



GASHA

Gakimaru's researched and standard library for C++

Copyright (c) 2014 Itaqaki Mamoru

Released under the [MIT license](#). <https://github.com/qakimaru/qasha> examples/blob/master/LICENSE

サンプルプログラム用リポジトリ

このリポジトリは、「GASHA」のサンプルプログラム用です。

本書にて、ライブラリ「GASHA」を解説します。

「GASHA」とは？

個人制作による、オープンソースのC++用基本ライブラリです。

主にゲーム開発での利用を想定し、暗黙的なメモリ確保／解放を行わない点が特徴の一つです。

高速性もしくは生産性を追求し、品質と開発効率の向上を目的としています。

名前の由来

名前の由来は「がしゃどくろ」から。読み方は「ガシャ」。

その「骨格」的な意味合い(?)がライブラリを象徴し、自身のハンドル名「Gakimaru」の頭文字を重ねて命名しました。

ライブラリのソースコードにはMITライセンスを適用しています。

ご自由にお使い下さい。

C++11の新しい構文、ライブラリもふんだんに取り入れて作成していますので、参考程度にでもご活用頂ければ幸いです。

バグや改善案などのご連絡は大いに歓迎いたします。

「GASHA」の主な特徴

C++11を基盤に、新規に作成したライブラリです。

標準化の進んだC++11のライブラリを利用することにより、プラットフォーム／コンパイラ依存のコードが少ない点が大きな特徴です。

ゲーム開発での利用を想定して開発した基本ライブラリです。

ゲーム開発向けの基本的なプログラミング要素を主体に開発しています。

メモリ安全なコンテナとアルゴリズム(暗黙的なメモリ確保・解放を行い)、多様なメモリアロケータ、利便性の高いデバッグ機能、一般的なスレッドセーフ設計などにより、生産性の向上と品質向上、パフォーマンスの向上に寄与します。

テンプレートクラス／関数を多用し、パフォーマンスを優先した設計です。

C++の柔軟なテンプレートを活用し、C++の「ゼロオーバーヘッドの原則」に倣って、不要な処理がパフォーマンスに影響しないように設計しています。

また、メタプログラミングの対応も強化し、コンパイル時のCRC計算や素数計算などに対応し、実行時のパフォーマンス向上に寄与します

- 例えば、各種コンテナクラスはマルチスレッド制御のためのインターフェースを備えていますが、マルチスレッド制御が不要なら、コンパイル時にそのコードが消滅するように構成しています。

マルチプラットフォームで共通利用可能なソースコードを強化します。

コンパイル時にコンパイラの種類とバージョンを自動判別し、コンパイラ固有仕様をC++11仕様に偽装するなどして、ソースコードレベルでの共通性を高めています。

- 例えば、alignof(), alignas(), thread_localなどのC++11仕様を、非対応のコンパイラでも使用できるようにしています。
- 現状では、Visual C++ 12.0(Visual Studio 2013)のWin32, x64、および、GCC4.6に対応しています。

プロジェクト固有のライブラリのカスタマイズに対応しています。

より柔軟に多様なプラットフォーム／プロジェクトでライブラリを共通利用できるように、ライブラリの挙動をカスタマイズし易いように構成しています。

プロジェクトごとに使用するライブラリの機能や、ワークバッファのサイズなどを調整できます。

複数プロジェクトでライブラリを共通利用することを想定したリポジトリ構成です。

Gitのサブモジュール機能を活用し、多数のプロジェクトでライブラリを共通利用する環境を想定してリポジトリを構成しています。プロジェクトごとに、ライブラリの動作要件をカスタマイズし易いように、幾つかのリポジトリを組み合わせて構成しています。

他のライブラリと競合しないように構成しています。

ネームスペースは `gasha`、マクロ名は `GASHA_***`、ヘッダーファイルは `gasha/***.h` と徹底しており、他のライブラリと競合しにくいようにしています。

個人的な学習・調査・研究もライブラリの目的の一つです。

個人的に学習・調査・研究した結果をライブラリに反映させています。

そのため、結果的に実用性がないものも含んでいます。(例:幾つかの非効率なソートアルゴリズム)

今後のライブラリの追加実装は、関心と余力に応じて、気まぐれに行ってきます。

サンプルプログラム実行手順

【クイックスタート】

1. `git` で `gasha_examples` リポジトリのクローンを作成

```
$ git clone https://github.com/gakimaru/gasha_examples.git
```

2. サブモジュールのアップデート

```
$ cd gasha_examples
$ git submodule update --init
```

これにより、サブモジュールとして配置された三つのリポジトリのクローンが更新されます。

```
| sub/gasha ライブラリ本体
```

`sub/gasha_settings` プロジェクト固有のライブラリ挙動カスタマイズ用
`sub/gasha_src` ライブラリソース

3. ライブラリのビルド

【Visual C++の場合】

`proj/gasha_examples.sln` を開き、[ビルド]→[パッチビルド]メニューを実行し、すべての(もしくは任意の)プロジェクト／構成／プラットフォームを選んでビルドを実行して下さい。
各サンプルプログラム個別のソリューションファイルもあります。

【Unix系環境+GCCの場合】

`proj/mk_all.sh` を実行して下さい。

```
$ cd proj
$ ./mk_all.sh all
```

リビルドする場合は、一度 `./mk_all.sh clean` を実行してから `./mk_all.sh all` を実行して下さい。なお、`Makefile` 自体は、各サンプルプログラム個別に用意しています。

4. サンプルプログラムの実行

【Visual C++の場合】

`exe/vc/` 以下のディレクトリから、`(サンプル名).exe` を実行することで、サンプルプログラムを実行可能です。
`exe/vc/` 以下のディレクトリには、サンプルプログラムをまとめて実行し、結果をログファイルに記録するためのバッチファイルも用意しています。
また、Visual Studio 上から、「スタートアッププロジェクト」を設定すれば、デバッグ実行することも可能です。

【Unix系環境+GCCの場合】

`exe/gcc/` 以下のディレクトリから、`(サンプル名)` の実行ファイルを実行することで、サンプルプログラムを実行可能です。
`exe/gcc/` 以下のディレクトリには、サンプルプログラムをまとめて実行し、結果をログファイルに記録するためのシェルスクリプトも用意しています。

- 【注意】: 各サンプルプログラムは、開発過程で機能やパフォーマンスの評価に使用したものです。
網羅的な機能チェックで処理が長いものもあり、難解かもしれません。
ライブラリをご理解いただく際には、ソースコード(ヘッダーファイル)と関連資料をご確認頂ければ幸いです。

対象コンパイラ

ライブラリの利用、および、サンプルプログラムの実行には、C++11仕様に対応したコンパイラが必要です。

【推奨コンパイラ】

- Visual C++ 12.0 (Visual Studio 2013) 以降
- GCC 4.6 以降

開発には下記のコンパイラを用いています。

- Visual C++ 12.0 (Visual Studio 2013) ※Win32, x64 プラットフォームで開発
- GCC 4.8.2 (32bit版Cygwin環境)

今後 Clang への対応を検討しています。

構成リポジトリ

GASHAは、下記のように複数のリポジトリで構成しています。

- `gasha` ライブラリ本体用リポジトリ
- `gasha_settings` プロジェクト固有のライブラリ挙動カスタマイズ用リポジトリ
- `gasha_src` ライブラリソース用リポジトリ

- [gasha_examples](#) サンプルプログラム用リポジトリ
- [gasha_proj](#) ライブラリビルド用リポジトリ

Gitの「サブモジュール」の機能を活用し、プロジェクトのソースコード管理に対象バージョンのライブラリを組み込むことができるよう構成しています。
また、プロジェクトごとにライブラリの挙動をカスタマイズすることや、プロジェクトに応じてライブラリのソースコードを隠蔽することに対応可能な構成です。

基本ディレクトリ構成とサブモジュール構成

サンプルプログラム＆ライブラリビルド用リポジトリ

```
[gasha_examples]      ... サンプルプログラム用リポジトリ
|
|-[exe]            ... 実行ファイル用
|-[proj]           ... 各サンプルプログラムのビルドプロジェクト用
|-[src]            ... 各サンプルプログラムのソースファイル用
`-[sub]            ... サブモジュール用
|
|-[gasha]          ... ライブラリ本体用リポジトリ(https://github.com/gakimaru/gasha)
|-[gasha_settings] ... プロジェクト固有のライブラリ挙動カスタマイズ用(https://github.com/gakimaru/gasha\_settings)
`-[gasha_src]       ... ライブラリソース用リポジトリ(https://github.com/gakimaru/gasha\_src)
```

ライブラリ単体リビルト用リポジトリ

```
[gasha_proj]        ... ライブラリビルド用リポジトリ
|
|-[proj]           ... ライブラリビルドプロジェクト用
`-[sub]            ... サブモジュール用
|
|-[gasha]          ... ライブラリ本体用リポジトリ(https://github.com/gakimaru/gasha)
|-[gasha_settings] ... プロジェクト固有のライブラリ挙動カスタマイズ用(https://github.com/gakimaru/gasha\_settings)
`-[gasha_src]       ... ライブラリソース用リポジトリ(https://github.com/gakimaru/gasha\_src)
```

ライブラリ概要

<注釈>

- 以降の説明で「テンプレート関数」もしくは「テンプレートクラス」と表記しているものは、ライブラリファイルをリンクせずに使用可能です。
- 以降の説明で「`※T`」の注釈記号が付いているものは、特殊なテンプレートクラスです。
コンパイル・リンク時間の短縮、および、クラス修正時の影響範囲の抑制(再コンパイル対象の抑制)を目的として、ソースファイルを「宣言部」(`.h`)、「インライン関数／テンプレート関数定義部」(`.inl`)、「関数定義部」(`.cpp.h`) の三つに分けています。
それぞれ適切な場所でインクルードし、テンプレートクラスの明示的なインスタンス化を行う事で、ビルトを効率化できます。
なお、面倒なら全部まとめてインクルードすれば、明示的なインスタンス化をせずに使用可能です。
- 以降の説明で「`※TLS`」の注釈記号が付いているものは、TLS領域を使用します。
この機能が有効だと、スレッド生成時にスレッドに割り当てられるメモリが増えます。
一つの機能につきTLS領域1～4個(4/8バイト×1～4個)ほど使用しています。

環境系

- [gasha/build_settings/build_settings.h](#) ビルド設定(統括) ※1
- [gasha/build_settings/project_first_settings.h](#) プロジェクト固有カスタマイズ設定(先行設定) ※2 ※4
- [gasha/build_settings/project_last_settings.h](#) プロジェクト固有カスタマイズ設定(最終設定) ※2 ※5
- [gasha/build_settings/build_settings_diag.h](#) ビルド設定診断 ※3

<説明>

- コンパイラの種類やバージョンを判別し、必要に応じて `nullptr`, `override`, `alignas`, `thread_local` などのC++11仕様に合わせた処理の独自実装版を有効化し、コード互換性の向上に寄与します。
- 同様に、`_FUNCTION_`, `_PRETTY_FUNCTION_` などの標準的なマクロも各コンパイラで共通利用可能にします。
- プロジェクト固有のカスタマイズが、対象プラットフォーム上で動作するかチェックする関数 `buildSettingsDiagnosticTest()` を用意しています。
- 現状、x86/x64系CPU／Windows + VC++12.0(VS2013)／Cygwin + GCC4.8.2 の環境でしか動作確認しておらず、他の環境に適合する状態ではありませんが、対象環境を拡張し易いように構成しています。

