補

P A F E E

Pragmatic Application Framework for Embedded Environment

ユーザズマニュアル

Rev 0.1

目 次

1. 概要

PAFEE とは

PAFEEは組込みソフトウェア向けの C++アプリケーションフレームークです。
標準 C++ライブラリ（STL を含む）は汎用的なコンテナ、文字列クラス、ソートアルゴリズムなどの便利な機能を持っていますがテンプレートをベースにしているためコードサイズの増大が懸念されます。 また動的メモリ管理機構の使用を前提としており、単に標準の new/delete を使ってしまうとメモリの断片化も懸念されます。 このような事情から、スペックの低いハードウェアで稼動する組込みソフトウェアでは使いにくいケースが多くあります。 そこで組込みソフトウェアに特化したフレームワークとしてPAFEEが誕生しました。
以下の特徴を持っています。

- C++のクラスライブラリ
- 組込み向けに設計されたパフォーマンスが高く柔軟にカスタマイズできるメモリ管理機構
- 拡張性、再利用性や保守性を向上させるイベントディスパッチ機構
- テンプレートの使用有無が選択可能なコレクションクラス
- 最小限の例外処理（全く使用しないことも可能）
- ITRON(TOPPERS)、Linux、Windows の差異を吸収するOS抽象化クラス
- オープンソース

ターゲット

PAFEEがターゲットとするのは、中規模の組込みソフトウェアです。 8ビットマイコンで動作するような小規模の組込みソフトウェアは対象外です。またスペック的にはほとんどPCと代わらないようなCPUで動く大規模な組込みソフトウェアでも利用は可能ですが、利用するメリットがあるのはメモリアロケータなどの一部のクラスに限定されるでしょう。
32ビットCPUでの使用を推奨します。 16ビットCPUでも動作するように考慮していますが動作評価は基本的に行っていません。
利用者のターゲットとしては次の3タイプを想定しています。

- 組込み屋で今までC言語、アセンブラで開発していたが、これから C++に切り替えていこうとしている人
- PCアプリケーションの世界から組込みソフトウェアの世界にやってきた人
- 組込みソフトウェアで C++を活用している人

コンセプト

何よりも実践的に使えるフレームワークであることを目指しています。 フレームワークをデザインする上で様々な考慮すべき点がありますが、多くのトレードオフを決定する際の基準は以下のようなものです。

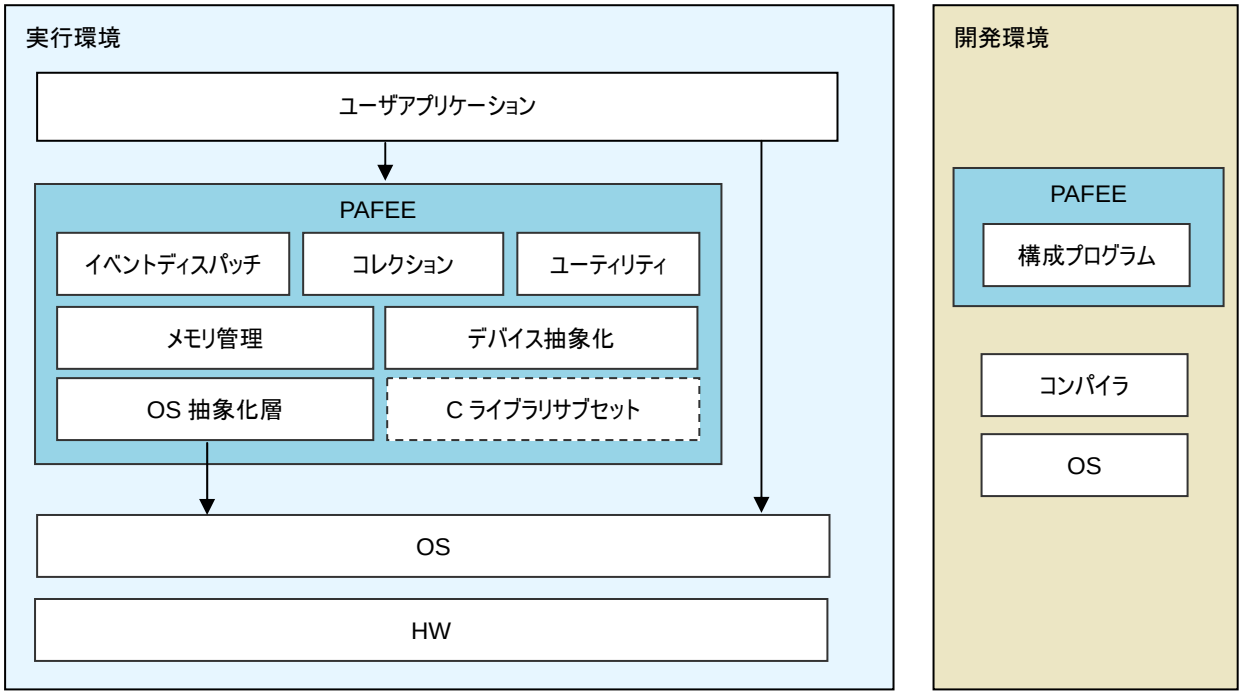
パフォーマンス ＞ 使い易さ・分かり易さ ＞ 敷居の低さ ＞ オブジェクト指向的美しさ

STL は優れたライブラリですが、テンプレートを多用しジェネリックプログラミングというオブジェクト指向とはまた別の概念が必要です。
今までC言語だけやってきた組込み屋がいきなり習得するには敷居が高いように見えます。 そこで PAFEE ではテンプレートを必須とせず、必要に応じて使えるように設計されています。 また PC のアプリケーションの世界からやってきた人が不自由なく組込みソフトウェアを実装できるように一通りのコレクションクラスや文字列クラスを提供しています。

2. システム構成

PAFEE は大きく分けて、イベントディスパッチ、コレクション、ユーティリティ、メモリ管理、デバイス抽象化、OS抽象化、Cライブラリサブセットから構成されます。 ライブラリではなくソースコードの形で提供され、ユーザアプリケーションの Makefie に組み込む形で使用します。 必要なクラスに関連するファイルを収集するために PAFEE 構成プログラムを提供しています。

PAFEE がタスクや排他制御などの OS の機能を利用する場合には OS 抽象化層のクラスを利用しています。
現在では ITRON(TOPPERS), Linux,Windows が利用可能です。 OS 抽象化層の持つ機能は限定的です。
基本的に PAFEE が必要とする API だけをサポートしています。



サポート対象の OS ： ITRON(TOPPERS JSP)、Windows2000/Xp/Vista、Linux2.4/2.6 系（pthread 必須）、
推奨 CPU: 32ビット CPU
（int 長が32ビットの場合に最もパフォーマンスがでる、16ビット、64ビット CPU では評価しない）

2.1 開発環境

2.1.1 コンパイラ

特にコンパイラは限定していません。 PAFEE の評価は Linux,ITRON 環境においては GNU の g++を使用し、Windows 環境では VisualC++6.0/2008 を使用しています。

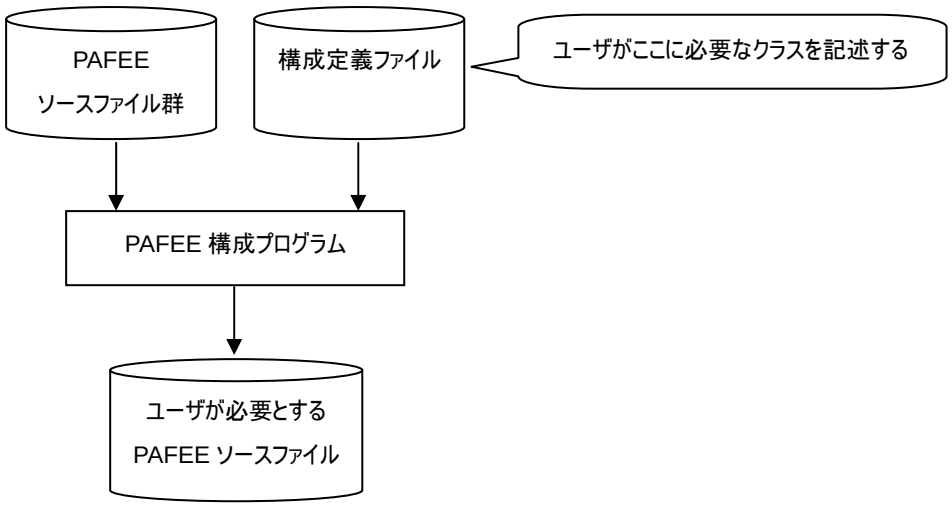
2.1.2 ホスト OS

主に Windows 上での開発を想定していますが Linux 上でも可能です。 ただし構成プログラムは Windows 版のみ提供されます。

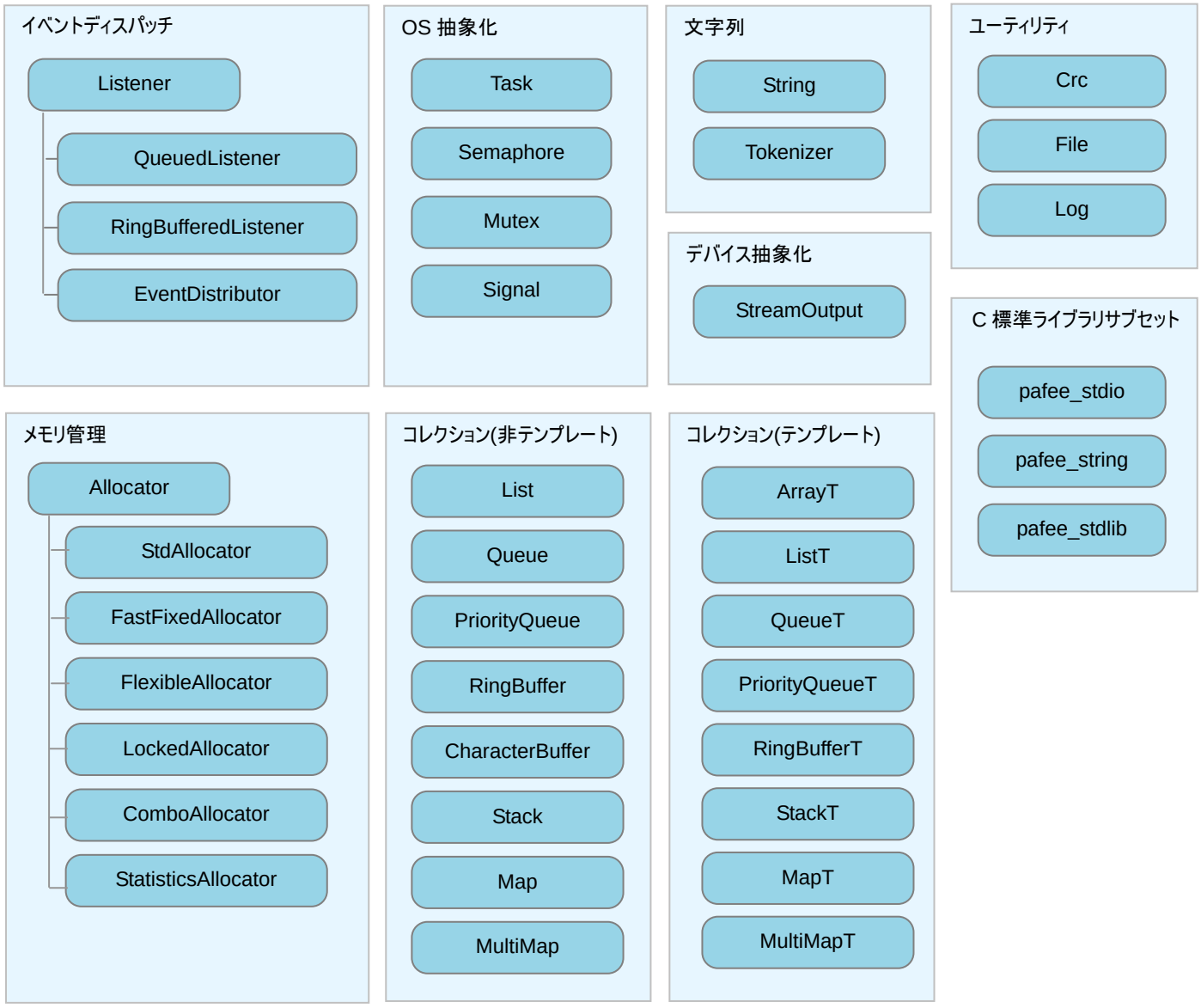
2.1.3 開発の流れ

PAFEEを使ったアプリケーション開発の流れは以下のようになります。
より詳細な説明は別紙『PAFEE 構成ツールマニュアル』を参照下さい。

1） ユーザが構成定義ファイルに必要なクラスを記述します。
2） PAFEE 構成プログラムを実行します。
3） 必要なファイルが収集され、出力される（ファイル間の依存関係が考慮されている）
4） ユーザアプリケーションの Makefile に出力された PAFEE のファイルを組み入れる（もしくはユーザ自身でライブラリ化する）
5） コンパイラでビルドし実行ファイルを生成する。

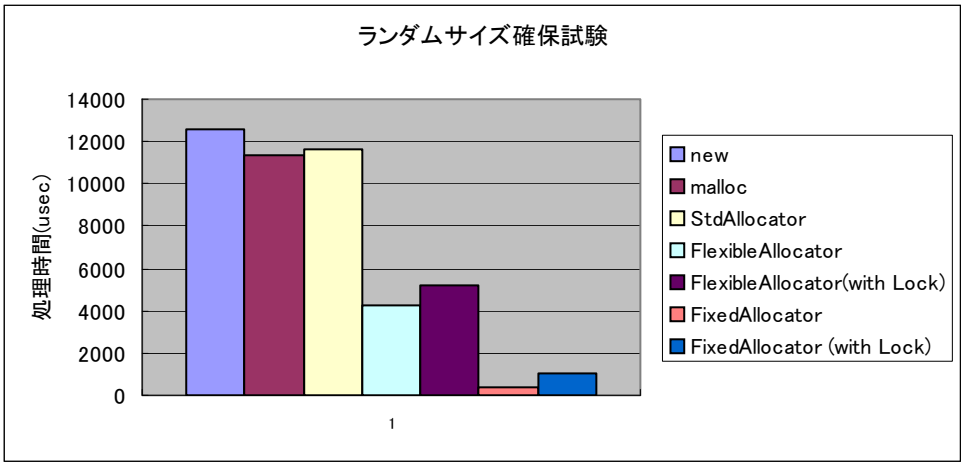


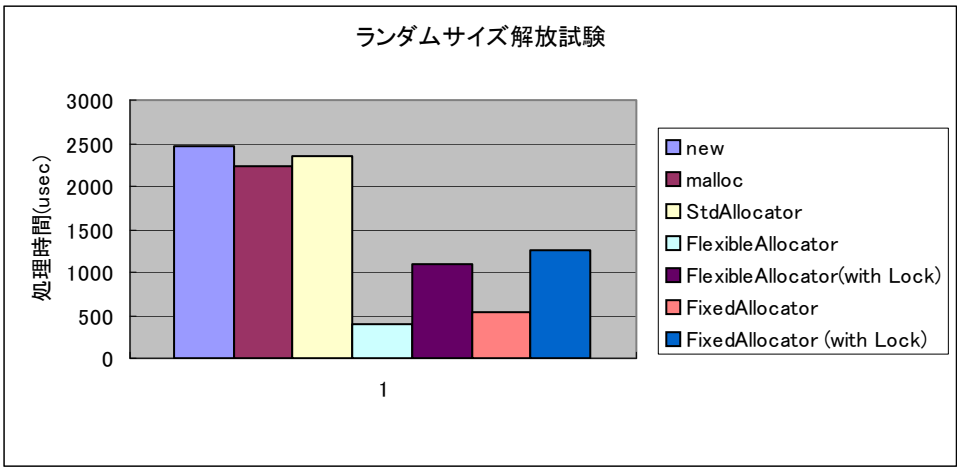
2.2 クラス構成


イベントディスパッチ
Listener
QueuedListener
RingBufferedListener
EventDistributor
OS 抽象化
Task
Semaphore
Mutex
Signal
文字列
String
Tokenizer
デバイス抽象化
StreamOutput
ユーティリティ
Crc
File
Log
C 標準ライブラリサブセット
pafee_stdio
pafee_string
pafee_stdlib
メモリ管理
Allocator
StdAllocator
FastFixedAllocator
FlexibleAllocator
LockedAllocator
ComboAllocator
StatisticsAllocator
コレクション(非テンプレート)
List
Queue
PriorityQueue
RingBuffer
CharacterBuffer
Stack
Map
MultiMap
コレクション(テンプレート)
ArrayT
ListT
QueueT
PriorityQueueT
RingBufferT
StackT
MapT
MultiMapT

3. メモリ管理

PAFEE のメモリ管理の特徴は高いパフォーマンスと柔軟にカスタマイズできる点です。
以下の評価データは Linux を搭載した ARM9 ボード(108MHz 動作)でのパフォーマンスを評価した結果です。

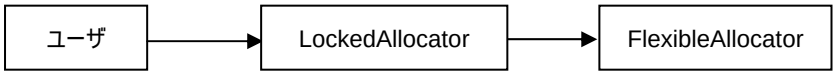




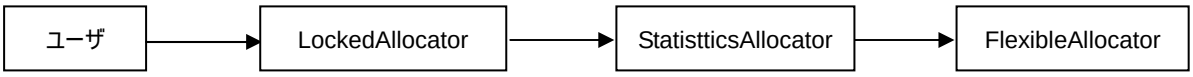
標準の new や malloc に対して非常に高速に動作しています。ただし MMU は利用することは出来ません。
OS に付随して提供される標準的なアロケータでは様々なパターンに対してある程度のパフォーマンスを発揮できるように設計されているため、実行速度を優先したい場合やメモリの利用効率を高めたいといったニーズに細かく対応することが出来ません。
PAFEE は目的別のアロケータを提供するとともに、それらを柔軟に組み合わせる仕組みが用意されています。
バックエンドのアロケータとして標準のアロケータを使用することもできるので既存のメモリ管理機構と共存することも出来ます。

シチュエーション毎に利用例を解説していきます。

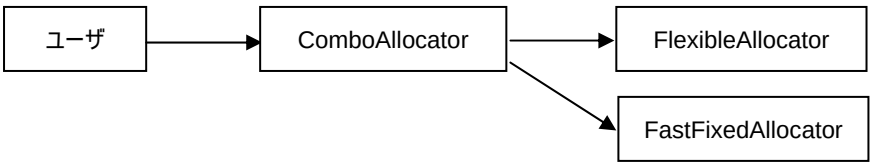
PAFEE のアロケータは排他制御機能を持ちません。 必要な場合には排他制御機能を提供する LockedAllocator を間に挟む形で使います。 以下の例では FlexibleAllocator に排他制御機能を持たせています。
これは LockedAllocator の Initialize 時にバックエンドのアロケータとして FlexibleAllocator を渡すことで実現できます。



さらに統計情報を取りたい場合は StatistticsAllocator を間に挿入します。 これは LockedAllocator のバックエンドアロケータに StatistticsAllocator を、StatistticsAllocator のバックエンドに FlexibleAllocator を指定することで実現できます



また ComboAllocator を使えば基本的には FlexibleAllocator を使い、特定のサイズ要求があったときだけ、固定長の FastFixedAllocator を使うといった形態も可能です。



このように用途に応じた最適な形でメモリ管理を構成することが可能です。

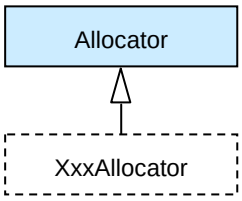
PAFEE では以下のメモリアロケータを提供しています。

Allocator	全てのアロケータの基底クラスとなるアロケータ。 実装は持ちません
StdAllocator	malloc/free の単なるラッパクラス
FastFixedAllocator	固定長メモリアロケータ。 非常に高速ですが、場合によってはメモリ利用効率が落ちます。
FlexibleAllocator	可変長メモリアロケータ。 メモリ利用効率がよいですが FastFixedAllocator より低速です。
ComboAllocator	複数のアロケータを組み合わせるアロケータ。他のアロケータの前段として使用します。
LockedAllocator	排他制御機能を提供するアロケータ。他のアロケータの前段として使用します。
StatisticsAllocator	統計情報機能を提供するアロケータ。他のアロケータの前段として使用します。

3.1 Allocator

メモリアロケータを抽象化するクラスです。

実装をもたず、アロケータの統一的なインタフェースを提供します。



3.1.1 クラスメンバー覧

コンストラクタ

Allocator	デフォルトコンストラクタ

操作

なし

オーバライド可能な関数

Allocate	メモリを割り当てます
Deallocate	メモリを解放します
IsOwnAddress	指定アドレスが自身の管理するメモリ領域かチェックします

データメンバ

なし

3.1.2 クラスメンバ詳細

Allocator

Allocator オブジェクトを構築します。

```
Allocator::Allocator ();
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します。

Allocate

メモリを割り当てます

```
void* Allocator::Allocate(
                        unsigned long    size ///< [in] size to allocate
                        )
```

パラメータ

size

[in] 確保するサイズを指定します。

戻り値

成功すれば割り当てたメモリのポインタを返します。

失敗すれば NULL を返します。

例外 (PAFEE_EXCEPTION が定義されている場合のみ)

派生クラスの実装によります

解説

メモリを割り当てます。

Deallocate

メモリを解放します

```
void Allocator::Deallocate(
                        void* ptr ///< [in] poiner to deallocate
                        )
```

パラメータ

ptr

[in] 解放するメモリのポインタ。 NULL を指定した場合には何もしません。

戻り値

なし

例外 (PAFEE_EXCEPTION が定義されている場合のみ)

派生クラスの実装によります

解説

メモリを解放します

IsOwnAddress

指定アドレスがそのオブジェクト自身が管理するメモリ領域かチェックします

```
bool Allocator::IsOwnAddress(
                        void* ptr ///< [in] pointer to check
                        )
```

パラメータ

ptr

[in] チェックするポインタ

戻り値

オブジェクト自身が管理するメモリの場合は true、そうでなければ false を返します。

例外 (PAFEE_EXCEPTION が定義されている場合のみ)

派生クラスの実装によります

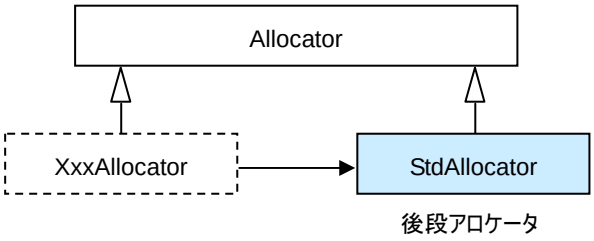
解説

指定アドレスがそのオブジェクト自身が管理するメモリ領域かチェックします

3.2 StdAllocator

malloc/free をラップするアロケータ。

最後段のアロケータまたはメモリプール確保用としての利用を想定しています。



3.2.1 クラスメンバー覧

コンストラクタ

Allocator	デフォルトコンストラクタ

操作

Malloc	メモリを割り当てる静的なメンバです。
Free	メモリを解放する静的なメンバです。

オーバライド可能な関数

Allocate	メモリを割り当てます
Deallocate	メモリを解放します
IsOwnAddress	指定アドレスが自身の管理するメモリ領域かチェックします

データメンバ

なし

3.2.2 クラスメンバ詳細

StdAllocator

StdAllocator オブジェクトを構築します。

```
StdAllocator::StdAllocator ();
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します

Malloc

メモリを割り当てます

```
void* StdAllocator::Malloc(
                        unsigned long    size ///< [in] size to allocate
                        )
```

