialized;//初期化済み
            static category* m_poolPtr;//要素のプール（ポインタ）
            static byte m_pool[];//要素のプール（バッファ）
                                 //※バッファとポインタを分けているのは、コンストラクタの実行を防止するため
            static std::bitset<POOL_NUM> m_isAlreadyPool;//要素の初期化済みフラグ
      };
public:
      //オペレータ
      bool operator ==(const category& rhs) const { return m_value == rhs.m_value; }
      bool operator !=(const category& rhs) const { return m_value != rhs.m_value; }
private:
      //コピーオペレータ
      category& operator=(const category& rhs)
```

```
	{
		memcpy(this, &rhs, sizeof(*this));
		return *this;
	}
public:
	//キャストオペレータ
	operator int() const { return static_cast<int>(m_value); }//値（カテゴリ）
	operator value_t() const { return m_value; }//値（カテゴリ）
	operator const char*() const { return m_name; }//名前
public:
	//アクセッサ
	value_t value() const { return m_value; }//値（カテゴリ）取得
	const char* name() const { return m_name; }//名前取得
	bool isAssigned() const { return m_isAssigned; }//割り当て済みカテゴリか？
	bool isReserved() const { return !m_isAssigned; }//予約カテゴリか？
	bool forLog() const { return m_forLog; }//ログ出力可能か？
	bool forNotice() const { return m_forNotice; }//画面通知可能か？
	bool forMask() const { return m_forMask; }//出力レベルマスク可能か？
public:
	//メソッド
	//コンテナ要素を取得（ショートカット用）
	static const category& get(const value_t value){ return container::get(value); }
public:
	//デフォルトコンストラクタ
	category() :
		m_name(nullptr),
		m_value(UCHAR_MAX),
		m_isAssigned(false),
		m_forLog(false),
		m_forNotice(false),
		m_forMask(false)
	{
		container::initializeOnce();//コンテナ初期化（一回限り）
	}
	//コピーコンストラクタ
	category(const category& src) :
		m_name(src.m_name),
		m_value(src.m_value),
		m_isAssigned(src.m_isAssigned),
		m_forLog(src.m_forLog),
		m_forNotice(src.m_forNotice),
		m_forMask(src.m_forMask)
	{
	}
	//コンストラクタ
	category(const value_t value, const char* name, const bool is_assigned, const bool for_log, const bool for_notice,
		const bool for_mask) :
		m_name(name),
		m_value(value),
		m_isAssigned(is_assigned),
		m_forLog(for_log),
		m_forNotice(for_notice),
		m_forMask(for_mask)
	{
		assert(value >= BEGIN && value <= END);
		container::set(m_value, *this);//コンテナに登録
	}
	category(const value_t value) :
		m_name(nullptr),
		m_value(value),
		m_isAssigned(false),
		m_forLog(false),
		m_forNotice(false),
		m_forMask(false)
	{
```

```
            assert(value >= BEGIN && value <= END);
            *this = container::get(m_value);//コンテナから取得して自身にコピー
      }
      //デストラクタ
      ~category()
      {}
private:
      //フィールド
      const char* m_name;//名前
      const value_t m_value;//値（カテゴリ）
      const bool m_isAssigned;//割り当て済みカテゴリか？
      const bool m_forLog;//ログ出力可能か？
      const bool m_forNotice;//画面通知可能か？
      const bool m_forMask;//出力レベルマスク可能か？
};
//---------------------------------------
//カテゴリ定義用テンプレートクラス：割り当て済みカテゴリ用
template<unsigned char V, bool for_log, bool for_notice>
class category_assigned : public category
{
public:
      //定数
      static const value_t VALUE = V;//値（カテゴリ）
      static_assert(VALUE >= ASSIGNED_MIN && VALUE <= ASSIGNED_MAX, "out of range of category");//値の範囲チェック
      static const bool IS_ASSIGNED = true;//割り当て済みカテゴリか？
      static const bool FOR_LOG = for_log;//ログ出力可能か？
      static const bool FOR_NOTICE = for_notice;//画面通知可能か？
      static const bool FOR_MASK = true;//出力レベルマスク可能か？
public:
      //コンストラクタ
      category_assigned(const char* name) :
            category(VALUE, name, IS_ASSIGNED, FOR_LOG, FOR_NOTICE, FOR_MASK)
      {}
};
//---------------------------------------
//カテゴリ定義用テンプレートクラス：予約カテゴリ用
template<unsigned char V, bool for_log, bool for_notice>
class category_reserved: public category
{
public:
      //定数
      static const value_t VALUE = V;//値（カテゴリ）
      static_assert(VALUE >= RESERVED_MIN && VALUE <= RESERVED_MAX, "out of range of category");//値の範囲チェック
      static const bool IS_ASSIGNED = false;//割り当て済みカテゴリか？
      static const bool FOR_LOG = for_log;//ログ出力可能か？
      static const bool FOR_NOTICE = for_notice;//画面通知可能か？
      static const bool FOR_MASK = true;//出力レベルマスク可能か？
public:
      //コンストラクタ
      category_reserved(const char* name) :
            category(VALUE, name, IS_ASSIGNED, FOR_LOG, FOR_NOTICE, FOR_MASK)
      {}
};
//---------------------------------------
//カテゴリ定義用テンプレートクラス：特殊カテゴリ用
template<unsigned char V, bool for_log, bool for_notice, bool for_mask>
class category_special : public category
{
public:
      //定数
      static const value_t VALUE = V;//値（カテゴリ）
      static_assert(VALUE >= SPECIAL_MIN && VALUE <= SPECIAL_MAX, "out of range of category");//値の範囲チェック
      static const bool IS_ASSIGNED = true;//割り当て済みカテゴリか？
      static const bool FOR_LOG = for_log;//ログ出力可能か？
      static const bool FOR_NOTICE = for_notice;//画面通知可能か？
```

```
	static const bool FOR_MASK = for_mask;//出力レベルマスク可能か？
public:
	//コンストラクタ
	category_special(const char* name) :
		category(VALUE, name, IS_ASSIGNED, FOR_LOG, FOR_NOTICE, FOR_MASK)
	{}
};
//---------------------------------------
//カテゴリ定義クラス：終端用
class category_end : public category
{
public:
	//定数
	static const value_t VALUE = END;//値（カテゴリ）
	static const bool IS_ASSIGNED = true;//割り当て済みカテゴリか？
	static const bool FOR_LOG = false;//ログ出力可能か？
	static const bool FOR_NOTICE = false;//画面通知可能か？
	static const bool FOR_MASK = false;//出力レベルマスク可能か？
public:
	//コンストラクタ
	category_end() :
		category(VALUE, "(END)", IS_ASSIGNED, FOR_LOG, FOR_NOTICE, FOR_MASK)
	{}
};
//---------------------------------------
//カテゴリ定数
#define define_assignedCategory(value) (category::ASSIGNED_MIN + value)
#define define_reservedCategory(value) (category::RESERVED_MIN + value)
#define define_specialCategory(value) (category::SPECIAL_MIN + value)
enum categoryEnum : category::value_t
{
	forAny = define_assignedCategory(0),//なんでも（カテゴリなし）
	forLogic = define_assignedCategory(1),//プログラム関係
	forResource = define_assignedCategory(2),//リソース関係
	for3D = define_assignedCategory(3),//3Dグラフィックス関係
	for2D = define_assignedCategory(4),//2Dグラフィックス関係
	forSound = define_assignedCategory(5),//サウンド関係
	forScript = define_assignedCategory(6),//スクリプト関係
	forGameData = define_assignedCategory(7),//ゲームデータ関係
	//ログレベル／画面通知レベル変更用
	forEvery = define_specialCategory(0),//全部まとめて変更
	//特殊なカテゴリ（プリント時専用）
	forCallPoint = define_specialCategory(1),//直近のコールポイントのカテゴリに合わせる（なければforAny扱い）
	forCriticalCallPoint = define_specialCategory(2),
                                        //直近の重大コールポイントのカテゴリに合わせる（なければforAny扱い）
};
//---------------------------------------
//カテゴリ定義
#define declare_assignedCategory(value, for_log, for_notice) ¥
	struct category_##value : public category_assigned<value, for_log, for_notice> ¥
	{ ¥
		category_##value () : ¥
			category_assigned<value, for_log, for_notice>(#value) ¥
		{} ¥
	}
#define declare_reservedCategory(value, for_log, for_notice) ¥
	struct category_##value : public category_reserved<value, for_log, for_notice> ¥
	{ ¥
		category_##value () : ¥
			category_reserved<value, for_log, for_notice>(#value) ¥
		{} ¥
	}
#define declare_specialCategory(value, for_log, for_notice, for_mask) ¥
	struct category_##value : public category_special<value, for_log, for_notice, for_mask> ¥
	{ ¥
```

```
            category_##value () : ¥
                  category_special<value, for_log, for_notice, for_mask>(#value) ¥
            {} ¥
      }
//※以下、ヘッダーで公開する必要なし
declare_assignedCategory(forAny, true, true);//なんでも（カテゴリなし）
declare_assignedCategory(forLogic, true, true);//プログラム関係
declare_assignedCategory(forResource, true, true);//リソース関係
declare_assignedCategory(for3D, true, true);//3D グラフィックス関係
declare_assignedCategory(for2D, true, true);//2D グラフィックス関係
declare_assignedCategory(forSound, true, true);//サウンド関係
declare_assignedCategory(forScript, true, true);//スクリプト関係
declare_assignedCategory(forGameData, true, true);//ゲームデータ関係
//ログレベル／画面通知レベル変更用
declare_specialCategory(forEvery, false, false, true);//全部まとめて変更
//特殊なカテゴリ（プリント時専用）
declare_specialCategory(forCallPoint, true, true, false);
                                                //直近のコールポイントのカテゴリに合わせる（なければ forAny 扱い）
declare_specialCategory(forCriticalCallPoint, true, true, false);
                                                //直近の重大コールポイントのカテゴリに合わせる（なければ forAny 扱い）
//----------------------------------------
//カテゴリコンテナの静的変数をインスタンス化
bool category::container::m_isInitialized = false;
category* category::container::m_poolPtr = nullptr;
category::byte category::container::m_pool[(POOL_NUM)* sizeof(category)];
std::bitset<category::POOL_NUM> category::container::m_isAlreadyPool;
//----------------------------------------
//カテゴリコンテナ初期化（一回限り）
void category::container::initializeOnce()
{
      //初期化済みチェック
      if (m_isInitialized)
            return;
      //静的変数を初期化
      m_isInitialized = true;
      m_isAlreadyPool.reset();
      memset(m_pool, 0, sizeof(m_pool));
      m_poolPtr = reinterpret_cast<category*>(m_pool);
      //要素を初期化
      for (category::value_t value = 0; value < category::NUM; ++value)
      {
            category(value, "(undefined)", false, false, false, false);
            m_isAlreadyPool[value] = false;
      }
      //割り当て済みカテゴリを設定（コンストラクタで要素を登録）
      category_forAny();//なんでも（カテゴリなし）
      category_forLogic();//プログラム関係
      category_forResource();//リソース関係
      category_for3D();//3D グラフィックス関係
      category_for2D();//2D グラフィックス関係
      category_forSound();//サウンド関係
      category_forScript();//スクリプト関係
      category_forGameData();//ゲームデータ関係
      //ログレベル／画面通知レベル変更用
      category_forEvery();//全部まとめて変更
      //特殊なカテゴリ（プリント時専用）
      category_forCallPoint();//直近のコールポイントのカテゴリに合わせる（なければ forAny 扱い）
      category_forCriticalCallPoint();//直近の重大コールポイントのカテゴリに合わせる（なければ forAny 扱い）
      category_end();//終端
}
//----------------------------------------
//カテゴリ用変数
static category s_categoryForInitialize;//初期化処理実行のためのインスタンス
//----------------------------------------
//カテゴリコンテナ列挙
```