<注釈>

- ※1: `build_settings.h` が幾つかの設定ファイルをインクルードしてまとめています。通常、このファイルを強制インクルードファイルに含めて利用します。プリコンパイル済みヘッダーに含めることも推奨します。
- ※2(1): ユーザー側(ライブラリの利用者側)で、ライブラリに対するプロジェクト固有のカスタマイズを行います。
例えば「SSE命令をどこまで使用するか?」、「CRC-32の生成多項式はCRC-32Cを採用するか?」、「コンテナクラスの明示的インスタンス化でビルド効率を最適化するか?」といった設定を行います。
- ※2(2): この設定ファイルはライブラリ本体とは別のリポジトリで管理しており、ユーザー側に更新権限があるものとして位置付けています。
- ※3(1): 対象マシン上で、カスタマイズ要件通りの動作が可能か診断する関数 `buildSettingsDiagnosticTest()` を用意しています。
例えば、SSE4.2命令の使用が設定されていれば、そのマシン上でSSE4.2命令が使用可能か判定します。
その代わり、そうしたCPU依存の命令の使用が設定されているなら、各処理系では(例えばCRC32計算など)、対象マシンの対応状態を確認せずに命令を使用し、処理を効率化します。
- ※3(2): `buildSettingsDiagnosticTest()` は、対象プラットフォームやC++11対応状態なども合わせて表示します。
ライブラリビルド時の環境と、現在のプロジェクトの環境をそれぞれ確認できます。
- ※3(3): ビルド設定関係のファイル一式と下記ユーティリティの `cpuid` のソースだけ抜き出せば、`buildSettingsDiagnosticTest()` を実行できます。
これを利用すれば、本ライブラリが動作しない環境であっても、コンパイラの状態や実行環境の確認に使うことができます。
- ※4: ライブラリのネームスペースを独自にカスタマイズできます。
- ※5: ライブラリの挙動を設定できます。コンパイラスイッチとしてのマクロを有効化／無効化することで設定します。
ユーザー側で任意にカスタマイズできるものです。

<資料>

- [本当にちょっとしたプログラミング Tips.pdf](#)
[コーディングに関するTips]-[#defineマクロの活用]

ユーティリティ系

- `gasha/utility.h` [テンプレート]汎用ユーティリティ ※1
- `gasha/type_traits.h` [テンプレート関数]型特性ユーティリティ ※2
- `gasha/limits.h` [テンプレートクラス]限界値ユーティリティ ※3
- `gasha/chrono.h` 時間処理ユーティリティ ※4
- `gasha/cpuid.h` CPU情報ユーティリティ ※5

<説明>

- 汎用的なユーティリティ関数／クラスです。
メタプログラミングにも対応し、コンパイル時の情報取得に活用できます。

<注釈>

- ※1: 任意の数の引数を指定可能な `min()`, `max()` 関数や、値を交換するための `swapValues()` 関数などを扱います。
- ※2: 配列の要素数や次元数を取得するための `extentof()`, `rankof()` などを扱います。
標準ライブラリの `<type_traits>` と異なり、変数を引数にとって結果を返します。
- ※3: 各型の限界値を `numeric_limits` で取得できます。
`std::numeric_limits` を継承し、指定の型の限界値を、静的メンバ `MIN`(最小値), `MAX`(最大値) で取得できます。
(`std::numeric_limits` は、`constexpr` に対応したコンパイラなら、`min()`, `max()` 関数を静的に扱えます。)
また、「値全域を扱える符号付きの型」`contained_signed_type`、「値の範囲型」`range_type`, `signed_range_type` なども扱えます。

前者は、対象が符号なし型なら一段上の符号付き型を返し(例: `unsigned char` なら `short`、`char` なら `char`)、
後者は、同精度の符号無し型および一段上の符号付き型を返します(例: `char` なら `unsigned char`, `short`)。

- ※4: 处理時間計測を簡単にするためのクラス `elapsedTime` などを扱います。
プログラムが起動してからの経過時間を取得する関数 `nowElapsedTime` もあります。
- ※5: x86系CPUの `cpuid` 命令を実行します。
VC++の `_cpuid()` 関数に合わせて、GCC用でも使用できるようにしています。

<資料>

- [効果的なテンプレートテクニック.pdf](#)
[メタプログラミング]

算術系: 基本

- `gasha/basic_math.h` [テンプレート関数含む] 基本算術 ※1
- `gasha/fast_math.h` [テンプレートクラス／関数含む] 高速算術 ※2

<説明>

- 基本的な算術処理と、その高速化版を実装しています。
- メタプログラミングに対応し、コンパイル時に計算結果を算出することができるものも多数あります。
- 高速化は常に効果があるものではなく、ハードウェアやコンパイラおよびコンパイラー依存します。

<注釈>

- ※1(1): べき乗、素数判定、素数計算、ビット数／MSB／ LSB計算といった基本的な算術系処理を扱います。
- ※1(2): 高速演算と同じインターフェースを実装することを目的に、`std::sqrt` を透過的に呼び出す平方根 `sqr` を実装しています。
- ※1(3): 基本的なベクトル演算関数も用意しています。テンプレートで n次元ベクトルのノルム、合成、差分、距離、正規化、スカラ倍、内積、外積を計算できます。
- ※1(4): 基本的な行列演算関数も用意しています。テンプレートで n×m行列の加算、減算、乗算を計算できます。
- ※2(1): 高速な四則演算、平方根、ベクトル演算を用意しています。
演算子による演算や関数呼び出しの際に、値を `fastA` (fast arithmetic) クラスでラップすることで、高速演算が適用されるというインターフェースです。
- ※2(2): 必ず効果があるわけではないので、使用の際は、プラットフォームやコンパイラに合わせた十分な検証が必要です。
- ※2(3): 高速処理の内容は、例えば、SSE命令有効時には SSE命令を使用し、無効時は通常計算を行うといったものです。
SSE命令に限らず、他のプラットフォームに合わせた高速演算を拡張できる構造です。
- ※2(4): プラットフォーム固有の高速演算命令 (SSE命令など) を隠蔽しているので、ソースコードレベルで互換性のある高速演算として利用できます。
- ※2(5): 例えば、SSE命令有効化状態で `float` 型に `fastA` を適用すると、内部で `_m128` 型で値を保持します。
演算は高速になりますが、情報量が増えることには注意が必要です。
- ※2(6): 【注意】高速演算は必ず効果のあるものではなく、コンパイラの最適化に任せた方が良い結果になることが多いので、コンパイル・実行環境に合わせて、よく検証した上で使用する必要があります。
コンパイル時にインライン展開を有効にしないと、適切な評価ができない点も注意が必要です。
- ※2(7): ベクトル演算と行列演算は、`fastA` を明示的に適用せずとも、SSE命令有効時、かつ、適合する型 (`float[3], float[4]` など) の場合は、SSE命令を使用します。

<資料>

- [効果的なテンプレートテクニック.pdf](#)
[メタプログラミング]
- [プログラム最適化Tips.pdf](#)
[ハードウェアに基づく最適化]-[SIMD演算の活用]
[ハードウェアに基づく最適化]-[CPU命令パイプラインに関する最適化]

算術系: CRC計算

- `gasha/crc32.h` CRC32計算

<説明>

- CRC計算を扱います。現状、32ビットのみに対応しています。
- C++11の `constexpr`, `ユーザー定義リテラル` による、メタプログラミング版(コンパイル時計算)に対応しています。
GCCは、`constexpr` が4.6から、`ユーザー定義リテラル` が4.7から対応しています。
Visual C++ 12.0(VS2013)はどちらも未対応ですが、`constexpr` は次期バージョンで対応予定です。
- ランタイム時のCRC計算は、設定に応じて、高速なSSE命令に対応します。
ただし、SSE命令使用時の多項式はCRC-32Cに限定します。
SSE命令を使用しないように指定することもできます。

<資料>

- **効果的なテンプレートテクニック.pdf**
[メタプログラミング]-[constexprの活用: CRC値の算出]

文字列系

- `gasha/fast_string.h` 高速文字列処理 ※1

<説明>

- 基本的な文字列処理の高速化版を実装しています。
- 高速化は常に効果があるものではなく、ハードウェアやコンパイラーおよびコンパイラオプションに依存します。

<注釈>

- ※1(1): SSE命令を用いた `strlen`, `strcmp`, `strcmp`, `strchr`, `strrchr`, `strstr`, `srtccpy`, `strncpy` を実装しています。
- ※1(1): 【注意】関数によっては、対象とする文字列の長さなども関係し、コンパイラーの標準関数の方が効率的です。
対象プラットフォームとコンパイラー(+最適化オプション)で検証の上、効果のある関数のみ有効化して使用する必要があります。

<資料>

- **プログラム 最適化Tips.pdf**
[ハードウェアに基づく最適化]-[SIMD演算の活用]
[ハードウェアに基づく最適化]-[CPU命令パイプラインに関する最適化]

アルゴリズム系: ソート

【交換ソート】

- `gasha/bubble_sort.h` [テンプレート関数]バブルソート ※1
- `gasha/shaker_sort.h` [テンプレート関数]シェーカーソート ※1
- `gasha/odd_even_sort.h` [テンプレート関数]奇遇転置ソート ※1
- `gasha/shear_sort.h` [テンプレート関数]シェアソート
- `gasha/comb_sort.h` [テンプレート関数]コムソート ※R
- `gasha/gnome_sort.h` [テンプレート関数]ノームソート ※1

【分割交換ソート】

- `gasha/quick_sort.h` [テンプレート関数]クイックソート

【選択ソート】

- `gasha/selection_sort.h` [テンプレート関数]選択ソート ※1
- `gasha/heap_sort.h` [テンプレート関数]ヒープソート

【挿入ソート】

- `gasha/insertion_sort.h` [テンプレート関数]挿入ソート ※1 ※3 ※R
- `gasha/shell_sort.h` [テンプレート関数]シェルソート ※3 ※R

【マージソート】

- `gasha(inplace_merge_sort.h)` [テンプレート関数] インプレースマージソート ※1

【混成ソート】

- `gasha/intro_sort.h` [テンプレート関数] イントロソート ※3 ※R

【分布ソート】

(非比較ソート)

- `gasha/radix_sort.h` [テンプレート関数] 基数ソート ※1 ※2 ※4 ※R

【整列状態確認】

- `gasha/is_ordered.h` [テンプレート関数] 整列状態確認 ※3 ※5

<説明>

- 配列もしくはコンテナのデータをソートします。
- アルゴリズムを使い分けて利用できます。通常は「イントロソート」(大抵は最速)、コードサイズを削減したければ「コムソート」または「シェルソート」(コムソートの方が小さくシェルソートの方が速い)、連結リストなら「挿入ソート」、メモリ度外視で大量データを高速ソートしたければ「基数ソート」(大量データでは最速)といった具合です。

<注釈>

- ※1: 安定ソートです。
- ※2: 外部ソートです。内部で一時的にメモリ確保します。
- ※3: 配列以外のコンテナ(イテレータ)にも対応しています。
- ※4: 整数キー専用です。浮動小数点の大小比較などはできません。
- ※5: 順序が逆転している要素数を計上する関数も用意しています。
- ※R:(Recommended) 推奨アルゴリズムです。特に処理効率／メモリ効率(小さいコードサイズ)が良いものです。
データ構造やデータ量、処理要件、プログラムサイズの状況等に基づいて、この中から最適なアルゴリズムを選択できます。

<補足: マージソートについて>

重要なアルゴリズムの「マージソート」(外部ソート)に未対応です。

分布ソートのように小さなワークメモリで済まず、多くのデータをコピー可能なバッファが必要になることや、環境や対象データ量に合わせて実装方法を変えるべきアルゴリズムであるといった理由から、対応しませんでした。