パラメータ

size

[in] 確保するサイズを指定します。

戻り値

成功すれば割り当てたメモリのポインタを返します。

失敗すれば NULL を返します。

例外

なし

解説

malloc を使ってメモリを割り当てます。静的メンバ関数です。

Free

メモリを解放します

```
void StdAllocator::Free(
                        void* ptr ///< [in]    pointer to deallocate
                        )
```

パラメータ

ptr

[in] 解放するメモリのポインタ。 NULL を指定した場合には何もしません。

戻り値

なし

例外

なし

解説

free を使ってメモリを解放します。静的メンバ関数です。

Allocate

メモリを割り当てます

```
void* StdAllocator::Allocate(
                        unsigned long    size ///< [in] size to allocate
                        )
```

パラメータ

size

[in] 確保するサイズを指定します。

戻り値

成功すれば割り当てたメモリのポインタを返します。

失敗すれば NULL を返します。

例外

なし

解説

malloc を使ってメモリを割り当てます。

Deallocate

メモリを解放します

```
void StdAllocator::Deallocate(
                        void* ptr ///< [in]    pointer to deallocate
                        )
```

パラメータ

ptr

[in] 解放するメモリのポインタ。 NULL を指定した場合には何もしません。

戻り値

なし

例外

なし

解説

free を使ってメモリを解放します

IsOwnAddress

指定アドレスがそのオブジェクト自身が管理するメモリ領域かチェックします

```
bool StdAllocator::sOwnAddress(
                        void* ptr ///< [in]    pointer to check
                        )
```

パラメータ

ptr

[in] チェックするポインタ

戻り値

オブジェクト自身が管理するメモリの場合は true、そうでなければ false を返します。

例外 (PAFEE_EXCEPTION が定義されている場合のみ)

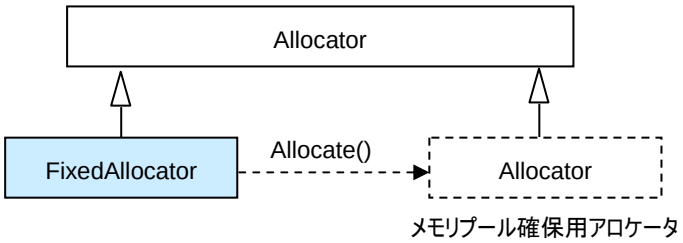
派生クラスの実装によります

解説

指定アドレスがそのオブジェクト自身が管理するメモリ領域かチェックします

3.3 FastFixedAllocator

固定長メモリプールを扱うアロケータです。 予め設定した固定サイズのメモリを割り当てることができます。
Addison-Welsey Longman 氏が考案した Loki::SmallObjAllocator のアルゴリズムをベースにしており、O（1）時間での高速なメモリの割当て／解放が可能です。 またブロック単位での管理領域も消費しません。
Allocator クラスを継承し割当、解放関連のメンバをオーバライドしています。
初期化時にはメモリプールを確保するアロケータを指定することも可能です。



3.3.1 クラスメンバー覧

コンストラクタ

FastFixedAllocator	デフォルトコンストラクタ

操作

Initialize	初期化します

オーバライド可能な関数

Allocate	メモリを割り当てます
Deallocate	メモリを解放します
IsOwnAddress	指定アドレスが自身の管理するメモリ領域かチェックします

データメンバ

なし

3.3.2 クラスメンバー覧

FastFixedAllocator

FastFixedAllocator オブジェクトを構築します。

コンストラクトした後は Initialize()を使って管理するメモリ領域を割り当てる必要があります。

```
FastFixedAllocator ();
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します

Initialize

メモリを確保して管理できるように初期化します。

```
bool FixedAllocator::Initialize(
                                    unsigned long blockSize,        ///< [in] block size
                                    unsigned long blockCount,       ///< [in] block count (max 65535)
                                    Allocator* pAllocator           ///< [in] allocator
)
```

パラメータ

blockSize

[in] ブロックサイズを指定します。

blockCount

[in] ブロック数を指定します。最大 65535 が指定可能です。

pAllocator

[in] メモリプールを確保するアロケータを指定します。

戻り値

成功すれば true を返します。

失敗すれば false を返します。

例外

本関数内で例外をスローすることはありませんが、指定したアロケータが例外をスローする場合があります。

解説

指定されたアロケータからブロックサイズxブロック数分のメモリを確保し管理します。

Allocate

メモリを割り当てます

```
void* FixedAllocator::Allocate(
                        unsigned long    size ///< [in] size to allocate
                        )
```

パラメータ

size

[in] 確保するサイズを指定します。 ただし固定長のメモリ管理なのでブロックサイズ未満のサイズを指定しても、ブロックサイズのメモリが割り当てられます。 ブロックサイズを超えたサイズを指定した場合は NULL を返します。

戻り値

成功すれば割り当てたメモリのポインタを返します。

失敗すれば NULL を返します。

例外 (PAFEE_EXCEPTION が定義されている場合のみ)

空きメモリが無い場合に std::bad_alloc()例外をスローします。

解説

メモリを割り当てます。

Deallocate

メモリを解放します

```
void FixedAllocator::Deallocate(
                        void* ptr ///< [in] pointer to deallocate
                        )
```

パラメータ

ptr

[in] 解放するメモリのポインタ。 NULL を指定した場合には何もしません。

戻り値

なし

例外

なし

解説

メモリを解放します

IsOwnAddress

指定アドレスがそのオブジェクト自身が管理するメモリ領域かチェックします

```
bool FixedAllocator::IsOwnAddress(
                        void* ptr ///< [in]    pointer to check
                        )
```

パラメータ

ptr

[in] チェックするポインタ

戻り値

オブジェクト自身が管理するメモリの場合は true、そうでなければ false を返します。

例外

なし

解説

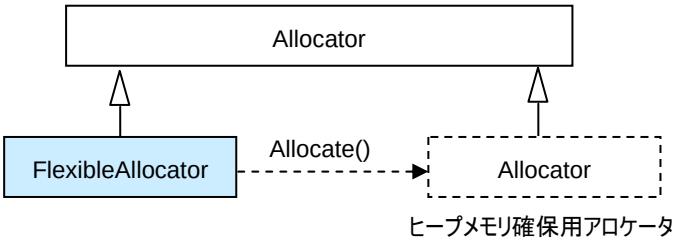
指定アドレスがそのオブジェクト自身が管理するメモリ領域かチェックします

3.4 FlexibleAllocator

可変長のメモリ管理を行うことのできるアメモリの利用効率が高いアロケータです。

O（N）時間でのメモリの割当て、O（1）時間での解放が可能です。 空きメモリが十分にある状態だとメモリ割当てもO（1）時間で動作可能です。 アロケーション毎に管理領域として24バイト消費します。

できるだけ断片化が発生しにくいように考慮されています。



3.4.1 クラスメンバー覧

コンストラクタ

FlexibleAllocator	デフォルトコンストラクタ

操作

Initialize	初期化します

オーバライド可能な関数

Allocate	メモリを割り当てます
Deallocate	メモリを解放します
IsOwnAddress	指定アドレスが自身の管理するメモリ領域かチェックします

データメンバ

なし

3.4.2 クラスメンバー覧

FlexibleAllocator

FlexibleAllocator オブジェクトを構築します。

コンストラクトした後は Initialize()を使って管理するメモリ領域を割り当てる必要があります。

```
FlexibleAllocator::FlexibleAllocator
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します

Initialize

メモリを確保して管理できるように初期化します。

アロケータを渡してそこから管理対象のヒープメモリを確保するバージョンと予め用意した領域のアドレスを渡すバージョンがあります。

```
bool FlexibleAllocator::Initialize(
                        unsigned long heapSize,
                        unsigned long maxAllocateSize,
                        unsigned long minBlockSize,
                        Allocator* pAllocator=NULL
                        );
void FlexibleAllocator::Initialize(
                        unsigned long heapAddr,
                        unsigned long heapSize,
                        unsigned long maxAllocateSize,
                        unsigned long minBlockSize
                        );
```

パラメータ

heapSize

[in] 管理対象のヒープサイズ指定します。

heapAddr

[in] 管理対象のヒープアドレスを指定します。

minAllocateSize

[in] 最低限割り当てるサイズを指定します。

maxAllocateSize

[in] ブロック数を指定します。最大 65535 が指定可能です。

pAllocator

[in] ヒープを確保するアロケータを指定します。

戻り値

成功すれば true を返します。

失敗すれば false を返します。

例外

なし

解説

指定されたアロケータからブロックサイズxブロック数分のメモリを確保し管理します。

Allocate

メモリを割り当てます

```
void* FlexibleAllocator::Allocate(
                        unsigned long    size ///< [in] size to allocate
                        )
```

パラメータ

size

[in] 確保するサイズを指定します。

戻り値

成功すれば割り当てたメモリのポインタを返します。

失敗すれば NULL を返します。

例外 (PAFEE_EXCEPTION が定義されている場合のみ)

空きメモリが無い場合に std::bad_alloc()例外をスローします。

解説

メモリを割り当てます。

Deallocate

メモリを解放します

```
void FixedAllocator::Deallocate(
                        void* ptr ///< [in] pointer to deallocate
                        )
```

パラメータ

ptr

[in] 解放するメモリのポインタ。 NULL を指定した場合には何もしません。

戻り値

なし

例外

なし

解説

メモリを解放します

IsOwnAddress

指定アドレスがそのオブジェクト自身が管理するメモリ領域かチェックします

```
bool FixedAllocator::IsOwnAddress(
                        void* ptr ///< [in]    pointer to check
                        )
```

パラメータ

ptr

[in] チェックするポインタ

戻り値

オブジェクト自身が管理するメモリの場合は true、そうでなければ false を返します。

例外

なし

解説

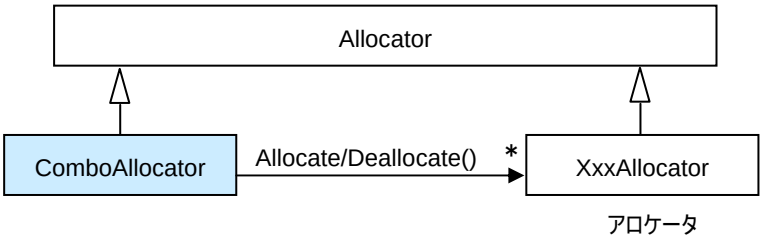
指定アドレスがそのオブジェクト自身が管理するメモリ領域かチェックします

3.5 ComboAllocator

複数のアロケータを束ねるアロケータです。

異なるブロックサイズを持つ複数の FixedAllocator を登録すれば、確保要求サイズに応じた適切なアロケータを選択してメモリを確保させることができます。

ComboAllocator 自体はメモリを確保することは出来ないので AddAllocato()を使ってアロケータを登録しておく必要があります。



3.5.1 クラスメンバー覧

コンストラクタ

ComboAllocator	デフォルトコンストラクタ

操作

AddAllocator	アロケータを追加します

オーバライド可能な関数

Allocate	メモリを割り当てます
Deallocate	メモリを解放します
IsOwnAddress	指定アドレスが自身の管理するメモリ領域かチェックします

データメンバ

なし

3.5.2 クラスメンバ詳細

ComboAllocator

ComboAllocator オブジェクトを構築します。

```
ComboAllocator::ComboAllocator ();
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します

AddAllocator

アロケータを追加します。

```
bool ComboAllocator::AddAllocator (
                        Allocator* pAllocator,                  ///< [in] Allocator
                        unsigned long minAllocationSize,    ///< [in] minimum allocation size of the allocator
                        unsigned long maxAllocationSize    ///< [in] maximum allocation size of the allocator
)
```

パラメータ

pAllocator

[in] 登録するアロケータのポインタ

minAllocationSize

[in] 登録するアロケータが割当可能な最小メモリサイズを指定します

maxAllocationSize

[in] 登録するアロケータが割当可能な最大メモリサイズを指定します

戻り値

成功すれば true を返します。

失敗すれば false を返します。

例外

なし

解説

アロケータを登録します。Allocate()が呼ばれたときにはここで指定されたサイズから適切なアロケータを選択します。

最大10個のアロケータを登録することができます。

Allocate

メモリを割り当てます

```
void* ComboAllocator::Allocate(
                        unsigned long    size ///< [in]    size to allocate
                        )
```

パラメータ

size

[in] 確保するサイズを指定します。

戻り値

成功すれば割り当てたメモリのポインタを返します。

失敗すれば NULL を返します。

例外

追加されたアロケータの実装によります

解説

追加されたアロケータの中から要求サイズを割り当てられるアロケータを選択してメモリを確保します。

アロケータは追加された順番に検索し、最初に要求サイズを割当可能なアロケータから確保します。

Deallocate

メモリを解放します

```
void ComboAllocator::Deallocate(
                        void* ptr ///< [in]    pointer to deallocate
                        )
```

パラメータ

ptr

[in] 解放するメモリのポインタ。 NULL を指定した場合には何もしません。

戻り値

なし

例外

追加されたアロケータの実装によります。

解説

指定ポインタを管理しているアロケータを選択し、解放させます。

IsOwnAddress

指定アドレスがそのオブジェクト自身が管理するメモリ領域かチェックします

```
bool ComboAllocator::IsOwnAddress(
                        void* ptr ///< [in]    pointer to check
                        )
```

パラメータ

ptr

[in] チェックするポインタ

戻り値

オブジェクト自身が管理するメモリの場合は true、そうでなければ false を返します。

例外 (PAFEE_EXCEPTION が定義されている場合のみ)

追加されたアロケータの実装によります

解説

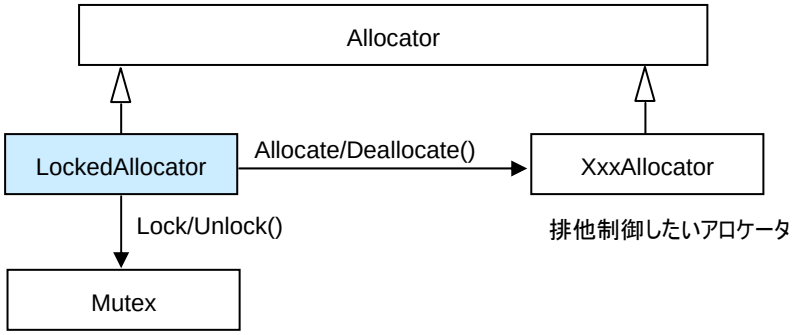
追加されたアロケータの中に渡されたアドレスを管理するものがあるかチェックします。

3.6 LockedAllocator

排他制御機構を提供するアロケータです。

排他制御したいアロケータを本クラスに登録して使います。

Allocate と Deallocate を一つのミューテックスを使って排他制御します。



3.6.1 クラスメンバー覧

コンストラクタ

LockedAllocator	デフォルトコンストラクタ

操作

Initialize	初期化します。

オーバライド可能な関数

Allocate	メモリを割り当てます
Deallocate	メモリを解放します
IsOwnAddress	指定アドレスが自身の管理するメモリ領域かチェックします

データメンバ

なし

3.6.2 クラスメンバ詳細

LockedAllocator

LockedAllocator オブジェクトを構築します。

```
LockedAllocator::LockedAllocator ();
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します

Initialize

メモリを確保して管理できるように初期化します。

```
bool LockedAllocator::Initialize(
                                    unsigned long blockSize,        ///< [in] block size
                                    unsigned long blockCount,       ///< [in] block count (max 65535)
                                    Allocator* pBackendAllocator    ///< [in] back-end allocator
)
```

パラメータ

blockSize

[in] ブロックサイズを指定します。

blockCount

[in] ブロック数を指定します。

pBackendAllocator

[in] 後段アロケータを指定します。 NULL を指定した場合は標準の StdAllocator が使われます。

戻り値

成功すれば true を返します。

失敗すれば false を返します。

例外 (PAFEE_EXCEPTION が定義されている場合のみ)

本関数内で例外をスローすることはありませんが、後段アロケータが例外をスローする場合があります。

解説

指定された後段アロケータからブロックサイズxブロック数分のメモリを確保し管理します。

Allocate

メモリを割り当てます

```
void* LockedAllocator::Allocate(
                        unsigned long    size ///< [in] size to allocate
                        )
```

パラメータ

size

[in] 確保するサイズを指定します。

戻り値

成功すれば割り当てたメモリのポインタを返します。

失敗すれば NULL を返します。

例外

登録されたアロケータの実装によります

解説

排他制御をかけて登録されたアロケータの Allocate()を呼び出します。

Deallocate

メモリを解放します

```
void LockedAllocator::Deallocate(
                        void* ptr ///< [in]    pointer to deallocate
                        )
```

パラメータ

ptr

[in] 解放するメモリのポインタ。 NULL を指定した場合には何もしません。

戻り値

なし

例外

登録されたアロケータの実装によります。

解説

排他制御をかけて登録されたアロケータの Deallocate()を呼び出します。

IsOwnAddress

指定アドレスがそのオブジェクト自身が管理するメモリ領域かチェックします

```
bool LockedAllocator::IsOwnAddress(
                        void* ptr ///< [in]    pointer to check
                        )
```

パラメータ

ptr

[in] チェックするポインタ

戻り値

オブジェクト自身が管理するメモリの場合は true、そうでなければ false を返します。

例外

登録されたアロケータの実装によります

解説

登録されたアロケータの中に渡されたアドレスを管理するものがあるかチェックします。

3.7 StatisticsAllocator

統計情報を記録するアロケータです。

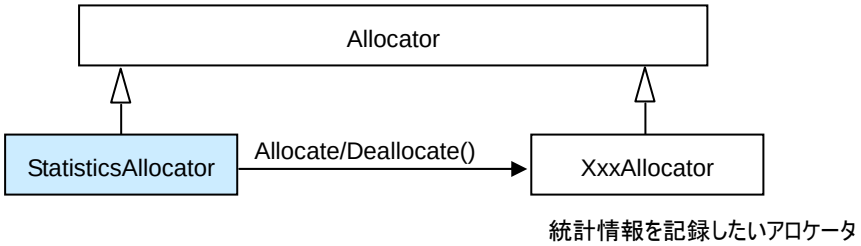
メモリの確保サイズ別の要求回数を記録、表示することができます。

記録対象のアロケータを本クラスに登録して使います。

以下の統計情報を記録できます。

- メモリ確保要求数
- メモリ解放要求数
- 特定のサイズ範囲でのメモリ確保要求数

  （この情報から要求頻度の高いサイズを知ることができます。 サイズ範囲は初期化時に指定する解像度ビット数で指定します）



3.7.1 クラスメンバー覧

コンストラクタ

StatisticsAllocator	デフォルトコンストラクタ

操作

Initialize	初期化します。
GetStatistics	統計情報を取得します
SetAllocator	記録対象のアロケータを設定します
SetResolution	

オーバライド可能な関数

Allocate	メモリを割り当てます
Deallocate	メモリを解放します
IsOwnAddress	指定アドレスが自身の管理するメモリ領域かチェックします

データメンバ

なし

3.7.2 クラスメンバ詳細

StatisticsAllocator

StatisticsAllocator オブジェクトを構築します。

```
StatisticsAllocator::StatisticsAllocator ();
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します

Initialize

初期化します。

```
bool StatisticsAllocator::Initialize(
                        int resolutionBits,             ///< [in] record resolution bits
                        Allocator* pAllocator           ///< [in] allocator
)
```

パラメータ

resolutionBits

[in] 記録するメモリ確保サイズの解像度をビットで指定します。

例えばここに 4 を指定すると 4bit 幅、16 バイトの解像度で記録します。

pAllocator

[in] 記録対象のアロケータを指定します

戻り値

成功すれば true を返します。

失敗すれば false を返します。

例外

なし

解説

指定したアロケータを登録し記録解像度を設定します。

記録できるメモリ確保要求サイズのバリエーションは最大256個までです。

例えば解像度を1に設定した場合には、そのサイズの要求を正確にカウントすることができますが、256以上のサイズのバリエーションがある場合には256個を超えたサイズは記録できません。

解像度に4を設定した場合は記録サイズが16バイトの精度になり20バイトや24バイトの確保要求も同じ16バイトとしてカウントされます。 この場合は256x4個のサイズバリエーションが記録可能です。

確保するサイズのバリエーションが多いときには解像度を大きく設定することで記録範囲を広げることができます。

256以上のサイズバリエーションの確保要求が発生した場合には記録されず統計情報の overflow がカウントされます。

SetAllocator

記録対象のアロケータを登録します

```
void StatisticsAllocator::SetAllocator (
                        Allocator* pAllocator ///< [in] Allocator
                        )
```

パラメータ

pAllocator

[in] 記録対象のアロケータ

戻り値

なし

例外

なし

解説

指定されたアロケータを登録します

SetResolution

記録解像度を設定します

```
void StatisticsAllocator::SetResolution (
                        int resolutionBits      ///< [in] record resolution bits
)
```

パラメータ

resolutionBits

[in] 記録するメモリ確保サイズの解像度をビットで指定します。

例えばここに 4 を指定すると 4bit 幅、16 バイトの解像度で記録します。

戻り値

なし

例外

なし

解説

記録解像度を設定します。 詳細については Initialize()の解説を参照下さい。

Allocate

メモリを割り当てます

```
void* StatisticsAllocator::Allocate(
                        unsigned long    size ///< [in] size to allocate
                        )
```

パラメータ

size

[in] 確保するサイズを指定します。

戻り値

成功すれば割り当てたメモリのポインタを返します。

失敗すれば NULL を返します。

例外

登録されたアロケータの実装によります

解説

登録されたアロケータの Allocate()を呼び出して統計情報を更新します

Deallocate

メモリを解放します

```
void StatisticsAllocator::Deallocate(
                        void* ptr ///< [in]    pointer to deallocate
                        )
```

パラメータ

ptr

[in] 解放するメモリのポインタ。 NULL を指定した場合には何もしません。

戻り値

なし

例外

登録されたアロケータの実装によります。

解説

登録されたアロケータの Deallocate()を呼び出して統計情報を更新します

IsOwnAddress

指定アドレスがそのオブジェクト自身が管理するメモリ領域かチェックします