```
void printCategoryAll()
{
	for (auto& obj : category::container())//C++11 スタイル
	//for (auto ite = category::container::begin(); ite != category::container::end(); ++ite)//旧来のスタイル
	//for (auto ite = category::container::cbegin(); ite != category::container::cend(); ++ite)//旧来のスタイル
	{
		//const category& obj = ite;//イテレータを変換（イテレータのままでもアロー演算子で直接値操作可能）
		printf("category=%d, name=¥"%s¥", isAssigned=%d, isReserved=%d, forLog=%d, forNotice=%d, forMask=%d¥n",
		obj.value(), obj.name(), obj.isAssigned(), obj.isReserved(), obj.forLog(), obj.forNotice(), obj.forMask());
	}
}
```

【メッセージ固定バッファ】

```
#include <cstddef>//std::size_t 用
#include <thread>//C++11 スレッド
#include <chrono>//C++11 時間
//※CPoolAllocator を使用

//--------------------------------------
//メッセージ固定バッファ
class messageBuffer
{
public:
	//定数
	static const std::size_t BUFFER_SIZE = 2048;//1 個あたりのバッファサイズ
	static const std::size_t BUFFER_NUM = 16;//バッファの個数
public:
	//型
	typedef CPoolAllocatorWithBuff<BUFFER_SIZE, BUFFER_NUM> pool_t;//プールバッファ型
public:
	//メッセージ用固定バッファ取得
	//※成功するまで（他のスレッドが解放するまで）リトライする
	void* assignBuffer()
	{
		void* p = nullptr;
		while (1)
		{
			p = m_pool.alloc(BUFFER_SIZE);
			if (p)
				break;
			std::this_thread::sleep_for(std::chrono::milliseconds(0));
		}
		return p;
	}
	//メッセージバッファ返却
	void releaseBuffer(void* p)
	{
		m_pool.free(p);
	}
public:
	//コンストラクタ
	messageBuffer()
	{}
	//デストラクタ
	~messageBuffer()
	{}
private:
	//フィール
	static pool_t m_pool;//バッファのプール
};
//メッセージ固定バッファ用静的変数インスタンス化
messageBuffer::pool_t messageBuffer::m_pool;//バッファのプール
```

【メッセージクラス定義】

```
#define TLS_IS_WINDOWS//【MS 固有仕様】TLS の宣言を Windows スタイルにする時はこのマクロを有効にする
```

```
#define USE_FUNCSIG//【MS 固有仕様】関数名に__FUNCSIG__を使用する時にこのマクロを有効にする
#define USE_STRCPY_S//【MS 固有仕様】strcpy_s を使用する時にこのマクロを有効にする

#include <assert.h>//assert 用
#include <memory.h>//memcpy 用
#include <string.h>//vsprintf 用
#include <stdarg.h>//valist 用

//スレッドローカルストレージ修飾子
//※C++11 仕様偽装
#ifdef TLS_IS_WINDOWS
#define thread_local __declspec(thread)//Windows 仕様
#else//TLS_IS_WINDOWS
#define thread_local __thread//POSIX 仕様
#endif//TLS_IS_WINDOWS

//---------------------------------------
//表示属性
//※コールポイントの集計に関する設定も扱う
enum msgAttrEnum : unsigned char//属性
{
	//表示属性
	withSeqNo = 0x01,//シーケンス番号表示（未実装）
	withLevel = 0x02,//レベル名表示
	withCategory = 0x04,//カテゴリ名表示
	withoutColor = 0x08,//カラーなし
	//コールポイント集計属性
	withoutProfile = 0x10,//プロファイラへの記録無効化（未実装）
	withoutThreadProfile = 0x20,//プロファイラでのスレッド別集計無効化（未実装）
};
//---------------------------------------
//メッセージ操作
enum msgCtrlEnum :unsigned char
{
	//コールポイントスタック表示操作
	simplePrint = 0x00,//シンプル
	withFuncName = 0x01,//関数名を共に表示
	withFileName = 0x02,//ファイル名を共に表示
};
//---------------------------------------
//メッセージクラス
class message
{
public:
	//型
	typedef char buff_t;//メッセージバッファ用
	typedef unsigned char attr_t;//属性型
	typedef unsigned char control_t;//メッセージ操作型
public:
	//定数
	static const attr_t DEFAULT_ATTR = withLevel | withCategory;//デフォルト表示属性
	static const level::value_t DEFAULT_LOG_LEVEL = asNormal;//デフォルトログレベル
	static const level::value_t DEFAULT_NOTOICE_LEVEL = asCritical;//デフォルト画面通知レベル
public:
	//アクセッサ
	level::value_t getLevel() const { return m_level; }//レベル取得
	category::value_t getCategory() const { return m_category; }//カテゴリ取得
	const char* getName() const { return m_name; }//処理名取得
	const char* getSrcFileName() const { return getFileName(m_srcFileName); }//ソースファイル名取得
	const char* getFuncName() const { return m_funcName; }//関数名取得
	bool hasPushed() const { return m_hasPushed; }//コールポイントスタックにプッシュ済み
	bool attrHasChanged() const { return m_attrHasChanged; }//一時表示属性変更済み
	bool logLevelHasChanged() const { return m_logLevelHasChanged; }//一時ログレベル変更済み
	bool noticeLevelHasChanged() const { return m_noticeLevelHasChanged; }//一時画面通知レベル変更済み
public:
```

```
	//メソッド
	//初期化（一回限り）
	static void initializeOnce();
	//適切なカテゴリに調整
	//※forCallPoint, forCriticalCallPointに基づく変換
	static category::value_t adjustProperCategory(const category::value_t category_)
	{
		if (category_ == forCallPoint)//直前のコールポイントのカテゴリ
		{
			const message* call_point = getLastCallPoint();
			if (call_point)
				return call_point->getCategory();
			return forAny;//コールポイントがなければ forAny 扱い
		}
		else if (category_ == forCriticalCallPoint)//直近のクリティカルコールポイントのカテゴリ
		{
			const message* call_point = getLastCriticalCallPoint();
			if (call_point)
				return call_point->getCategory();
			return forAny;//コールポイントがなければ forAny 扱い
		}
		return category_;
	}
	//-----「表示属性」系処理
	//現在の表示属性を取得
	static attr_t getAttrG(){ return m_attrG; }
	//【一時表示属性版】
	attr_t getAttr() const { return m_attr; }
	//【コールポイント版】
	static attr_t getAttrG_CP()
	{
		message* call_point = getLastCallPointWithAttrHasChanged();
		if (!call_point)
			return getAttrG();//コールポイント無かったら通常の表示属性
		return call_point->getAttr();
	}
	//【一時表示属性のコールポイント版】
	level::value_t getAttr_CP()
	{
		return m_attrHasChanged ? getAttr() : getAttrG_CP();
	}
	//現在の表示属性を変更
	//static void setAttrG(const attr_t attr){ m_attrG = attr; }//禁止
	static attr_t addAttrG(const attr_t attr){ m_attrG |= attr; return m_attrG; }
	static attr_t delAttrG(const attr_t attr){ m_attrG &= ~attr; return m_attrG; }
	//【一時表示属性版】
	//void setAttr(const attr_t attr){ copyAttr(); m_attr = attr; }//禁止
	attr_t addAttr(const attr_t attr){ copyAttr(); m_attr |= attr; return m_attr; }
	attr_t delAttr(const attr_t attr){ copyAttr(); m_attr &= ~attr; return m_attr; }
	//現在の表示属性をリセット
	//※デフォルトに戻す
	static void resetAttrG(){ m_attrG = DEFAULT_ATTR; }
	//【一時表示属性版】
	void resetAttr(){ copyAttr(); m_attr = DEFAULT_ATTR; }
	//現在の表示属性を一時表示属性にコピー
	void copyAttr()
	{
		if (m_attrHasChanged)
			return;
		message* call_point = getLastCallPointWithAttrHasChanged();
		m_attrHasChanged = true;
		if (m_hasPushed)
			++m_attrHasChangedG;
		if (call_point)
			m_attr = call_point->m_attr;//一時表示属性
```