しかし、マージソートはかなり高速であり、分散・並列処理との相性も抜群です。超大量データのソートにも使えます。

適度な粒度に分割したものを並列に(他のアルゴリズムを使って)ソートし、それをマージソートするのが特に効率的です。

STLの `std::stable_sort` がマージソートなので、メモリに余裕がある場合は選択肢の一つとして検討して下さい。

<資料>

- [プログラム最適化Tips.pdf](#)
[データ構造とアルゴリズムによる最適化]-[ソートアルゴリズム]

アルゴリズム系:探索

- `gasha/linear_search.h` [テンプレート関数] 線形探索 ※1
- `gasha/binary_search.h` [テンプレート関数] 二分探索 ※1 ※2

<説明>

- 配列もしくはコンテナのデータを探索します。

<注釈>

- ※1: 配列以外のコンテナ(イテレータ)にも対応しています。
- ※2: ソート済み状態を前提とした探索です。重複キーは必ず先頭にマッチします。

<資料>

- [プログラム最適化Tips.pdf](#)
[データ構造とアルゴリズムによる最適化]-[探索アルゴリズム]

マルチスレッド系: 基本

- `gasha/thread_id.h` スレッドID ※1 ※TLS

<説明>

- マルチスレッドの基本情報を扱います。

<注釈>

- ※1: C++11標準の `std::this_thread::id()` によるスレッド識別は遅いので、TLSを利用してスレッドIDを高速に参照可能にします。
- 加えて、スレッドに名前を付けて扱うことができます。デバッグに便利です。

マルチスレッド系: ロック制御

- `gasha/spin_lock.h` スピンロック ※1
- `gasha/lw_spin_lock.h` サイズ軽量スピンロック ※1 ※2
- `gasha/dummy_lock.h` ダミーロック ※1 ※3
- `gasha/lock_guard.h` [テンプレートクラス]スコープロック ※4
- `gasha/unique_lock.h` [テンプレートクラス]単一ロック ※5

<説明>

- スレッド間のロック制御を行います。
- 単純なスピンロックによる実装です。ミューテックスのようにコンテキストスイッチを発生させずに待機するため、短時間のロックであれば、マルチコア環境で最適に動作します。
- シングルコア環境ではブリエンプション(タイムスライスを使い切る)までコンテキストスイッチが発生しないため、CPU占有時間が長くなり、非効率です。長い処理のロックの場合も同様です。
- 対策として、一定回数のスピンでコンテキストスイッチを行う仕組みを組み込んでいます。

<注釈>

- ※1: スコープロックパターン制御(`lock_guard`, `unique_lock`)に対応しています。
- ※2: 通常版が4バイトに対して、軽量版が1バイトです。通常版の方が若干高速です。
- ※3: ロッククラスと同じインターフェースを備えますが、実際にはロック制御しません。
コンテナのロック制御を無効化する時などに使用します。
- ※4: スコープロックによるロック制御を行います。コンストラクタでロックを自動的に取得します。
- ※5: スコープロックによるロック制御を行います。コンストラクタでロックを取得するかどうかを選べます。

<資料>

- [プログラム最適化Tips.pdf](#)
[データ構造とアルゴリズムによる最適化]-[マルチスレッドを最適化するアルゴリズム]
- [マルチスレッドプログラミングの基礎.pdf](#)
[様々な同期手法]-[排他制御:スピンロック]

サンプルプログラムを掲載していますが、ライブラリでは、更に洗練された状態で実装しています。

マルチスレッド系: 共有ロック制御

- `gasha/shared_spin_lock.h` 共有スピンロック ※1
- `gasha/simple_shared_spin_lock.h` 単純共有スピンロック ※1 ※2
- `gasha/unshared_spin_lock.h` 非共有スピンロック ※1 ※3
- `gasha/dummy_shared_lock.h` ダミー共有ロック ※1 ※4
- `gasha/shared_lock_guard.h` [テンプレートクラス]スコープ共有ロック ※5
- `gasha/unique_shared_lock.h` [テンプレートクラス]単一共有ロック ※6

<説明>

- 共有ロック(リードロック)と排他ロック(ライトロック)を使い分けたロック制御を行います。

- 共有ロックは複数のスレッドが同時に取得できますが、排他ロックは一つのスレッドしか取得できません。
- 一定回数のスピンでコンテキストスイッチを行います。
- 【注】C++14/17, Boost C++ に合わせて、インターフェースを修正する可能性があります。

<注釈>

- ※1: スコープロックパターン制御(lock_guard, shared_lock_guard, unique_shared_lock)に対応しています。
- ※2: 排他ロックを優先しません。通常版は、排他ロック取得待ちが発生すると、後続の共有ロックが取得待ち状態になります。
単純版は、共有ロックが混み合うと、排他ロックが取得できなくなります。
- ※3: 共有ロッククラスと同じインターフェースを備えますが、実際には全て排他ロックします。
- ※4: 共有ロッククラスと同じインターフェースを備えますが、実際にはロック制御しません。
コンテナのロック制御を無効化する時などに使用します。
- ※5: スコープロックによる共有ロック制御を行います。コンストラクタでロックを自動的に取得します。
- ※6: スコープロックによる共有ロック制御を行います。コンストラクタでロックを取得するかどうかを選べます。

<資料>

- [プログラム最適化Tips.pdf](#)
[データ構造とアルゴリズムによる最適化]-[マルチスレッドを最適化するアルゴリズム]
- [マルチスレッドプログラミングの基礎.pdf](#)
[様々な同期手法]-[排他制御:リード・ライトロック]

マルチスレッド系: 共有データ

- `gasha/shared_stack.h` [テンプレートクラス] 共有スタック ※1 ※T
- `gasha/shared_queue.h` [テンプレートクラス] 共有キュー ※1 ※T

<説明>

- マルチスレッドでデータを共有するためのテンプレートクラスです。
- コンテナとしては機能しません。至ってシンプルなpush/poll/pop/enqueue/dequeueのみに対応します。
- 内部では固定領域のプールアロケータを保持しており、安全なメモリ操作が可能です。

<注釈>

- ※1: 共有にはロック制御を伴いますが、使用する同期オブジェクトをテンプレート引数で指定します。
デフォルトは `spinLock` ですが、`std::mutex` (C++11標準ライブラリ)などに変更することができます。
`dummyLock` を指定すれば、ロック制御を無効化し、シングルスレッドで高速に動作させることができます。

<資料>

- [プログラム最適化Tips.pdf](#)
[データ構造とアルゴリズムによる最適化]-[マルチスレッドを最適化するアルゴリズム]
- サンプルプログラムを掲載していますが、ライブラリでは、更に洗練された状態で実装しています。

マルチスレッド系: ロックフリー共有データ

- `gasha/lf_stack.h` [テンプレート] ロックフリースタック ※1 ※T
- `gasha/lf_queue.h` [テンプレート] ロックフリーキュー ※1 ※T
- `gasha/tagged_ptr.h` [テンプレート] タグ付きポインター

<説明>

- ロックフリーアルゴリズムで実装しています。
- マルチスレッドでデータを共有するためのテンプレートクラスですが、ロック制御をしません。
- コンテナとしては機能しません。至ってシンプルなpush/poll/pop/enqueue/dequeueのみに対応します。
- 内部では固定領域のプールアロケータを保持しており、安全なメモリ操作が可能です。
- 「キヤッショコヒーレンシ問題」、「ABA問題」に対処済みです。
- スレッドが混み合うような状況下では、通常のロック制御よりも効率的に動作します。
- 逆に競合が少ない場面では、スピントロックによるロック制御の方が高速です。
- ロックフリーアルゴリズムは、シングルコアのマルチスレッド環境でも効果を発揮します。

<注釈>

- ※1: ロックフリーアルゴリズム特有の「ABA問題」に対処するために、「タグ付きポインター」を使用します。
タグ付きポインターは、ポインターの一部にタグとなる情報を仕込んで、ポインターの誤認を防ぐ手法です。
32ビット環境では、ポインターを64ビットに拡張して32ビットのタグを扱うことでABA問題をほぼ完全に防ぎます。
64ビット環境では、アラインメントの隙間(2~4ビットほど)か、実質的な有効アドレス範囲外の上位8ビットなどを利用します。

<資料>

- [プログラム最適化Tips.pdf](#)
[データ構造とアルゴリズムによる最適化]-[マルチスレッドを最適化するアルゴリズム]

ABA問題やキャッシュコヒーレンシ問題についての解説も記述しています。
サンプルプログラムを掲載していますが、ライブラリでは、更に洗練された状態で実装しています。

メモリ操作系: 基本

- [gasha/memory.h](#) メモリ操作基本 ※1 ※T
- [gasha/allocator_common.h](#) アロケータ共通設定・処理 ※2 ※T

<説明>

- 基本的なメモリ操作を扱います。

<注釈>

- ※1: アラインメント計算、アラインメント対応の標準メモリアロケータに対応しています。
アラインメント対応のメモリアロケータはコンパイラ標準のものを使用しますが、VC++に合わせて、GCCでも `_aligned_malloc()` と `_aligned_free()` を使えるようにします。その内部では `posix_memalign()` を使用しています。
- ※2: 各種メモリアロケータのための基本的な定数(デフォルトアラインメントなど)、および、明示的なコンストラクタ/デストラクタ呼び出し関数に対応しています。

メモリ操作系: アロケータ

- [gasha/stack_allocator.h](#) [テンプレートクラス] スタックアロケータ ※1 ※T
- [gasha/dual_stack_allocator.h](#) [テンプレートクラス] 双方向スタックアロケータ ※2 ※T
- [gasha/mono_allocator.h](#) [テンプレートクラス] 単一アロケータ ※3 ※T
- [gasha/pool_allocator.h](#) [テンプレートクラス] プールアロケータ ※4 ※T
- [gasha/std_allocator.h](#) [テンプレートクラス] 標準アロケータ ※5 ※T

<説明>

- 用途に合わせて使い分け可能な、幾つかのメモリアロケータを用意しています。
- 標準アロケータ以外は、いずれも固定バッファを内部に持ち、その範囲内でメモリ割り当てを行います。
- 「指定サイズのメモリ確保」、「型指定のメモリ確保とコンストラクタ呼び出し」、「配列型指定のメモリ確保とコンストラクタ呼び出し」(`[new[]演算子`と異なり、デフォルトコンストラクタ以外の呼び出しも可)、「メモリ破棄とデストラクタ呼び出し」に対応しています。
- テンプレート引数に `spinLock` や `std::mutex` などを与えることにより、排他制御を有効化し、スレッドセーフなメモリ確保が可能となります。デフォルトは `dummyLock` で、排他制御しません。
- 任意のアラインメント指定に対応します。

<注釈>

- ※1(1): 典型的なスタックアロケータです。個々のメモリ確保要求を管理しないため、メモリの解放ができませんが、その分メモリ効率がよく、また、高速に動作します。
- ※1(2): 使用メモリを強制的にクリアするか、指定のサイズに強制的に戻す事ができます。
- ※1(3): 「スマートスタックアロケータ」として機能拡張して使用することができます。
スタックアロケータは個々のメモリ解放ができませんが、メモリ確保数を管理しており、メモリ解放が呼び出されたらその数を減らします(メモリは解放されません)。
スマートスタックアロケータは、メモリ確保数が 0 になった時に、自動的にメモリをクリアします。
- ※2(1): 前方と後方の両側からメモリ確保を行うスタックアロケータです。「現在の割り当て方向」(前方からの「正順」か後方から