```
bool StatisticsAllocator::IsOwnAddress(
                        void* ptr ///< [in]    pointer to check
                        )
```

パラメータ

ptr

[in] チェックするポインタ

戻り値

オブジェクト自身が管理するメモリの場合は true、そうでなければ false を返します。

例外

登録されたアロケータの実装によります

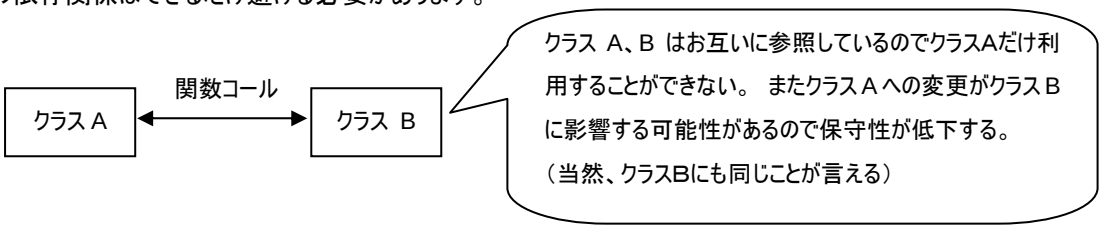
解説

登録されたアロケータの IsOwnAddress()をコールします

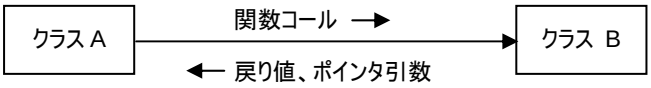
4. イベントディスパッチ

PAFEE のイベントディスパッチ機構を使うことによって、ソフトウェアの拡張性や保守性、再利用性を高めることができます。
JAVE でいうところの EventListener、 .NET フレームワークでのイベント機構に近い機能を提供しますが、それらと大きく異なる点はイベントの伝達方法を柔軟にスタマイズできるというところです。
以降に PAFEE のイベントディスパッチモデルに至るまでの流れを説明していきます。

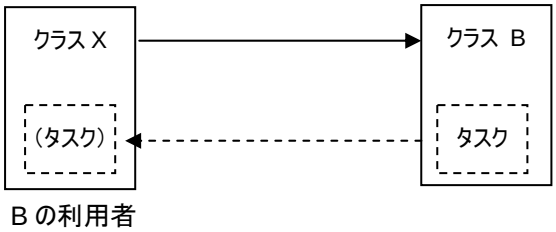
ソフトウェアの拡張性や保守性、再利用性を高めるにはモジュール間の結合度を疎にすることがポイントです。
双方向の依存関係はできるだけ避ける必要があります。



ここで依存関係を片方向にすると拡張性や保守性、再利用性が大きく向上しますがクラス間で双方向の情報のやりとりがある場合、情報をどうやって取得するかという問題がでてきます。
全て同期的に処理できるのであれば、下図のように関数の戻り値、または引数でポインタ渡しといった手段でクラス B からの情報を引き出すことが出来ます。



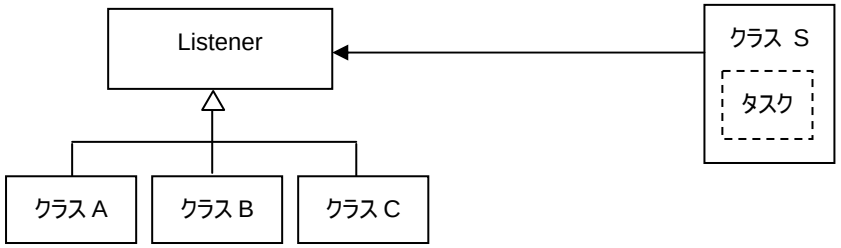
しかし、結果がクラス B から非同期に返される場合はこのモデルではうまくいきません。
そこで OS の提供するメッセージ通信の仕組みを使えば結果を非同期に受取ることができます。
ITRON だとメールボックスやデータキュー、Windows だと Window メッセージ、Linux だとメッセージキューなどの仕組みになります。



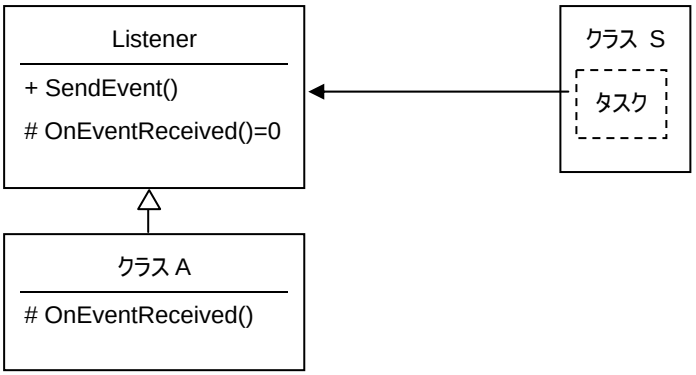
このモデルだとクラス B の利用者はメッセージを受取るハンドルまたは ID をクラス B に伝えてやればクラス B からのメッセージを受取ることが出来ます。 つまりメッセージを受取る仕組みがあればどのようなクラスでもクラス B が利用できます。
クラスBは利用者のクラスへの依存関係が無いので保守性、再生産性も向上します。
しかし、メッセージを受取るには常にタスクを必要とします。 例えばメッセージを受取って単に LED を点灯させるだけの処理でもタスクを用意する必要があり、クラス B 内のタスクの実行コンテキストで処理させるということが出来ません。
またメッセージ通信の仕組みも自由に選択できません。 クラス B がメールボックスで実装してあるなら、受け取り側もメールボックスで受信する必要があり、クラスBの実装によって通信方法が固定されることになります。

さらに異なる OS に移植する場合にはメッセージ通信部分を改修する作業が必要になってしまいます。
PAFEE ではこの問題を解決するため Listener という抽象クラスを提供します。

Listener クラスはイベント（ここからはメッセージの代わりにイベントという言葉を使います）を受取るクラスを抽象化します。
イベントを送信するクラス（ここではクラス S）は常に Listener に対してイベントを送信するようにします。
逆にイベントを受取りたいクラスは Listener から派生させて作ります。　つまりクラス S は Listener にだけ依存関係があり、Listener から派生したクラスには依存関係を持たないため、関連を疎にすることができます。

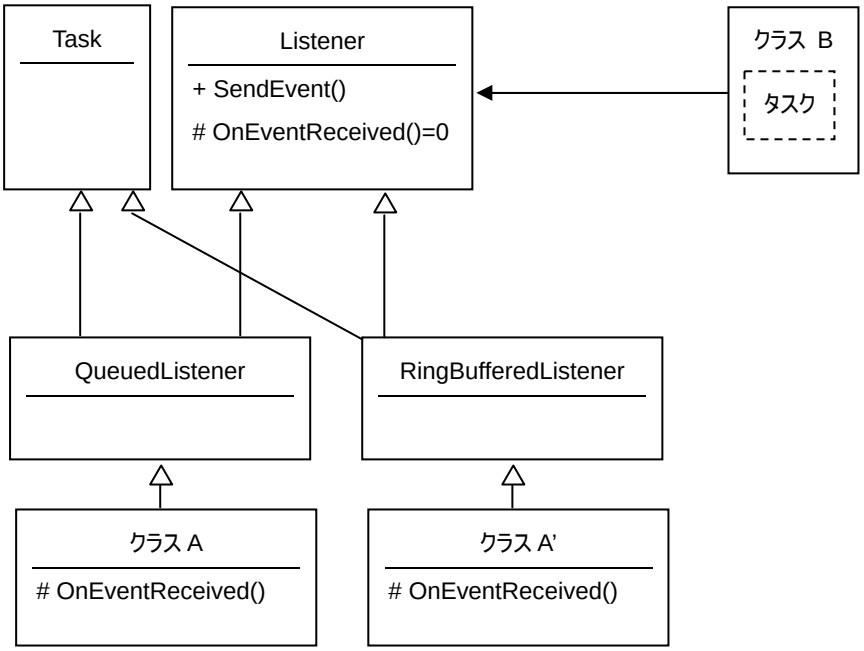


Listener は SendEvent()というメンバ関数と OnEventReceived()という純粋仮想関数だけを持つ非常に小さなクラスです。
SendEvent()の処理は単に OnEventReceived()をコールするだけです。 イベントを受取るクラス（ここではクラス A）は OnEventReceived()を実装し、イベントを処理します。 このモデルではイベントの処理は同期モデルになります。 つまりイベントはクラス B のタスクの実行コンテキストで処理されます。

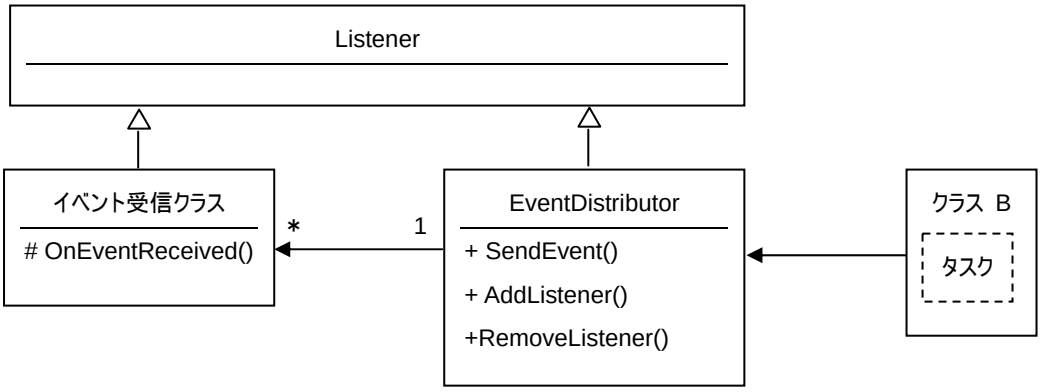


イベントを非同期に処理したい場合には Listener から継承した QueuedListener または RingBufferedListener を用います。　これらのクラスは Task クラスも継承しているのでクラス内のタスクでイベントを非同期に処理することができます。

QueuedListener と RingBufferedListener はイベントのキューイング機構が異なります。QueuedListener はイベントをリスト構造で保持し、メモリの利用効率が良いという特徴があります。 RingBufferedListener はその名が指すようにリングバッファでイベントをキューイングしています。 イベントパラメータのサイズにムラがある場合にはメモリの利用効率は良くないですが、非常に高速に動作します。



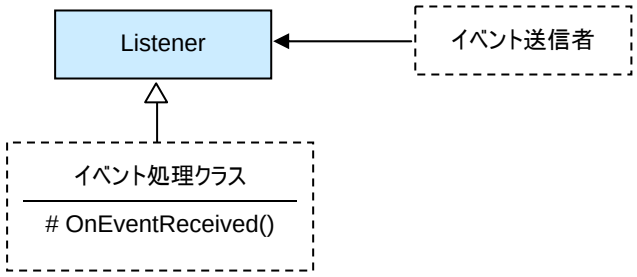
Java や.NET では addEventListener()のようなメソッドで複数のイベント送信先を登録できる仕組みを提供していますが、PAFEE の Listener には複数の宛先を登録する機能を持たせていません。 これは多くの場合、イベントの宛先は単一なのに常に複数のイベント配信のための処理で無駄なオーバヘッドが発生するのを避けるためです。 その代わりに EventDistributor というクラスを提供しています。 複数宛先へのイベント送信が必要な場合にはこの EventDistributor を間に挟む形で使います。



4.1 Listener

イベントの受信処理を抽象化するクラスです。

本クラスから直接派生した場合は同期モデル（イベント送信側の実行コンテキスト）でイベントを処理します。



4.1.1 クラスメンバー覧

コンストラクタ

Listener	デフォルトコンストラクタ

操作

なし

オーバライド可能な関数

SendEvent	イベントを本クラスに送信します
OnEventReceived	イベントハンドラです。 派生クラスで実装してください

データメンバ

なし

4.1.2 クラスメンバ詳細

Listener

コンストラクタ

```
Listener:: Listener ();
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します。

SendEvent

イベントを本クラスに送信します

```
bool Listener:: SendEvent (
                                    unsigned long event,    ///< [in] event id
                                    const void* pParam,   ///< [in] parameter buffer
                                    int paramLength         ///< [in] parameter length
)
```

パラメータ

event

[in] イベント ID

pParam

[in] イベントパラメータのポインタ

paramLength

[in] イベントパラメータ長

戻り値

true: 成功

false: 失敗

例外

派生クラスの実装によります

解説

イベントを本クラスに送信します。

OnEventReceived

イベントハンドラです。派生クラスで実装して下さい。

```
void Listener::OnEventReceived (
                                    unsigned long event,    ///< [in] event id
                                    const void* pParam,    ///< [in] parameter buffer
                                    int paramLength         ///< [in] parameter length
)
```

パラメータ

event

[in] イベント ID

pParam

[in] イベントパラメータのポインタ

paramLength

[in] イベントパラメータ長

戻り値

なし

例外

派生クラスの実装によります

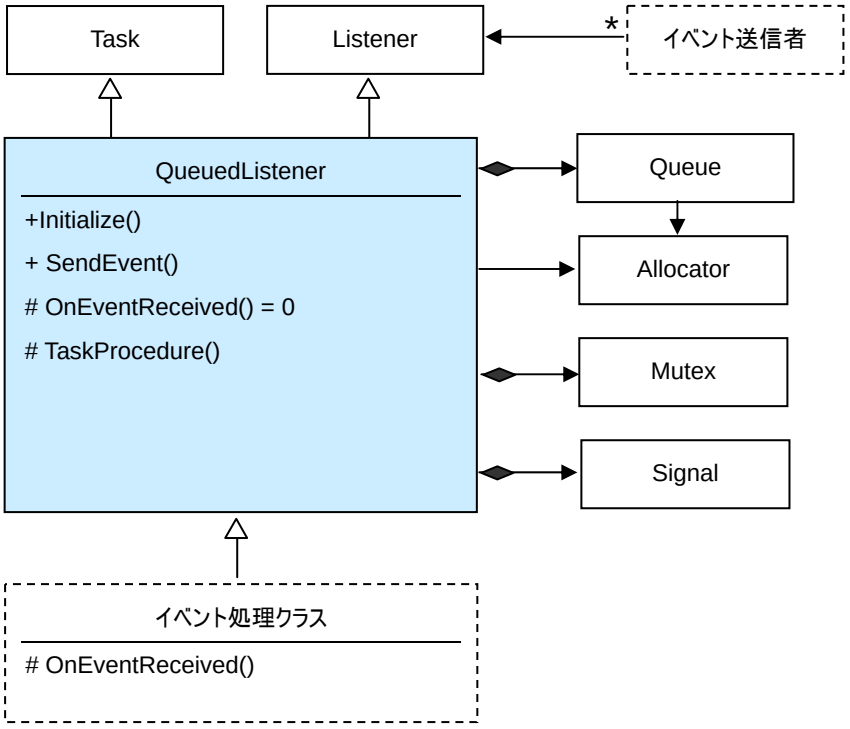
解説

イベントを受信した時に呼ばれます。

4.2 QueuedListener

イベントを受取って非同期に処理するクラスです。 イベントは Queue クラスのインスタンスに蓄積され、内部のクラスの実行コンテキストで処理されます。

本タスクから派生させて OnEventReceived()を実装するだけでイベントの非同期処理が実現できます。



4.2.1 クラスメンバー覧

コンストラクタ

QueuedListener	デフォルトコンストラクタ

操作

なし

オーバライド可能な関数

Initialize	初期化します
SendEvent	イベントを本クラスに送信します
OnEventReceived	イベントハンドラです。 派生クラスで実装してください

データメンバ

なし

4.2.2 クラスメンバ詳細

QueuedListener

```
QueuedListener:: QueuedListener ();
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します。

Initialize

初期化します

```
// For Windows/Linux
bool QueuedListener:: Initialize(
                            TASK_PRIORITY priority,   ///< [in] task priority
                            TASK_SCHEDULING schedule, ///< [in] task scheduling
                            int stackSize,            ///< [in] stack size for task
                            Allocator* pAllocator     ///< [in] allocator for queue
)

// For ITRON
bool QueuedListener::Initialize(
                            ID taskID,        ///< [in] task id
                            ID signalID,      ///< [in] signal id
                            ID mutexID,       ///< [in] mutex id
                            Allocator* pAllocator     ///< [in] allocator for queue
)
```

パラメータ

priority

[in] タスクの優先度。設定値は OS 依存です。

schedule

[in] スケジュリングポリシ。Linux でのみ有効で SCHED_OTHER, SCHED_RR, SCHED_FIFO が設定可能です。
Windows の場合には 0 を設定して下さい。

stackSize

[in] タスクのスタックサイズ

taskID

[in] コンフィグレータで設定したタスク ID

signalID

[in] コンフィグレータで設定したシグナル ID

mutexID

[in] コンフィグレータで設定したミューテックス ID

pAllocator

[in] リングバッファを確保するアロケータ

戻り値

true: 成功

false: 失敗

例外

本クラスでは発生させませんが、アロケータが例外をスローする可能性があります

解説

QueuedListener 内のタスク、キューを適切に初期化します。

イベント処理を開始するには本クラスが継承している PafeeTask クラスの Start メソッドをコールする必要があります。

SendEvent

イベントを本クラスに送信します。

```
bool QueuedListener:: SendEvent (
                                    unsigned long event,    ///< [in] event id
                                    const void* pParam,    ///< [in] parameter buffer
                                    int paramLength         ///< [in] parameter length
)
```

パラメータ

event

[in] イベント ID

pParam

[in] イベントパラメータのポインタ

paramLength

[in] イベントパラメータ長

戻り値

true: 成功

false: 失敗

例外

派生クラスの実装によります

解説

イベントを本クラスに送信します。 イベントは非同期に処理されます。

排他制御がかけられているので同時に複数のタスクからコールされても安全です。

OnEventReceived

イベントハンドラです。派生クラスで実装して下さい。

```
void QueuedListener::OnEventReceived (
                                    unsigned long event,    ///< [in] event id
                                    const void* pParam,   ///< [in] parameter buffer
                                    int paramLength         ///< [in] parameter length
)
```

パラメータ

event

[in] イベント ID

pParam

[in] イベントパラメータのポインタ

paramLength

[in] イベントパラメータ長

戻り値

なし

例外

派生クラスの実装によります

解説

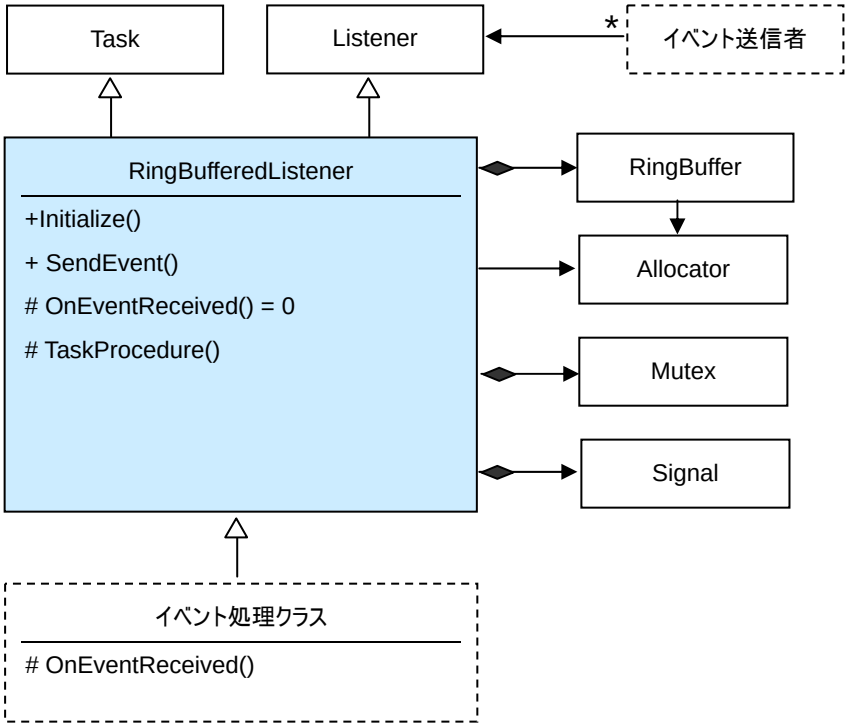
イベントを受信した時に呼ばれます。

4.3 RingBufferedListener

イベントを受取って非同期に処理するクラスです。 イベントは RingBuffer クラスのインスタンスに蓄積され、内部のクラスの実行コンテキストで処理されます。

本タスクから派生させて OnEventReceived()を実装するだけでイベントの非同期処理が実現できます。

リングバッファを用いているので QueuedListener より高速に動作しますが、イベントパラメータのサイズにバラツキがある場合にはメモリの利用効率がよくありません。



4.3.1 クラスメンバー覧

コンストラクタ

RingBufferedListener	デフォルトコンストラクタ

操作

なし

オーバライド可能な関数

Initialize	初期化します
SendEvent	イベントを本クラスに送信します
OnEventReceived	イベントハンドラです。 派生クラスで実装してください

データメンバ

なし

4.3.2 クラスメンバ詳細

RingBufferedListener

RingBufferdListener:: RingBufferdListener ()

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します。

Initialize

初期化します

```
// For Windows/Linux
bool RingBufferedListener::Initialize(
        int capacity,             ///< [in] capacity of ring buffer (max element count)
        int elementSize,          ///< [in] element size of ring buffer
        TASK_PRIORITY priority,   ///< [in] task priority
        TASK_SCHEDULING schedule =0,///< [in] task scheduling
        int stackSize = 0,        ///< [in] stack size
        Allocator* pAllocator=NULL///< [in] allocator for ring buffer
);
// For ITRON
bool RingBufferedListener::Initialize(
        int capacity,    ///< [in] capacity of ring buffer (max element count)
        int elementSize, ///< [in] element size of ring buffer
        ID taskID,       ///< [in] task ID
        ID signalID,     ///< [in] signal ID
        ID mutexID,      ///< [in] mutex ID
        Allocator* pAllocator=NULL///< [in] allocator for ring buffer
);
```

パラメータ

capacity

[in] リングバッファの容量(最大要素数)です。

elementSize

[in] リングバッファの要素の最大値を指定します。イベントの格納用に 4 バイト消費するので実際に扱うデータサイズに 4 を足した値を設定して下さい。

priority

[in] タスクの優先度。設定値は OS 依存です。

schedule

[in] スケジュリングポリシ。Linux でのみ有効で SCHED_OTHER, SCHED_RR, SCHED_FIFO が設定可能です。Windows の場合には 0 を設定して下さい。

stackSize

[in] タスクのスタックサイズ

taskID

[in] コンフィグレータで設定したタスク ID

signalID

[in] コンフィグレータで設定したシグナル ID

mutexID

[in] コンフィグレータで設定したミューテックス ID

pAllocator

[in] リングバッファを確保するアロケータ

戻り値

true: 成功

false: 失敗

例外

本クラスでは発生させませんが、アロケータが例外をスローする可能性があります

解説

QueuedListener 内のタスク、キューを適切に初期化します。

イベント処理を開始するには本クラスが継承している PafeeTask クラスの Start メソッドをコールする必要があります。

SendEvent

イベントを本クラスに送信します。

```
bool RingBufferedListener:: SendEvent (
                                    unsigned long event,    ///< [in] event id
                                    const void* pParam,    ///< [in] parameter buffer
                                    int paramLength         ///< [in] parameter length
)
```

パラメータ

event

[in] イベント ID

pParam

[in] イベントパラメータのポインタ

paramLength

[in] イベントパラメータ長

戻り値

true: 成功

false: 失敗

例外

派生クラスの実装によります

解説

イベントを本クラスに送信します。 イベントは非同期に処理されます。

排他制御がかけられているので同時に複数のタスクからコールされても安全です。

OnEventReceived

イベントハンドラです。派生クラスで実装して下さい。

```
void RingBufferedListener::OnEventReceived (
                                    unsigned long event,    ///< [in] event id
                                    const void* pParam,    ///< [in] parameter buffer
                                    int paramLength         ///< [in] parameter length
)
```

パラメータ

event

[in] イベント ID

pParam

[in] イベントパラメータのポインタ

paramLength

[in] イベントパラメータ長

戻り値

なし

例外

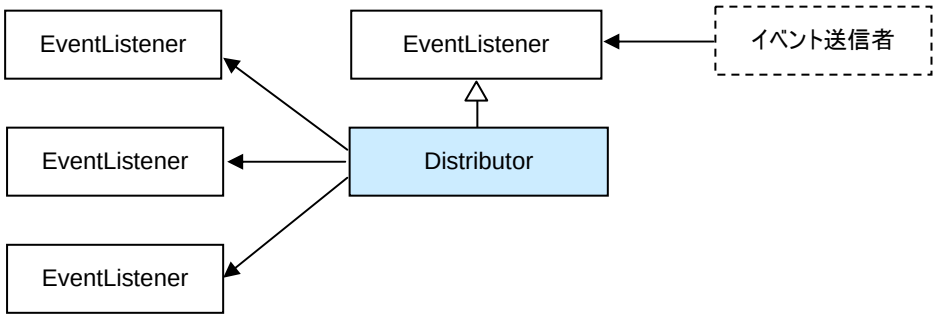
派生クラスの実装によります

解説

イベントを受信した時に呼ばれます。

4.4 EventDistributor

イベントを登録された複数の宛先に配信するクラスです。



4.4.1 クラスメンバー覧

コンストラクタ

Distributor	デフォルトコンストラクタ

操作

なし

オーバライド可能な関数

AddListener	イベントリスナを登録します
RemoveListener	イベントリスナを削除します
SendEvent	イベントを本クラスに送信します

データメンバ

なし

4.4.2 クラスメンバ詳細

EventDistributor