```
        else
            m_attr = m_attrG;//一時表示属性
    }
    //一時表示属性のコピー状態を解除
    //※本来の表示属性に戻す
    void releaseAttr()
    {
        if (!m_attrHasChanged)
            return;
        m_attrHasChanged = false;
        if (m_hasPushed)
            --m_attrHasChangedG;
    }
    //-----「ログレベル」系処理
    //現在のログレベルを取得
    static level::value_t getLogLevelG(const category::value_t category_)
    {
        const category& o_category = category::get(category_);
        assert(o_category.forMask() == true);
        assert(o_category.forLog() == true || o_category.forNotice() == true);
        return m_logLevelG[category_];
    }
    //【コールポイント版】
    static level::value_t getLogLevelG_CP(const category::value_t category_)
    {
        message* call_point = getLastCallPointWithLogLevelHasChanged();
        if (!call_point)
            return getLogLevelG(category_);//コールポイント無かったら通常のログレベル
        return call_point->getLogLevel(category_);
    }
    //【一時ログレベルの有効判定】
    //※本来のレベルより低い一時ログレベルは扱えない
    bool isValidTempLogLevel(const category::value_t category_)
    {
        const category& o_category = category::get(category_);
        assert(o_category.forMask() == true);
        assert(o_category.forLog() == true || o_category.forNotice() == true);
        return m_logLevelHasChanged && level::calcValueAsOutput(m_logLevelG[category_])
                                        <= level::calcValueAsOutput(m_logLevel[category_]);
    }
    //【一時ログレベル版】
    level::value_t getLogLevel(const category::value_t category_)
    {
        return isValidTempLogLevel(category_) ? m_logLevel[category_] : m_logLevelG[category_];
    }
    //【一時ログレベルのコールポイント版】
    level::value_t getLogLevel_CP(const category::value_t category_)
    {
        return isValidTempLogLevel(category_) ? m_logLevel[category_] : getLogLevelG_CP(category_);
    }
    //現在のログレベルを変更
    //※指定の値以上のレベルのメッセージのみをログ出力する
    //※変更前のログレベルを返す（forEvery 指定時は forAny のログレベルを返す）
    static level::value_t setLogLevelG(const level::value_t level_, const category::value_t category_)
    {
        const level& o_level = level::get(level_);
        assert(o_level.forMask() == true);
        const category& o_category = category::get(category_);
        assert(o_category.forMask() == true);
        if (category_ == forEvery)
        {
            const level::value_t prev = m_logLevelG[forAny];
            for (level::value_t& value : m_logLevelG)
                value = level_;
            return prev;
```

```
        }
        assert(o_category.forLog() == true);
        const level::value_t prev = m_logLevelG[category_];
        m_logLevelG[category_] = level_;
        return prev;
}
//【一時ログレベル版】
level::value_t setLogLevel(const level::value_t level_, const category::value_t category_)
{
        const level& o_level = level::get(level_);
        assert(o_level.forMask() == true);
        const category& o_category = category::get(category_);
        assert(o_category.forMask() == true);
        copyLogLevel();//更新前に現在のログレベルをコピー
        if (category_ == forEvery)
        {
                const level::value_t prev = m_logLevel[forAny];
                for (int category_tmp = category::NORMAL_MIN; category_tmp <= category::NORMAL_MAX; ++category_tmp)
                {
                        m_logLevel[category_tmp] =
                                level::calcValueAsOutput(m_logLevelG[category_tmp]) > level::calcValueAsOutput(level_) ?
                                        m_logLevelG[category_tmp] ://本来のレベルより低くすることはできない
                                        level_;
                }
                return prev;
        }
        assert(o_category.forLog() == true);
        const level::value_t prev = m_logLevel[category_];
        m_logLevel[category_] =
                level::calcValueAsOutput(m_logLevelG[category_]) > level::calcValueAsOutput(level_) ?
                        m_logLevelG[category_] ://本来のレベルより低くすることはできない
                        level_;
        return prev;
}
//現在のログレベルをリセット
//※デフォルトに戻す
static void resetLogLevelG(const category::value_t category_)
{
        setLogLevelG(DEFAULT_LOG_LEVEL, category_);
}
//【一時ログレベル版】
void resetLogLevel(const category::value_t category_)
{
        setLogLevel(DEFAULT_LOG_LEVEL, category_);
}
//現在のログレベルを全てリセット
//※デフォルトに戻す
static void resetLogLevelAllG()
{
        //resetLogLevelG(forEvery);//初期化処理でも呼び出されるので、共通処理を使わず単純に更新
        for (level::value_t& value : m_logLevelG)
                value = DEFAULT_LOG_LEVEL;
}
//【一時ログレベル版】
void resetLogLevelAll()
{
        resetLogLevel(forEvery);
}
//ログレベルを一時ログレベルにコピー
void copyLogLevel()
{
        if (m_logLevelHasChanged)
                return;
        message* call_point = getLastCallPointWithLogLevelHasChanged();
        m_logLevelHasChanged = true;
```

```
        if (m_hasPushed)
            ++m_logLevelHasChangedG;
        if (call_point)
            memcpy(m_logLevel, call_point->m_logLevel, sizeof(m_logLevel));//一時ログレベル
        else
            memcpy(m_logLevel, m_logLevelG, sizeof(m_logLevel));//一時ログレベル
    }
    //一時ログレベルのコピー状態を解除
    //※本来のログレベルに戻す
    void releaseLogLevel()
    {
        if (!m_logLevelHasChanged)
            return;
        m_logLevelHasChanged = false;
        if (m_hasPushed)
            --m_logLevelHasChangedG;
    }
    //-----「画面通知レベル」系処理
    //現在の画面通知レベルを取得
    static level::value_t getNoticeLevelG(const category::value_t category_)
    {
        const category& o_category = category::get(category_);
        assert(o_category.forMask() == true);
        assert(o_category.forLog() == true || o_category.forNotice() == true);
        return m_noticeLevelG[category_];
    }
    //【コールポイント版】
    static level::value_t getNoticeLevelG_CP(const category::value_t category_)
    {
        message* call_point = getLastCallPointWithNoticeLevelHasChanged();
        if (!call_point)
            return getNoticeLevelG(category_);//コールポイント無かったら通常の画面通知レベル
        return call_point->getNoticeLevel(category_);
    }
    //【一時画面通知レベルの有効判定】
    //※本来のレベルより低い一時画面通知レベルは扱えない
    bool isValidTempNoticeLevel(const category::value_t category_)
    {
        const category& o_category = category::get(category_);
        assert(o_category.forMask() == true);
        assert(o_category.forLog() == true || o_category.forNotice() == true);
        return m_noticeLevelHasChanged && level::calcValueAsOutput(m_noticeLevelG[category_])
                                                <= level::calcValueAsOutput(m_noticeLevel[category_]);
    }
    //【一時画面通知レベル版】
    level::value_t getNoticeLevel(const category::value_t category_)
    {
        return isValidTempNoticeLevel(category_) ? m_noticeLevel[category_] : m_noticeLevelG[category_];
    }
    //【一時画面通知レベルのコールポイント版】
    level::value_t getNoticeLevel_CP(const category::value_t category_)
    {
        return isValidTempNoticeLevel(category_) ? m_noticeLevel[category_] : getNoticeLevelG_CP(category_);
    }
    //現在の画面通知レベルを変更
    //※指定の値以上のレベルのメッセージのみをログ出力する
    //※変更前の画面通知レベルを返す（forEvery 指定時は forAny の画面通知レベルを返す）
    static level::value_t setNoticeLevelG(const level::value_t level_, const category::value_t category_)
    {
        const level& o_level = level::get(level_);
        assert(o_level.forMask() == true);
        const category& o_category = category::get(category_);
        assert(o_category.forMask() == true);
        if (category_ == forEvery)
        {
```

```
                const level::value_t prev = m_noticeLevelG[forAny];
                for (level::value_t& value : m_noticeLevelG)
                      value = level_;
                return prev;
          }
          assert(o_category.forNotice() == true);
          const level::value_t prev = m_noticeLevelG[category_];
          m_noticeLevelG[category_] = level_;
          return prev;
    }
    //【一時画面通知レベル版】
    level::value_t setNoticeLevel(const level::value_t level_, const category::value_t category_)
    {
          const level& o_level = level::get(level_);
          assert(o_level.forMask() == true);
          const category& o_category = category::get(category_);
          assert(o_category.forMask() == true);
          copyNoticeLevel();//更新前に現在の画面通知レベルをコピー
          if (category_ == forEvery)
          {
                const level::value_t prev = m_noticeLevel[forAny];
                for (int category_tmp = category::NORMAL_MIN; category_tmp <= category::NORMAL_MAX; ++category_tmp)
                {
                      m_noticeLevel[category_tmp] =
                          level::calcValueAsOutput(m_noticeLevelG[category_tmp]) > level::calcValueAsOutput(level_) ?
                            m_noticeLevelG[category_tmp] ://本来のレベルより低くすることはできない
                            level_;
                }
                return prev;
          }
          assert(o_category.forNotice() == true);
          const level::value_t prev = m_noticeLevel[category_];
          m_noticeLevel[category_] =
                level::calcValueAsOutput(m_noticeLevelG[category_]) > level::calcValueAsOutput(level_) ?
                m_noticeLevelG[category_] ://本来のレベルより低くすることはできない
                level_;
          return prev;
    }
    //現在の画面通知レベルをリセット
    //※デフォルトに戻す
    static void resetNoticeLevelG(const category::value_t category_)
    {
          setNoticeLevelG(DEFAULT_NOTOICE_LEVEL, category_);
    }
    //【一時画面通知レベル版】
    void resetNoticeLevel(const category::value_t category_)
    {
          setNoticeLevel(DEFAULT_NOTOICE_LEVEL, category_);
    }
    //現在の画面通知レベルを全てリセット
    //※デフォルトに戻す
    static void resetNoticeLevelAllG()
    {
          //resetNoticeLevelG(forEvery);//初期化処理でも呼び出されるので、共通処理を使わず単純に更新
          for (level::value_t& value : m_noticeLevelG)
                value = DEFAULT_NOTOICE_LEVEL;
    }
    //【一時画面通知レベル版】
    void resetNoticeLevelAll()
    {
          resetNoticeLevel(forEvery);
    }
    //画面通知レベルを一時画面通知レベルにコピー
    void copyNoticeLevel()
    {
```