の「逆順」)を切り替えて扱います。メモリ確保時に割り当て方向を指定することも可能です。

- ※2(2): メモリのクリア、強制サイズ変更は、正順／逆順それぞれに対して操作可能です。
 - ※2(3): 「スマートスタックアロケータ」の機能は、正順／逆順それぞれに対して機能します。
 - ※3: 先着一つにしかメモリを確保しないアロケータです。
- `call_once` のような使い方ができますが、`call_once` のように先着スレッドの初期化の完了を待つ機能はありません。降着はアロケートに失敗するのみです。
- ※4(1): 固定ブロック×n個のメモリを管理します。固定ブロックのサイズを超えるメモリを確保することはできませんが、個々にメモリ解放することができます。
 - 固定ブロックの管理は効率的に処理できるため、高速に動作します。
 - ※4(2): ブロックサイズの範囲内であれば、任意の型や配列の確保が可能です。ただし、それがどんなサイズであれ、必ず1ブロックを消費します。
 - また、連続したブロックをまとめて扱うようなことはできません。
 - ※5: 内部で `malloc()` / `free()` もしくは `_aligned_malloc()` / `_aligned_free()` を扱います。
 - 他のアロケータと同じインターフェースを持つので、アライメント指定時や配列各保持に便利に利用できます。

<資料>

- [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.pdf](#)
[スタックアロケータ] / [プールアロケータ] / [フレームアロケータ] / [スマートスタックアロケータ]

サンプルプログラムを多数掲載していますが、ライブラリでは、更に洗練された状態で実装しています。(基本的な考え方はそのままに、実装内容は大きく変更しています)

- [ゲーム制御のためのメモリ管理方針.pdf](#)
[メモリ効率向上のための方針: 少しでも無駄のないメモリ確保を行うために]-[プールアロケータ]

プールアロケータの詳細な解説を記述しています。

メモリ操作系: ロックフリーアロケータ

- `gasha/lf_stack_allocator.h` [テンプレートクラス] ロックフリースタックアロケータ ※T
- `gasha/lf_dual_stack_allocator.h` [テンプレートクラス] ロックフリー双方向スタックアロケータ ※T
- `gasha/lf_mono_allocator.h` ロックフリー単一アロケータ
- `gasha/lf_pool_allocator.h` [テンプレートクラス] ロックフリープールアロケータ ※T

<説明>

- 各種メモリアロケータのロックフリー版です。マルチスレッドで効率的に動作します。

メモリ操作系: スコープアロケータ

- `gasha/scoped_stack_allocator.h` [テンプレートクラス] スコープスタックアロケータ
- `gasha/scoped_dual_stack_allocator.h` [テンプレートクラス] スコープ双方向スタックアロケータ

<説明>

- スタックアロケータ／双方向スタックアロケータ、および、そのロックフリー版を一時利用するためのクラスです。
- 既存のスタック系アロケータをコンストラクタで渡して使用します。
- 通常のスタック系アロケータとして振る舞います。
- 処理ブロック(スコープ)を抜ける時に、デストラクタで元の状態に戻します。

<資料>

- [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.pdf](#)
[フレームアロケータ] / [共通アロケータインターフェース]

「一時スタックアロケータ」という扱いでサンプルプログラムを掲載していますが、ライブラリでは、更に洗練された状態で実装しています。(基本的な考え方はそのままに、実装内容は大きく変更しています)

メモリ操作系: アロケータアダプター

- `gasha/allocator_adapter.h` [テンプレートクラス] アロケータアダプター
- `gasha/i_allocator_adapter.h` [テンプレートクラス] アロケータアダプターインターフェース

<説明>

- 多様なメモリアロケータに共通インターフェースを適用するために使用します。
- 動的にメモリアロケータを変更したい場合に利用できます。
- 各種アロケータは仮想化(virtualメソッド)に対応しておらず、共通インターフェースは実装していませんが、「アダプター」(テンプレートクラス)を通すことによって対応します。
- なお、各種アロケータは、コードレベルでは共通のインターフェースを持っており、テンプレートの共通利用は可能です。

<資料>

- [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.pdf](#)
[共通アロケータインターフェース]

サンプルプログラムを掲載していますが、ライブラリでは、更に洗練された状態で実装しています。(基本的な考え方はそのままに、実装内容は大きく変更しています)

メモリ操作系:多態アロケータ

- `gasha/new.h` 多態アロケータ対応標準new/delete演算子 ※1
- `gasha/poly_allocator.h` 多態アロケータ ※2 ※TLS

<説明>

- 【注意】:この機能を有効化すると、標準`new / delete` 演算子をオーバーライドします。使用時の際には、プロジェクト全体の方針を決めて、慎重に導入する必要があります。
- 標準`new / delete` 演算子によるメモリ確保／解放に対して、任意のアロケータを適用することができます。
- 例えば、STLを使用する際に、(STLの`Allocator` クラスを作成することなく)ローカル領域を割り当てるようなことが可能です。
- 多態アロケータ`polyAllocator` のコンストラクタに何らかのアロケータのアダプターを渡すことにより、`new / delete` 演算子が扱うアロケータが切り替わります。
- 多態アロケータのデストラクタで、変更前の状態に戻ります。
- TLS(Thread Local Storage)を活用しており、アロケータの変更が他のスレッドに影響しない仕組みです。
- `new / delete` 演算子の代わりに`GASHA_NEW` / `GASHA_DELETE` マクロを使用することにより、対象クラス／構造体のアラインメントを保証します。
更に、アロケータした時のソースファイル／行／関数／プログラム経過時間といったデバッグ情報も収集できます。
- デバッグ支援機能として、メモリ確保／解放時に反応するコールバック処理(オブザーバー)を登録することができます。この時、現在のアロケータやデバッグ情報を確認することができます。
- 多態アロケータのデフォルトのアロケータアダプターは、「標準アロケータ」(アライメント非対応版)です。

<注釈>

- ※1: 多態アロケータを利用する際にインクルードします。`GASHA_NEW` / `GASHA_DELETE` マクロが定義されています。
- ※2: 多態アロケータクラスの本体ですが、`<new.h>` がインクルードしているため、直接使用する必要がありません。

<資料>

- [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.pdf](#)
[共通アロケータインターフェース] / [STLなどの標準ライブラリを便利に活用するテクニック]

サンプルプログラムを掲載していますが、ライブラリでは、更に洗練された状態で実装しています。(基本的な考え方はそのままに、実装内容は大きく変更しています)

コンテナ系:擬似コンテナ(外部データコンテナ)

【配列】

- `gasha/dynamic_array.h` [テンプレートクラス] 動的配列 ※1 ※3 ※T
- `gasha/ring_buffer.h` [テンプレートクラス] リングバッファ ※1 ※4 ※T

【連結リスト】

- `gasha/linked_list.h` [テンプレートクラス] 双方向連結リスト ≈2 ≈5 ≈T
- `gasha/singly_linked_list.h` [テンプレートクラス] 片方向連結リスト ≈2 ≈6 ≈T

【二分探索木】

- `gasha/rb_tree.h` [テンプレートクラス] 赤黒木 ≈2 ≈7 ≈T

<説明>

- メモリ管理を行わず、外部のデータを操作する疑似コンテナです。
- STLライクなインターフェースで操作可能です。
- データ構造とアルゴリズムを分離して扱えるため、より自由度の高い活用ができます。
既存のユーザー定義データにアルゴリズムを適用するような操作が可能です。
- テンプレート引数に `shareSpinLock` などを与えることにより、ロック制御が可能となります。ただし、暗黙的なロック制御はしないので、`lockScoped` などのメソッドを任意に使用する必要があります。
デフォルトは `dummySharedLock` で、ロック制御が無効化されます（ロック制御の処理自体は記述できます）。

<注釈>

- ≈1: 配列系コンテナは、外部から既存の配列を受け取って扱います。
- ≈2: 連結リスト／木系コンテナは、外部からノードを受け取り、連結・解除・ソート・回転（平衡化）などのアルゴリズムを適用します。
- ≈3: 動的配列 `dynamic_array` は、STL の `std::vector`（動的配列）をモデルにしたインターフェースを実装しています。
- ≈4: リングバッファ `ring_buffer` は、STL の `std::deque`（双方向キュー）をモデルにしたインターフェースを実装しています。
- ≈5: 双方向連結リスト `linked_list` は、STL の `std::list`（双方向連結リスト）をモデルにしたインターフェースを実装しています。
- ≈6: 片方向連結リスト `singly_linked_list` は、STL の `std::forward_list`（C++11: 片方向連結リスト）をモデルにしたインターフェースを実装しています。
- ≈7: 赤黒木 `rb_tree` は、STL の `std::multimap/std::multiset` をモデルにしたインターフェースを実装しています。（キー重複を許容します。）

<資料>

- [プログラム最適化Tips.pdf](#)
[データ構造とアルゴリズムによる最適化]-[データ構造]

赤黒木の平衡化アルゴリズムの解説を記述しています。

サンプルプログラムを多数掲載していますが、ライブラリでは、更に洗練された状態で実装しています。

コンテナ系: コンテナ(内部データコンテナ)

- `gasha/binary_heap.h` [テンプレートクラス] 二分ヒープ ≈1 ≈T
- `gasha/hash_table.h` [テンプレートクラス] 開番地法ハッシュテーブル ≈2 ≈T

<説明>

- コンテナが保持する固定バッファのみで処理を行い、動的なメモリ確保／解放を行わないコンテナです。
- STLライクなインターフェースで操作可能です。
- テンプレート引数に `spinLock` や `std::mutex`、`shareSpinLock`（共有ロックはハッシュテーブルのみ）などを与えることにより、排他制御します。
デフォルトは `dummyLock` または `dummySharedLock` で、ロック制御しません。

<注釈>

- ≈1: 二分ヒープ `binary_heap` は、STL の `std::priority_queue`（優先度付きキュー）をモデルにしたインターフェースを実装しています。
- ≈2: 開番地法ハッシュテーブル `hash_table` は、STL の `std::unordered_map`（C++11: ハッシュテーブル）をモデルにしたインターフェースを実装しています。
キー重複を許容しません。重複キーの登録は失敗するか、置き換えにするかを指定できます。
なお、STLは（おそらく）連鎖法ハッシュテーブルです。

<資料>

- [プログラム最適化Tips.pdf](#)

[データ構造とアルゴリズムによる最適化]-[データ構造]

サンプルプログラムを多数掲載していますが、ライブラリでは、更に洗練された状態で実装しています。

コンテナ系: コンテナアダプタ

- gasha/priority_queue.h [テンプレートクラス] 優先度付きキュー ※1 ※T

<説明>

- 内部でコンテナを使用し、アルゴリズムを適用するコンテナアダプタです。
- テンプレート引数に `spinLock` や `std::mutex` などを与えることにより、排他制御します。
デフォルトは `dummyLock` で、ロック制御しません。

<注釈>

- ※1: 優先度付きキュー `priority_queue` は、STL を特にモデルとしていません。
STL の優先度付きキューと異なり、エンキューの順序性をデキュー時に保証します。
内部で使用するデフォルトのコンテナは二分ヒープ(`binary_heap`)です。
`push/pop/upHeap/downHeap` インターフェースを条件としてコンテナを使用します。

<資料>

- [プログラム最適化Tips.pdf](#)

[データ構造とアルゴリズムによる最適化]-[データ構造]

サンプルプログラムを掲載していますが、ライブラリでは、更に洗練された状態で実装しています。

デバッグ機能系: デバッグログ

- gasha/print.h ログ出力操作 ※1
- gasha/debug_log.h ログ操作
- gasha/log_level.h ログレベル
- gasha/log_category.h ログカテゴリ
- gasha/log_mask.h ログレベルマスク ※TLS
- gasha/log_attr.h ログ属性 ※TLS
- gasha/log_queue_monitor.h ログキュー モニター ※2