```
EventDistributor::EventDistributor ();
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します。

AddListener

イベントリスナを登録します。

```
bool EventDistributor:: AddListener(
                                    Listener* pListener    ///< [in] Event listener
)
```

パラメータ

pListener

[in] イベントリスナのポインタ

戻り値

true: 成功

false: 失敗

例外

なし

解説

イベントを受取るリスナを登録します。

RemoveListener

イベントのリスナの登録を削除します。

```
bool EventDistributor:: AddListener(
                                    Listener* pListener    ///< [in] Event listener
)
```

パラメータ

pListener

[in] イベントリスナのポインタ

戻り値

true: 成功

false: 失敗

例外

なし

解説

イベントのリスナの登録を削除します。

SendEvent

イベントを本クラスに送信します。

```
bool EventDistributor:: SendEvent (
                                    unsigned long event,    ///< [in] event id
                                    const void* pParam,   ///< [in] parameter buffer
                                    int paramLength         ///< [in] parameter length
)
```

パラメータ

event

[in] イベント ID

pParam

[in] イベントパラメータのポインタ

paramLength

[in] イベントパラメータ長

戻り値

true: 成功

false: 失敗

例外

派生クラスの実装によります

解説

登録されている全てのイベントリスナの SendEvent()をコールしてイベントを配信します。

5. 非テンプレート版コレクション

テンプレートを使わない汎用的なデータ構造を提供します。

5.1 List

Pafee::List は可変長データを格納する双方向リンクリストクラスです

要素を追加する毎にメモリを確保して格納します

必要条件

ヘッダー：PafeeList.h

関連ファイル：

5.1.1 クラスメンバー覧

コンストラクタ

List	コンストラクタ

操作

Initialize	初期化します
operator =	既存のリストをコピーします
GetCount	List に格納されている要素数を取得します
IsEmpty	リストが空かチェックします
GetWriteBuffer	省コピーバージョンで追加する要素のアドレスを取得します
AddFront	リストの先頭に要素を追加します
AddBack	リストの末尾に要素を追加します
GetFrontLength	先頭要素のデータの長さを取得します
GetBackLength	末尾要素のデータの長さを取得します
GetLengthAt	pos の示す要素位置のデータの長さを取得します
GetFront	先頭要素のデータを取得します
GetBack	末尾要素のデータを取得します
GetFrontPosition	先頭要素の位置を取得します
GetBackPosition	末尾要素の位置を取得します
GetNextLength	次の要素のデータの長さを取得します
GetPreviousLength	前の要素のデータの長さを取得します
GetNext	現在の位置の要素のデータを取得して、pos を次の要素の位置にします
GetPrevious	現在の位置の要素のデータを取得して、pos を前の要素の位置にします
GetAt	現在の位置の要素のデータを取得します
GetAt	現在の位置の要素のデータを取得します
SetAt	現在の位置にデータをセットします
RemoveFront	先頭の要素を削除します
RemoveBack	末尾の要素を削除します
RemoveAt	pos の位置の要素を削除します
RemoveAll	要素を全て削除します

5.1.2 クラスメンバ詳細

List

空のリストを構築します

```
List::List(
        Allocator* pAllocator ///< [in] Memory Allocator
);
List::List(
        const List& lstc, ///< [in] initial value of List
        Allocator* pAllocator ///< [in] Memory Allocator
);
```

パラメータ

pAllocator

[in] メモリを確保するアロケータクラスを指定します

stc

[in] コピーList オブジェクト

戻り値

なし

解説

空のリストオブジェクトを作成します

// Allocator の指定をしない場合
Pafee::List myList;
//Allocator を指定した場合
Pafee::List myListp(&FixedAllocator);
//既存のリストを代入し初期化する場合
Pafee::List myListc = yourList;

参照

Pafee::Allocator クラス

Initialize

リストを初期化します

```
bool List::Initialize(
        Allocator* pAllocator ///< [in]    Memory Allocator
);
```

パラメータ

pAllocator

[in] メモリを確保するアロケータクラス

戻り値

bool

初期化成功：true 失敗:false

解説

生成時、メモリアロケータの指定を行わなかった場合、1 回のみ本関数で初期化することができます

参照

Pafee::Allocator クラス

operator=

代入します

```
List& List::operator =(
        const List& lstc ///< [in] Object of copied list
);
```

解説

代入します

IsEmpty

リストが空かチェックします

```
bool List::IsEmpty();
```

パラメータ

なし

戻り値

bool

リストが空：true リストが空でない：false

解説

リストが空かどうかチェックします

GetCount

List に格納されている要素数を取得します

```
int List::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

GetWriteBuffer

省コピーバージョンで追加する要素のアドレスを取得します

```
void* List::GetWriteBuffer(
        int length ///< [in] Buffer Size
);
```

パラメータ

length

[in] 追加する要素のサイズ(バイト数)

戻り値

void*

追加する要素のアドレス。NULL の場合、追加する要素のアドレス取得に失敗

解説

省コピーバージョンで要素を追加する際に予め、追加する要素のアドレスを取得します

GetWriteBuffer は必要な領域を確保し、確保した先頭のアドレスを戻します

領域の確保に失敗した場合は NULL を返します

```
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
char* pGetBuffer = myList.GetWriteBuffer(pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myList.AddFront( pafee_strlen(setData) );
```

参照

Pafee::List::GetWriteBuffer

Pafee::pafee_memcpy

Pafee::pafee_strlen

AddFront

リストの先頭に要素を追加します

```
bool List::AddFront(
        const void* pData, ///< [in] Write Thing
        int length ///< [in] Write Size
);
bool List::AddFront(
        int length ///< [in] data size
);
```

パラメータ

pData

[in] 追加する要素の先頭アドレス

length

[in] 追加する要素のサイズ(バイト数)

戻り値

bool

リスト追加成功:true リスト追加失敗:false

解説

リストの先頭に要素を追加します。省コピーバージョンは予め、GetWriteBufer 関数で取得したアドレスに、データを入れておく必要があります。

```
char setData[16];
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
myList.AddFront(setData, pafee_strlen(setData));
//省コピーバージョン
char* pGetBuffer = myList.GetWriteBuffer(pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myList.AddFront(pafee_strlen(setData) );
```

参照

Pafee::List::GetWriteBuffer

Pafee::pafee_memcpy

Pafee::pafee_strlen

AddBack

リストの末尾に要素を追加します

```
bool List::AddBack(
        const void *pData, ///< [in]Write Thing
        int length          ///< [in]Write Size
);
bool List::AddBack(
        int length ///< [in] data size
);          //省コピーバージョン
```

パラメータ

pData

[in] 追加する要素の先頭アドレス

length

[in] 追加する要素のサイズ(バイト数)

戻り値

bool

リスト追加成功:true リスト追加失敗:false

解説

リストの末尾に要素を追加します。省コピーバージョンは予め、GetWriteBuufer 関数で取得したアドレスに、データを入れておく必要があります

```
char setData[16];
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
myList.AddBack(setData, pafee_strlen(setData));
//省コピーバージョン
char* pGetBuffer = myList.GetWriteBuffer(pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myList.Add(pafee_strlen(setData) );
```

参照

Pafee::List::GetWriteBuffer

Pafee::pafee_memcpy

Pafee::pafee_strlen

GetFrontLength

先頭要素のデータの長さを取得します

```
int List::GetFrontLength(void);
```

パラメータ

なし

戻り値

int

先頭要素のデータの長さ

解説

リストの要素が空の場合は、0 を返します

GetBackLength

末尾要素のデータの長さを取得ます

```
int List::GetBackLength(void);
```

パラメータ

なし

戻り値

int

末尾要素のデータの長さ

解説

リストの要素が空の場合は、0 を返します

GetLengthAt

pos の示す位置の要素の長さを取得します

```
int List::GetLengthAt(
        List::POSITION pos ///< [in] Specified Position
);
```

パラメータ

pos

[in] リスト内の要素の位置

戻り値

int

データの長さ

解説

リストの要素の位置を指定して対象要素のデータの長さを取得します

参照

List::POSITION GetFrontPosition(void)

List::POSITION GetBackPosition(void)

int GetNextLength(List::POSITION& pos)

int GetPreviousLength (List::POSITION& pos)

bool GetNext(List::POSITION& pos, void* pBuffer, int* pLength, int bufSize=INT_MAX)

void* GetNext(List::POSITION& pos, int* pLength)

bool GetPrevious(List::POSITION &pos, void* pBuffer, int* pLength, int bufSize=INT_MAX)

void* GetPrevious(List::POSITION &pos, int* pLength)

bool GetAt(List::POSITION pos, void* pBuffer, int* pLength, int bufSize=INT_MAX)

void* GetAt(List::POSITION pos, int* pLength)

bool SetAt(List::POSITION& pos, const void* pData, int length)

GetFront

先頭要素のデータを取得します

```
bool List::GetFront(
        void* pBuffer, ///< [out] Head Element
        int* pLength, ///< [out] Head Element Size
        int bufSize ///< [in] Prepared buffer sizes
);
```

パラメータ

pBuffer

[out] 取得したデータを入れるバッファのアドレス

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

bufSize

[in] 取得したデータを入れるバッファのサイズ

戻り値

bool

成功：true 失敗:false

解説

リストの要素が空の場合、false を返します

bufSize より *pLength の方が大きい場合、false を返します

GetFront（省コピーバージョン）

先頭要素のデータを取得します

```
void* List::GetFront(
        int* pLength ///< [out] Head Element Size
);
```

パラメータ

*pLength

[out] 取得するデータの長さを入れるバッファのアドレス

戻り値

void*

取得するデータの先頭アドレス

解説

リストの要素が空の場合、NULL を返します

```
char buf[256], *p;
int Length;
p = myList.GetFront(&Length);
if ( p == NULL )
  //取得失敗
  return -1;
if ( Length > 256 )
  return -1;
pafee_memcpy ( buf, p, Length );
```

GetBack

末尾要素のデータを取得します

```
bool List::GetBack(
        void* pBuffer, ///< [out] End Element
        int* pLength, ///< [out] End Element Size
        int bufSize ///< [in] Prepared buffer sizes
);
```

パラメータ

pBuffer

[out] 取得したデータを入れるバッファのアドレス

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

bufSize

[in] 取得したデータを入れるバッファのサイズ

戻り値

bool

成功：true 失敗:false

解説

リストの要素が空の場合、false を返します

bufSize より *pLength の方が大きい場合、false を返します

GetBack（省コピーバージョン）

末尾要素のデータを取得します

```
void* List::GetBack(
        int* pLength ///< [out] End Element Size
);
```

パラメータ

*pLength

[out] 取得するデータの長さを入れるバッファのアドレス

戻り値

void*

取得するデータの先頭アドレス

解説

リストの要素が空の場合、NULL を返します

```
char buf[256], *p;
int Length;
p = myList.GetBack(&Length);
if ( p == NULL )
    //取得失敗
    return -1;
if ( Length > 256 )
    return -1;
    pafee_memcpy ( buf, p, Length );
```

GetFrontPosition

先頭要素の位置を取得します

```
List::POSITION List::GetFrontPosition(void);
```

パラメータ

なし

戻り値

POSITION

先頭要素の位置

解説

リストの要素が空の場合、NULL を返します

GetBackPosition

末尾要素の位置を取得します

```
List::POSITION List::GetBackPosition(void);
```

パラメータ

なし

戻り値

POSITION

末尾要素の位置

解説

リストの要素が空の場合、NULL を返します

GetNextLength

次の要素のデータの長さを取得します

```
int List::GetNextLength(
        List::POSITION& pos ///< [in] Specified Position
);
```

パラメータ

pos

[in] リストの位置

戻り値

int

次の要素のデータの長さ

解説

本関数を呼び出しても pos の位置は変わりません

GetPreviosLength

前の要素のデータの長さを取得します

```
int List::GetPreviousLength(
        List::POSITION& pos ///< [in] Specified Position
);
```

パラメータ

pos

[in] リストの位置

戻り値

int

前の要素のデータの長さ

解説

本関数を呼び出しても pos の位置は変わりません

GetNext

現在の位置の要素のデータを取得して、pos を次の要素の位置にします

```
bool List::GetNext(
        List::POSITION &pos, ///< [in/out] Next Node Pointer
        void* pBuffer, ///< [out] Spesified Position Element
        int* pLength, ///< [out] Spesified Position Element Size
        int bufSize ///< [in] Prepared buffer sizes
);
```

パラメータ

pos

[in/out] リストの位置

pBuffer

[out] 取得したデータを入れるバッファのアドレス

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

bufSize

[in] 取得したデータを入れるバッファのサイズ

戻り値

bool

成功:true 失敗:false

解説

本関数は現在の pos の位置のデータを取得します

データを取得後、次の位置を指すように pos を更新します

bufSize < *pLenght のとき false を返します

要素が空のとき false を返します

```
char buf[256];
int Length;
POSITION pos = myList.GetFrontPosition();
bool rtn = myList.GetNext( pos, Buffer, &Length );
if ( rtn == false )
{
  //取得失敗
}
```

GetNext（省コピーバージョン）

現在の位置の要素のデータを取得して、pos を次の要素の位置にします

```
void* List::GetNext(
        List::POSITION &pos, ///< [in/out] Next Node Pointer
        int* pLength ///< [out] Spesified Position Element Size
);
```

パラメータ

pos

[in/out] リストの位置

pLength

[out] 要素のデータの長さ

戻り値

void*

成功：現在の要素のバッファアドレス 失敗：NULL

解説

本関数は現在の pos の位置のデータを取得します

データを取得後、次の位置を指すように pos を更新します

要素が空のとき NULL を返します

```
char buf[256];
char* pBuf;
int Length;
POSITION pos = myList.GetFrontPosition();
char* pBuf = myList.GetNext( pos, &Length );
if ( pBuf == NULL ) { //取得失敗 }
if ( Length > 256 ) { //エラー   }
pafee_memcpy ( buf, pBuf, Lenfth );
```

GetPrevious

現在の位置の要素のデータを取得して、pos を前の要素の位置にします

```
bool List::GetPrevious(
        List::POSITION &pos, ///< [in/out] Next Node Pointer
        void* pBuffer, ///< [out] Spesified Position Element
        int* pLength, ///< [out] Spesified Position Element Size
        int bufSize ///< [in] Prepared buffer sizes
);
```

パラメータ

pos

[in/out] リストの位置

pBuffer

[out] 取得したデータを入れるバッファのアドレス

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

bufSize

[in] 取得したデータを入れるバッファのサイズ

戻り値

bool

成功:true 失敗:false

解説

本関数は現在の pos の位置のデータを取得します

データを取得後、前の位置を指すように pos を更新します

bufSize < *pLenght のとき false を返します

要素が空のとき false を返します

```
char buf[256];
int Length;
POSITION pos = myList.GetFrontPosition();
bool rtn = myList. GetPrevious( pos, Buffer, &Length );
if ( rtn == false )
{
  //取得失敗
}
```

GetPrevious（省コピーバージョン）

現在の位置の要素のデータを取得して、pos を前の要素の位置にします

```
void* List::GetPrevious(
        List::POSITION &pos, ///< [in/out] Next Node Pointer
        int* pLength ///< [out] Spesified Position Element Size
);
```

パラメータ

pos

[in/out] リストの位置

pLength

[out] 要素のデータの長さ

戻り値

void*

成功：現在の要素のバッファアドレス 失敗：NULL

解説

本関数は現在の pos の位置のデータを取得します。

データを取得後、前の位置を指すように pos を更新します。

要素が空のとき NULL を返します。

```
char buf[256];
char* pBuf;
int Length;
POSITION pos = myList.GetBackPosition();
char* pBuf = myList.GetPrevious( pos, &Length );
if ( pBuf == NULL ) { //取得失敗 }
if ( Length > 256 ) { //エラー  }
pafee_memcpy ( buf, pBuf, Lenfth );
```

GetAt

現在の位置の要素のデータを取得します

```
bool List::GetAt(
        List::POSITION pos,   ///< [in] Specified Position
        void* pBuffer, ///< [out] Spesified Position Element
        int* pLength, ///< [out] Spesified Position Element Size
        int bufSize ///< [in] Prepared buffer sizes
);
```

パラメータ

pos

[in] リストの位置

pBuffer

[out] 取得したデータを入れるバッファのアドレス

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

bufSize

[in] 取得したデータを入れるバッファのサイズ

戻り値

bool

成功:true 失敗:false

解説

bufSize < *pLenght のとき false を返します

要素が空のとき false を返します

```
char buf[256];
int Length;
POSITION pos = myList.GetFrontPosition();
bool rtn = myList. GetAt( pos, Buffer, &Length );
if ( rtn == false )
{
  //取得失敗
}
```

GetAt（省コピーバージョン）

現在の位置の要素のデータを取得します

```
void* List::GetAt(
        List::POSITION pos, ///< [in] Specified Position
        int* pLength ///< [out] Spesified Position Element Size
);
```

パラメータ

pos

[in] リストの位置

pLength

[out] 要素のデータの長さ

戻り値

void*

成功：現在の要素のバッファアドレス 失敗：NULL

解説

要素が空のとき NULL を返します

```
char buf[256];
char* pBuf;
int Length;
POSITION pos = myList.GetFrontPosition();
char* pBuf = myList.GetAt( pos, &Length );
if ( pBuf == NULL ) { //取得失敗 }
if ( Length > 256 ) { //エラー  }
pafee_memcpy ( buf, pBuf, Lenfth );
```

SetAt

現在の位置にデータをセットします

```
bool List::SetAt(
        List::POSITION& pos, ///< [in] Specified Position
        const void* pData,     ///< [in] Spesified Position Element
        int length             ///< [in] Spesified Position Element Size
);
```

パラメータ

pos

[in] リストの位置

pData

[in] 追加するデータのアドレス

length

[in] 追加するデータの長さ

戻り値

bool

成功:true 失敗:false

解説

要素が空のとき false を返します

```
char buf[256];
int Length;
pafee_memcpy(buf,” Hello World” );
Length = pafee_strlen(” Hello World” );
bool rtn = myList. SetAt( pos, buf, &Length );
if ( rtn == false )
{
  //取得失敗
}
```

RemoveFront

先頭の要素を削除します。

```
void List::RemoveFront(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

RemoveBack

末尾の要素を削除します

```
void List::RemoveBack(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

RemoveAt

pos の位置の要素を削除します

```
void List::RemoveAt(
        List::POSITION pos ///< [in]    Specified Position
);
```

パラメータ

pos

[in] リストの位置

戻り値

なし

解説

要素が空のときなしもしません

RemoveAll

要素を全て削除します

```
void List::RemoveAt(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

5.2 Queue

Pafee::Queue は可変長データの FIFO（FirstIn-FirstOut）を提供します

List のラッパーとして実装されています

必要条件

ヘッダー：PafeeQueue.h

関連ファイル： PafeeList.h,PafeeList.cpp

5.2.1 クラスメンバー覧

コンストラクタ

Queue	コンストラクタ

操作

operator =	既存のリストをコピーします
GetCount	Queue に格納されている要素数を取得します
IsEmpty	リストが空かチェックします
GetWriteBuffer	省コピーバージョンで追加する要素のアドレスを取得します
Enqueue	リストの先頭に要素を追加します
Dequeue	先頭要素のデータを取得します
Peek	先頭要素のデータを取得します
PeekLength	先頭要素のデータの長さを取得します
RemoveFront	先頭の要素を削除します
RemoveAll	要素を全て削除します

5.2.2 クラスメンバ詳細

Queue

キューを構築します

```
Queue::Queue(
        Allocator* pAllocator ///< [in] Memory Allocator
);
Queue::Queue(
        const Queue& quec, ///< [in] initial value of Queue
        Allocator* pAllocator ///< [in] Memory Allocator
);
```

パラメータ

pAllocator

[in] メモリを確保するアロケータクラス

quec

[in] コピーQueue オブジェクト

戻り値

なし

解説

キューオブジェクトを作成します

```
// Allocator の指定をしない場合
Pafee::Queue myQueue;
//Allocator を指定した場合
Pafee::Queue myQueue( &FixedAllocator);
//既存のキューを代入し初期化する場合
Pafee::QueuemyQueue = yourQueue;
```

参照

Pafee::Allocator クラス

GetCount

Queue に格納されている要素数を取得します

```
int Queue::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

IsEmpty

キューが空かチェックします

```
bool Queue::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

キューが空：true　キューが空でない：false

解説

キューが空かどうかチェックします

GetWriteBuffer

省コピーバージョンで追加する要素のアドレスを取得します

```
void* Queue::GetWriteBuffer(
    int length //< [in] Buffer length
);
```

パラメータ

length

[in] 追加する要素のサイズ(バイト数)

戻り値

void*

追加する要素のアドレス

解説

省コピーバージョンで要素を追加する際に予め、追加する要素のアドレスを取得します

GetWriteBuffer は必要な領域を確保し、確保した先頭のアドレスを戻します

領域の確保に失敗した場合は NULL を返します

```
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
char* pGetBuffer = myQueue.GetWriteBuffer(pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myQueue.Enqueue( pafee_strlen(setData) );
```

参照

Pafee::pafee_memcpy

Pafee::pafee_strlen

Enqueue

キューの先頭に要素を追加します

```
bool Queue::Enqueue(
        void* pData, // < [in] Data to set
        int length     // < [in] Length to set
);
bool Queue::Enqueue(
        int length // < [in] Length to set
);          //省コピーバージョン
```

パラメータ

pData

[in] 追加する要素の先頭アドレス

length

[in] 追加する要素のサイズ(バイト数)

戻り値

bool

キュー追加成功:true キュー追加失敗:false

解説

キューに要素を追加します。省コピーバージョンは予め、GetWriteBuufer 関数で取得したアドレスに、データを入れておく必要があります

```
char setData[16];
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
myQueue.Enqueue(setData, pafee_strlen(setData));
//省コピーバージョン
char* pGetBuffer = myQueue.GetWriteBuffer(pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myQueue.Enqueue(pafee_strlen(setData) );
```

参照

Pafee::List

Pafee::pafee_memcpy

Pafee::pafee_strlen

Dequeue

要素のデータを取得します

```
bool Queue::Dequeue(
        void* pBuffer, ///< [out] Head Position Element
        int* pLength, ///< [out] Head Position Element Size
        int bufSize ///< [in] Prepared buffer sizes
);
```

パラメータ

pBuffer

［out］取得したデータを入れるバッファのアドレス

pLength

［out］取得したデータの長さを入れるバッファのアドレス

bufSize

［in］取得したデータを入れるバッファのサイズ

戻り値

bool

成功：true　失敗:false

解説

キューの要素が空の場合、false を返します

bufSize より *pLength の方が大きい場合、false を返します

Peek

要素のデータを取得します

```
bool Queue::Peek(
        void* pBuffer,          // < [out] A buffer to put provided data in
        int*    pLength,        // < [out] The length that was gotten
        int bufSize=INT_MAX // < [in] Length to get
);
```

パラメータ

pBuffer

[out] 取得したデータを入れるバッファのアドレス

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

bufSize

[in] 取得したデータを入れるバッファのサイズ

戻り値

bool

成功：true 失敗:false

解説

取得した要素をキューから削除しません

キューの要素が空の場合、false を返します

bufSize より *pLength の方が大きい場合、false を返します

Peek（省コピーバージョン）

要素のデータを取得します

```
void* Queue::Peek(
        int* pLength // < [out] The length that was gotten
);
```