```
            if (m_noticeLevelHasChanged)
                  return;
            message* call_point = getLastCallPointWithLogLevelHasChanged();
            m_noticeLevelHasChanged = true;
            if (m_hasPushed)
                  ++m_noticeLevelHasChangedG;
            if (call_point)
                  memcpy(m_noticeLevel, call_point->m_noticeLevel, sizeof(m_noticeLevel));//一時ログレベル
            else
                  memcpy(m_noticeLevel, m_noticeLevelG, sizeof(m_noticeLevel));//一時ログレベル
      }
      //一時画面通知レベルのコピー状態を解除
      //※本来の画面通知レベルに戻す
      void releaseNoticeLevel()
      {
            if (!m_noticeLevelHasChanged)
                  return;
            m_noticeLevelHasChanged = false;
            if (m_hasPushed)
                  --m_noticeLevelHasChangedG;
      }
protected:
      //-----「コールポイント操作」系処理（共通内部処理）
      //コールポイントスタックに自身をプッシュ
      void pushCallPoint()
      {
            if (m_hasPushed)
                  return;
            m_hasPushed = true;
            assert(m_callPointStackHead != this);
            m_callPointStackNext = m_callPointStackHead;
            m_callPointStackHead = this;
      }
      //コールポイントスタックから（自身を）ポップ
      void popCallPoint()
      {
            if (!m_hasPushed)
                  return;
            m_hasPushed = false;
            assert(m_callPointStackHead == this);
            m_callPointStackHead = m_callPointStackNext;
            m_callPointStackNext = nullptr;
      }
public:
      //-----「コールポイント操作」系処理
      //直前のコールポイントを取得
      static message* getLastCallPoint()
      {
            return m_callPointStackHead;
      }
      //直近のクリティカルコールポイントを取得
      static message* getLastCriticalCallPoint()
      {
            message* node = m_callPointStackHead;
            while (node)
            {
                  if (node->getLevel() == asCritical)
                        break;
                  node = node->m_callPointStackNext;
            }
            return node;
      }
      //直近で表示属性を更新したコールポイントを取得
      static message* getLastCallPointWithAttrHasChanged()
      {
```

```
            if (m_attrHasChangedG == 0)
                  return nullptr;
            message* node = m_callPointStackHead;
            while (node)
            {
                  if (node->attrHasChanged())
                        break;
                  node = node->m_callPointStackNext;
            }
            return node;
      }
      //直近でログレベルを更新したコールポイントを取得
      static message* getLastCallPointWithLogLevelHasChanged()
      {
            if (m_logLevelHasChangedG == 0)
                  return nullptr;
            message* node = m_callPointStackHead;
            while (node)
            {
                  if (node->logLevelHasChanged())
                        break;
                  node = node->m_callPointStackNext;
            }
            return node;
      }
      //直近で画面通知レベルを更新したコールポイントを取得
      static message* getLastCallPointWithNoticeLevelHasChanged()
      {
            if (m_noticeLevelHasChangedG == 0)
                  return nullptr;
            message* node = m_callPointStackHead;
            while (node)
            {
                  if (node->noticeLevelHasChanged())
                        break;
                  node = node->m_callPointStackNext;
            }
            return node;
      }
protected:
      //----「メッセージバッファ操作」系処理（共通内部処理）
      //メッセージバッファ確保
      static void beginBuffer()
      {
            if (m_messageBuff)
                  freeBuffer();
            messageBuffer allocator;//メッセージ固定バッファ
            m_messageBuff = reinterpret_cast<buff_t*>(allocator.assignBuffer());
            m_messageBuffSize = messageBuffer::BUFFER_SIZE;
            m_messageBuffUsed = 0;
            if (!m_messageBuff)
                  m_messageBuffSize = 0;
      }
      //メッセージバッファ解放
      static void freeBuffer()
      {
            if (!m_messageBuff)
                  return;
            messageBuffer allocator;//メッセージ固定バッファ
            allocator.releaseBuffer(m_messageBuff);
            m_messageBuff = nullptr;//メッセージバッファ
            m_messageBuffSize = 0;//メッセージバッファサイズ
            m_messageBuffUsed = 0;//メッセージバッファ使用量
      }
      //メッセージバッファフラッシュ
```

```
static void flushBuffer(const level& o_level, const bool flush_log, const bool flush_notice,
                                                                    const bool log_without_color = false)
{
      if (!m_messageBuff)
            return;
      //【仮処理】本来はメッセージをキューイングしてロギング処理／画面通知処理に渡す
      if (flush_log)
      {
            if (log_without_color)
            {
                  fprintf(stdout, m_messageBuff);
            }
            else
            {
                  color col(o_level.changeColor());
                  fprintf(stdout, m_messageBuff);
            }
      }
      if (flush_notice)
      {
            color col(o_level.changeColorForNotice());
            fprintf(stderr, m_messageBuff);
      }
      freeBuffer();//バッファ解放
}
//バッファリング
static int vbuffer(const char* fmt, va_list args)
{
      std::size_t remain = m_messageBuffSize - m_messageBuffUsed;
      if (!m_messageBuff || remain == 0)
            return 0;
#ifdef USE_STRCPY_S
      const int ret = vsprintf_s(m_messageBuff + m_messageBuffUsed, remain, fmt, args);
#else//USE_STRCPY_S
      const int ret = vsprintf(m_messageBuff + m_messageBuffUsed, fmt, args);
#endif//USE_STRCPY_S
      m_messageBuffUsed += ret;
      return ret;
}
static int buffer(const char* fmt, ...)
{
      va_list args;
      va_start(args, fmt);
      const int ret = vbuffer(fmt, args);
      va_end(args);
      return ret;
}
//出力判定
static void isAllowPrint(bool& is_allow_log, bool& is_allow_notice, const level::value_t log_level,
            const level::value_t notice_level, const level::value_t level_, const category::value_t category_)
{
      const level& o_level = level::get(level_);
      assert(o_level.forLog() == true || o_level.forNotice() == true);
      const category& o_category = category::get(category_);
      assert(o_category.forLog() == true || o_category.forNotice() == true);
      const level& o_log_level = level::get(log_level);
      const level& o_notice_level = level::get(notice_level);
      is_allow_log = (o_level >= o_log_level && o_level.forLog() && o_category.forLog());
      is_allow_notice = (o_level >= o_notice_level && o_level.forNotice() && o_category.forNotice());
}
//-----「メッセージ出力」系処理（共通内部処理）
//メッセージ出力：レベルを引数指定
//※va_list を引数にとるバージョン
static int vprintCommon(const attr_t attr, const level::value_t log_level, const level::value_t notice_level,
                  const level::value_t level_, const category::value_t category_, const char* fmt, va_list args)
```

```
{
	bool is_allow_log;
	bool is_allow_notice;
	isAllowPrint(is_allow_log, is_allow_notice, log_level, notice_level, level_, category_);
	if (!is_allow_log && !is_allow_notice)
		return 0;
	const level& o_level = level::get(level_);
	const category& o_category = category::get(category_);
	beginBuffer();
	const bool without_color = (attr & withoutColor) ? true : false;
	const attr_t attr_tmp = attr & (withLevel | withCategory);
	if (is_allow_log && attr_tmp != 0)
	{
		if (attr_tmp == withLevel)
			buffer("[%s]", o_level.name());
		else if (attr_tmp == withCategory)
			buffer("[%s]", o_category.name());
		else//if (attr_tmp == (withLevel | withCategory))
			buffer("[%s,%s]", o_level.name(), o_category.name());
		flushBuffer(o_level, true, false, without_color);
		beginBuffer();
		buffer(" ");
	}
	int ret = vbuffer(fmt, args);
	flushBuffer(o_level, is_allow_log, is_allow_notice, true);
	return ret;
}
//-----「コールポイントスタック出力」系処理（共通内部処理）
//コールポイントスタックをリスト表示
static void printCPStackCommon(const attr_t attr, const level::value_t log_level, const level::value_t level_,
			const category::value_t category_, const char* name, const control_t control)
{
	bool is_allow_log;
	bool is_allow_notice;
	isAllowPrint(is_allow_log, is_allow_notice, log_level, asSilentAbsolutely, level_, category_);
	if (!is_allow_log)
		return;
	const level& o_level = level::get(level_);
	const category& o_category = category::get(category_);
	const bool without_color = (attr & withoutColor) ? true : false;
	const attr_t attr_tmp = attr & (withLevel | withCategory);
	beginBuffer();
	buffer("============================================================¥n");
	if (attr_tmp != 0)
	{
		if (attr_tmp == withLevel)
			buffer("[%s]", o_level.name());
		else if (attr_tmp == withCategory)
			buffer("[%s]", o_category.name());
		else//if (attr_tmp == (withLevel | withCategory))
			buffer("[%s,%s]", o_level.name(), o_category.name());
	}
	if (!without_color)
	{
		flushBuffer(o_level, true, false);
		beginBuffer();
	}
	buffer(" Call point stack at ¥"%s¥"¥n", name);
	buffer("----------------------------------------¥n");
	const message* call_point = m_callPointStackHead;
	while (call_point)
	{
		buffer("  ¥"%s¥" ", call_point->getName());
		if (attr_tmp != 0)
		{
```

```
                        const level& o_level_tmp = level::get(call_point->getLevel());
                        const category& o_category_tmp = category::get(call_point->getCategory());
                        if (!without_color)
                        {
                              flushBuffer(o_level, true, false, true);
                              beginBuffer();
                        }
                        if (attr_tmp == withLevel)
                              buffer("[%s]", o_level_tmp.name());
                        else if (attr_tmp == withCategory)
                              buffer("[%s]", o_category_tmp.name());
                        else//if (attr_tmp == (withLevel | withCategory))
                              buffer("[%s,%s]", o_level_tmp.name(), o_category_tmp.name());
                        if (!without_color)
                        {
                              flushBuffer(o_level_tmp, true, false, false);
                              beginBuffer();
                        }
                  }
                  const control_t control_tmp = control & (withFuncName | withFileName);
                  if (control_tmp != 0)
                  {
                        buffer(" ... ");
                        if (control_tmp & withFuncName)
                              buffer("%s", call_point->getFuncName());
                        if (control_tmp == (withFuncName | withFileName))
                              buffer(" : ");
                        if (control_tmp & withFileName)
                              buffer(" %s", call_point->getSrcFileName());
                  }
                  buffer("¥n");
                  call_point = call_point->m_callPointStackNext;
            }
            if (!without_color)
            {
                  flushBuffer(o_level, true, false, true);
                  beginBuffer();
            }
            buffer("============================================================¥n");
            flushBuffer(o_level, true, false, without_color);
      }
public:
      //-----「メッセージ出力」系処理
      //メッセージ出力／ログ出力／画面通知メソッド：レベルを引数指定
      //※vprint***/vlog***/vnotice***
      //※va_list を引数にとるバージョン
      static int vprintG(const level::value_t level_, const category::value_t category_, const char* fmt, va_list args)
      {
            const category::value_t category_adj = adjustProperCategory(category_);
            const attr_t attr_adj = getAttrG_CP();
            const level::value_t log_level = getLogLevelG_CP(category_adj);
            const level::value_t notice_level = getNoticeLevelG_CP(category_adj);
            return vprintCommon(attr_adj, log_level, notice_level, level_, category_adj, fmt, args);
      }
      int vprint(const level::value_t level_, const category::value_t category_, const char* fmt, va_list args)
                                                                              //【一時ログ／画面通知レベル版】
      {
            const category::value_t category_adj = adjustProperCategory(category_);
            const attr_t attr_adj = getAttr_CP();
            const level::value_t log_level = getLogLevel_CP(category_adj);
            const level::value_t notice_level = getNoticeLevel_CP(category_adj);
            return vprintCommon(attr_adj, log_level, notice_level, level_, category_adj, fmt, args);
      }
      int vprint(const level::value_t level_, const char* fmt, va_list args)//【一時ログ／画面通知レベル版】
      {
```