<説明>

- `print()` 文、`put()` 文により、ログにメッセージを出力できます。
- ログ出力時に、ログレベルとログカテゴリの指定を行います。ログレベルマスクの設定により、許可されたログカテゴリのログレベルのメッセージのみが実際に出力されます。
- ログカテゴリは、多様なゲーム要素ごとにログレベルを変えるような目的で利用します。
- ログカテゴリは、任意に追加可能です。例えば、「ミニゲーム用」、「開発者A専用」といったカテゴリを追加できます。開発者、コンテンツ制作、QAスタッフに応じて、必要なログだけが出力されるように設定できます。
- 重大なログレベルに対しては、ゲーム画面への通知も設定できます。(ゲーム画面に表示するための「コンソール」を任意に定義する必要があります)
- ゲーム中に動的にログレベル(マスク)を変更することができます。一時的にログ出力を抑えることや、より詳細なログを出力するようなことが可能です。
- ログレベルマスクのシリализ/デシリализ(セーブ/ロード)に対応しています。開発者、コンテンツ制作、QAスタッフに応じたログレベルマスクのデフォルトを用意するといった使い方ができます。
- 局所的なログレベルマスクの変更に対応しています。例えば、プログラム中の一部の処理ブロックだけログの出力を抑え、処理ブロックから抜ける時に元に戻すなどが可能です。
- ログ属性を扱うことができます。ログ属性により、ログ出力に付随してログID、ゲーム時間、ログレベル名、ログカテゴリ名などを表示できます。
- ログ属性もログレベルマスクのように、グローバルな設定変更と局所的な設定変更に対応しています。
- 局所的なログレベルマスク/ログ属性の変更は、TLSにより、他のスレッドに影響しません。
- 専用のワークバッファを使用して出力するため、スタックサイズの小さなスレッドからも問題なく出力できます。その代り、ワークバッファのサイズを超えるメッセージはそのサイズで切られます。
- なお、ワークバッファのサイズはビルド設定でカスタマイズ可能です。

- ・ビルド設定により、ログ出力に「ログキュー」を使用するように設定できます。ログキューを使用すると、ログ出力時点ではキューイングのみを行い、ログ出力の専用スレッドが実際の出力を行うようになります。
 - これにより、マルチスレッドでのログ出力の衝突を回避できます。また、ログ出力によるパフォーマンス低下を極力抑えることができます。
 - ・マルチスレッド間で処理の到達順序を正確にログに反映するために、順序性を保証したいタイミングで「ログの出力予約」を行い、遅延してログ出力を行うことができます。(ログキュー使用時のみ利用可能です。)
 - ・ビルド構成によりデバッグログが無効化されると、各処理クラスは、何もしない空の処理クラスになります。デバッグログを利用する側で #ifdef で判定せども、コンパイル時の最適化により、デバッグログのコードが消滅します。
 - ・ビルド構成によりデバッグログが無効化されると、ログキュー系のクラスはその存在自体消滅するため、利用する側で #ifdef で判定する必要があります。
- 他の処理のように空の処理クラスにすると、ログ出力スレッドの扱いがかえって煩雑になりかねないため、他の処理クラスと扱いを変えています。

<注釈>

- ・※1: このファイルだけをインクルードすれば、ログ出力が可能です。ログレベルマスク操作やログ属性操作などの必要性に応じて他のファイルをインクルードして使用します。
- ・※2: ログキューを使用する場合、「ログキューモニタ」の処理を専用スレッドで実行する必要があります。スレッドを終了させるためには、ログ操作クラスを使用して中止を要求します。

<資料>

- ・[効果的なデバッグログとアサーション.pdf](#)
[効果的なデバッグログとアサーション]-[処理仕様]-[デバッグメッセージ]

サンプルプログラムを掲載していますが、ライブラリでは、更に洗練された状態で実装しています。

サンプルプログラムではログ属性やログレベルマスクなどの多数の機能を「コールポイント」に統合していますが、ライブラリではそれぞれ独立した構成にしています。

デバッグ機能系: コンソール

- ・`gasha/i_console.h` コンソールインターフェース
- ・`gasha/tty_console.h` TTY端末 ※1
- ・`gasha/win_console.h` Windowsコマンドプロンプト
- ・`gasha/vs_console.h` Visual Studio出力ウインドウ
- ・`gasha/dummy_console.h` ダミーコンソール ※2
- ・`gasha/mem_console.h` メモリコンソール ※3
- ・`gasha/std_console.h` 標準コンソール ※4
- ・`gasha/console_color.h` コンソールカラー

<説明>

- ・デバッグログの出力先です。
- ・ログレベルまたはログカテゴリに対して、任意のコンソールを設定できます。これにより、分かり易いログの振り分けを行うことができます。
- ・コンソールカラーによって抽象化したカラー設定を扱い、それをコンソールに合わせて処理しています。
TTY端末はエスケープシーケンスで着色し、Windowsコマンドプロンプトはスクリーン属性で着色しています。
- ・コンソールインターフェースを実装したユーザー定義クラスを用意すれば、重大なメッセージをゲーム画面に通知することも可能です。
- ・プラットフォームに応じて対応していないコンソールを使用した場合、そのプラットフォームで利用可能なコンソールに置き換えます。
例えば、GCC環境でWindowsコマンドプロンプトを使用した場合、その実装をTTY端末に置き換えます。
- ・デバッグログを通さず単独で使用することも可能です。メモリコンソールを利用する場合などに便利です。
- ・ビルド構成によりデバッグログが無効化されると、各コンソールの処理クラスは、何もしない空の処理クラスになります。コンソールを利用する側で #ifdef で判定せども、コンパイル時の最適化により、コンソールのコードが消滅します。

<注釈>

- ・※1: 出力先TTY端末のファイルディスクリプタを任意に指定できます。多数の端末にログを振り分ける事や、ログを直接ファイルに記録することができます。(ファイルディスクリプタのオープン／クローズは行いません。)
- ・※2: 出を行わないコンソールです。例えば、「画面通知専用ログカテゴリ」に対しては、ログレベルに設定されたコンソールを

無効化するために使用しています。

- ※3: リングバッファに対して出力します。任意のバッファサイズの任意のインスタンスを使用できます。排他制御も可能です。
ログ出力を通さず直接コンソールを操作すると、マルチスレッドでのデバッグに役立ちます。
- ※4: 標準コンソールは、プラットフォームに対して適切なコンソールのデフォルトです。

デバッグ機能系: コールポイント

- `gasha/call_point.h` コールポイント ※TLS

<説明>

- 簡易的なコールスタック(コールポイントスタック)のトレースに使用します。
- コールポイントにより、処理ブロックのプロファイル(パフォーマンス計測)を集計できます。なお、コールポイント使用時にプロファイルを明示的に有効化する必要があります。
- コールポイント使用時には、ログカテゴリと名前を指定します。名前はコールポイントスタックの表示とプロファイルの集計に用います。
- コールポイントを共通処理の制御に利用できます。例えば、アサーションが設定された共通処理に対して、特定の利用場面ではアサーションを無効化したい時に利用できます。
ログ出力やアサーションには、ログカテゴリに「直近のコールポイントに従う」という設定が可能です。要するに、共通処理のログ出力／アサーション制御を、呼び出し元に従うように設定できます。
- コールポイントのネストが深くなった時にも意図したコールポイント制御が行えるように、「クリティカルコールポイント」を設定することができます。
ログ出力やアサーションには、ログカテゴリに「直近のクリティカルコールポイントに従う」という設定が可能です。
- コールポイントスタックは、TLSにより、他のスレッドに影響しません。
- ビルド構成によりコールポイントが無効化されると、コールポイントは何もしない空の処理クラスになります。コールポイントを利用する側で #ifdef で判定せずとも、コンパイル時の最適化により、コールポイントのコードが消滅します。

<資料>

- [効果的なデバッグログとアサーション.pdf](#)
[効果的なデバッグログとアサーション]-[処理仕様]-[コールポイント]

サンプルプログラムを掲載していますが、ライブラリでは、更に洗練された状態で実装しています。

サンプルプログラムよりも、コールポイントの機能を大幅にシンプルにし、別途細かく独立した処理を用意しています。

デバッグ機能系: アサーション

- `gasha/assert.h` アサーション／ブレークポイント／ウォッチポイント ※1 ※TLS
- `gasha/debugger_break.h` デバッガ用ブレークポイント割り込み ※2
- `gasha/i_debug_pause.h` デバッグポーズインターフェース
- `gasha/std_debug_pause.h` 標準デバッグポーズ ※3
- `gasha/stdin_debug_pause.h` 標準入力デバッグポーズ ※4
- `gasha/dummy_debug_pause.h` ダミーデバッグポーズ ※5

<説明>

- ログ出力の機能を利用したアサーションです。
- アサーション違反を検出すると、メッセージと位置情報(判定式とファイル名と関数名)、コールポイントスタックをログ出力し、処理を停止します。
- アサーション指定時には、ログ出力と同様に、ログレベルとログカテゴリを指定します。
- ログレベルマスクによって無効化されたアサーションは、ログ出力だけではなく、処理の停止もしません。
- アサーションは、指定の判定式が偽(違反)の時に反応しますが、別途真の時に反応する「ウォッチポイント」、判定式無しで反応する「ブレークポイント」が利用可能です。
- 「デバッグポーズ」を設定することで、アサーション反応時のポーズ方法を任意に設定できます。
- デバッグポーズインターフェースを実装したユーザー定義クラスを用意すれば、アサーション違反時にゲームをポーズするようなことも可能です。
- デバッグポーズの設定変更は、TLSにより他のスレッドに影響しません。これにより、メインスレッドは直接ポーズし、他のスレッドではポーズをスルーもしくはポーズ要求するといった使い分けが可能です。

<注釈>

- ※1: このファイルだけをインクルードすれば、アサーションの利用が可能です。ログレベルマスク操作やログ属性操作などの必要性に応じて他のファイルをインクルードして使用します。
- ※2: プラットフォームに応じたブレークポイント割り込み処理が定義されています。また、ビルド設定により、「開発ツール有効化」状態の時だけブレークポイント割り込みを行います。それ以外の時は何もしません(停止しません)。
- ※3: デバッガ用ブレークポイント割り込みを行います。デフォルトです。
- ※4: TTY端末もしくはWindowsコマンドプロンプト上で、エンターキーの入力待ちとなります。標準入力が使えるプラットフォームでのみ利用可能です。プログラムは停止しないので、他のスレッドはどうし続けることに注意が必要です。
- ※5: 一切ポーズせずにスルーします。

<資料>

- [効果的なデバッグログとアサーション.pdf](#)

[効果的なデバッグログとアサーション]-[処理仕様]-[アサーション／ウォッチポイント／ブレークポイント]

デバッグ機能系 : シンプルアサーション

- [gasha/simple_assert.h](#) シンプルアサーション／ブレークポイント／ウォッチポイント

<説明>

- ログレベルやログカテゴリの指定が不要な、シンプルなアサーションです。
- アサーション違反時には、ブレークポイント割り込みを発生させます。
- ビルド設定により、「開発ツール有効化」状態の時だけブレークポイント割り込みを行います。それ以外の時は何もしません(停止しません)。