パラメータ

pLength

［out］取得したデータの長さを入れるバッファのアドレス

戻り値

void*

取得したデータのバッファのアドレス

解説

取得した要素をキューから削除しません

キューの要素が空の場合、NULL を返します

bufSize より *pLength の方が大きい場合、NULL を返します

PeekLength

先頭要素のデータの長さを取得します

```
int Queue::GetPeekLength(void);
```

パラメータ

なし

戻り値

int

取得要素のデータの長さ

解説

キューの要素が空の場合は、0 を返します

RemoveFront

先頭の要素を削除します

```
void Queue::RemoveFront(void);
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

RemoveAll

要素を全て削除します

```
void Queue::RemoveAt(void);
```

戻り値

なし

解説

要素が空のときなしもしません

5.3 PriorityQueue

Pafee::PriorityQueue は可変長データの優先順位付 FIFO（FirstIn-FirstOut）を提供します

List のラッパーとして実装されています

必要条件

ヘッダー：PafeePriorityQueue.h

関連ファイル： PafeeList.h,PafeeList.cpp

5.3.1 クラスメンバー覧

コンストラクタ

PriorityQueue	デフォルトコンストラクタ

操作

Initialize	既存のキューをコピーします
GetCount	PriorityQueue に格納されている要素数を取得します
IsEmpty	プライオリティキューが空かチェックします
GetWriteBuffer	省コピーバージョンで追加する要素のアドレスを取得します
Enqueue	プライオリティキューの先頭に要素を追加します
Dequeue	先頭要素のデータを取得します
Peek	先頭要素のデータを取得します
PeekLength	先頭要素のデータの長さを取得します
RemoveFront	先頭の要素を削除します
RemoveAll	要素を全て削除します

5.3.2 クラスメンバ詳細

ProrityQueue

空のプライオリティキューを構築します

```
PriorityQueue::PriorityQueue(
        int priorityRange=0, ///< [in] priorityRange
        Allocator* pAllocator ///< [in] Pointer of allocator
);
PriorityQueue::PriorityQueue(
        const PriorityQueue& pquec, ///< [in] initial value of Queue
        Allocator* pAllocator ///< [in] Memory Allocator
);
```

パラメータ

priorityRange

［in］優先順位の範囲

pAllocator

［in］メモリを確保するアロケータクラス

quec

［in］コピーPriorityQueue オブジェクト

戻り値

なし

解説

空のプライオリティキューオブジェクトを作成します

// Allocator の指定をしない場合
Pafee::PriorityQueue myPQueue;
//Allocator を指定した場合
Pafee::PriorityQueue myPQueue(&FixedAllocator);
//既存のリストを代入し初期化する場合
Pafee::Priority myPQueue = yourPQueue;

参照

Pafee::Allocator クラス

Initialize

リストを初期化します。

```
bool PriorityQueue::Initialize(
        int priorityRange, ///< [in] priorityRange
        Allocator* pAllocator ///< [in] Pointer of allocator
);
```

パラメータ

priorityRange

［in］優先順位の範囲

pAllocator

［in］メモリを確保するアロケータクラス

戻り値

bool

初期化成功：true　失敗:false

解説

生成時、メモリアロケータの指定を行わなかった場合、1 回のみ本関数で初期化することができます

参照

Pafee::Allocator クラス

GetCount

PriorityQueue に格納されている要素数を取得します

```
int PriorityQueue::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

IsEmpty

プライオリティキューが空かチェックします

```
bool PriorityQueue::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

プライオリティキューが空：true　プライオリティキューが空でない：false

解説

プライオリティキューが空かどうかチェックします

GetWriteBuffer

省コピーバージョンで追加する要素のアドレスを取得します

```
void* PriorityQueue::GetWriteBuffer(
        int priority, ///< [in] priority
        int length    ///< [in] Data length
);
```

パラメータ

priority

[in] 追加する要素の優先順位

length

[in] 追加する要素のサイズ(バイト数)

戻り値

void*

追加する要素のアドレス。NULL の場合、追加する要素のアドレス取得に失敗

解説

省コピーバージョンで要素を追加する際に予め、追加する要素のアドレスを取得します

GetWriteBuffer は必要な領域を確保し、確保した先頭のアドレスを戻します

領域の確保に失敗した場合は NULL を返します

```
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
char* pGetBuffer = myQueue.GetWriteBuffer(1,pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myPQueue.Enqueue( pafee_strlen(setData) );
```

参照

Pafee::pafee_memcpy

Pafee::pafee_strlen

Enqueue

プライオリティーキューに要素を追加します

```
bool PriorityQueue::Enqueue (
		int priority,///< [in] priority
		void* pData,///< [in] Data Buffer Pointer
		int length ///< [in] Data length
);
bool PriorityQueue::Enqueue (
		int priority,///< [in] priority
		int length     ///< [in] Data length
);		//省コピーバージョン
```

パラメータ

priority

[in] 優先順位

pData

[in] 追加する要素の先頭アドレス

length

[in] 追加する要素のサイズ(バイト数)

戻り値

bool

キュー追加成功:true キュー追加失敗:false

解説

プライオリティキューに要素を追加します。省コピーバージョンは予め、GetWriteBuufer 関数で取得したアドレスに、データを入れておく必要があります

```
char setData[16];
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
myPQueue.Enqueue(1,setData, pafee_strlen(setData));
//省コピーバージョン
char* pGetBuffer = myPQueue.GetWriteBuffer(1,pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myPQueue.Enqueue(1,pafee_strlen(setData) );
```

参照

Pafee::pafee_memcpy

Pafee::pafee_strlen

Dequeue

要素のデータを取得します

```
bool PriorityQueue::Dequeue(
        int* pPriority,///< [out] Pointer of priority
        void* pBuffer, ///< [out] Data Buffer Pointer
        int* pLength,  ///< [out] Data Length Pointer
        int bufSize    ///< [in] Size of demand buffer
);
```

パラメータ

pPriority

［out］取得する要素の優先順位を格納するアドレス

pBuffer

［out］取得したデータを入れるバッファのアドレス

pLength

［out］取得したデータの長さを入れるバッファのアドレス

bufSize

［in］取得したデータを入れるバッファのサイズ

戻り値

bool

成功：true　失敗:false

解説

キューの要素が空の場合、false を返します

bufSize より *pLength の方が大きい場合、false を返します

Peek

要素のデータを取得します

```
bool PriorityQueue::Peek(
        int* pPriority,///< [out] Pointer of priority
        void* pBuffer, ///< [out] Data Buffer Pointer
        int* pLength,  ///< [out] Data Length Pointer
        int bufSize    ///< [in] Size of demand buffer
);
```

パラメータ

pPriority

［out］取得する要素の優先順位を格納するアドレス

pBuffer

［out］取得した要素を入れるバッファのアドレス

pLength

［out］取得した要素の長さを入れるバッファのアドレス

bufSize

［in］取得した要素を入れるバッファのサイズ

戻り値

bool

成功：true　失敗:false

解説

取得した要素をキューから削除しません

キューの要素が空の場合、false を返します

bufSize より *pLength の方が大きい場合、false を返します

Peek（省コピーバージョン）

要素のデータを取得します

```
void* PriorityQueue::Peek (
        int* pPriority, ///< [out] Pointer of priority
        int* pLength    ///< [out] Data Length Pointer
);
```

パラメータ

pPriority

［out］取得した要素の優先順位を格納するアドレス

pLength

［out］取得したデータの長さを入れるバッファのアドレス

戻り値

void*

取得した要素のバッファのアドレス

解説

取得した要素をキューから削除しません

キューの要素が空の場合、NULL を返します

bufSize より *pLength の方が大きい場合、NULL を返します

PeekLength

先頭要素のデータの長さを取得します

```
int PriorityQueue::PeekLength(
        int* pPriority ///< [out] Pointer of priority
);
```

パラメータ

pPriority

[out] 取得する要素のデータの長さを格納するアドレス

戻り値

int

取得要素のデータの長さ

解説

キューの要素が空の場合は、0 を返します

RemoveFront

先頭の要素を削除します。

```
void PriorityQueue::RemoveFront(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

RemoveAll

要素を全て削除します。

```
void PriorityQueue::RemoveAt(void)
```

戻り値

なし

解説

要素が空のときなしもしません

5.4 Stack

Pafee::Stack は可変長データの Stack を提供します

List のラッパーとして実装されています

必要条件

ヘッダー：PafeeStack.h

関連ファイル： PafeeList.h,PafeeList.cpp

5.4.1 クラスメンバー覧

コンストラクタ

Stack	コンストラクタ

操作

operator =	既存のスタックをコピーします
GetCount	Stack に格納されている要素数を取得します
IsEmpty	スタックが空かチェックします
GetWriteBuffer	省コピーバージョンで追加する要素のアドレスを取得します
Push	スタックに要素を Push します
Pop	スタックの要素を Pop します
Peek	先頭要素のデータを取得します
PeekLength	先頭要素のデータの長さを取得します
RemoveFront	先頭の要素を削除します
RemoveAll	要素を全て削除します

5.4.2 クラスメンバ詳細

Stack

スタックを構築します。

```
Stack::Stack(
        Allocator* pAllocator ///< [in] Memory Allocator
);
Stack::Stack(
        const Stack& stkc, ///< [in] initial value of Stack
        Allocator* pAllocator ///< [in] Memory Allocator
);
```

パラメータ

pAllocator

［in］メモリを確保するアロケータクラス

stkc

［in］コピーStack オブジェクト

戻り値

なし

解説

スタックオブジェクトを作成します

// Allocator の指定をしない場合
Pafee::Stack myStack;
//Allocator を指定した場合
Pafee::Stack myStack(&FixedAllocator);

//既存のスタックを代入し初期化する場合
Pafee::Stck myStack = yourStack;

参照

Pafee::Allocator クラス

GetCount

Stack に格納されている要素数を取得します

```
int Stack::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

IsEmpty

Stack が空かチェックします

```
bool Stack::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

スタックが空：true　スタックが空でない：false

解説

スタックが空かどうかチェックします

GetWriteBuffer（省コピーバージョン）

省コピーバージョンで追加する要素のアドレスを取得します

```
void* Stack::GetWriteBuffer(
        int length // < [in] Length to set
);
```

パラメータ

length

[in] 追加する要素のサイズ（バイト数）

戻り値

void*

追加する要素のアドレス。NULL の場合、追加する要素のアドレス取得に失敗

解説

省コピーバージョンで要素を追加する際に予め、追加する要素のアドレスを取得します

GetWriteBuffer は必要な領域を確保し、確保した先頭のアドレスを戻します

領域の確保に失敗した場合は NULL を返します

```
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
char* pGetBuffer = myStack.GetWriteBuffer(pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myStack.Push( pafee_strlen(setData) );
```

参照

Pafee::pafee_memcpy

Pafee::pafee_strlen

Push

スタックに要素を Push します

```
bool Stack::Push(
        void* pData,    /// < [in] Data to set
        int length    /// < [in] Length to set
)
bool Stack::Push(
        int length /// < [in] Length to set
)
```

パラメータ

pData

[in] Push する要素の先頭アドレス

length

[in] Push する要素のサイズ(バイト数)

戻り値

bool

Push 成功:true Push 失敗:false

解説

Stack に要素を Push します。省コピーバージョンは予め、GetWriteBuufer 関数で取得したアドレスに、データを入れておく必要があります

```
char setData[16];
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
//
myStack.Push(setData, pafee_strlen(setData));
//省コピーバージョン
char* pGetBuffer = myStack.GetWriteBuffer(pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myStack.Push(pafee_strlen(setData) );
```

参照

Pafee::pafee_memcpy

Pafee::pafee_strlen

Pop

Stack のデータを Pop します

```
bool Stack::Pop(
        void* pBuffer, ///< [out] Head Position Element
        int* pLength, ///< [out] Head Position Element Size
        int bufSize ///< [in] Prepared buffer sizes
)
```

パラメータ

pBuffer

[out] Pop したデータを入れるバッファのアドレス

pLength

[out] Pop したデータの長さを入れるバッファのアドレス

bufSize

[in] Pop したデータを入れるバッファのサイズ

戻り値

bool

成功:true 失敗:false

解説

スタックの要素が空の場合、false を返します

bufSize より *pLength の方が大きい場合、false を返します

Peek

先頭の要素のデータを取得します

```
bool Stack::Peek(
        void* pBuffer, ///< [out] Head Position Element
        int* pLength, ///< [out] Head Position Element Size
        int bufSize=INT_MAX ///< [in] Prepared buffer sizes
);
```

パラメータ

pBuffer

［out］取得したデータを入れるバッファのアドレス

pLength

［out］取得したデータの長さを入れるバッファのアドレス

bufSize

［in］取得したデータを入れるバッファのサイズ

戻り値

bool

成功：true　失敗:false

解説

取得した要素をスタックから削除しません

スタックの要素が空の場合、false を返します

bufSize より *pLength の方が大きい場合、false を返します

Peek（省コピーバージョン）

要素のデータを取得します

```
void* Stack::Peek
        int* pLength    /// < [out] The length that was gotten
);
```

パラメータ

pLength

［out］取得したデータの長さを入れるバッファのアドレス

戻り値

void*

取得したデータのバッファのアドレス

解説

取得した要素をスタックから削除しません

タックの要素が空の場合、NULL を返します

PeekLength

先頭要素のデータの長さを取得します

```
int Stack::GetPeekLength(void);
```

パラメータ

なし

戻り値

int

取得要素のデータの長さ

解説

スタックの要素が空の場合は、0 を返します

RemoveFront

先頭の要素を削除します

```
void Stack::RemoveFront(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

RemoveAll

要素を全て削除します

```
void Stack::RemoveAt(void)
```

戻り値

なし

解説

要素が空のときなしもしません

5.5 RingBuffer

Pafee::RingBuffer は固定長の連続したリングバッファを提供します

必要条件

ヘッダー：PafeeRingBuffer.h

5.5.1 クラスメンバー覧

コンストラクタ

RingBuffer	コンストラクタ

操作

Initialize	初期化します
operator =	既存のリングバッファをコピーします
GetCount	リングバッファに格納されている要素数を取得します
IsEmpty	リングバッファが空かチェックします
GetWriteBuffer	省コピーバージョンで追加する要素のアドレスを取得します
Enqueue	リングバッファの末尾に要素を追加します
Dequeue	リングバッファの先頭要素を取得します
Peek	先頭要素のデータを取得します
PeekLength	先頭要素のデータの長さを取得します
ClearAll	要素を全て削除します

5.5.2 クラスメンバ詳細

RingBuffer

リングバッファを構築します

```
RingBuffer::RingBuffer(
        int capacity, ///< [in] Maximum size and memory allocator of number of elements
        int elementSize, ///< [in] and one element in can setting of the following as argument and ring buffer
        Allocator* pAllocator ///< [in] MemoryAllocator
);
RingBuffer::RingBuffer(
        RingBuffer& rbc, ///< [in] initial value of RingBuffer
        Allocator* pAllocator ///< [in] MemoryAllocator
);
```

パラメータ

capacity

[in] リングバッファの要素数を指定します

elementSize

[in] 要素のサイズを指定します

pAllocator

[in] メモリを確保するアロケータクラスを指定します

rbc

[in] コピーRingBuffer オブジェクト

戻り値

なし

解説

リングバッファオブジェクトを作成します

```
// Allocator の指定をしない場合
Pafee::RingBuffer myRing(100,20);
//Allocator を指定した場合
Pafee::RingBuffer myRing( 100,20,&FixedAllocator);
//既存のスタックを代入し初期化する場合
Pafee::RingBuffer myRing = yourRing;
```

参照

Pafee::Allocator クラス

Initialize

リングバッファを初期化します

```
bool RingBuffer::Initialize(
                int capacity,    ///< [in] capacity
                int elementSize,    ///< [in] elementSize
                Allocator* pAllocator ///< [in] MemoryAllocator
);
```

パラメータ

capacity

[in] リングバッファの要素数

elementSize

[in] 要素のサイズ

pAllocator

[in] メモリを確保するアロケータクラス

戻り値

bool

初期化成功：true、初期化失敗：false

解説

生成直後、1 回のみ本関数で初期化することができます

参照

Pafee::Allocator クラス

GetCount

RingBuffer に格納されている要素数を取得します

```
int RingBuffer::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

IsEmpty

RingBuffer が空かチェックします

```
bool RingBuffer::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

リングバッファが空：true リングバッファが空でない：false

解説

リングバッファが空かどうかチェックします

GetWriteBuffer

省コピーバージョンで追加する要素のアドレスを取得します

```
void* RingBuffer::GetWriteBuffer(void);
```

パラメータ

なし

戻り値

void*

追加する要素のアドレス。NULL の場合、追加する要素のアドレス取得に失敗

解説

省コピーバージョンで要素を追加する際に予め、追加する要素のアドレスを取得します
アドレスの取得に失敗した場合は NULL を返します

```
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
char* pGetBuffer = myRing.GetWriteBuffer(pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myRing.Enqueue( pafee_strlen(setData) );
```

参照

Pafee::pafee_memcpy
Pafee::pafee_strlen

Enqueue

リングバッファの末尾に要素を追加します

```
bool RingBuffer::Enqueue (
        void* pData,    ///< [in] Data
        int length    ///< [in] Data Length
);
bool RingBuffer::Enqueue (
        int length    ///< [in] Data Length
);          //省コピーバージョン
```

パラメータ

pData

[in] 追加する要素の先頭アドレス

length

[in] 追加する要素のサイズ(バイト数)

戻り値

bool

追加成功:true 追加失敗:false

解説

RingBuffer に要素を追加します。省コピーバージョンは予め、GetWriteBuufer 関数で取得したアドレスに、データを入れておく必要があります

```
char setData[16];
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
myRing.Enqueue(setData, pafee_strlen(setData));
//省コピーバージョン
char* pGetBuffer = myRing.GetWriteBuffer(pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myRing.Enqueueh(pafee_strlen(setData) );
```

参照

Pafee::pafee_memcpy

Pafee::pafee_strlen

Dequeue

リングバッファの先頭要素を取得します

```
bool RingBuffer::Dequeue(
        void* pBuffer, ///< [out] Prepared Buffer
        int* pLength, ///< [out] Data Length
        int bufSize=INT_MAX ///< [in] Prepared Buffer Length
);
```

パラメータ

pBuffer

[out] 取得したデータを入れるバッファのアドレス

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

bufSize

[in] 取得したデータを入れるバッファのサイズ

戻り値

bool

成功：true 失敗:false

解説

リングバッファの要素が空の場合、false を返します

bufSize より *pLength の方が大きい場合、false を返します

Dequeue（省コピーバージョン）

リングバッファの先頭要素を取得します

```
void* RingBuffer::Dequeue(
        int* pLength ///< [out] Data Length
);
```

パラメータ

*pLength

[out] 取得するデータの長さを入れるバッファのアドレス

戻り値

void*

取得するデータの先頭アドレス

解説

リングバッファの要素が空の場合、NULL を返します

```
char buf[256], *p;
int Length;
p = myRingBuffer.Dequeue(&Length);
if ( p == NULL )
    //取得失敗
    return -1;
pafee_memcpy ( buf, p, Length );
```

Peek

先頭の要素のデータを取得します

```
bool RingBuffer::Peek(
        void* pBuffer, ///< [out] Prepared Buffer
        int* pLength, ///< [out] Data Length
        int bufSize ///< [in] Prepared Buffer Length
);
```

パラメータ

pBuffer

[out] 取得したデータを入れるバッファのアドレス

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

bufSize

[in] 取得したデータを入れるバッファのサイズ

戻り値

bool

成功：true 失敗:false

解説

取得した要素をリングバッファから削除しません

本関数はリードバッファポインタを更新しません

リングバッファの要素が空の場合、false を返します

Peek（省コピーバージョン）

先頭要素のデータを取得します

```
void* RingBuffer::Peek(
        int* pLength ///< [out] Data Length
);
```

パラメータ

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

戻り値

void*

取得したデータのバッファのアドレス

解説

取得した要素をリングバッファから削除しません

リングバッファの要素が空の場合、NULL を返します

PeekLength

先頭要素のデータの長さを取得します

```
int RingBuffer::PeekLength(void);
```

パラメータ

なし

戻り値

int

取得要素のデータの長さ

解説

リングバッファの要素が空の場合は、0 を返します

ClearAll

要素を全て削除します

```
void RingBuffer::ClearAll(void);
```

戻り値

なし

解説

要素が空のときなしもしません

5.6 CharacterBuffer

5.7 Map

Pafee::Map はキーと値をペアで管理するクラスです

必要条件

ヘッダー：PafeeMap.h

5.7.1 クラスメンバー覧

コンストラクタ

Map	コンストラクタ

操作

Initialize	初期化します
operator =	既存のリストをコピーします
GetCount	Map に格納されている要素数を取得します
IsEmpty	Map が空かチェックします
GetWriteBuffer	省コピーバージョンで追加する要素のアドレスを取得します
Set	Map に要素を追加します
Get	指定キーの要素のデータを取得します
operator []	指定キーの要素のデータを取得します
GetStartPosition	先頭要素の位置を取得します
GetNext	現在の位置の要素のデータを取得して、pos を次の要素の位置にします
Remove	要素を削除します
RemoveAll	要素を全て削除します

5.7.2 クラスメンバ詳細

Map

Map を構築します

```
Map::Map(
        int hashSize, ///< [in] hash size
        Allocator* pAllocator ///< [in] Memory Allocator
)
Map::Map(
        const Map& mpc, ///< [in] initial value of Map
        Allocator* pAllocator ///< [in] Memory Allocator
)
```

パラメータ

hashSize

[in] hash テーブルのサイズ

pAllocator

[in] メモリを確保するアロケータクラス

mpc

[in] コピーMap オブジェクト

戻り値

なし

解説

空の Map オブジェクトを作成します

```
// Allocator の指定をしない場合
Pafee::Map myMap(100);
//Allocator を指定した場合
Pafee::Map myMap( 100,&FixedAllocator);
//既存のリストを代入し初期化する場合
Pafee::List myMap = yourMap;
```

参照

Pafee::Allocator クラス

Initialize

Map を初期化します

```
bool Map::Initialize(
        int hashSize, ///< [in] hash size
        Allocator* pAllocator ///< [in] Memory Allocator
);
```

パラメータ

hashSize

[in] hash テーブルのサイズ

pAllocator

[in] メモリを確保するアロケータクラス

戻り値

bool

初期化成功：true 失敗:false

解説

生成直後、1 回のみ本関数で初期化することができます

参照

Pafee::Allocator クラス

IsEmpty

Map が空かチェックします

```
bool Map::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

Map が空：true Map が空でない：false

解説

Map が空かどうかチェックします

GetCount

Map に格納されている要素数を取得します

```
int Map::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

GetWriteBuffer

省コピーバージョンで追加する要素のアドレスを取得します

```
void* Map::GetWriteBuffer(
        const char* szKey, ///< [in] Character Key
        int length ///< [in] Data Length
);
void* Map::GetWriteBuffer(
        const int key, ///< [in] Number Key
        int length ///< [in] Data Length
);
```

パラメータ

szKey,Key

[in] 追加する要素のキー。

length

[in] 追加する要素のサイズ(バイト数)

戻り値

void*

追加する要素のアドレス。NULL の場合、追加する要素のアドレス取得に失敗

解説

省コピーバージョンで要素を追加する際に予め、追加する要素のアドレスを取得します

GetWriteBuffer は必要な領域を確保し、確保した先頭のアドレスを戻します

領域の確保に失敗した場合は NULL を返します

```
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
char* pGetBuffer = myMap.GetWriteBuffer( “ABC” ,pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myMap.Set( pafee_strlen(setData) );
```

参照

Pafee::List

Pafee::pafee_memcpy

Pafee::pafee_strlen

Set

Map に要素を追加します