```
	return vprint(level_, m_category, fmt, args);
}
int vprint(const char* fmt, va_list args)//【一時ログ／画面通知レベル版】
{
	return vprint(m_level, m_category, fmt, args);
}
static int vlogG(const level::value_t level_, const category::value_t category_, const char* fmt, va_list args)
{
	const category::value_t category_adj = adjustProperCategory(category_);
	const attr_t attr_adj = getAttrG_CP();
	const level::value_t log_level = getLogLevelG_CP(category_adj);
	const level::value_t notice_level = asSilentAbsolutely;
	return vprintCommon(attr_adj, log_level, notice_level, level_, category_adj, fmt, args);
}
int vlog(const level::value_t level_, const category::value_t category_, const char* fmt, va_list args)
                                                                                    //【一時ログレベル版】
{
	const category::value_t category_adj = adjustProperCategory(category_);
	const attr_t attr_adj = getAttr_CP();
	const level::value_t log_level = getLogLevel_CP(category_adj);
	const level::value_t notice_level = asSilentAbsolutely;
	return vprintCommon(attr_adj, log_level, notice_level, level_, category_adj, fmt, args);
}
int vlog(const level::value_t level_, const char* fmt, va_list args)//【一時ログレベル版】
{
	return vlog (level_, m_category, fmt, args);
}
int vlog(const char* fmt, va_list args)//【一時ログレベル版】
{
	return vlog(m_level, m_category, fmt, args);
}
static int vnoticeG(const level::value_t level_, const category::value_t category_, const char* fmt, va_list args)
{
	const category::value_t category_adj = adjustProperCategory(category_);
	const attr_t attr_adj = getAttrG_CP();
	const level::value_t log_level = asSilentAbsolutely;
	const level::value_t notice_level = getNoticeLevelG_CP(category_adj);
	return vprintCommon(attr_adj, log_level, notice_level, level_, category_adj, fmt, args);
}
int vnotice(const level::value_t level_, const category::value_t category_, const char* fmt, va_list args)
                                                                                    //【一時画面通知レベル版】
{
	const category::value_t category_adj = adjustProperCategory(category_);
	const attr_t attr_adj = getAttr_CP();
	const level::value_t log_level = asSilentAbsolutely;
	const level::value_t notice_level = getNoticeLevel_CP(category_adj);
	return vprintCommon(attr_adj, log_level, notice_level, level_, category_adj, fmt, args);
}
int vnotice(const level::value_t level_, const char* fmt, va_list args)//【一時画面通知レベル版】
{
	return vnotice(level_, m_category, fmt, args);
}
int vnotice(const char* fmt, va_list args)//【一時画面通知レベル版】
{
	return vnotice(m_level, m_category, fmt, args);
}
//メッセージ出力／ログ出力／画面通知メソッド：レベルを引数指定
//※print***/log***/notice***
//※可変長引数バージョン
static int printG(const level::value_t level_, const category::value_t category_, const char* fmt, ...)
{
	va_list args;
	va_start(args, fmt);
	const int ret = vprintG(level_, category_, fmt, args);
	va_end(args);
```

```
	return ret;
}
int print(const level::value_t level_, const category::value_t category_, const char* fmt, ...)
                                                                    //【一時ログ／画面通知レベル版】
{
	va_list args;
	va_start(args, fmt);
	const int ret = vprint(level_, category_, fmt, args);
	va_end(args);
	return ret;
}
int print(const level::value_t level_, const char* fmt, ...)//【一時ログ／画面通知レベル版】
{
	va_list args;
	va_start(args, fmt);
	const int ret = vprint(level_, fmt, args);
	va_end(args);
	return ret;
}
int print(const char* fmt, ...)//【一時ログ／画面通知レベル版】
{
	va_list args;
	va_start(args, fmt);
	const int ret = vprint(fmt, args);
	va_end(args);
	return ret;
}
static int logG(const level::value_t level_, const category::value_t category_, const char* fmt, ...)
{
	va_list args;
	va_start(args, fmt);
	const int ret = vlogG(level_, category_, fmt, args);
	va_end(args);
	return ret;
}
int log(const level::value_t level_, const category::value_t category_, const char* fmt, ...)
                                                                    //【一時ログレベル版】
{
	va_list args;
	va_start(args, fmt);
	const int ret = vlog(level_, category_, fmt, args);
	va_end(args);
	return ret;
}
int log(const level::value_t level_, const char* fmt, ...)//【一時ログレベル版】
{
	va_list args;
	va_start(args, fmt);
	const int ret = vlog(level_, fmt, args);
	va_end(args);
	return ret;
}
int log(const char* fmt, ...)//【一時ログレベル版】
{
	va_list args;
	va_start(args, fmt);
	const int ret = vlog(fmt, args);
	va_end(args);
	return ret;
}
static int noticeG(const level::value_t level_, const category::value_t category_, const char* fmt, ...)
{
	va_list args;
	va_start(args, fmt);
	const int ret = vnoticeG(level_, category_, fmt, args);
```

```
            va_end(args);
            return ret;
      }
      int notice(const level::value_t level_, const category::value_t category_, const char* fmt, ...)
                                                                                  //【一時画面通知レベル版】
      {
            va_list args;
            va_start(args, fmt);
            const int ret = vnotice(level_, category_, fmt, args);
            va_end(args);
            return ret;
      }
      int notice(const level::value_t level_, const char* fmt, ...)//【一時画面通知レベル版】
      {
            va_list args;
            va_start(args, fmt);
            const int ret = vnotice(level_, fmt, args);
            va_end(args);
            return ret;
      }
      int notice(const char* fmt, ...)//【一時画面通知レベル版】
      {
            va_list args;
            va_start(args, fmt);
            const int ret = vnotice(fmt, args);
            va_end(args);
            return ret;
      }
      //メッセージ出力／ログ出力／画面通知メソッド定義マクロ
      //※vprint***/vlog***/vnotice***
      //※va_list を引数にとるバージョン
#define declare_vprintMethods(level_) ¥
      static int vprintAs##level_##G(const category::value_t category_, const char* fmt, va_list args) ¥
      { ¥
            return vprintG(as##level_, category_, fmt, args); ¥
      } ¥
      int vprintAs##level_(const category::value_t category_, const char* fmt, va_list args) ¥
                                                                        /*【一時ログ／画面通知レベル版】*/ ¥
      { ¥
            return vprint(as##level_, category_, fmt, args); ¥
      } ¥
      int vprintAs##level_(const char* fmt, va_list args)/*【一時ログ／画面通知レベル版】*/ ¥
      { ¥
            return vprint(as##level_, fmt, args); ¥
      } ¥
      static int vlogAs##level_##G(const category::value_t category_, const char* fmt, va_list args) ¥
      { ¥
            return vlogG(as##level_, category_, fmt, args); ¥
      } ¥
      int vlogAs##level_(const category::value_t category_, const char* fmt, va_list args)/*【一時ログレベル版】*/ ¥
      { ¥
            return vlog(as##level_, category_, fmt, args); ¥
      } ¥
      int vlogAs##level_(const char* fmt, va_list args)/*【一時ログレベル版】*/ ¥
      { ¥
            return vlog(as##level_, fmt, args); ¥
      } ¥
      static int vnoticeAs##level_##G(const category::value_t category_, const char* fmt, va_list args) ¥
      { ¥
            return vnoticeG(as##level_, category_, fmt, args); ¥
      } ¥
      int vnoticeAs##level_(const category::value_t category_, const char* fmt, va_list args) ¥
                                                                        /*【一時画面通知レベル版】*/ ¥
      { ¥
            return vnotice(as##level_, category_, fmt, args); ¥
```

```
	} ¥
	int vnoticeAs##level_(const char* fmt, va_list args)/*【一時画面通知レベル版】*/ ¥
	{ ¥
		return vnotice(as##level_, fmt, args); ¥
	}
	//メッセージ出力／ログ出力／画面通知メソッド定義マクロ
	//※print***/log***/notice***
	//※可変長引数バージョン
#define declare_printMethods(level_) ¥
	static int printAs##level_##G(const category::value_t category_, const char* fmt, ...) ¥
	{ ¥
		va_list args; ¥
		va_start(args, fmt); ¥
		const int ret = vprintAs##level_##G(category_, fmt, args); ¥
		va_end(args); ¥
		return ret; ¥
	} ¥
	int printAs##level_(const category::value_t category_, const char* fmt, ...)/*【一時ログ／画面通知レベル版】*/ ¥
	{ ¥
		va_list args; ¥
		va_start(args, fmt); ¥
		const int ret = vprintAs##level_(category_, fmt, args); ¥
		va_end(args); ¥
		return ret; ¥
	} ¥
	int printAs##level_(const char* fmt, ...)/*【一時ログ／画面通知レベル版】*/ ¥
	{ ¥
		va_list args; ¥
		va_start(args, fmt); ¥
		const int ret = vprintAs##level_(fmt, args); ¥
		va_end(args); ¥
		return ret; ¥
	} ¥
	static int logAs##level_##G(const category::value_t category_, const char* fmt, ...) ¥
	{ ¥
		va_list args; ¥
		va_start(args, fmt); ¥
		const int ret = vlogAs##level_##G(category_, fmt, args); ¥
		va_end(args); ¥
		return ret; ¥
	} ¥
	int logAs##level_(const category::value_t category_, const char* fmt, ...)/*【一時ログレベル版】*/ ¥
	{ ¥
		va_list args; ¥
		va_start(args, fmt); ¥
		const int ret = vlogAs##level_(category_, fmt, args); ¥
		va_end(args); ¥
		return ret; ¥
	} ¥
	int logAs##level_(const char* fmt, ...)/*【一時ログレベル版】*/ ¥
	{ ¥
		va_list args; ¥
		va_start(args, fmt); ¥
		const int ret = vlogAs##level_(fmt, args); ¥
		va_end(args); ¥
		return ret; ¥
	} ¥
	static int noticeAs##level_##G(const category::value_t category_, const char* fmt, ...) ¥
	{ ¥
		va_list args; ¥
		va_start(args, fmt); ¥
		const int ret = vnoticeAs##level_##G(category_, fmt, args); ¥
		va_end(args); ¥
		return ret; ¥
	} ¥
```