デバッグ機能系 : プロファイル

- [gasha/profiler.h](#) プロファイル

<説明>

- コールポイントの処理開始から終了(デストラクタ)、までの処理時間を計測し、記録します。
- コールポイントを通さず、任意に利用する事も可能です。
- スレッドごとに、各処理の処理時間の集計を行います。処理には必ず名前を指定し、同じ名前の処理の集計回数、合計処理時間、最長処理時間、最短処理時間、平均処理時間を記録します。
- スレッドの判別はスレッド名で行うため、スレッドの生成と破棄を繰り返すようなスレッドであっても、同じ名前のスレッドであれば、一つのスレッドとして記録します。
複数のスレッドに同じ名前を付けた場合、同じスレッドとしてひとまとめに集計する点に注意が必要です。
- threadId クラスでスレッドに名前を付けていないと、記録できない点に注意が必要です。
- スレッドセーフな処理構造により、多数のスレッドの集計が入り乱れても問題なく記録できます。
- 毎フレーム一定のタイミングで、スレッドごとに、「集計」を実行し、記録した処理時間を集計します。
- 集計には2種類あり、「全体集計」と「期間集計」があります。
- 期間集計は、例えば1秒(30/60frame)に一回呼び出して使用します。これにより、直近の1秒内での合計／最長／最短／平均処理時間(および処理回数)を取得できるようになります。
リアルタイムなパフォーマンス表示を行う際、毎フレーム目まぐるしく変動する情報を表示するよりも、一定間隔で情報を更新した方が視認し易くなります。
- 集計した全スレッドの全プロファイルを取得するインターフェースを備えます。その際、情報取得用の配列を受け渡し、コピーを行うため、マルチスレッドで安全に情報を参照できます。
また、情報受け取る際には、ソート方法を指定することもできます。デフォルトは、期間集計の最長処理時間降順(大きい順)です。
- プロファイル記録のためのバッファのサイズは固定です。ビルド設定によって変更することができます。
- ビルド構成によりプロファイルが無効化されると、プロファイルは何もしない空の処理クラスになります。プロファイルを利用する側で #ifdef で判定せざとも、コンパイル時の最適化により、プロファイルのコードが消滅します。
- プロファイルの処理自体あまり高速ではないので(十分な最適化を行ってはいますが)、大量に記録すると低速化を招きかねません。利用のしそうには注意が必要です。

<資料>

- [効果的なデバッグログとアサーション.pdf](#)

[効果的なデバッグログとアサーション]-[処理仕様]-[コールポイント]

サンプルプログラムを掲載していますが、ライブラリでは、更に洗練された状態で実装しています。

シングルトン系

- `gasha singleton.h` [テンプレートクラス] シングルトン ※1 ※T
- `gasha singleton_debug.h` [テンプレートクラス] シングルトンデバッグ用処理 ※2 ※T
- `gasha dummy_singleton_debug.h` シングルトンデバッグ用ダミー処理 ※3 ※T

<説明>

- 任意のクラスをシングルトン化するためのテンプレートクラスです。
 - 対象クラスのオブジェクトをシングルトン内の固定バッファで扱うため、インスタンスの生成が必ず成功することを保証します。
 - シングルトンは対象クラスのプロキシーとして振る舞います。アロー演算子(`->`)で対象クラスのメンバーに透過的にアクセスできます。
 - 使用時の設定により、マルチスレッドセーフ(ロック制御)の仕組みを有効化できます。
 - 使用時の設定により、シングルトンデバッグの仕組みを有効化できます。
- この仕組みを用いると、シングルトンにアクセス中の処理を、アクセスが発生した順にリストアップすることが可能です。

<注釈>

- ※1: シングルトン本体です。テンプレートクラスが静的メンバー変数を持つため、明示的なインスタンス化を行うことを推奨します。
- ※2: シングルトンクラスを定義する時に指定可能なクラスです。
デバッグ機能が無効な時は何もしないクラスになります。
このクラスは、シングルトンが生成／削除された時の時間、同時アクセス中の処理名とスレッド情報のリストを管理します。
所定のメンバー関数を実装したクラスであれば、自作も可能です。
- ※3: シングルトンのデバッグ機能を無効化するためのクラスです。シングルトンにデフォルトで指定されます。

<資料>

- [効率化と安全性のためのロック制御.pdf](#)
[シングルトンの設計]

サンプルプログラムを掲載していますが、ライブラリでは、更に洗練された状態で実装しています。(基本的な考え方はそのままに、実装内容は大きく変更しています)

【検討事項】今後対応を検討しているライブラリ要素

以下、検討事項を箇条書きします。メモ書きです。

まだ最適な構造・手法を確立していないものです。

- 【検討事項: メモリ操作系】`gasha/memory_manager.h` 統合メモリマネージャ
 - ヒープメモリ対応。
 - ヒープ領域とページ領域を分けて管理し、境界の動的な変動に対応。どちらかのメモリ不足時に自動的に変動する。
 - バディシステムによるページ管理に対応。
 - 256バイト以下のメモリを動的にページ割り当てるプールアロケータに対応。
 - 任意の構造体の配列を動的にページ割り当てるスラップアロケータに対応。
 - ヒープメモリの各ノード(メモリブロック)の管理情報部をメモリブロック本体から切り離し、ページ領域で扱う。
 - これにより、メモリ効率の向上(アライメント調整し易くなる)、高速探索(キャッシュ効率がよくなる)、コンパクション(メモリ再配置)、メモリマネージャレベルでのスマートポインタ対応(参照カウンタによるガベージコレクション)、デバッグ情報管理などに対応する。
- 【検討事項: コンテナ系】`gasha/linked_list.h` [テンプレートクラス] 片方向／双方向連結リスト
 - 高速外部ソート対応。(基数ソートを片方向連結リストに対応する)
- 【検討事項: コンテナ系】`gasha/aggregate.h` [テンプレートクラス] 集合
 - STLの集合(std::set)とは全く異なり、他のコンテナの部分集合を扱うことを目的としたクラス。
 - 例えば、ゲームのインベントリデータでありがちな、部分的なカテゴリの抜き出しや一時的なソートなどを扱い易くする。
 - 簡易トランザクション機能を検討。通常の部分集合は元の集合を直接参照するが、トランザクション有効時は値のコピーを作成する。

- 成し、元のデータに影響せずに値を変更する。
- 簡易トランザクションは、最終的なコミット(元の集合に反映)・ロールバック(変更を破棄)の機能を持つ。
- トランザクションの機能は、マルチスレッドの処理を最適化するための「コピー・オンライン」としても利用可能とする。
- 【課題】SQLiteやRDBなどの連携も考慮するか？SQL、トランザクション、カーソルなどの対応？

- 【検討事項:コンテナ系】`gasha/chain_hash_table.h` [テンプレートクラス] 連鎖法ハッシュテーブル

- ハッシュテーブルに赤黒木でノードを連結し、重複許可と検索性能向上を実現したハッシュテーブルを構成。
- 【課題】イテレータには非対応？

- 【検討事項:コンテナ系】`gasha/abstract_syntax_tree.h` [テンプレートクラス] 抽象構文木

- 抽象構文木の対応。
- 計算式を解析・演算するための独自パーサーを、ユーザー側で開発するための基本処理とする。
- 想定する用途：ツール開発(パーサー部分)に利用。ランタイムパーサーに利用。計算式の演算に利用。
ゲームデータ用Excelに記述したゲーム制御用の計算式など利用可能。
- 下記処理との組み合わせ対応。計算式を含んだJSONパーサーの独自実装が一つの目標。
- 構文解析に対応。(パーサー用ステートマシンを汎用化？)
- パーサーの構文エラー通知に対応。
- 抽象構文木の中置記法から前置記法(ポーランド記法)、後置記法(逆ポーランド記法)への変換に対応。
- 中置記法、前置記法、後置記法の演算に対応。
- シリアル化／デシリアル化に対応。
- 四則演算、ビット演算、論理演算、三項演算子に対応。
- 文字列に対応。
- 組み込み関数(算術系)に対応。
- ユーザー定義関数に対応。
- (できれば)変数、制御文、ループ分に対応。
- 演算時のオブジェクトバインディングに対応。
- 基本的にコレーチンには非対応。(拡張は考慮)

- 【検討事項:コンテナ系】`gasha/graph.h` [テンプレートクラス] グラフ

- 重み付き有向グラフの対応。(一方通行のある経路探索に適用可能)
- マトリックス形式データ、エッジ(2ノード連結情報)リスト形式データのどちらか、もしくは、それぞれの対応。
- 「扉の閉閉」のような、一方通行のON/OFFも考慮。
- ダイクストラ法に対応。主に経路探索の事前計算用。データのシリアル化／デシリアル化も考慮。
- A*法に対応。主にランタイムの経路探索用。
- 所属ノード探索に対応。自分の座標や目標点の座標からノードを素早く算出することを想定。
- 高速ノード探索のためのインデックステーブルもしくはハッシュテーブルの対応。
- 高速最短経路探索のためのヒープアルゴリズム検証。(二項ヒープ、フィボナッチヒープ、ペアリングヒープ、Skewヒープなど)

- 【検討事項:コンテナ系】`gasha/hierarchy.h` [テンプレートクラス] 階層

- 階層構造データ。
- 親ノードへの連結、両隣の兄弟ノードへの連結、最初の子ノードへの連結情報を持つ。もしくは、そこから親と兄への連結を廃止。
- デザインパターンのCompositeパターンや、Visitorパターン、ChainOfResponsibilityパターンをサポート。
- シンググラフのようなデータへの応用を想定。

- 【検討事項:コンテナ系】`gasha/spatial_partitioning.h` [テンプレートクラス] 空間分割

- バイナリ空間分割木(BSP木)／四分木／八分木／kd木など検討。
- 衝突判定の絞り込み、描画対象の絞り込み、処理対象の間引きなど、処理効率化のためのデータ構造とアルゴリズムに対応。
- `std::set`のような集合を管理するコンテナとして扱い、指定範囲に該当する部分集合を返すなどの構造を検討。
部分集合取得時は、情報格納用の配列を渡すインターフェースや、近い順にソートして返すなどの機能を検討。
- 更には、階層構造と組み合わせ、シンググラフ上のオブジェクトのアニメーション順序・省略決定、移動・衝突判定の順序・省略決定、不透明／半透明描画の順序・省略決定など、処理要件に柔軟に対応できる構造を目指す。

- 【検討事項:環境／文字列系】`gasha/locale.h` ローカライズ

- 数ヶ国語のローカライズに対応した環境設定の対応。
- 日付、時間、数値(桁区切り、小数点)表示の対応。

- UTF-8, ShiftJIS, EUCコードの文字コード変換対応(デバッグログと組み合わせて使用)。
- 動的な言語切り替えをサポート。
- 対象言語に応じたリソースファイル選択のための、ファイルパス作成補助機能対応。

- 【検討事項:リソースビルダーフレームワーク】

- リソース構築のための共通フレームワーク。
- あらゆる種類のリソース構築要求を、同じインターフェースで統合的に管理する。
- リソースの種類とプロジェクト固有の各種リソース構築処理を関連づけ、処理の振り分けを行う。

- 【検討事項:ユニットテストフレームワーク】

- ユニットテスト／回帰テストのためのフレームワーク。
- プログラムの品質維持と、QAテストの効率化を目的とする。
- ユニットテストのための処理コードを各ソースコードに書いておけば、メイン関数などからの明示的な呼び出しを記述せずに実行できる仕組み。
- ゲームでの活用を想定し、任意のタイミングでのユニットテスト呼び出しに対応。(ステージや所持アイテムなど、複雑な状況を再現・変更して再テストするような時に便利)
- ユニットテストを複数のグループに分けて登録可能とし、任意のテスト時にはグループを指定する。

- 【検討事項:名前付きデータ参照】

- データの参照に名前を付け、名前と型だけ分かっていれば、どこからでもデータにアクセスできる仕組み。
- スクリプトのオブジェクトバイナリなどに利用することを想定。スクリプト用の処理が様々な処理をインクルードして複雑化することを防ぐ。

- 【検討事項:圧縮／解凍】

- ランタイムの可逆圧縮／解凍処理の対応。
- 圧縮効率よりもパフォーマンスを優先。
- 64KB(仮)ごとに分割して圧縮し、圧縮／解凍処理をマルチコアやGPGPUで並列に実行できる構造にする。