```
bool Map::Set(
		const char* szKey, ///< [in] Character Key
		void* pData, ///< [in] Data
		int length ///< [in] Data Length
);
bool Map::Set(
		int key, ///< [in] Number Key
		void* pData, ///< [in] Data
		int length ///< [in] Data Length
);
bool Map::Set(
		int length ///< [in] Data Length
);
```

パラメータ

szKey,Key

[in] 追加する要素のキー

pData

[in] 追加する要素の先頭アドレス

length

[in] 追加する要素のサイズ(バイト数)

戻り値

bool

リスト追加成功:true リスト追加失敗:false

解説

Map に要素を追加します。省コピーバージョンは予め、GetWriteBuufer 関数で取得したアドレスに、データを入れておく必要があります

キーが重複した場合データが上書きされます

```
char setData[16];
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
myMap.Set( “ABC”, setData, pafee_strlen(setData));
//省コピーバージョン
char* pGetBuffer = myMap.GetWriteBuffer( “ABC” ,pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myMap.Set(pafee_strlen(setData));
```

参照

Pafee::List

Pafee::pafee_memcpy

Pafee::pafee_strlen

Get

指定要素のデータを取得します

```
bool Map::Get(
		const char* szKey, ///< [in] Character key
		void* pBuffer, ///< [out] Prepared Buffer
		int* pLength, ///< [out] Data Length
		int bufSize ///< [in] Prepared Buffer Length
);
bool Map::Get(
		int key, ///< [in] Number Key
		void* pBuffer, ///< [out] Prepared Buffer
		int* pLength, ///< [out] Data Length
		int bufSize ///< [in] Prepared Buffer Length
);
```

パラメータ

szKey,Key

[in] 取得する要素のキー

pBuffer

[out] 取得したデータを入れるﾊﾞｯﾌｧのアドレス

pLength

[out] 取得したデータの長さを入れるﾊﾞｯﾌｧのアドレス

bufSize

[in] 取得したデータを入れるﾊﾞｯﾌｧのサイズ

戻り値

bool

成功：true 失敗:false

解説

Map の要素が空の場合、false を返します

bufSize より *pLength の方が大きい場合、false を返します

Get（省コピーバージョン）

指定要素のデータを取得します

```
void* Map::Get(
        const char* szKey, ///< [in] Character Key
        int* pLength ///< [out] Data Length
);
void* Map::Get(
        int key, ///< [in] Number Key
        int* pLength ///< [out] Data Length
);
```

パラメータ

szKey,Key

[in] 取得する要素のキー

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

戻り値

void*

取得したデータを入れるバッファのアドレス

解説

Map の要素が空の場合、NULL を返します

operator[]

指定要素のデータを取得します

```
void* Map::operator [] (
        const char* szKey ///< [in] Character Key
);
void* Map::operator [] (
        int key ///< [in] Number Key
);
```

パラメータ

szKey,Key

[in] 取得する要素のキー

戻り値

void*

取得したデータを入れるバッファのアドレス

解説

指定要素のデータを取得します

GetStartPosition

先頭要素の位置を取得します。

```
Map::POSITION GetStartPosition(void);
```

パラメータ

なし

戻り値

POSITION

先頭要素の位置

解説

リストの要素が空の場合、NULL を返します

GetNext

現在の位置の要素のデータを取得して、pos を次の要素の位置にします

```
bool Map::GetNext(
        Map::POSITION& pos, ///< [in/out] specified position
        void* pBuffer, ///< [out] Prepared Buffer
        int* pLength, ///< [out] Data Length
        int bufSize ///< [in] Prepared Buffer Length
);
```

パラメータ

pos

[in/out] リストの位置

pBuffer

[out] 取得したデータを入れるバッファのアドレス

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

bufSize

[in] 取得したデータを入れるバッファのサイズ

戻り値

bool

成功:true 失敗:false

解説

本関数は現在の pos の位置のデータを取得します

データを取得後、次の位置を指すように pos を更新します

bufSize < *pLenght のとき false を返します

要素が空のとき false を返します

```
char buf[256];
int Length;
POSITION pos = myMap.GetStartPosition();
bool rtn = myMap.GetNext( pos, Buffer, &Length );
if ( rtn == false )
{
  //取得失敗
}
```

GetNext（省コピーバージョン）

現在の位置の要素のデータを取得して、pos を次の要素の位置にします

```
void* Map::GetNext(
        Map::POSITION& pos, ///< [in/out] specified position
        int* pLength ///< [out] Data Length
);
```

パラメータ

pos

[in/out] リストの位置

pLength

[out] 要素のデータの長さ

戻り値

void*

成功：現在の要素のバッファアドレス 失敗：NULL

解説

本関数は現在の pos の位置のデータを取得します

データを取得後、次の位置を指すように pos を更新します

要素が空のとき NULL を返します

```
char buf[256];
char* pBuf;
int Length;
POSITION pos = myMap.GetStartPosition();
char* pBuf = myMap.GetNext( pos, &Length );
if ( pBuf == NULL ) { //取得失敗 }
pafee_memcpy ( buf, pBuf, Lenfth );
```

Remove

指定要素を削除します。

```
void Map::Remove(
		const char* szKey ///< [in] Character Key
);
void Map::Remove(
		int key ///< [in] Number Key
);
```

パラメータ

szKey,key

[in] 削除する要素のキー

戻り値

なし

解説

指定要素が空のときなしもしません

RemoveAll

要素を全て削除します

```
void RemoveAt(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

5.8 MultiMap

Pafee::MultiMap はキーと値をペアで管理するクラスです

Map との違いはキーの重複を許可するかどうかです

必要条件

ヘッダー：PafeeMultiMap.h

5.8.1 クラスメンバー覧

コンストラクタ

MultiMap	デフォルトコンストラクタ
MultiMap	コピーコンストラクタ

操作

Initialize	初期化します
operator =	既存の MultiMap をコピーします
GetCount	MultiMap に格納されている要素数を取得します
IsEmpty	MultiMap が空かチェックします
GetWriteBuffer	省コピーバージョンで追加する要素のアドレスを取得します
Set	MultiMap に要素を追加します
GetStartPosition	先頭要素の位置を取得します
GetNext	現在の位置の要素のデータを取得して、pos を次の要素の位置にします
Remove	指定要素を削除します
RemoveAll	要素を全て削除します

5.8.2 クラスメンバ詳細

MultiMap

空の MultiMap を構築します

```
MultiMap::MultiMap(
        int hashSize, ///< [in] hash size
        Allocator* pAllocator ///< [in] Memory Allocator
);
MultiMap::MultiMap(
        const MultiMap& mmpc, ///< [in] initial value of MultiMap
        Allocator* pAllocator ///< [in] Memory Allocator
);
```

パラメータ

hashSize

[in] hash テーブルのサイズを指定します

pAllocator

[in] メモリを確保するアロケータクラスを指定します

mmpc

[in] コピーMultiMap オブジェクト

戻り値

なし

解説

空の MultiMap オブジェクトを作成します

// Allocator の指定をしない場合
Pafee:MultiMap myMMap(100);
//Allocator を指定した場合
Pafee::MultiMap myMMap(100,&FixedAllocator);
//既存のリストを代入し初期化する場合
Pafee::MultiMup myMMap = yourMMap;

参照

Pafee::Allocator クラス

Initialize

MultiMap を初期化します

```
bool MultiMap::Initialize(
        int hashSize, ///< [in] hash size
        Allocator* pAllocator ///< [in] Memory Allocator
);
```

パラメータ

hashSize

[in] hash テーブルのサイズを指定します

pAllocator

[in] メモリを確保するアロケータクラスを指定します

戻り値

bool

初期化成功：true　失敗:false

解説

生成直後、1 回のみ本関数で初期化することができます

参照

Pafee::Allocator クラス

IsEmpty

MultiMap が空かチェックします。

```
bool MultiMap::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

MultiMap が空：true MultiMap が空でない：false

解説

MultiMap が空かどうかチェックします

GetCount

MultiMap に格納されている要素数を取得します

```
int MultiMap::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

GetWriteBuffer

省コピーバージョンで追加する要素のアドレスを取得します

```
void* MultiMap::GetWriteBuffer(
        const char* szKey, ///< [in] Character Key
        int length ///< [in] Data Length
);
void* MultiMap::GetWriteBuffer(
        const int key, ///< [in] Number Key
        int length ///< [in] Data Length
);
```

パラメータ

szKey,Key

[in] 追加する要素のキー

length

[in] 追加する要素のサイズ(バイト数)

戻り値

void*

追加する要素のアドレス。NULL の場合、追加する要素のアドレス取得に失敗

解説

省コピーバージョンで要素を追加する際に予め、追加する要素のアドレスを取得します

GetWriteBuffer は必要な領域を確保し、確保した先頭のアドレスを戻します

領域の確保に失敗した場合は NULL を返します

```
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
char* pGetBuffer = myMMap.GetWriteBuffer( “ABC” ,pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myMMap.Set( pafee_strlen(setData) );
```

参照

Pafee::List

Pafee::pafee_memcpy

Pafee::pafee_strlen

Set

MultiMap に要素を追加します

```
bool MultiMap::Set(
		const char* szKey, ///< [in] Character Key
		void* pData, ///< [in] Data
		int length ///< [in] Data Length
);
bool MultiMap::Set(
		int key, ///< [in] Number Key
		void* pData, ///< [in] Data
		int length ///< [in] Data Length
);
bool MultiMap::Set(
		int length ///< [in] Data Length
);
```

パラメータ

szKey,Key

[in] 追加する要素のキー

pData

[in] 追加する要素の先頭アドレス

length

[in] 追加する要素のサイズ(バイト数)

戻り値

bool

追加成功:true 追加失敗:false

解説

MultiMap に要素を追加します。省コピーバージョンは予め、GetWriteBuufer 関数で取得したアドレスに、データを入れておく必要があります。

キーが重複してもデータが追加されます

```
char setData[16];
pafee_memcpy( setData, “Hello World”, pafee_strlen( “Hello World” ) );
myMMap.Set( “ABC”, setData, pafee_strlen(setData));
//省コピーバージョン
char* pGetBuffer = myMMap.GetWriteBuffer( “ABC” ,pafee_strlen(setData));
pafee_memcpy( pGetBuffer, setData, pafee_strlen(setData));
myMMap.Set(pafee_strlen(setData));
```

参照

Pafee::List

Pafee::pafee_memcpy

Pafee::pafee_strlen

GetStartPosition

先頭要素の位置を取得します。

```
MultiMap::POSITION MultiMap::GetStartPosition(void);
MultiMap::POSITION MultiMap::GetStartPosition(
        const char* szKey ///< [in] Character key
);
MultiMap::POSITION MultiMap::GetStartPosition(
        int key ///< [in] Number key
);
```

パラメータ

szKey/key

[in] 取得する要素のキー

戻り値

MultiMap::POSITION

[in] 先頭要素の位置

解説

リストの要素が空の場合、NULL を返します

キーを指定した場合指定したキーに対応する要素のみを取得します

参照

MultiMap::GetNext

GetNext

現在の位置の要素のデータを取得して、pos を次の要素の位置にします

```
bool MultiMap::GetNext(
        MultiMap::POSITION& pos, ///< [in/out] specified position
        void* pBuffer, ///< [out] Prepared Buffer
        int* pLength, ///< [out] Data Length
        int bufSize ///< [in] Prepared Buffer Length
);
```

パラメータ

pos

[in/out] 位置

pBuffer

[out] 取得したデータを入れるバッファのアドレス

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

bufSize

[in] 取得したデータを入れるバッファのサイズ

戻り値

bool

成功:true、失敗:false

解説

本関数は現在の pos の位置のデータを取得します

データを取得後、次の位置を指すように pos を更新します

要素が空のとき false を返します

GetStartPosition でキーを指定した場合は、指定したキーに対応する要素のみを取得します

```
POSITION pos = myMMap.GetStartPosition();
while(1)
{
  bool rtn = myMap.GetNext( pos, Buffer, &Length );
  if ( rtn == false )
  break;
  pafee_memcpy ( buf, pBuf, Length);
}
```

GetNext（省コピーバージョン）

現在の位置の要素のデータを取得して、pos を次の要素の位置にします

```
void* MultiMap::GetNext(
        MultiMap::POSITION& pos, ///< [in/out] specified position
        int* pLength ///< [out] Data Length
);
```

パラメータ

pos

[in/out] リストの位置

pLength

[out] 取得したデータの長さを入れるバッファのアドレス

戻り値

void*

取得する要素のアドレス

解説

本関数は現在の pos の位置のデータを取得します

データを取得後、次の位置を指すように pos を更新します

要素が空のとき NULL を返します。

GetStartPosition でキーを指定した場合は、指定したキーに対応する要素のみを取得します

```
char buf[256];
int Length;
POSITION pos = myMMap.GetStartPosition();
while(1)
{
  char* pBuffer =  myMMap.GetNext( pos, &Length );
  if ( pBuffer == NULL )
  {
    break;
  }
  pafee_memcpy ( buf, pBuf, Length);
}
```

Remove

指定要素を削除します

```
void MultiMap::Remove(
		const char* szKey ///< [in] Character Key
);
void MultiMap::Remove(
		int key ///< [in] Number Key
);
```

パラメータ

[in] szKey,key

削除する要素のキー

戻り値

なし

解説

指定要素が空のときなしもしません

RemoveAll

要素を全て削除します

```
void MultiMap::RemoveAt(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

テンプレート版コレクションクラス

6. テンプレート版コレクション

テンプレートを使った汎用的なデータ構造を提供します。

6.1 ArrayT

6.2 ListT

Pafee::ListT はテンプレートを格納する双方向リンクリストクラスです

要素を追加する毎にメモリを確保して格納します

必要条件

ヘッダー：PafeeListT.h

関連ファイル：

6.2.1 クラスメンバー覧

コンストラクタ

ListT	コンストラクタ

操作

Initialize	初期化します
operator =	既存のリストをコピーします
GetCount	List に格納されている要素数を取得します
IsEmpty	リストが空かチェックします
AddFront	リストの先頭に要素を追加します
AddBack	リストの末尾に要素を追加します
GetFront	先頭要素のデータを取得します
GetBack	末尾要素のデータを取得します
GetFrontPosition	先頭要素の位置を取得します
GetBackPosition	末尾要素の位置を取得します
GetNext	現在の位置の要素のデータを取得して、pos を次の要素の位置にします
GetPrevious	現在の位置の要素のデータを取得して、pos を前の要素の位置にします
GetAt	現在の位置の要素のデータを取得します
SetAt	現在の位置にデータをセットします
RemoveFront	先頭の要素を削除します
RemoveBack	末尾の要素を削除します
RemoveAt	pos の位置の要素を削除します
RemoveAll	要素を全て削除します

6.2.2 クラスメンバ詳細

ListT

リストを構築します

```
ListT::ListT(
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
ListT::ListT(
        const ListT& lstc, ///< [in] initial value of List
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
```

パラメータ

pAllocator

[in] メモリを確保するアロケータクラス

lstc

[in] コピーListT オブジェクト

戻り値

なし

解説

リストオブジェクトを作成します

```
// Allocator の指定をしない場合
Pafee::ListT<int> myList;
//Allocator を指定した場合
Pafee::ListT<int> myListp( &FixedAllocator);
//既存のリストを代入し初期化する場合
Pafee::ListT myListc<int> = yourList;
```

参照

Pafee::Allocator クラス

Initialize

リストを初期化します

```
bool ListT::Initialize(
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
```

パラメータ

pAllocator

[in] メモリを確保するアロケータクラス

戻り値

bool

初期化成功：true 失敗:false

解説

生成時、メモリアロケータの指定を行わなかった場合、1 回のみ本関数で初期化することができます

参照

Pafee::Allocator クラス

IsEmpty

リストが空かチェックします

```
bool ListT::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

リストが空：true リストが空でない：false

解説

リストが空かどうかチェックします

GetCount

ListT に格納されている要素数を取得します

```
int ListT::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

AddFront

リストの先頭に要素を追加します

```
bool ListT::AddFront(
		const T& data ///< [in] Set value
);
```

パラメータ

data

[in] 追加する要素のクラス

戻り値

bool

リスト追加成功：true リスト追加失敗：false

解説

リストの先頭に要素を追加します

AddBack

リストの末尾に要素を追加します

```
bool ListT::AddBack(
        const T& data ///< [in] Set value
);
```

パラメータ

data

[in] 追加する要素のクラス

戻り値

bool

リスト追加成功：true リスト追加失敗：false

解説

リストの末尾に要素を追加します

GetFront

先頭要素のデータを取得します

```
T& ListT::GetFront(void);
```

パラメータ

なし

戻り値

T&

取得したデータ

解説

リストの要素が空の場合、例外が発生します

GetBack

末尾要素のデータを取得します

```
T& ListT::GetBack(void);
```

パラメータ

なし

戻り値

T&

取得したデータ

解説

リストの要素が空の場合、例外が発生します

GetFrontPosition

先頭要素の位置を取得します

```
ListT::POSITION ListT::GetFrontPosition(void);
```

パラメータ

なし

戻り値

POSITION

先頭要素の位置

解説

リストの要素が空の場合、NULL を返します

GetBackPosition

末尾要素の位置を取得します。

```
ListT::POSITION ListT::GetBackPosition(void);
```

パラメータ

なし

戻り値

POSITION

末尾要素の位置

解説

リストの要素が空の場合、NULL を返します

GetNext

現在の位置の要素のデータを取得して、pos を次の要素の位置にします

```
T& ListT::GetNext(
        POSITION& pos ///< [in/out] Position
);
```

パラメータ

pos

[in/out] リストの位置

戻り値

T&

取得したデータ

解説

本関数は現在の pos の位置のデータを取得します

データを取得後、次の位置を指すように pos を更新します

要素が空のとき例外が発生します

```
Element buf;
POSITION pos = myList.GetFrontPosition();
Element buf =  myList.GetNext( pos );
```

GetPrevious

現在の位置の要素のデータを取得して、pos を前の要素の位置にします

```
T& ListT::GetPrevious(
        POSITION &pos ///< [in/out] Next Node Pointer
);
```

パラメータ

pos

[in/out] リストの位置

戻り値

T&

取得したデータ

解説

本関数は現在の pos の位置のデータを取得します

データを取得後、次の位置を指すように pos を更新します

要素が空のとき例外が発生します

```
Element buf;
POSITION pos = myList.GetFrontPosition();
Element buf =  myList. GetPrevious ( pos );
```

GetAt

現在の位置の要素のデータを取得します

```
bool ListT::GetAt(
        POSITION& pos, ///< [in] Position
        T& Data    ///< [in] The data that was gotten
)
```

パラメータ

pos

[in] リストの位置

Data

[in] 取得したデータを入れるバッファ

戻り値

bool

成功:true 失敗:false

解説

pos の状態は変わりません

```
element buf;
POSITION pos = myList.GetFrontPosition();
bool rtn = myList. GetAt( pos, Buffer );
if ( rtn == false )
{
  //取得失敗
}
```

SetAt

現在の位置にデータをセットします。

```
bool ListT::SetAt(
        POSITION& pos, ///< [in] Position
        const T& Data    ///< [in] data to set
);
```

パラメータ

pos

[in] リストの位置

Data

[in] 追加するデータ

戻り値

bool

成功:true 失敗:false

解説

要素が空のとき false を返します

```
element buf;
// データのセット;
bool rtn = myList. SetAt( pos, buf );
if ( rtn == false )
{
  //取得失敗
}
```

RemoveFront

先頭の要素を削除します

```
void ListT::RemoveFront(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

RemoveBack

末尾の要素を削除します

```
void ListT::RemoveBack(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

RemoveAt

pos の位置の要素を削除します

```
void ListT::RemoveAt(
        POSITION pos ///< [in] Position
)
```

パラメータ

pos

リストの位置

戻り値

なし

解説

要素が空のときなしもしません

RemoveAll

要素を全て削除します

```
void ListT::RemoveAt(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

6.3 QueueT

Pafee::Queue はクラスデータの FIFO（FirstIn-FirstOut）を提供します

ListT のラッパーとして実装されています

必要条件

ヘッダー：PafeeQueueT.h

関連ファイル： PafeeListT.h,PafeeListT.cpp

6.3.1 クラスメンバー覧

コンストラクタ

QueueT	デフォルトコンストラクタ

操作

operator =	既存のリストをコピーします
GetCount	QueueT に格納されている要素数を取得します
IsEmpty	リストが空かチェックします
Enqueue	リストの先頭に要素を追加します
Dequeue	先頭要素のデータを取得します
Peek	先頭要素のデータを取得します
RemoveFront	先頭の要素を削除します
RemoveAll	要素を全て削除します

6.3.2 クラスメンバ詳細

QueueT

空のキューを構築します

```
QueueT ::QueueT(
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
QueueT ::QueueT(
        const QueueT& quec, ///< [in] initial value of Queue
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
```

パラメータ

pAllocator

[in] メモリを確保するアロケータクラス

quec

[in] コピーQueueT オブジェクト

戻り値

なし

解説

空のキューオブジェクトを作成します

```
// Allocator の指定をしない場合
Pafee::QueueT<Pafee::String> myQueue;
//Allocator を指定した場合
Pafee::QueueT<Pafee::String>    myQueue ( &FixedAllocator);
//既存のキューを代入し初期化する場合
Pafee::QueueT <Pafee::String> myQueue= yourQueue;
```

参照

Pafee::Allocator クラス

GetCount

QueueT に格納されている要素数を取得します

```
int QueueT ::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

IsEmpty

キューが空かチェックします

```
bool QueueT ::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

キューが空：true　キューが空でない：false

解説

キューが空かどうかチェックします

Enqueue

キューの先頭に要素を追加します

```
bool QueueT ::Enqueue(
        const T& data ///[in] Input data
);
```

パラメータ

data

[in] 追加する要素

戻り値

bool

キュー追加成功:true キュー追加失敗:false

解説

キューに要素を追加します

```
Pafee::String setData=" Hello World" ;
myQueue.Enqueue(setData);
```

Dequeue

要素のデータを取得します

```
bool QueueT ::Dequeue(
        T& data ///[out] Output data
);
```

パラメータ

data

[out] 取得したデータを入れるバッファ

戻り値

bool

成功：true 失敗:false

解説

キューの要素が空の場合、false を返します

Peek

要素のデータを取得します

```
T& QueueT ::Peek(void)
```

パラメータ

なし

戻り値

T&

取得したデータ

解説

取得した要素をキューから削除しません

キューの要素が空の場合、例外が発生します

RemoveFront

先頭の要素を削除します。

```
void QueueT ::RemoveFront(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

RemoveAll

要素を全て削除します。

```
void QueueT ::RemoveAt(void)
```

戻り値

なし

解説

要素が空のときなしもしません

6.4 PriorityQueueT

Pafee::PriorityQueueT はクラスデータの優先順位付 FIFO（FirstIn-FirstOut）を提供します

ListT のラッパーとして実装されています

必要条件

ヘッダー：PafeePriorityQueueT.h

関連ファイル： PafeeListT.h,PafeeListT.cpp

6.4.1 クラスメンバー覧

コンストラクタ

PriorityQueueT	コンストラクタ

操作

operator =	既存のリストをコピーします
Initialize	キューを初期化します
GetCount	PriorityQueue に格納されている要素数を取得します
IsEmpty	プライオリティキューが空かチェックします
Enqueue	プライオリティキューの先頭に要素を追加します
Dequeue	先頭要素のデータを取得します
Peek	先頭要素のデータを取得します
RemoveFront	先頭の要素を削除します
void RemoveAll	要素を全て削除します

6.4.2 クラスメンバ詳細

ProrityQueueT

空のプライオリティキューを構築します