```
	int noticeAs##level_(const category::value_t category_, const char* fmt, ...)/*【一時画面通知レベル版】*/ ¥
	{ ¥
		va_list args; ¥
		va_start(args, fmt); ¥
		const int ret = vnoticeAs##level_(category_, fmt, args); ¥
		va_end(args); ¥
		return ret; ¥
	} ¥
	int noticeAs##level_(const char* fmt, ...)/*【一時画面通知レベル版】*/ ¥
	{ ¥
		va_list args; ¥
		va_start(args, fmt); ¥
		const int ret = vnoticeAs##level_(fmt, args); ¥
		va_end(args); ¥
		return ret; ¥
	}
	//メッセージ出力／ログ出力／画面通知メソッド定義
	//※vprint***/vlog***/vnotice***
	//※va_list を引数にとるバージョン
	declare_vprintMethods(Normal);//通常メッセージ
	declare_vprintMethods(Verbose);//冗長メッセージ
	declare_vprintMethods(Detail);//詳細メッセージ
	declare_vprintMethods(Important);//重要メッセージ
	declare_vprintMethods(Warning);//警告メッセージ
	declare_vprintMethods(Critical);//重大（問題）メッセージ
	declare_vprintMethods(Absolute);//絶対（必須）メッセージ
	//メッセージ出力／ログ出力／画面通知メソッド定義
	//※print***/log***/notice***
	//※可変長引数バージョン
	declare_printMethods(Normal);//通常メッセージ
	declare_printMethods(Verbose);//冗長メッセージ
	declare_printMethods(Detail);//詳細メッセージ
	declare_printMethods(Important);//重要メッセージ
	declare_printMethods(Warning);//警告メッセージ
	declare_printMethods(Critical);//重大（問題）メッセージ
	declare_printMethods(Absolute);//絶対（必須）メッセージ
	//-----「コールポイントスタック出力」系処理
	//コールポイントスタックをリスト表示
	static void printCPStackG(const level::value_t level_, category::value_t category_, const char* name,
	                                                                const control_t control = simplePrint)
	{
		const category::value_t category_adj = adjustProperCategory(category_);
		const attr_t attr_adj = getAttrG_CP();
		const level::value_t log_level = getLogLevelG_CP(category_adj);
		printCPStackCommon(attr_adj, log_level, level_, category_adj, name, control);
	}
	void printCPStack(const level::value_t level_, category::value_t category_, const char* name,
	                                                                const control_t control = simplePrint)
	{
		const category::value_t category_adj = adjustProperCategory(category_);
		const attr_t attr_adj = getAttr_CP();
		const level::value_t log_level = getLogLevel_CP(category_adj);
		printCPStackCommon(attr_adj, log_level, level_, category_adj, name, control);
	}
#define declare_printCPMethods(level_) ¥
	static void printCPStackAs##level_##G(const category::value_t category_, const char* name, ¥
	                                                                const control_t control = simplePrint) ¥
	{ ¥
		printCPStackG(as##level_, category_, name, control); ¥
	} ¥
	void printCPStackAs##level_(const category::value_t category_, const char* name, ¥
	                                                                const control_t control = simplePrint) ¥
	                                                                      /*【一時ログ／画面通知レベル版】*/ ¥
	{ ¥
		printCPStack(as##level_, category_, name, control); ¥
```

```
	} ¥
	void printCPStackAs##level_(const char* name, const control_t control = simplePrint) ¥
                                                                                /*【一時ログ／画面通知レベル版】*/ ¥
	{ ¥
		printCPStack(as##level_, m_category, name, control); ¥
	}
	declare_printCPMethods(Normal);//通常メッセージ
	declare_printCPMethods(Verbose);//冗長メッセージ
	declare_printCPMethods(Detail);//詳細メッセージ
	declare_printCPMethods(Important);//重要メッセージ
	declare_printCPMethods(Warning);//警告メッセージ
	declare_printCPMethods(Critical);//重大（問題）メッセージ
	declare_printCPMethods(Absolute);//絶対（必須）メッセージ
public:
	//デフォルトコンストラクタ
	message():
		m_level(asNormal),//レベル
		m_category(forAny),//カテゴリ
		m_name(nullptr),//名前
		m_srcFileName(nullptr),//ソースファイル名
		m_funcName(nullptr),//関数名
		m_attr(0),//一時表示属性
		m_hasPushed(false),//コールポイントスタックにプッシュ済み
		m_attrHasChanged(false),//一時表示属性変更済み
		m_logLevelHasChanged(false),//一時ログレベル変更済み
		m_noticeLevelHasChanged(false),//一時画面通知レベル変更済み
		m_callPointStackNext(nullptr)//次のコールポイントスタック
	{
		initializeOnce();
	}
	//コンストラクタ
	message(const level::value_t level_, const category::value_t category_, const char* name = nullptr,
                                        const char* src_file_name = nullptr, const char* func_name = nullptr) :
		m_level(level_),//レベル
		m_category(category_),//カテゴリ
		m_name(name),//名前
		m_srcFileName(src_file_name),//ソースファイル名
		m_funcName(func_name),//関数名
		m_attr(0),//一時表示属性
		m_hasPushed(false),//コールポイントスタックにプッシュ済み
		m_attrHasChanged(false),//一時表示属性変更済み
		m_logLevelHasChanged(false),//一時ログレベル変更済み
		m_noticeLevelHasChanged(false),//一時画面通知レベル変更済み
		m_callPointStackNext(nullptr)//次のコールポイントスタック
	{
		assert(category_ >= category::NORMAL_MIN && category_ <= category::NORMAL_MAX);
	}
	//デストラクタ
	~message()
	{
		releaseAttr();//一時表示属性を解放
		releaseLogLevel();//一時ログレベルを解放
		releaseNoticeLevel();//一時画面通知レベルを解放
		popCallPoint();//コールスタックからポップ
	}
protected:
	//フィールド
	const level::value_t m_level;//レベル
	const category::value_t m_category;//カテゴリ
	const char* m_name;//処理名
	const char* m_srcFileName;//ソースファイル名
	const char* m_funcName;//関数名
	attr_t m_attr;//表示属性
	bool m_hasPushed;//コールポイントスタックにプッシュ済み
	bool m_attrHasChanged;//一時表示属性変更済み
```

```
	bool m_logLevelHasChanged;//一時ログレベル変更済み
	bool m_noticeLevelHasChanged;//一時画面通知レベル変更済み
	message* m_callPointStackNext;//次のコールポイントスタック
	level::value_t m_logLevel[category::NORMAL_NUM];//一時ログレベル
	level::value_t m_noticeLevel[category::NORMAL_NUM];//一時画面通知レベル
private:
	static bool m_isInitialized;//初期化済みフラグ
	static attr_t m_attrG;//表示属性
	static level::value_t m_logLevelG[category::NORMAL_NUM];//現在のログレベル
	static level::value_t m_noticeLevelG[category::NORMAL_NUM];//現在の画面通知レベル
	static thread_local int m_attrHasChangedG;//一時表示属性変更スタック数
	static thread_local int m_logLevelHasChangedG;//一時ログレベル変更スタック数
	static thread_local int m_noticeLevelHasChangedG;//一時画面通知レベル変更スタック数
	static thread_local message* m_callPointStackHead;//コールポイントスタックの先頭
	static thread_local buff_t* m_messageBuff;//メッセージバッファ
	static thread_local std::size_t m_messageBuffSize;//メッセージバッファサイズ
	static thread_local std::size_t m_messageBuffUsed;//メッセージバッファ使用量
};
//---------------------------------------
//メッセージの静的変数をインスタンス化
bool message::m_isInitialized = false;//初期化済みフラグ
message::attr_t message::m_attrG = 0;//表示属性
level::value_t message::m_logLevelG[category::NORMAL_NUM];//現在のログレベル
level::value_t message::m_noticeLevelG[category::NORMAL_NUM];//現在の画面通知レベル
thread_local int message::m_attrHasChangedG = 0;//一時表示属性変更スタック数
thread_local int message::m_logLevelHasChangedG = 0;//一時ログレベル変更スタック数
thread_local int message::m_noticeLevelHasChangedG = 0;//一時画面通知レベル変更スタック数
thread_local message* message::m_callPointStackHead = nullptr;//コールポイントスタックの先頭
thread_local message::buff_t* message::m_messageBuff = nullptr;//メッセージバッファ
thread_local std::size_t message::m_messageBuffSize = 0;//メッセージバッファサイズ
thread_local std::size_t message::m_messageBuffUsed = 0;//メッセージバッファ使用量
//---------------------------------------
//メッセージ初期化（一回限り）
void message::initializeOnce()
{
	//初期化済みチェック
	if (m_isInitialized)
		return;
	//静的変数を初期化
	m_isInitialized = true;
	resetAttrG();
	resetLogLevelAllG();
	resetNoticeLevelAllG();
}
//---------------------------------------
//メッセージ用変数
static message s_messageForInitialize;//初期化処理実行のためのインスタンス
```

【コールポイントクラス定義】

```
#include <stdint.h>//uintptr_t 用
#include <chrono>//C++11 時間
//※CThreadID（スレッド名 CRC 拡張版）を使用
//※共通関数 getFileName を使用

//---------------------------------------
//コールポイント用定数
enum controlEnum : unsigned char//操作設定
{
	recProfile = 0x00,//プロファイル記録
	noProfile = 0x01,//プロファイル不要
};
//---------------------------------------
//コールポイントクラス
class callPoint : public message
{
```