- 【検討事項:ジョブスケジューラ】

- 過剰なスレッドや、演算リソース使用量を制限するためのジョブスケジューラ。
- 例えば、上記の圧縮／解凍で、並列実行可能なスレッドを制限する。
- GPGPU対応。汎用性を考慮し、OpenGLベースにする。(検討すべき他のGPGPU技術:OpenGL Compute Shader, DirectX Direct Compute, NVIDIA CUDA, OpenAL)

- 【検討事項:誤り符号訂正】

- ハミング符号の実装。
- 通信データやその他のシリアルライズデータに利用可能。

- 【検討事項:暗号化／複合化】

- AES暗号化対応。
- SSE命令で高速化。

- 【検討事項:ビルド設定】

- 数ヶ国語のローカライズを考慮したビルド設定。

ライブラリの命名規則

ライブラリに用いている命名規則は、基本的には下記のとおりです。

- 【ファイル名】:すべて小文字でアンダースコアを区切り文字としたスネークケース。

- (例)word_and_word.h

- 【ネームスペース名】:すべて小文字でアンダースコアを区切り文字としたスネークケース。

- (例)word_and_word

- 【クラス名／関数名／メソッド名】: 小文字から始まるローワーキャメルケース。
 - (例) wordAndWord
- 【例外】: 標準ライブラリのクラスやメソッドと同質・同類・類似の場合、それがスネークケースであっても、同じ名前か同様の名前を用いる。
 - (例) push_backメソッド, lock_guardクラス, max_sizeRealメソッド(類似メソッド)
- 【例外】: クラス／構造体内で「型」を定義する場合、(標準ライブラリを真似て)「***_type」の表記を用いる。
 - (例) value_type
- 【例外】: 汎用的でプリミティブな型は、(標準ライブラリを真似て)「***_t」の表記を用いる。
 - (例) crc32_t
- 【混在の許容】: 一つのソースファイルやクラス内に、キャメルケースと例外(スネークケース)が混在することを問題とせず、むしろ独自拡張を識別可能なものとして扱う。
 - (例) dynamic_array::conrtainerクラスは、std::vectorに準拠し、assign, popo_backなどのメソッドを実装しており、独自メソッドは assignArray のようなキャメルケースの名前で実装。

ライブラリのソースファイル

ソースファイルは、Visual C++とGCCの互換性のために、下記の仕様で統一しています。

- 文字コード ... UTF-8(BOM付き)
 - Visual Studio が扱う標準的なユニコードは UTF-8(BOM付き)です。
 - GCCが扱う標準的なユニコードは UTF-8(BOMなし)ですが、4.3.x 以降、BOM付きにも対応しています。
- 改行コード ... LF
 - Windows系OSの標準的な改行コードは CR+LF ですが、Visual StudioはLFのみのソースファイルも扱えます。
 - Unix系OSの標準的な改行コードは LF です。

ビルド構成

ビルド構成

以下、本ライブラリが想定するビルド構成です。

コンパイルオプション(-Dオプション)で該当するマクロを指定すると反映されます。

- `DEBUG` フルデバッガ設定

マクロ: `GASHA_BUILD_CONFIG_IS_FULL_DEBUG`
 デバッガ機能: あり(冗長)
 デバッガログ: 有効
 アサーション: 有効
 開発ツール利用: 可能
 最適化: なし
 ファイルシステム: ローカル／ROM切り替え(データで設定)
 ユニットテスト: 可
 シンボル情報: あり

- `DEBUG_MODERATE` プログラム開発向け設定

マクロ: `GASHA_BUILD_CONFIG_IS_DEBUG_MODERATE`
 デバッガ機能: あり(冗長)
 デバッガログ: 有効
 アサーション: 有効

開発ツール利用: 可能

最適化: 小

ファイルシステム: ローカル／ROM切り替え(データで設定)

ユニットテスト: 可

シンボル情報: あり

- `DEBUG_OPT` コンテンツ制作・QA向け設定

マクロ: `GASHA_BUILD_CONFIG_IS_DEBUG_OPT`

デバッグ機能: あり

デバッグログ: 有効

アサーション: 有効

開発ツール利用: 不可

最適化: 最大

ファイルシステム: ローカル／ROM切り替え(データで設定)

ユニットテスト: 可

シンボル情報: あり

- `UNIT_TEST` 自動回帰テスト向け設定

マクロ: `GASHA_BUILD_CONFIG_IS_REGRESSION_TEST`

デバッグ機能: あり

デバッグログ: 有効

アサーション: 有効

開発ツール利用: 不可

最適化: 最大

ファイルシステム: ローカル／ROM切り替え(データで設定)

ユニットテスト: 専用

シンボル情報: あり

- `LOCAL_RELEASE` 製品テスト向け設定

マクロ: `GASHA_BUILD_CONFIG_IS_LOCAL_RELEASE`

デバッグ機能: なし

デバッグログ: 無効

アサーション: 無効

開発ツール利用: 不可

最適化: 最大

ファイルシステム: ローカル／ROM切り替え(データで設定)

ユニットテスト: 不可

シンボル情報: あり

- `RELEASE` 製品向け設定

マクロ: `GASHA_BUILD_CONFIG_IS_RELEASE`

デバッグ機能: なし

デバッグログ: 無効

アサーション: 無効

開発ツール利用: 不可

最適化: 最大

ファイルシステム: ROMのみ

ユニットテスト: 不可

シンボル情報: なし

注釈

- 現状のライブラリは、`DEBUG` と `RELEASE` 以外のビルド構成を用意していません。
また、ファイルシステムには対応しておらず、マクロの定義のみを行っています。
これらは、整然とした開発環境の構築を目的に、扱うべきビルド構成とその内訳を整理したものです。

ビルド構成に関するマクロ

以下、ビルド構成に応じて指定されるマクロです。

このマクロに基づいて、処理が有効化／無効化されてライブラリがビルドされます。

ライブラリを使用するプロジェクト(ユーザー側)でも使用可能です。

- `GASHA_DEBUG_FEATURE_IS_ENABLED` ... デバッグ機能有効化
- `GASHA_VERBOSE_DEBUG_IS_ENABLED` ... 冗長デバッグ機能有効化
- `GASHA_LOG_IS_ENABLED` ... デバッガログ有効化 ※1
- `GASHA_ASSERTION_IS_ENABLED` ... アサーション／ブレークポイント／ウォッチポイント有効化 ※2
- `GASHA_CALLPOINT_IS_ENABLED` ... コールポイント機能有効化 ※2
- `GASHA_PROFILE_IS_AVAILABLE` ... プロファイル機能有効化
- `GASHA_DEV_TOOLS_IS_AVAILABLE` ... 開発ツール利用可能 ※3
- `GASHA_NO_OPTIMIZED` ... 最適化なし ※4
- `GASHA_OPTIMIZED_MODERATELY` ... 適度に最適化 ※4
- `GASHA_OPTIMIZED` ... 最大限の最適化 ※4
- `GASHA_FILE_SYSTEM_IS_ROM` ... ROM専用ファイルシステム
- `GASHA_FILE_SYSTEM_IS_FLEXIBLE` ... ローカル／ROM切り替えファイルシステム(ローカルデータで設定)
- `GASHA_UNITE_TEST_ENABLED` ... ユニットテスト(の仕組みが)有効
- `GASHA_IS_REGRESSION_TEST` ... 回帰テストモード有効 ※5
- `GASHA_HAS_SYMBOLS` ... シンボル情報あり ※6
- `GASHA_IS_STRIPPED_SYMBOLS` ... シンボル情報なし ※6

注釈

- ※1: デバッガログ無効化時は、コンソール(端末等へのログ出力用)も無効化されます。
- ※2: アサーション／ブレークポイント／ウォッチポイント／コールポイント有効時は、自動的にデバッガログを有効にします。
- ※3: このオプションは、「Visual Studio出力カウントをどの程度表示するか」などの判定に用います。
- ※4: これらは、ビルドの最適化状態を示すことを目的としたマクロであり、実際にこのマクロに基づいて最適化するわけではありません。
- ビルド構成ごとに、別途最適化オプションが設定されます。
- なお、このマクロに基づいて、最適化コードの切り替えを行う処理を組んでも構いません。
- ※5: 「回帰テスト」はユニットテストの自動実行です。
- ※6: これもビルドの状態を示すことを目的としたマクロであり、実際にこのマクロに基づいてシンボル情報の組み込み／削除を行うわけではありません。

ターゲットプラットフォーム

現状、x86系Windows、x64系Windows、Cygwinに対応しています。

Visual C++ では、`Win32` と `x64` という名称のプラットフォームです。

ディレクトリ構成

サブモジュール／ライブラリ本体

```
[gasha_examples/gasha_proj] ... ライブラリサンプルプログラム／ビルドプロジェクト用リポジトリ
|
|- [sub] ... サブモジュール用
  |
  `-[gasha] ... サブモジュール：ライブラリ本体用リポジトリ
    |
    |- [include] ... インクルードファイル用
      |
      `-[gasha] ... ネームスペースディレクトリ
        |
        |-*.*h/*.*inl/*.*cpp.h ... 各種インクルードファイル
        |
        `-[build_settings] ... ビルド設定用
          |
          |- build_settings.h ... ビルド設定ファイル（代表）
          |
          |-[compiler_auto_settings.h] ... ビルド設定：コンパイラ自動判別・設定
          |-[platform_auto_settings.h] ... ビルド設定：プラットフォーム自動判別・設定
          |-[platform_auto_settings_***.h] ... ビルド設定：プラットフォーム自動判別・設定（osやプラットフォームごとのファイル）
```

```

|      |- language_auto_settings.h     ... ビルド設定 : 言語仕様自動判別・設定
|      |- build_configuration.h     ... ビルド設定 : ビルド構成 (デバッグ機能の有効化など)
|      |- adjust_build_settings.h   ... ビルド設定 : 調整用 (主にプロジェクト固有設定による設定上の矛盾・過不足の調整)
|      |- builtin_functions.h       ... ビルド設定 : (ビルド設定に基づく) 標準的な組み込み関数・マクロ (ソースファイル名)
|      |- builtin_functions.inl    ... (同上)
|      |- build_settings_diag.h    ... ビルド設定診断
|      `-- build_settings_diag.inl ... (同上)

|
`-[lib]           ... ライブラリファイル用
|
|
`-[vc]            ... Visual C++用
|
|
`-- gasha_x86.lib        ... x86リリースビルト用
`-- gasha_x86_debug.lib  ... x86デバッグビルト用
`-- gasha_x64.lib        ... x64リリースビルト用
`-- gasha_x64_debug.lib  ... x64デバッグビルト用

|
`-[gcc]            ... GCC用
|
|
`-- gasha_x86.a        ... x86リリースビルト用
`-- gasha_x86_debug.a  ... x86デバッグビルト用
                               (x86版Cygwinで開発したため、x64版は現状なし)

|
`-[proj]
|
`-- Makefile_common ... GCC用メイクファイルの共通部分
                           (ライブラリ／サンプルプログラム共用)

```

サブモジュール／ライブラリ拳動設定

```

[gasha_examples/gasha_proj] ... ライブラリサンプルプログラム／ビルドプロジェクト用リポジトリ
|
`-[sub]           ... サブモジュール用
|
`-[gasha_settings] ... サブモジュール : プロジェクト固有のライブラリ拳動カスタマイズ用
|
`-[include]       ... ライブラリインクルードファイル用
|
`-[gasha]          ... ネームスペースディレクトリ
                   (ライブラリファイルパスの競合を避けるためのサブディレクトリ)
`-[build_settings] ... ビルド設定用
|
`-- project_first_settings.h ... プロジェクト固有のライブラリ設定 (先行設定)
|
`-- project_last_settings.h ... プロジェクト固有のライブラリ設定 (最終設定)
|
`-- project_default_includes.h ... プロジェクト固有のデフォルトインクルード設定

```

サブモジュール／ライブラリソース

```

[gasha_examples/gasha_proj] ... ライブラリサンプルプログラム／ビルドプロジェクト用リポジトリ
|
`-[sub]           ... サブモジュール用
|
`-[gasha_src]    ... サブモジュール : ライブラリソース用リポジトリ
|
`-[proj]          ... ライブラリビルドプロジェクト用
|
`-- standard.h/.cpp ... 強制インクルード／プリコンパイル済みヘッダー
|
`-- gasha.vcxproj ... Visual C++用ライブラリビルドプロジェクト
`-- Makefile       ... GCC用ライブラリビルドメイクファイル
`-- mk.sh          ... GCC用ライブラリビルドシェルスクリプト

|
`-[src]           ... ライブラリソースファイル用
`-...