```
PriorityQueueT ::PriorityQueueT(
        int priorityRange=0, ///< [in] Priority Range
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
PriorityQueueT ::PriorityQueueT(
        const PriorityQueueT& pqtuec, ///< [in] initial value of Queue
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
```

パラメータ

priorityRange

［in］優先順位の範囲

pAllocator

［in］メモリを確保するアロケータクラス

pqtuec

［in］コピーPriorityQueueT オブジェクト

戻り値

なし

解説

プライオリティキューオブジェクトを作成します

// Allocator の指定をしない場合

Pafee::PriorityQueueT<Pafee::String> myPQueue;

//Allocator を指定した場合

Pafee::PriorityQueueT<Pafee::String> myPQueue(&FixedAllocator);

//既存のリストを代入し初期化する場合

Pafee::Priority myPQueue<Pafee::String> = yourPQueue;

参照

Pafee::Allocator クラス

Initialize

キューを初期化します

```
bool PriorityQueueT ::Initialize(
        int priorityRange, ///< [in] Priority Range
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
```

パラメータ

priorityRange

[in] 優先順位の範囲

pAllocator

[in] メモリを確保するアロケータクラス

戻り値

bool

初期化成功:true 失敗:false

解説

生成時、メモリアロケータの指定を行わなかった場合、1 回のみ本関数で初期化することができます

参照

Pafee::Allocator クラス

GetCount

PriorityQueue に格納されている要素数を取得します

```
int PriorityQueueT ::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

IsEmpty

プライオリティキューが空かチェックします

```
bool PriorityQueueT ::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

プライオリティキューが空：true　プライオリティキューが空でない：false

解説

プライオリティキューが空かどうかチェックします

Enqueue

キューに要素を追加します

```
bool PriorityQueueT ::Enqueue (
        int priority, ///< [in] Priority
        const T& Data ///< [in] Data
);
```

パラメータ

priority

[in] 優先順位

Data

[in] 追加する要素

戻り値

bool

キュー追加成功:true キュー追加失敗:false

解説

プライオリティキューに要素を追加します

Dequeue

要素のデータを取得します

```
bool PriorityQueueT ::Dequeue(
        int* pPriority, ///< [out] Priority
        T& Buffer ///< [out] Data
);
```

パラメータ

pPriority

[out] 取得する要素の優先順位を格納するアドレス

Buffer

[out] 取得したデータを入れるバッファ

戻り値

bool

成功：true 失敗:false

解説

キューの要素が空の場合、false を返します

Peek

要素のデータを取得します

```
bool PriorityQueueT ::Peek(
        int* pPriority, ///< [out] Priority
        T& Buffer ///< [out] Data
);
```

パラメータ

pPriority

[out] 取得する要素の優先順位を格納するアドレス

Buffer

[out] 取得した要素を入れるバッファ

戻り値

bool

成功：true 失敗:false

解説

取得した要素をキューから削除しません

キューの要素が空の場合、false を返します

RemoveFront

先頭の要素を削除します

```
void PriorityQueueT ::RemoveFront(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

RemoveAll

要素を全て削除します。

```
void PriorityQueueT ::RemoveAt(void)
```

戻り値

なし

解説

要素が空のときなしもしません。

6.5 StackT

Pafee::StackT はテンプレートクラスの Stack を提供します。

ListT のラッパーとして実装されています。

必要条件

ヘッダー：PafeeStackT.h

関連ファイル： PafeeListT.h,PafeeListT.cpp

6.5.1 クラスメンバー覧

コンストラクタ

StackT	デフォルトコンストラクタ

操作

operator =	既存のスタックをコピーします
GetCount	StackT に格納されている要素数を取得します
IsEmpty	スタックが空かチェックします
Push	スタックに要素を Push します
Pop	スタックの要素を Pop します
Peek	先頭要素のデータを取得します
RemoveFront	先頭の要素を削除します
RemoveAll	要素を全て削除します

6.5.2 クラスメンバ詳細

StackT

空のスタックを構築します。

```
StackT ::StackT(
        Allocator* pAllocator=NULL ///< [in] memory allocator
);
StackT ::StackT(
        const StackT& stkc, ///< [in] initial value of Stack
        Allocator* pAllocator=NULL ///< [in] Memory allocator
);
```

パラメータ

pAllocator

メモリを確保するアロケータクラス

stkc

コピーStackT オブジェクト

戻り値

なし

解説

スタックオブジェクトを作成します。

// Allocator の指定をしない場合

Pafee::StackT<Pafee::String> myStack;

//Allocator を指定した場合

Pafee::StackT<Pafee::String> myStack(&FixedAllocator);

//既存のスタックを代入し初期化する場合

Pafee::StckT<Pafee::String> myStack = yourStack;

参照

Pafee::Allocator クラス

GetCount

StackT に格納されている要素数を取得します

```
int StackT ::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

IsEmpty

StackT が空かチェックします。

```
bool StackT ::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

スタックが空：true　スタックが空でない：false

解説

スタックが空かどうかチェックします

Push

スタックに要素を Push します

```
bool StackT ::Push(
        const T& Data ///< [in] Push data
);
```

パラメータ

Data

Push する要素

戻り値

bool

Push 成功:true Push 失敗:false

解説

StackT に要素を Push します。

```
Pafee::String setData=”Hello World”;
Pafee::StackT<Pafee::String> myStack;
myStack.Push(setData);
```

Pop

StackT のデータを Pop します

```
bool StackT ::Pop(
        T& Buffer ///< [out] Pop data
);
```

パラメータ

Buffer

Pop した要素を入れるバッファ

戻り値

Bool

成功：true　失敗:false

解説

スタックの要素が空の場合、false を返します

Peek

先頭の要素のデータを取得します

```
bool StackT ::Peek(
        T& Buffer ///< [in] Get data
);
```

パラメータ

Buffer

取得したデータを入れるバッファ

戻り値

bool

成功：true　失敗:false

解説

取得した要素をスタックから削除しません

スタックの要素が空の場合、false を返します

RemoveFront

先頭の要素を削除します

```
void StackT ::RemoveFront(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

RemoveAll

要素を全て削除します

```
void StackT ::RemoveAt(void)
```

戻り値

なし

解説

要素が空のときなしもしません

6.6 RingBufferT

Pafee::RingBufferT はテンプレートクラスの連続したリングバッファを提供します

必要条件

ヘッダー：PafeeRingBufferT.h

6.6.1 クラスメンバー覧

コンストラクタ

RingBufferT	デフォルトコンストラクタ
RingBufferT	コピーコンストラクタ

デストラクタ

~RingBufferT	デストラクタ

操作

Initialize	初期化します
operator =	既存のリングバッファをコピーします
GetCount	リングバッファに格納されている要素数を取得します
IsEmpty	リングバッファが空かチェックします
Enqueue	リングバッファの末尾に要素を追加します
Dequeue	リングバッファの先頭要素を取得します
Peek	先頭要素のデータを取得します
ClearAll	要素を全て削除します

6.6.2 クラスメンバ詳細

RingBufferT

空のリングバッファを構築します

```
RingBufferT ::RingBufferT(
        int capacity=0, ///< [in] and one element in can setting of the following as argument and ring buffer
        Allocator* pAllocator=NULL ///< [in] MemoryAllocator
);
RingBufferT ::RingBufferT(
        RingBufferT& rbc, ///< [in] Initialize RingBuffer
        Allocator* pAllocator=NULL ///< [in] MemoryAllocator
);
```

パラメータ

capacity

[in] リングバッファの要素数

pAllocator

[in] メモリを確保するアロケータクラス

rbc

[in] コピーRingBuffer オブジェクト

戻り値

なし

解説

空のリングバッファオブジェクトを作成します

```
// Allocator の指定をしない場合
Pafee::RingBufferT<int> myRing(20);
//Allocator を指定した場合
Pafee::RingBufferT<int> myRing( 20,&FixedAllocator);
//既存のスタックを代入し初期化する場合
Pafee::RingBuffer<int> myRing = yourRing;
```

参照

Pafee::Allocator クラス

Initialize

リングバッファを初期化します

```
bool RingBufferT ::Initialize(
        int capacity,    ///< [in] capacity
        Allocator* pAllocator=NULL ///< [in] MemoryAllocator
);
```

パラメータ

capacity

リングバッファの要素数を指定します

pAllocator

メモリを確保するアロケータクラスを指定します

戻り値

bool

成功：true、失敗：false

解説

生成直後、1 回のみ本関数で初期化することができます

参照

Pafee::Allocator クラス

GetCount

RingBufferT に格納されている要素数を取得します

```
int RingBufferT::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

IsEmpty

RingBufferT が空かチェックします

```
bool RingBufferT::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

リングバッファが空：true リングバッファが空でない：false

解説

リングバッファが空かどうかチェックします

Enqueue

リングバッファの末尾に要素を追加します

```
bool RingBufferT ::Enqueue (
        const T& Data    ///< [in] Data
);
```

パラメータ

Data

[in] 要素

戻り値

bool

追加成功:true 追加失敗:false

解説

RingBufferT に要素を追加します

Pafee::RingBufferT<Pafee::String> myRing;

Pafee::String setData=”Hello World”;

myRing.Enqueue(setData);

Dequeue

リングバッファの先頭要素を取得します

```
bool RingBufferT ::Dequeue(
        T& Buffer ///< [out] Prepared Buffer
);
```

パラメータ

Buffer

[out] 取得したデータを入れるバッファ

戻り値

bool

成功：true 失敗:false

解説

リングバッファの要素が空の場合、false を返します

Peek

先頭の要素のデータを取得します

```
bool RingBufferT ::Peek(
        T& Buffer ///< [out] Prepared Buffer
);
```

パラメータ

Buffer

[out] 取得したデータを入れるバッファ

戻り値

bool

成功：true 失敗:false

解説

取得した要素をリングバッファから削除しません

本関数はリードバッファポインタを更新しません

リングバッファの要素が空の場合、false を返します

ClearAll

要素を全て削除します

```
void RingBufferT ::ClearAll(void);
```

戻り値

なし

解説

要素が空のときなしもしません

6.7 MapT

Pafee::MapT はキーとクラスをペアで管理するクラスです

必要条件

ヘッダー：PafeeMapT.h

6.7.1 クラスメンバ詳細

コンストラクタ

MapT	コンストラクタ

操作

Initialize	初期化します
operator =	既存のリストをコピーします
GetCount	MapT に格納されている要素数を取得します
IsEmpty	MapT が空かチェックします
Set	MapT に要素を追加します
Get	指定キーの要素のデータを取得します
operator □	指定キーの要素のデータを取得します
Position	先頭要素の位置を取得します
GetNext	現在の位置の要素のデータを取得して、pos を次の要素の位置にします
Remove	指定要素を削除します
RemoveAll	要素を全て削除します

6.7.2 クラスメンバ詳細

MapT

MapT を構築します。

```
MapT ::MapT(
        int hashSize=0, ///< [in] Hash table size
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
MapT ::MapT(
        const MapT& mptc, ///< [in] initial value of Map
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
```

パラメータ

hashSize

[in] hash テーブルのサイズを指定します

pAllocator

[in] メモリを確保するアロケータクラスを指定します

mptc

[in] コピーMapT オブジェクト

戻り値

なし

解説

MapT オブジェクトを作成します。

// Allocator の指定をしない場合

Pafee::MapT<Pafee::String> myMap(100);

//Allocator を指定した場合

Pafee::MapT<Pafee::String> myMap(100,&FixedAllocator);

//既存のリストを代入し初期化する場合

Pafee::MapT<Pafee::String> myMap = yourMap;

参照

Pafee::Allocator クラス

Initialize

MapT を初期化します。

```
bool MapT ::Initialize(
        int hashSize, ///< [in] Hash table size
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
```

パラメータ

hashSize

[in] hash テーブルのサイズを指定します

pAllocator

[in] メモリを確保するアロケータクラスを指定します

戻り値

bool

初期化成功:true 失敗:false

解説

生成直後、1 回のみ本関数で初期化することができます

参照

Pafee::Allocator クラス

IsEmpty

MapT が空かチェックします

```
bool MapT ::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

MapT が空：true MapT が空でない：false

解説

MapT が空かどうかチェックします

GetCount

MapT に格納されている要素数を取得します

```
int MapT::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

Set

MapT に要素を追加します

```
bool MapT::Set(
		const char* szKey, ///[in] Key
		const T& data ///[in] Set data
);
bool MapT::Set(
		int key, ///[in] Key
		const T& data ///[in] Set data
);
```

パラメータ

szKey,Key

[in] 追加する要素のキー

data

[in] 追加する要素

戻り値

bool

マップ追加成功:true マップ追加失敗:false

解説

MapT に要素を追加します

キーが重複した場合データが上書きされます

```
Pafee::MapT<Pafee::String> myMap;
Pafee::String setData="Hello World";
myMap.Set(“ABC”, setData);
```

Get

指定要素のデータを取得します。

```
bool MapT::Get(
		const char* szKey, ///[in] Key
		T& data ///[in] Set data
);
bool MapT::Get(
		int key, ///< [in] Key
		T& data ///< [in] Set data
);
```

パラメータ

szKey,Key

[in] 取得する要素のキー

data

[in] 取得したデータを入れるバッファ

戻り値

bool

成功：true 失敗:false

解説

MapT の要素が空の場合、false を返します

operator[]

指定要素のデータを取得します

```
T& MapT::operator [] (
        const char* szKey ///< [in] Key
);
T& MapT::operator [](
        int key ///< [in] Key
);
```

パラメータ

szKey,Key

[in] 取得する要素のキー

戻り値

なし

解説

指定要素のデータを取得します

GetStartPosition

先頭要素の位置を取得します

```
MapT::POSITION MapT::GetStartPosition(void);
```

パラメータ

なし

戻り値

POSITION

先頭要素の位置

解説

リストの要素が空の場合、NULL を返します

GetNext

現在の位置の要素のデータを取得して、pos を次の要素の位置にします

```
bool MapT::GetNext(
        POSITION& pos, ///< [in/out] position
        T& data ///< [out] Data
);
```

パラメータ

pos

[in/out] リストの位置

data

[out] 取得したデータを入れるバッファ

戻り値

bool

成功:true 失敗:false

解説

本関数は現在の pos の位置のデータを取得します

データを取得後、次の位置を指すように pos を更新します

```
Pafee::String Buffer;
POSITION pos = myMap.GetStartPosition();
bool rtn = myMap.GetNext( pos, Buffer );
if ( rtn == false )
{
  //取得失敗
}
```

Remove

指定要素を削除します

```
void MapT::Remove(
        const char* szKey ///< [in] Key to delete data
);
void MapT::Remove(
        int key ///< [in] Key to delete data
);
```

パラメータ

szKey,key

[in] 削除する要素のキー

戻り値

なし

解説

指定要素が空のときなしもしません

RemoveAll

要素を全て削除します

```
void MapT::RemoveAt(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

6.8 MultiMapT

Pafee::MultiMapT はキーとクラスをペアで管理するクラスです

MapT との違いはキーの重複を許可するかどうかです

必要条件

ヘッダー：PafeeMultiMapT.h

6.8.1 クラスメンバー覧

コンストラクタ

MultiMapT	コンストラクタ

操作

Initialize	初期化します
operator =	既存の MultiMapT をコピーします
GetCount	MultiMapT に格納されている要素数を取得します
IsEmpty	MultiMapT が空かチェックします
Set	MultiMapT に要素を追加します
GetStartPosition	先頭要素の位置を取得します
bool GetNext	現在の位置の要素のデータを取得して、pos を次の要素の位置にします
void Remove	指定要素を削除します
void RemoveAll	要素を全て削除します

6.8.2 クラスメンバ詳細

MultiMapT

MultiMapT を構築します

```
MultiMapT::MultiMapT(
        int hashSize=0,///[in] Hash table size
        Allocator* pAllocator=NULL ///[in] Memory allocator
);
MultiMapT::MultiMapT(
        const MultiMapT& mmpc, ///< [in] initial value of MultiMap
        Allocator* pAllocator=NULL ///< [in] Memory Allocator
);
```

パラメータ

hashSize

[in] テーブルのサイズ

pAllocator

[in] 確保するアロケータクラス

mmpc

[in] MultiMapT オブジェクト

戻り値

なし

解説

MultiMapT オブジェクトを作成します

// Allocator の指定をしない場合

Pafee:MultiMapT<Pafee::String> myMMap(100);

//Allocator を指定した場合

Pafee::MultiMapT<Pafee::String> myMMap(100,&FixedAllocator);

//既存のリストを代入し初期化する場合

Pafee::MultiMup<Pafee::String> myMMap = yourMMap;

参照

Pafee::Allocator クラス

Initialize

MultiMapT を初期化します

```
bool MultiMapT::Initialize(
        int hashSize,///[in] Hash table size
        Allocator* pAllocator=NULL ///[in] Memory allocator
);
```

パラメータ

hashSize

[in] テーブルのサイズを指定します

pAllocator

[in] 確保するアロケータクラスを指定します

戻り値

bool

初期化成功:true 失敗:false

解説

生成直後、1 回のみ本関数で初期化することができます

参照

Pafee::Allocator クラス

IsEmpty

MultiMapT が空かチェックします

```
bool MultiMapT::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

MultiMapT が空：true MultiMap が空でない：false

解説

MultiMapT が空かどうかチェックします

GetCount

MultiMapT に格納されている要素数を取得します

```
int MultiMapT::GetCount(void);
```

パラメータ

なし

戻り値

int

要素数

解説

戻り値が 0 の場合 IsEmpty()が true の場合と等価です

Set

MultiMapT に要素を追加します

```
bool MultiMapT::Set(
		const char* szKey,///[in] Key
		const T& data///[in] Set data
);
bool MultiMapT::Set(
		int key,///[in] Key
		const T& data///[in] Set data
);
```

パラメータ

szKey,Key

[in] 要素のキー

data

[in] セットするデータ

戻り値

bool

追加成功:true 追加失敗:false

解説

MultiMapT に要素を追加します

キーが重複してもデータが追加されます

```
Pafee::String setData=”Hello World”;
myMMap.Set(“ABC”, setData);
```

GetStartPosition

先頭要素の位置を取得します

```
POSITION MultiMapT::GetStartPosition(void);
POSITION MultiMapT::GetStartPosition(
        const char* szKey///[in] Key
);
POSITION MultiMapT::GetStartPosition(
        int key ///[in] Key
);
```

パラメータ

szKey,Key

[in] 要素のキー

戻り値

MultiMapT::POSITION

先頭要素の位置

解説

マップの要素が空の場合、NULL を返します

キーを指定した場合、指定したキーに対応する要素のみを取得します

参照

MultiMapT::GetNext

GetNext

現在の位置の要素のデータを取得して、pos を次の要素の位置にします

```
bool MultiMapT::GetNext(
        POSITION& pos, ///[in/out] Position
        T& data///[out] Get data
);
```

パラメータ

Pos

[in/out] リストの位置

data

[out] 取得するデータ

戻り値

bool

成功:true 失敗:false

解説

本関数は現在の pos の位置のデータを取得します

データを取得後、次の位置を指すように pos を更新します

GetStartPosition でキーを指定した場合は、指定したキーに対応する要素のみを取得します

```
Pafee::String data
POSITION pos = myMMap.GetStartPosition();
while(1)
  {
  bool myMMap.GetNext( pos, data);
  if ( rtn == false )
  {
    break;
  }
  //data をコピー
}
```

Remove

指定要素を削除します

```
void MultiMapT::Remove(
        const char* szKey///< [in] Key
);
void MultiMapT::Remove(
        int key ///< [in] Key
);
```

パラメータ

szKey,key

[in] 削除する要素のキー

戻り値

なし

解説

指定要素が空のときなしもしません

RemoveAll

要素を全て削除します

```
void MultiMapT::RemoveAt(void)
```

パラメータ

なし

戻り値

なし

解説

要素が空のときなしもしません

7. 文字列

文字列処理機能を提供します。

7.1 String

Pafee::String は文字列を扱うクラスです

必要条件

ヘッダー：PafeeString.h

7.1.1 クラスメンバー覧

コンストラクタ

String(Allocator* pAllocator = NULL)	デフォルトコンストラクタ
String(const String& src, Allocator* pAllocator = NULL)	コピーコンストラクタ
String(const char* szSrc, Allocator* pAllocator = NULL)	コンストラクタ

操作

Initialize	初期化します
operator =	String に代入します
operator +	加算します
operator +=	String に追加します
operator ==	同じか比較します
operator !=	異なるか比較します
operator <	右辺が左辺より大きいか比較します
operator >	右辺が左辺より小さいか比較します
operator <=	右辺が左辺以上か比較します
operator >=	右辺が左辺以下か比較します
Compare	比較します
CompareNoCase	大文字小文字を区別せず比較します
Find	先頭から検索します
ReverseFind	末尾から検索します
format	指定の書式に変換します
Insert	挿入します
Delete	削除します
Substring	部分文字列を取得します
Trim	指定文字を取り除きます
ToLower/ToUppe	小文字/大文字に変換します

GetBuffer	文字列の先頭アドレスを取得します
GetLength	String の長さを取得します
IsEmpty	String が空かチェックします

7.1.2 クラスメンバ詳細

String

文字列クラスを構築します

```
String(Allocator* pAllocator = NULL);
String(const String& src, Allocator* pAllocator = NULL);
String(const char* szSrc, Allocator* pAllocator = NULL);
```

パラメータ

pAllocator

[in] メモリを確保するアロケータクラスを指定します

src

[in] コピーString オブジェクトを指定します

szSRc

[in] 文字列のアドレスを指定します

戻り値

なし

解説

文字列ブジェクトを作成します。

```
// Allocator の指定をしない場合
Pafee::String myStr;
//Allocator を指定した場合
Pafee::String myStr(&FixedAllocator);
//既存の String を代入し初期化する場合
Pafee::String myString = yourString;
//リテラル文字列で初期化する場合
Pafee::String myString =”Hello World";
```

参照

Pafee::Allocator クラス

Initialize

String を初期化します。

```
void Initialize(Allocator* pAllocator);
```

パラメータ

pAllocator

[in] メモリを確保するアロケータクラスを指定します

戻り値

なし

解説

生成直後、1 回のみ本関数で初期化することができます

参照

Pafee::Allocator クラス

operator=

String に代入します

```
String& operator = (const String& src);
String& operator = (const char* szSrc);
String& operator = (char src);
```

パラメータ

src/szSrc

[in] String クラスに代入する String オブジェクト、リテラル文字列及び文字

戻り値

自身の文字列

解説

String に代入します

operator+

加算します

```
friend String operator + (const String& src1, const String& src2);
friend String operator + (const String& src1, const char* szSrc2);
friend String operator + (const char* szSrc1, const String& src2);
friend String operator + (const String& src1, char src2);
friend String operator + (char src1,    const String& src2);
```

パラメータ

src/szSrc

[in] オブジェクト、リテラル文字列及び文字

戻り値

加算した文字列

解説

String オブジェクト、リテラル文字列及び文字を加算します

```
String myStr = yourStrA + yourStrB;
String myStr = yourStr + “World”;
String myStr= “Hello” + yourStr;
String myStr= yourStr + ‘W’;
String myStr =’H’ + yourStr;
```

operator+=

String に追加します

```
String& operator += (const String& src);
String& operator += (const char* szSrc);
String& operator += (const char    src);
```

パラメータ

src/szSrc

[in] String オブジェクト、リテラル文字列及び文字

戻り値

自身の文字列

解説

String オブジェクトに String オブジェクト、リテラル文字列及び文字を追加します

```
String   myStr += yourStr;
String myStr += “Hello World”;
String myStr += ‘C’;
```

opetator==

同じか比較します

```
friend bool operator == (const String& src1, const String& src2);
friend bool operator == (const String& src1, const char* szSrc2);
friend bool operator == (const char* szSrc1, const String& src2);
friend bool operator == (const String& src1, char src2);
friend bool operator == (char src1, const String& src2);
```

パラメータ

src/szSrc

[in] String オブジェクト、リテラル文字列及び文字

戻り値

Src1 ≠ src2 :戻り値≠0

Src1 ＝ src2 :戻り値＝0

解説

String オブジェクト、リテラル文字列及び文字を同じかどうか比較します