```
public:
	//型
	typedef uintptr_t key_t;//キー型
	typedef std::chrono::high_resolution_clock::time_point clock_t;//クロック型
	typedef double duration_t;//処理時間型
	typedef unsigned char control_t;//操作設定型
public:
	//アクセッサ
	key_t getKey() const{ return m_key; }//キー
	clock_t getBegintime() const{ return m_beginTime; }//処理開始時間
public:
	//メソッド
	//経過時間取得
	duration_t getElapsedTime() const
	{
		const clock_t end_time = std::chrono::high_resolution_clock::now();
		return static_cast<double>(
			std::chrono::duration_cast<std::chrono::microseconds>(end_time - m_beginTime).count()) / 1000000.;
	}
public:
	//コンストラクタ
	callPoint(const level::value_t level_, const category::value_t category_, const char* name,
			const controlEnum control, const key_t key, const char* src_file_name, const char* func_name) :
		message(level_, category_, name, src_file_name, func_name),
		m_key(key),
		m_control(control),
		m_beginTime(std::chrono::high_resolution_clock::now())
	{
		pushCallPoint();//コールポイントをプッシュ
	}
	//デストラクタ
	~callPoint()
	{
		if (!(m_control & noProfile))
		{
			//スレッド名取得
			CThreadID thread_id;

			//パフォーマンス記録
			CProfiler profiler;
			profiler.addProfile(m_key, m_name, thread_id, getFileName(m_srcFileName), m_funcName,
				getElapsedTime());
		}
		//popCallPoint();//コールポイントをポップ　※ポップは親クラスが行うのでここでは不要
	}
private:
	//フィールド
	const key_t m_key;//キー
	const control_t m_control;//操作設定
	const clock_t m_beginTime;//処理開始時間
};
template<level::value_t L>
class callPointAsLevel : public callPoint
{
public:
	//定数
	static const level::value_t LEVEL = L;//レベル
public:
	//コンストラクタ
	callPointAsLevel(const category::value_t category_, const char* name, const controlEnum control, const key_t key,
		const char* src_file_name, const char* func_name) :
		callPoint(LEVEL, category_, name, control, key, src_file_name, func_name)
	{}
	//デストラクタ
	~callPointAsLevel()
```

```
    {}
};
#define declare_callPointClass(level_) using callPointAs##level_ = callPointAsLevel<as##level_>
declare_callPointClass(Normal);//通常メッセージ
declare_callPointClass(Verbose);//冗長メッセージ
declare_callPointClass(Detail);//詳細メッセージ
declare_callPointClass(Important);//重要メッセージ
declare_callPointClass(Warning);//警告メッセージ
declare_callPointClass(Critical);//重大（問題）メッセージ
declare_callPointClass(Absolute);//絶対（必須）メッセージ
#define __CPARGS() reinterpret_cast<callPoint::key_t>(GET_CONCATENATED_SOURCE_FILE_NAME(), ¥
                                                      GET_CONCATENATED_SOURCE_FILE_NAME(), GET_FUNC_NAME()
```

【関数定義】

```
//---------------------------------------
//関数
//現在の表示属性を取得
message::attr_t getAttr(){ return message::getAttrG(); }
//現在の表示属性を変更
//void setAttr(const message::attr_t attr){ message::setAttrG(attr); }
message::attr_t addAttr(const message::attr_t attr){ return message::addAttrG(attr); }
message::attr_t delAttr(const message::attr_t attr){ return message::delAttrG(attr); }
//現在の表示属性をリセット
void resetAttr(){ message::resetAttrG(); }
//現在のログレベルを取得
level::value_t getLogLevel(const category::value_t category_)
{
    return message::getLogLevelG(category_);
}
//現在のログレベルを変更
//※指定の値以上のレベルのメッセージのみをログ出力する
level::value_t setLogLevel(const level::value_t level_, const category::value_t category_)
{
    return message::setLogLevelG(level_, category_);
}
//現在のログレベルをリセット
void resetLogLevel(const category::value_t category_)
{
    message::resetLogLevelG(category_);
}
//現在のログレベルを全てリセット
void resetLogLevelAll()
{
    message::resetLogLevelAllG();
}
//現在の画面通知レベルを取得
level::value_t getNoticeLevel(const category::value_t category_)
{
    return message::getNoticeLevelG(category_);
}
//現在の画面通知レベルを変更
//※指定の値以上のレベルのメッセージのみをログ出力する
level::value_t setNoticeLevel(const level::value_t level_, const category::value_t category_)
{
    return message::setNoticeLevelG(level_, category_);
}
//現在の画面通知レベルをリセット
void resetNoticeLevel(const category::value_t category_)
{
    message::resetNoticeLevelG(category_);
}
//現在の画面通知レベルを全てリセット
void resetNoticeLevelAll()
{
    message::resetNoticeLevelAllG();
```

```
}
//メッセージ出力／ログ出力／画面通知関数：レベルを引数指定
//※vprint***/vlog***/vnotice***
//※va_list を引数にとるバージョン
static int vprint(const level::value_t level_, const category::value_t category_, const char* fmt, va_list args)
{
	return message::vprintG(level_, category_, fmt, args);
}
static int vlog(const level::value_t level_, const category::value_t category_, const char* fmt, va_list args)
{
	return message::vlogG(level_, category_, fmt, args);
}
static int vnotice(const level::value_t level_, const category::value_t category_, const char* fmt, va_list args)
{
	return message::vnoticeG(level_, category_, fmt, args);
}
//メッセージ出力／ログ出力／画面通知関数：レベルを引数指定
//※print***/log***/notice***
//※可変長引数バージョン
static int print(const level::value_t level_, const category::value_t category_, const char* fmt, ...)
{
	va_list args;
	va_start(args, fmt);
	const int ret = message::vprintG(level_, category_, fmt, args);
	va_end(args);
	return ret;
}
static int log(const level::value_t level_, const category::value_t category_, const char* fmt, ...)
{
	va_list args;
	va_start(args, fmt);
	const int ret = message::vlogG(level_, category_, fmt, args);
	va_end(args);
	return ret;
}
static int notice(const level::value_t level_, const category::value_t category_, const char* fmt, ...)
{
	va_list args;
	va_start(args, fmt);
	const int ret = message::vnoticeG(level_, category_, fmt, args);
	va_end(args);
	return ret;
}
//メッセージ出力／ログ出力／画面通知関数定義マクロ
//※vprint***/vlog***/vnotice***
//※va_list を引数にとるバージョン
#define declare_vprintFuncs(level_) ¥
static int vprintAs##level_(const category::value_t category_, const char* fmt, va_list args) ¥
{ ¥
	return message::vprintAs##level_##G(category_, fmt, args); ¥
} ¥
static int vlogAs##level_(const category::value_t category_, const char* fmt, va_list args) ¥
{ ¥
	return message::vlogAs##level_##G(category_, fmt, args); ¥
} ¥
static int vnoticeAs##level_(const category::value_t category_, const char* fmt, va_list args) ¥
{ ¥
	return message::vnoticeAs##level_##G(category_, fmt, args); ¥
}
//メッセージ出力／ログ出力／画面通知関数定義マクロ
//※print***/log***/notice***
//※可変長引数バージョン
#define declare_printFuncs(level_) ¥
static int printAs##level_(const category::value_t category_, const char* fmt, ...) ¥
{ ¥
```

```
	va_list args; ¥
	va_start(args, fmt); ¥
	const int ret = message::vprintAs##level_##G(category_, fmt, args); ¥
	va_end(args); ¥
	return ret; ¥
} ¥
static int logAs##level_(const category::value_t category_, const char* fmt, ...) ¥
{ ¥
	va_list args; ¥
	va_start(args, fmt); ¥
	const int ret = message::vlogAs##level_##G(category_, fmt, args); ¥
	va_end(args); ¥
	return ret; ¥
} ¥
static int noticeAs##level_(const category::value_t category_, const char* fmt, ...) ¥
{ ¥
	va_list args; ¥
	va_start(args, fmt); ¥
	const int ret = message::vnoticeAs##level_##G(category_, fmt, args); ¥
	va_end(args); ¥
	return ret; ¥
}
//メッセージ出力／ログ出力／画面通知関数定義
//※vprint***/vlog***/vnotice***
//※va_list を引数にとるバージョン
declare_vprintFuncs(Normal);//通常メッセージ
declare_vprintFuncs(Verbose);//冗長メッセージ
declare_vprintFuncs(Detail);//詳細メッセージ
declare_vprintFuncs(Important);//重要メッセージ
declare_vprintFuncs(Warning);//警告メッセージ
declare_vprintFuncs(Critical);//重大（問題）メッセージ
declare_vprintFuncs(Absolute);//絶対（必須）メッセージ
//メッセージ出力／ログ出力／画面通知関数定義
//※print***/log***/notice***
//※可変長引数バージョン
declare_printFuncs(Normal);//通常メッセージ
declare_printFuncs(Verbose);//冗長メッセージ
declare_printFuncs(Detail);//詳細メッセージ
declare_printFuncs(Important);//重要メッセージ
declare_printFuncs(Warning);//警告メッセージ
declare_printFuncs(Critical);//重大（問題）メッセージ
declare_printFuncs(Absolute);//絶対（必須）メッセージ
//コールポイントスタック表示
void printCPStack(const level::value_t level_, category::value_t category_, const char* name,
                                                    const message::control_t control = simplePrint)
{
	message::printCPStackG(level_, category_, name, control);
}
#define declare_printCPFunc(level_) ¥
static void printCPStackAs##level_(const category::value_t category_, const char* name, const message::control_t control
= simplePrint) ¥
{ ¥
	message::printCPStackAs##level_##G(category_, name, control); ¥
}
declare_printCPFunc(Normal);//通常メッセージ
declare_printCPFunc(Verbose);//冗長メッセージ
declare_printCPFunc(Detail);//詳細メッセージ
declare_printCPFunc(Important);//重要メッセージ
declare_printCPFunc(Warning);//警告メッセージ
declare_printCPFunc(Critical);//重大（問題）メッセージ
declare_printCPFunc(Absolute);//絶対（必須）メッセージ
```

【プロファイラクラス定義】

```
#include <stdint.h>//uintptr_t 用
#include <unordered_map>//C++11 ハッシュテーブル
```

```
#include <map>//STL map
#include <string>//std::string
//※CThreadID（スレッド名 CRC 拡張版）を使用
//※CSpinLock を使用
//※CStackAllocatorWithBuff を使用
//※CStackAllocAdp を使用
//※CTempPolyStackAllocator を使用