```

実行ファイル

```
[gasha_examples]          ... ライブラリサンプルプログラム用リポジトリ
|
|-[exe]                  ... 実行ファイル用
|
| - run_all.bat          ... Visual C++ 版全サンプルプログラム一括実行＆実行ログ記録用バッチファイル
| - run_all.sh            ... GCC版全サンプルプログラム一括実行＆実行ログ記録用バッチファイル
|
| -[vc]                   ... Visual C++でビルドした実行ファイルの置き場
| |
| | - run_all.bat          ... x86&x64全サンプルプログラム一括実行＆実行ログ記録用バッチファイル
| |
| | -[x86]                 ... x86(32bit)向け
| |
| | | - (サンプル名) .exe    ... サンプルプログラム実行ファイル
| | | - run_all.bat          ... リリースビルド＆デバッグビルド全サンプルプログラム一括実行＆実行ログ記録用バッチファイル
| |
| | | -[bat]                ... サンプルプログラム実行と実行ログ記録用
| | |
| | | | - (サンプル名) .bat   ... サンプルプログラム個別実行＆実行ログ記録用バッチファイル
| | | | - run_all.bat          ... 全サンプルプログラム一括実行＆実行ログ記録用バッチファイル
| |
| | | `-[log]
| | |
| | | | - (サンプル名) .log     ... サンプルプログラム個別実行ログファイル
| |
| | `-[Debug]               ... デバッグビルド用
| | | -...                  (親フォルダ（リリースビルド用）と同様の構成)
|
| `-[x64]                  ... x64(64bit)向け
| | - ...                  (x86版と同様の構成)
|
`-[gcc]                   ... GCCでビルドした実行ファイルの置き場
|
`-[x86]                   ... x86(32bit)向け
| - ...                  (Visual C++用と同様の構成)
| (シェルスクリプトの代わりにバッチファイル使用)
| (x86版Cygwinで開発したため、x64版は現状なし)
```

各サンプルプログラムのソース

```
[gasha_examples]          ... ライブラリサンプルプログラム用リポジトリ
|
|-[src]                  ... サンプルプログラムソースファイル用
|
| -[ (サンプル名) ]      ... 各サンプルプログラムソースファイル用
| |
| | - *.*.cpp/.h
```

ライブラリおよび各サンプルプログラムのビルドプロジェクト

```
[gasha_examples]          ... ライブラリサンプルプログラム用リポジトリ
|
|-[proj]                 ... ライブラリ／サンプルプログラムビルドプロジェクト用
|
| -[ (サンプル名) ]      ... 各サンプルプログラムビルドプロジェクト用
| |
| | - main.cpp            ... サンプルプログラム用メイン関数
| | - standard.h/.cpp     ... サンプルプログラム用強制インクルード／プリコンパイル済みヘッダー
| |
| | - (サンプル名) .vcxproj ... Visual C++用サンプルプログラムビルドプロジェクト
| | - Makefile             ... GCC用サンプルプログラムビルドメイクファイル
| | ` - mk.sh              ... GCC用サンプルプログラムビルドシェルスクリプト
|
| - Makefile_example_common ... GCC用サンプルプログラムメイクファイルの共通部分
|
| - gasha_examples.sln     ... Visual C++用ライブラリ＆全サンプルプログラムビルドソリューション
` - mk_all.sh              ... GCC用ライブラリ＆全サンプルプログラムビルドシェルスクリプト
```

ライブラリおよび各サンプルプログラムのビルドプロジェクト

```
[gasha_proj]           ... ライブラリビルドプロジェクト用リポジトリ
|
|-[proj]              ... ライブラリビルドプロジェクト用
|
|- gasha.sln          ... Visual C++用ライブラリビルドソリューション
`- mk.sh              ... GCC用ライブラリビルドシェルスクリプト
```

ライブラリの利用手順

- 前提1: **GASHA**もしくはその派生ライブラリをgitリポジトリで管理するものとします。

- 前提2: ライブラリを複数プロジェクトで共通利用するものとします。

1. ライブラリ用リポジトリとプロジェクト固有のライブラリ挙動カスタマイズ用リポジトリをプロジェクトに配置

サブモジュールとして、ライブラリ用のリポジトリを配置して下さい。

```
project/sub/gasha ... ライブラリ本体用リポジトリ(https://github.com/gakimaru/gasha)
project/sub/gasha_settings ... プロジェクト固有のライブラリ挙動カスタマイズ用
(https://github.com/gakimaru/gasha\_settings)
```

ライブラリ自体をプロジェクト向けにビルドする場合は、ライブラリソース用リポジトリも配置して下さい。

```
project/sub/gasha_src ... ライブラリソース用リポジトリ(https://github.com/gakimaru/gasha\_src)
```

2. インクルードパスを設定

下記のパスをプロジェクトのインクルードパスに追加して下さい。

```
project/sub/gasha/include ... ライブラリ用
project/sub/gasha_setting/include ... ライブラリ挙動設定用
```

3. ライブラリパスとライブラリファイルを設定

下記のいずれか一つのファイルをプロジェクトのライブラリファイルに追加して下さい。

【Visual C++用】

```
project/sub/gasha/lib/vc/gasha_x86.lib ... x86リリースビルド用
project/sub/gasha/lib/vc/gasha_x86_debug.lib ... x86デバッグビルド用
project/sub/gasha/lib/vc/gasha_x64.lib ... x64リリースビルド用
project/sub/gasha/lib/vc/gasha_x64_debug.lib ... x64デバッグビルド用
```

【GCC用】

```
project/sub/gasha/lib/gcc/gasha_x86.a ... x86リリースビルド用
project/sub/gasha/lib/gcc/gasha_x86_debug.a ... x86デバッグビルド用
(x86版Cygwinで開発したため、x64版は現状なし)
```

4. 【推奨】強制インクルードとプリコンパイル済みヘッダーを設定

プラットフォーム／言語設定を暗黙的に全ソースファイルに反映させるために、「強制インクルード」を使用することを推奨します。更に、強制インクルードファイルを「プリコンパイル済みヘッダー」にすることで、コンパイル速度を高速化することを、合わせて推奨します。

【設定例: standard.h】

プロジェクトファイル (*.vcxproj, Makefile) と同じディレクトリに standard.h を配置して下さい。
standard.h の内容(この一行のみ)

```
#include <gasha/build_settings/build_settings.h>
```

【設定例: standard.cpp】(Visual C++のみ必要なファイル)

プロジェクトファイル (*.vcxproj, Makefile) と同じディレクトリに standard.cpp を配置して下さい。

standard.cpp の内容(この一行のみ)

```
#include <standard.h>
```

【Visual C++の場合】

プロジェクトのプロパティから、[C/C++]→[詳細設定]ページの設定を下記のように変更して下さい。

- [必ず使用されるインクルードファイル]に standard.h を指定

プロジェクトのプロパティから、[C/C++]→[プリコンパイル済みヘッダー]ページの設定を下記のように変更して下さい。

- [プリコンパイル済みヘッダー]に「使用(/Yu)」を指定
- [プリコンパイル済みヘッダーファイル]に standard.h を指定

更に、standard.cpp のプロパティから、[C/C++]→[プリコンパイル済みヘッダー]ページの設定を下記のように変更して下さい。

- [プリコンパイル済みヘッダー]に「作成(/Yc)」を指定
- [プリコンパイル済みヘッダーファイル]に standard.h を指定

【GCCの場合】

g++コマンドでプリコンパイル済みヘッダーファイル standard.h.gch を作成して下さい。

【例】

```
$ g++ (-std=c++11 や -g などのコンパイルオプション) -x c++-header standard.h
```

standard.h.gch はインクルードパスが通った場所に配置して下さい。

コンパイル時には、g++コマンドに -include オプションを指定して下さい。

.h.gch ではなく、.h ファイルを指定します。

【例】

```
$ g++ (-std=c++11 や -g などのコンパイルオプション) -include standard.h -c xxx.cpp -o xxx.o
```

5. サブモジュールのプランチ(もしくはタグ／バージョン)を確定してコミット

サブモジュールの挙動設定ファイルの変更やライブラリのビルドなどを行った後、プロジェクトをコミットして下さい。

サブモジュール用のフォルダがコミット対象になります。

これにより、プロジェクトで使用するサブモジュール(ライブラリのリポジトリ)のバージョンが設定されます。

サンプルプログラム用プロジェクト

サンプルプログラム用プロジェクト

```
[gasha_examples] ... ライブラリサンプルプログラム用リポジトリ
|
`-[proj]
  |           【ビルト設定】
  |-[example_build_settings] ... ビルド設定テスト
  |
  |           【ユーティリティ】
  |-[example_utility]     ... 汎用ユーティリティテスト
  |-[example_type_traits] ... 型特性ユーティリティテスト
  |
  |           【算術】
  |-[example_basic_math]   ... 基本算術処理テスト
  |-[example_fast_math]    ... 高速算術処理テスト
  |-[example_crc32]        ... CRC32計算処理テスト
  |
  |           【文字列】
  |-[example_fast_string]  ... 高速文字列処理テスト
  |
```

```

|                               【アルゴリズム】
|[example_sort_and_search]    ... ソート&探索処理テスト
|
|                               【スレッド】
|[example_thread_id]          ... スレッドIDテスト
|[example_shared_data]        ... マルチスレッド共有データ／ロックフリー共有データテスト
|
|                               【メモリ】
|[example_allocator]          ... アロケータテスト
|
|                               【擬似コンテナ（外部データコンテナ）】
|[example_dynamic_array]      ... 動的配列コンテナテスト
|[example_reing_buffer]        ... リングバッファコンテナテスト
|[example_linked_list]         ... 双方向連結リストコンテナテスト
|[example_singly_linked_list]  ... 片方向連結リストコンテナテスト
|[example_rb_tree]             ... 赤黒木コンテナテスト
|
|                               【コンテナ（内部データコンテナ）】
|[example_hahs_table]          ... 開番地法ハッシュテーブルコンテナテスト
|
|                               【コンテナアダプタ】
|[example_priority_queue]      ... 二分ヒープコンテナ&優先度付きキュー コンテナアダプタテスト
|
|                               【シングルトン】
|[example_singleton]            ... シングルトンテスト

```

免責事項

本ライブラリの実績は不十分であり、品質を保証できる状態ではありません。

ライブラリの不具合には基本的に対応できませんので、ご利用の際は、自己責任をお願いします。

なお、MITライセンスに基づき、自由な改変や商用利用も可能ですので、独自に改変・修正の上でご利用頂けます。

不具合を見つかった場合には、教えて頂けると幸いです。

関連資料

[ゲームシステムのアーキテクチャと開発環境\(GitHub\)](#)

[ゲームシステムのアーキテクチャと開発環境\(DropBox\)](#)

資料にはサンプルプログラムも多数掲載しており、ライブラリのプログラムと同じものも含んでいます。

サンプルプログラム作成とそれにに基づく資料作成が先行し、その後ライブラリにまとめ直しているので、両者のソースコードには多少の相違があります。

■■以上