//-------------------------------------
//プロファイラクラス
class CProfiler
{
public:
	//定数
	static const std::size_t BUFF_SIZE = 64 * 1024;//記録用のバッファ
public:
	//型
	typedef CStackAllocatorWithBuff<BUFF_SIZE> buffer_t;//バッファ型
	typedef uintptr_t key_t;//キー型
	//プロファイル
	struct PROFILE
	{
		key_t m_key;//キー
		std::string m_name;//処理名
		std::string m_srcFileName;//ソースファイル名
		std::string m_funcName;//関数名
		struct TIME
		{
			double m_sum;//処理時間の合計
			float m_max;//最大の処理時間
			float m_min;//最小の処理時間
			int m_count;//計測回数
			float average() const { return m_count == 0 ? 0 :
				static_cast<float>(m_sum / static_cast<double>(m_count)); }
			//計測時間集計
			void add(const double time)
			{
				++m_count;
				m_sum += time;
				const float time_f = static_cast<float>(time);
				if (m_max == 0.f || m_max < time_f)
					m_max = time_f;
				if (m_min == 0.f || m_min > time_f)
					m_min = time_f;
			}
			//計測時間集計
			void add(const TIME& src)
			{
				m_count += src.m_count;
				m_sum += src.m_sum;
				if (m_max == 0.f || m_max < src.m_max)
					m_max = src.m_max;
				if (m_min == 0.f || m_min > src.m_min)
					m_min = src.m_min;
			}
			//計測時間リセット
			void reset()
			{
				m_sum = 0.;
				m_max = 0.f;
				m_min = 0.f;
				m_count = 0;
			}
			//コピー演算子
			TIME& operator=(TIME& rhs)
```

```
		{
			memcpy(this, &rhs, sizeof(*this));
			return *this;
		}
		//ムーブコンストラクタ
		TIME(const TIME&& src):
			m_sum(src.m_sum),
			m_max(src.m_max),
			m_min(src.m_min),
			m_count(src.m_count)
		{}
		//デフォルトコンストラクタ
		TIME() :
			m_sum(0.),
			m_max(0.f),
			m_min(0.f),
			m_count(0)
		{}
	};
	//計測時間情報
	struct MEASURE
	{
		TIME m_total;//処理全体の集計
		TIME m_frame;//フレーム内の集計
		bool m_inCountOnFrame;//フレームの集計があったか？
		int m_frameCount;//計測フレーム数
		float m_time;//（前回計測時の）処理時間
		//計測時間を加算
		void add(const double time)
		{
			m_time = static_cast<float>(time);
			m_frame.add(time);
			m_inCountOnFrame = true;
		}
		//フレーム集計
		void sumOnFrame()
		{
			if (m_inCountOnFrame)
			{
				m_inCountOnFrame = false;
				m_total.add(m_frame);
				m_frame.reset();
				++m_frameCount;
			}
		}
		//ムーブコンストラクタ
		MEASURE(MEASURE&& src) :
			m_total(std::move(src.m_total)),
			m_frame(std::move(src.m_frame)),
			m_inCountOnFrame(src.m_inCountOnFrame),
			m_frameCount(src.m_frameCount),
			m_time(src.m_time)
		{}
		//コンストラクタ
		MEASURE() :
			m_total(),
			m_frame(),
			m_inCountOnFrame(false),
			m_frameCount(0),
			m_time(0.f)
		{}
	};
	MEASURE m_measure;//計測情報
	//スレッド情報
	struct THREAD_INFO
```

```
	{
		crc32_t m_nameCrc;//スレッド名の CRC
		std::string m_name;//スレッド名
		MEASURE m_measure;//計測情報
		//計測時間を加算
		void add(const double time)
		{
			m_measure.add(time);
		}
		//フレーム集計
		void sumOnFrame()
		{
			m_measure.sumOnFrame();
		}
		//ムーブコンストラクタ
		THREAD_INFO(THREAD_INFO&& src) :
			m_nameCrc(src.m_nameCrc),
			m_name(std::move(src.m_name)),
			m_measure(std::move(src.m_measure))
		{}
		//コンストラクタ
		THREAD_INFO(const crc32_t name_crc, const char* name) :
			m_nameCrc(name_crc),
			m_name(name),
			m_measure()
		{}
	};
	typedef std::map<crc32_t, THREAD_INFO> threadList_t;//マップ型
	threadList_t m_threadList;
	//計測時間を加算
	void add(const double time, const CThreadID& thread_id)
	{
		m_measure.add(time);

		//スレッドごとの集計
		const crc32_t thread_name_crc = thread_id.getNameCrc();
		const char* thread_name = thread_id.getName();
		THREAD_INFO* thread_info = nullptr;
		{
			threadList_t::iterator ite = m_threadList.find(thread_name_crc);
			if (ite != m_threadList.end())
				thread_info = &ite->second;
			else
			{
				m_threadList.emplace(thread_name_crc,
					std::move(THREAD_INFO(thread_name_crc, thread_name)));
				ite = m_threadList.find(thread_name_crc);
				if (ite != m_threadList.end())
					thread_info = &ite->second;
			}
		}
		if (thread_info)
		{
			thread_info->add(time);
		}
	}
	//フレーム集計
	void sumOnFrame()
	{
		if (m_measure.m_inCountOnFrame)
		{
			m_measure.sumOnFrame();

			for (auto& ite : m_threadList)
			{
```

```
                        ite.second.sumOnFrame();
                    }
                }
            }
            //ムーブコンストラクタ
            //※要素追加の際の無駄なオブジェクト生成と破棄を防ぐ
            //  (std::string 向けの対処：std::move()を使って受け渡し)
            PROFILE(PROFILE&& src) :
                m_key(src.m_key),
                m_name(std::move(src.m_name)),
                m_srcFileName(std::move(src.m_srcFileName)),
                m_funcName(std::move(src.m_funcName)),
                m_measure(std::move(src.m_measure)),
                m_threadList(std::move(src.m_threadList))
            {}
            //コンストラクタ
            PROFILE(const key_t key, const char* name, const char* src_file_name, const char* func_name) :
                m_key(key),
                m_name(name),
                m_srcFileName(src_file_name),
                m_funcName(func_name),
                m_measure()
            {}
    };
    typedef std::unordered_map<key_t, PROFILE> table_t;//ハッシュテーブル型
    typedef table_t::iterator iterator;//イテレータ型
    typedef table_t::const_iterator const_iterator;//イテレータ型
public:
    //アクセッサ
    std::size_t getBuffTotal()const{ return m_buff.getTotal(); }
    std::size_t getBuffUsed()const{ return m_buff.getUsed(); }
    std::size_t getBuffRemain()const{ return m_buff.getRemain(); }
public:
    //メソッド
    //プロファイル追加／集計
    void addProfile(const key_t key, const char* name, const CThreadID& thread_id, const char* src_file_name,
                                                        const char* func_name, const double time)
    {
        m_lock.lock();
        CTempPolyStackAllocator allocator(m_buff);
        PROFILE* profile = nullptr;
        {
            table_t::iterator ite = m_table->find(key);
            if (ite != m_table->end())
                profile = &ite->second;
            else
            {
                m_table->emplace(key, std::move(PROFILE(key, name, src_file_name, func_name)));
                ite = m_table->find(key);
                if (ite != m_table->end())
                    profile = &ite->second;
            }
        }
        if (profile)
        {
            profile->add(time, thread_id);
        }
        m_lock.unlock();
    }
    //フレーム集計
    void sumOnFrame()
    {
        m_lock.lock();
        for (auto& ite : *m_table)
        {
```

```
                PROFILE* profile = &ite.second;
                profile->sumOnFrame();
            }
            m_lock.unlock();
        }
        //イテレータ取得
        std::size_t size() const { return m_table->size(); }
        //table_t::iterator begin() { return m_table->begin(); }
        //table_t::iterator end() { return m_table->end(); }
        table_t::const_iterator begin() const { return m_table->begin(); }
        table_t::const_iterator end() const { return m_table->end(); }
        table_t::const_iterator cbegin() const { return m_table->cbegin(); }
        table_t::const_iterator cend() const { return m_table->cend(); }
        //テーブル生成
        void createTable()
        {
            m_lock.lock();
            if (!m_table)
            {
                CTempPolyStackAllocator allocator(m_buff);
                m_table = new table_t();
                m_table->reserve(1024);
            }
            m_lock.unlock();
        }
        //テーブル破棄
        void destroyTable()
        {
            m_lock.lock();
            if (m_table)
            {
                CTempPolyStackAllocator allocator(m_buff);
                delete m_table;
                m_table = nullptr;
                m_buff.clearN();
            }
            m_lock.unlock();
        }
        //テーブルをリセット
        void resetTable()
        {
            destroyTable();
            createTable();
        }
    public:
        //コンストラクタ
        CProfiler()
        {
            createTable();
        }
        //デストラクタ
        ~CProfiler()
        {
            //シングルトンなのでデストラクタではなにもしない
        }
    private:
        //フィールド
        static table_t* m_table;//ハッシュテーブル
        static buffer_t m_buff;//記録用バッファ
        static CSpinLock m_lock;//ロック
    };
    //-----------------------------------
    //プロファイラの静的変数をインスタンス化
    CProfiler::table_t* CProfiler::m_table;//ハッシュテーブル
    CProfiler::buffer_t CProfiler::m_buff;//記録用バッファ
```

```
CSpinLock CProfiler::m_lock;//ロック
```

▼ システムの依存関係

このサンプルプログラムは、別紙に掲載する他のサンプルプログラムを必要とする。
下記のプログラムである。

・ スレッド ID クラス
  - 別紙の「効率化と安全性のためのロック制御」に掲載。
  - ただし、本サンプルでは、さらにスレッド名の CRC を算出して保持する処理を追加。
  - CRC 算出処理については別紙の「効果的なテンプレートテクニック」に掲載。

・ スピンロッククラス
  - 別紙の「効率化と安全性のためのロック制御」に掲載。

・ プールアロケータクラス
  - 別紙の「効率化と安全性のためのロック制御」に掲載。
  - 「様々なメモリ管理手法と共通アロケータインターフェース」にも掲載しているが、こちらはロックの仕組みを外している。本書のシステムはロックが必須。

・ スタックアロケータクラス／スタックアロケータアダプタークラス
  - 別紙の「様々なメモリ管理手法と共通アロケータインターフェース」に掲載。

・ 多態アロケータクラス／グローバル多態アロケータ
  - 別紙の「様々なメモリ管理手法と共通アロケータインターフェース」に掲載。

■■以上■■

■ 索引

索引項目が見つかりません。

効果的なデバッグログとアサーション

以　上