```
Pafee::String strA=”Hello”;
Pafee::String strB=”World”;

if ( strA == strB ){/*同じ時*/}
if ( strA == “Hello” ){/*同じ時*/}
if ( “Hello” == strB ){/*同じ時*/}
if ( strA == ‘A’ ){/*同じ時*/}
if ( ‘B’ == strB ){/*同じ時*/}
```

opetator!=

異なるか比較します

```
friend bool operator != (const String& src1, const String& src2);
friend bool operator != (const String& src1, const char* szSrc2);
friend bool operator != (const char* szSrc1, const String& src2);
friend bool operator != (const String& src1, char src2);
friend bool operator != (char src1, const String& src2);
```

パラメータ

src/szSrc

[in] String オブジェクト、リテラル文字列及び文字

戻り値

Src1 ≠ src2 :戻り値＝0

Src1 ＝ src2 :戻り値≠0

解説

String オブジェクト、リテラル文字列及び文字が異なるかどうか比較します

```
Pafee::String strA=”Hello”;
Pafee::String strB=”World”;
if ( strA != strB ){/*異なる*/}
if ( strA != “Hello” ){/*  異なる*/}
if ( “Hello” != strB ){/*  異なる*/}
if ( strA != ‘A’ ){/*  異なる*/}
if ( ‘B’ != strB ){/*  異なる*/}
```

opetator<

より大きいか比較します

```
friend int operator < (const String& src1, const String& src2);
friend int operator < (const String& src1, const char* szSrc2);
friend int operator < (const char* szSrc1, const String& src2);
friend int operator < (const String& src1, char src2);
friend int operator < (char src1, const String& src2);
```

パラメータ

src/szSrc

[in] string オブジェクト、リテラル文字列及び文字

戻り値

Src1 < src2 ：戻り値=0

Src1 < src2 ：戻り値≠0

解説

String オブジェクト、リテラル文字列及び文字がより大きいか比較します

```
Pafee::String strA="Hello";
Pafee::String strB="World";
if ( strA < strB ){/*より大きい*/}
if ( strA < "Hello" ){/*  より大きい*/}
if ( "Hello" < strB ){/*  より大きい*/}
if ( strA < 'A' ){/*  より大きい*/}
if ( 'B' < strB ){/*  より大きい*/}
```

opetator>

より小さいか比較します

```
friend int operator > (const String& src1, const String& src2);
friend int operator > (const String& src1, const char* szSrc2);
friend int operator > (const char* szSrc1, const String& src2);
friend int operator > (const String& src1, char src2);
friend int operator > (char src1, const String& src2);
```

パラメータ

src/szSrc

[in] String オブジェクト、リテラル文字列及び文字

戻り値

Src1 > src2 ：戻り値=0

Src1 > src2 ：戻り値≠0

解説

String オブジェクト、リテラル文字列及び文字がより小さいか比較します

```
Pafee::String strA="Hello";
Pafee::String strB="World";
if ( strA > strB ){/*より小さい*/}
if ( strA > "Hello" ){/*より小さい*/}
if ( "Hello" > strB ){/*より小さい*/}
if ( strA > 'A' ){/*より小さい*/}
if ( 'B' > strB ){/*より小さい*/}
```

opetator<=

右辺が左辺以上か比較します

```
friend int operator <= (const String& src1, const String& src2);
friend int operator <= (const String& src1, const char* szSrc2);
friend int operator <= (const char* szSrc1, const String& src2);
friend int operator <= (const String& src1, char src2);
friend int operator <= (char src1, const String& src2);
```

パラメータ

src/szSrc

[in] String オブジェクト、リテラル文字列及び文字

戻り値

Src1 <= src2 ：戻り値=0

Src1 <= src2 ：戻り値≠0

解説

String オブジェクト、リテラル文字列及び文字を右辺が左辺以上か比較します

```
Pafee::String strA="Hello";
Pafee::String strB="World";
if ( strA <= strB ){/*以上*/}
if ( strA <= "Hello" ){/*  以上*/}
if ( "Hello" <= strB ){/*  以上*/}
if ( strA <= 'A' ){/*  以上*/}
if ( 'B' <= strB ){/*  以上*/}
```

opetator>=

右辺が左辺以下か比較します

```
friend int operator >= (const String& src1, const String& src2);
friend int operator >= (const String& src1, const char* szSrc2);
friend int operator >= (const char* szSrc1, const String& src2);
friend int operator >= (const String& src1, char src2);
friend int operator >= (char src1, const String& src2);
```

パラメータ

src/szSrc

[in] String オブジェクト、リテラル文字列及び文字

戻り値

Src1 >= src2 ：戻り値=0

Src1 >= src2 ：戻り値≠0

解説

String オブジェクト、リテラル文字列及び文字を右辺が左辺以下か比較します

```
Pafee::String strA="Hello";
Pafee::String strB="World";
if ( strA >= strB ){/*以下*/}
if ( strA >= "Hello" ){/*  以下*/}
if ( "Hello" >= strB ){/*  以下*/}
if ( strA >= 'A' ){/*  以下*/}
if ( 'B' >= strB ){/*  以下*/}
```

Compare

比較します

```
friend int Compare(const String& src1, const String& src2);
friend int Compare(const String& src1, const char* szSrc2);
friend int Compare(const char* szSrc1, const String& src2);
friend int Compare(const String& src1, char src2);
friend int Compare(char src1, const String& src2);
```

パラメータ

src/szSrc

[in] String オブジェクト、リテラル文字列及び文字

戻り値

src1>src2 ：戻り値>0

src1<src2 ：戻り値<0

src1=src2 ：戻り値=0

解説

String オブジェクト、リテラル文字列及び文字を比較します

CompareNoCase

大文字小文字を区別せず比較します

```
friend int CompareNoCase(const String& src1, const String& src2);
friend int CompareNoCase(const String& src1, const char* szSrc2);
friend int CompareNoCase(const char* szSrc1, const String& src2);
friend int CompareNoCase(const String& src1, char src2);
friend int CompareNoCase(char src1, const String& src2);
```

パラメータ

src/szSrc

[in] String オブジェクト、リテラル文字列及び文字

戻り値

src1>src2 ：戻り値>0

src1<src2 ：戻り値<0

src1=src2 ：戻り値=0

解説

String オブジェクト、リテラル文字列及び文字を大文字小文字、区別せず比較します

Find

先頭から検索します

```
int Find(const String& target, int from = 0);
int Find(const char* szTarget, int from =0);
int Find(char target, int from =0);
```

パラメータ

target/szTarger

[in] 文字列

戻り値

int

検索した文字及び文字列の位置

解説

文字列及び文字を先頭から検索します

Pafee::String str=”Hello World”

int pos = str.Find(“Wo”);

ReversFind

末尾ら検索します

```
int ReverseFind(const String& target);
int ReverseFind(const char* szTarget);
int ReverseFind(char targetChar);
```

パラメータ

target/szTarger/ targetChar

[in]検索する文字列/文字

戻り値

int

検索した文字及び文字列の位置

解説

文字列及び文字を末尾から検索します

Pafee::String str=”Hello World”

int pos = str. ReverseFind (“Wo”);

Format

指定の書式に変換します

```
void Format(const char* format, ...);
```

パラメータ

format

[in] 書式

…

[in] パラメータ

戻り値

なし

解説

書式は pafee_printf と同じです

```
Pafee::String str;
char[] vl =”Height”;
Str.Format(“%s:%d”,vl,4); //”Height:4”
```

参照

Pafee::pafee_printf

Insert

挿入します

```
void Insert(int indexAt, const String& src);
void Insert(int indexAt, const char* szSrc);
void Insert(int indexAt, char src);
```

パラメータ

indexAt

　挿入位置

src/sxSrc

　挿入する文字及び文字列

戻り値

　なし

解説

String オブジェクト、リテラル文字列、文字を挿入します

Delete

削除します

```
void Delete(int index, int count=1);
void DeleteLeft(int count);
void DeleteRight(int count);
```

パラメータ

index

[in] 削除位置

count

[in] 削除数

戻り値

なし

解説

文字を削除します

Substring

部分文字列を取得します

```
String Substring(int from, int count);
String Substring(int from)
String Left(int count)
String Right(int count)
```

パラメータ

index

[in] 削除位置

count

[in] 削除数

戻り値

部分文字列

解説

部分文字列のために新しいインスタンスを作成します

自身のオブジェクトには変更はありません

Trim

指定文字を取り除きます

```
String& Trim(const char* szTarget = " ¥t")
String& Trim(char target)
String& TrimLeft(const char* szTarget = " ¥t");
String& TrimLeft(char target)
String& TrimRight(const char* szTarget = " ¥t");
String& TrimRight(char target)
```

パラメータ

szTarget/target

[in] 取り除く文字

戻り値

自身の文字列

解説

指定文字を取り除きます

ToLower/ToUpper

小文字/大文字に変換します

```
String& ToLower(void);
String& ToUpper(void);
```

パラメータ

なし

戻り値

自身の文字列

解説

小文字/大文字に変換します

GetBuffer

文字列のバッファを取得します

```
char* GetBuffer(void);
```

パラメータ

なし

戻り値

文字列のバッファアドレス

解説

連続した領域の先頭アドレスを取得します

GetLength

文字列の長さを取得します

```
int GetLength(void);
```

パラメータ

なし

戻り値

int

文字列の長さ

解説

文字列の長さを取得します

IsEmpty

文字列が空かチェックします

```
bool IsEmpty(void);
```

パラメータ

なし

戻り値

bool

文字列が空：true　文字列が空でない：false

解説

文字列が空かチェックします

7.2 Tokenizer

Pafee::Tokenizer は区切り文字で区切られた文字列を抽出するクラスです

必要条件

ヘッダー：PafeeTokenizer.h

7.2.1 クラスメンバー覧

コンストラクタ

Tokenizer(Allocator* pAllocator=NULL)	デフォルトコンストラクタ
Tokenizer(const char* szTarget, char token=',', char quote='¥"', char* szEndMark="¥r¥n", Allocator* pAllocator=NULL);	コンストラクタ

操作

Split	初期化します
GetCount	分割された文字列の数を取得します
IsEmpty	空かチェックします
Get	指定インディクスにある分割された文字列を取得します
GetInt	指定インディクスにある分割された文字列を数値に変換して取得します
GetLength	指定インディクスにある分割された文字列の長さを取得します

7.2.2 クラスメンバ詳細

Tokenizer

Tokenizer クラスを構築します

```
Tokenizer::Tokenizer(
        Allocator* pAllocator ///< [in] Memory Allocator
);
Tokenizer::Tokenizer(
        const char* szTarget, ///< [in] Character string to handle
        char token,           ///< [in] Delimiter
        char quote,           ///< [in] Quote
        char* szEndMark ,     ///< [in] The last sign
        Allocator* pAllocator ///< [in] Memory Allocator
);
```

パラメータ

pAllocator

[in] メモリを確保するアロケータクラス

szTarget

[in] 対象文字列

token

[in] 区切り文字

quote

[in] この文字で囲まれた文字列は対象としません

szEndMark

終了文字列

戻り値

なし

解説

Tokenizer クラスを構築します

Split

初期化します

```
int Tokenizer::Split(
        const char* szTarget, ///< [in] Character string to handle
        char token,           ///< [in] Delimiter
        char quote,           ///< [in] Quote
        char* szEndMark        ///< [in] The last sign
)
```

パラメータ

szTarget

[in] 対象文字列

token

[in] 区切り文字

quote

[in] この文字で囲まれた文字列は対象としません

szEndMark

[in] 終了文字列

戻り値

int

処理した長さを返します

解説

文字列、区切り文字を指定しオブジェクトを初期化します

```
Tokenize Tknz;
char szTg[]= "ip=192.168.1.2¥r¥n"user=¥"Bob¥"¥r¥n";
Tknz.Split ( szTg,'=', '¥"' );
```

GetCount

分割された文字列の数を取得します

```
int Tokenizer::GetCount(void);
```

パラメータ

なし

戻り値

int

分割された文字列の数

解説

分割された文字列の数を取得します

IsEmpty

Tokenizer 空かチェックします

```
bool Tokenizer::IsEmpty(void);
```

パラメータ

なし

戻り値

bool

tokenizer が空：true tokenizer が空でない：false

Get

指定インディクスにある分割された文字列を取得します

```
bool Tokenizer::Get(
        int index,          ///< [in] index
        char* szBuffer, ///< [in] Buffer
        int bufferSize    ///< [in] bufferSize
);
char* Tokenizer::Get(
        int index          ///< [in] index
);
```

パラメータ

index

[in] 何番目の分割した文字列か

szBuffer

[in] 取得する文字列を格納するバッファ

bufferSize

[in] 取得する文字列のバッファサイズ

戻り値

boot

取得成功:true 取得失敗:false

解説

取得する文字列が bufferSize より大きい場合 bufferSize-1 の文字列を返し終端に null 文字を挿入します

GetInt

指定インディクスにある分割された文字列を数値に変換して取得します

```
int Tokenizer::GetInt(
        int index,    ///< [in] index
        int base    ///< [in] A cardinal number
);
```

パラメータ

index

[in] 何番目の分割した文字列か

Base

[in] 基数

戻り値

int

取得数値

解説

奇数は pafee_strtol の指定方法と同じです

参照

Pafee::pafee_strtol

GetLength

指定インディクスにある分割された文字列の長さを取得します

```
int Tokenizer::GetLength(
        int index ///< [in] index
);
```

パラメータ

index

[in] 何番目の分割した文字列か

戻り値

int

文字列の長さ

解説

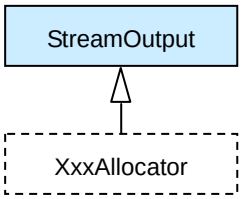
指定インディクスにある分割された文字列の長さを取得します

8. デバイス抽象化

デバイスを抽象化するクラスを提供します。 現状ではストリームデータの出力クラスのみの提供となっています。

8.1 StreamOutput

ストリーム出力デバイスを抽象化するクラスです。
実装をもたず、ストリーム出力の統一的なインタフェースを提供します。



8.1.1 クラスメンバー覧

コンストラクタ

StreamOutput	デフォルトコンストラクタ

操作
なし

オーバライド可能な関数

Write	文字列、バイナリデータを出力します

データメンバ
なし

8.1.2 クラスメンバ詳細

StreamOutput

StreamOutput オブジェクトを構築します。

```
StreamOutput:: StreamOutput()
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します。

Write

文字列、バイナリデータを出力します。

```
virtual void Write(
                        const char *fmt,        ///< [in] format string
                        ...                     ///< [in] arguments
                        ) = 0;

virtual void Write(
                        int length,             ///< [in] length of data
                        const char* pOutput     ///< [in] data to output
                        ) = 0;
```

パラメータ

fmt

[in] フォーマット文字列。printf の書式フォーマットに準拠します。

…

[in] 可変引数

length

[in] データ長

pOutput

[in] 出力データ

戻り値

なし

例外

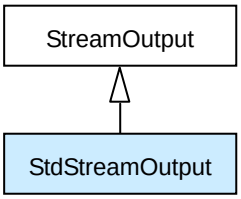
なし

解説

文字列、バイナリデータを出力します。

8.2 StdStreamOutput

標準出力を使って文字列、バイナリデータを出力するクラスです。



8.2.1 クラスメンバー覧

コンストラクタ

StdStreamOutput	デフォルトコンストラクタ

操作

なし

オーバライド可能な関数

Write	標準出力を使って文字列、バイナリデータを出力します

データメンバ

なし

8.2.2 クラスメンバ詳細

StdStreamOutput

StdStreamOutput オブジェクトを構築します。

```
StdStreamOutput:: StreamOutput()
```

パラメータ

なし

戻り値

なし

例外

なし

解説

オブジェクトを構築します。

Write

標準出力を使って文字列、バイナリデータを出力します。

```
virtual void Write(
                        const char *fmt,        ///< [in] format string
                        ...                     ///< [in] arguments
                        ) ;

virtual void Write(
                        int length,             ///< [in] length of data
                        const char* pOutput     ///< [in] data to output
                        ) ;
```

パラメータ

fmt

[in] フォーマット文字列。printf の書式フォーマットに準拠します。

…

[in] 可変引数

length

[in] データ長

pOutput

[in] 出力データ

戻り値

なし

例外

なし

解説

文字列、バイナリデータを出力します。

例） stream.Write(“data=%d¥n”, data);

9. ユーティリティ

9.1 CRC

Pafee::Crc は巡回冗長符号を生成します

必要条件

ヘッダー：PafeeCrc.h

9.1.1 クラスメンバー覧

コンストラクタ

Crc	コンストラクタ

操作

Encode16	16 ビットの Crc コードを生成します
Encode32	32 ビットの Crc コードを生成します

9.1.2 クラスメンバ詳細

Crc

Crc クラスを構築します

```
Crc();
```

パラメータ

なし

戻り値

なし

解説

Crc クラスを構築します

Encode16

16 ビットの Crc コードを生成します

```
static unsigned short Crc::Encode16(
        void* pData, ///< [in] Point of data to be computed
        int length ///< [in] Length of data in byte
);
```

パラメータ

pData

[in] 対象データのバッファアドレス

length

[in] 対象データの長さ（バイト）

戻り値

static unsigned short

生成した Crc コード

解説

16 ビットの Crc コードを生成します

Encode32

32 ビットの Crc コードを生成します

```
static unsigned long Crc::Encode32(
        void* pData,    ///< [in] Point of data to be computed
        int length    ///< [in] Length of data in byte
);
```

パラメータ

pData

[in] 対象データのバッファアドレス

length

[in] 対象データの長さ（バイト）

戻り値

unsigned long

生成した Crc コード

解説

32 ビットの Crc コードを生成します

9.2 File

Pafee::File はファイルの読み書きの操作を行います

必要条件

ヘッダー：PafeeFile.h

9.2.1 クラスメンバー覧

コンストラクタ

File	コンストラクタ

操作

Open	ファイルをオープンします
Close	ファイルをクローズします
GetLength	ファイルのサイズを取得します
Read	ファイルを読み出します
Write	ファイルに書き込みます
Flush	バッファをフラッシュします
Seek	ファイルを Seek します

9.2.2 クラスメンバ詳細

File

File クラスを構築します

```
File();
```

パラメータ

なし

戻り値

なし

解説

File クラスを構築します

Open

ファイルをオープンします

```
bool File::Open(
        const char* filePath, ///< [in]file path
        const OPENMODE mode ///< [in]File::OPENMODE
);
```

パラメータ

filePath

[in] オープンするファイル名

mode

[in] オープンモード

0	MODE_READ	r に相当
1	MODE_WRITE	w に相当
2	MODE_APPEND	a に相当
3	MODE_READPLUS	r+に相当
4	MODE_WRITEPLUS	w+に相当
5	MODE_APPENDPLUS	a+に相当
6	MODE_READ_BINARY	rb に相当
7	MODE_WRITE_BINARY	wb に相当
8	MODE_APPEND_BINARY	ab に相当
9	MODE_READPLUS_BINARY	r+b に相当
10	MODE_WRITEPLUS_BINARY	w+b に相当
11	MODE_APPENDPLUS_BINARY	a+b に相当

戻り値

bool

成功:true 失敗:false

解説

ファイルをオープンします

参照

標準関数:fopen

Close

ファイルをクローズします

```
void File::Close(void);
```

パラメータ

なし

戻り値

なし

解説

ファイルをクローズします

参照

標準関数:fclose

GetLenth

ファイルの大きさを取得します

```
long File::GetLength(void);
```

パラメータ

なし

戻り値

long

ファイルの長さ

解説

ファイルの大きさを取得します

Read

ファイルから読み出します

```
int File::Read(
        void* pBuffer, /// < [out] The data which were read
        int length /// < [in] Length to read
);
```

パラメータ

pBuffer

[in] 読み出したデータを入れるバッファ

length

[in] 読み出す長さ

戻り値

int

読み出した大きさ

解説

ファイルから読み出します

参照

標準関数:fread

Write

ファイルに書き込みます

```
int File::Write(
        const void* pData, /// < [in] Data to write
        int length /// < [in] Length to write
);
```

パラメータ

pData

[in] ファイルに書き込むデータのアドレス

length

[in] ファイルに書き込む大きさ

戻り値

int

ファイルに書き込んだ大きさ

解説

書き込んだ大きさを取得します

参照

標準関数:fwrite

Flush

バッファをフラッシュします

```
void File::Flush(void);
```

パラメータ

なし

戻り値

なし

解説

バッファをフラッシュします

参照

標準関数:fflush

Seek

ファイル位置を移動します

```
bool File::Seek(
        int offset, ///< [in]    the offset from origin
        int origin    ///< [in] File::ORIGIN
);
```

パラメータ

bool

[in] 成功:true 失敗:false

offset

[in] origin からの距離

origin

SEEK_SET	ファイルの先頭
SEEK_CUR	ファイルの現在位置
SEEK_END	ファイルの終端

戻り値

なし

解説

ファイル位置を移動します

参照

標準関数:fseek

9.3 Log

Pafee::Log は汎用的なログ出力機能を提供します

必要条件

ヘッダー：PafeeLog.h

9.3.1 クラスメンバー覧

コンストラクタ

Log	コンストラクタ

操作

Write	ログを書き込みます
WriteX	ログを書き込みます
Dump	バイナリダンプを出力します
SetLevelString	ログレベル文字列を設定します
SetCategoryString	ログカテゴリ文字列を設定します
SetLogLevel	ログレベルを設定します　指定レベル以下のログを出力します
SetLogOutput	ログ出力先を設定します

9.3.2 クラスメンバ詳細

Log

Log クラスを構築します。

```
Log::Log(
        StreamOutput* pOutput ///<[in] File stream
);
```

パラメータ

なし

戻り値

なし

解説

Log クラスを構築します

参照

Pafee::StreamOutput

Write

ログを書き込みます

```
void Log::Write(
        int level,              ///<[in] Log level
        int category,           ///<[in] Log category
        const char* stringLog ///<[in] Write string
);
void Log::Write(int level, int category, const char* stringLog, char* p1, char* p2=NULL, char* p3=NULL, char*
p4=NULL);
void Log::Write(int level, int category, const char* stringLog, int n1);
void Log::Write(int level, int category, const char* stringLog, int n1, int n2);
void Log::Write(int level, int category, const char* stringLog, int n1, int n2, int n3);
void Log::Write(int level, int category, const char* stringLog, int n1, int n2, int n3, int n4);
```

パラメータ

level

[in] ログレベルを指定します

category

[in] カテゴリを指定します

stringLog

[in] ログ文字列を指定します

p1/p2/p3/p4/n1/n2/n3/n1

[in] パラメータを指定します

戻り値

なし

解説

ログを書き込みます

出力フォーマットイメージは以下の通りです

[レベル] [カテゴリ] ログ文字列 (P1,P2,P3,P4)

SetLogLevel で指定したレベル以下のログが出力されます

WriteX

ログを書き込みます(16 進表示)

```
void Log::WriteX(
        int level,                  ///<[in] Log level
        int category,               ///<[in] Log category
        const char* stringLog, ///<[in] Write string
        int n1, ///<[in] Numerical value argument(Hexadecimal expression)
        int n2   ///<[in] Numerical value argument(Hexadecimal expression)
);
void Log::WriteX(int level, int category, const char* stringLog, int n1, int n2, int n3);
void Log::WriteX(int level, int category, const char* stringLog, int n1, int n2, int n3, int n4);
```

パラメータ

level

[in] ログレベルを指定します

category

[in] カテゴリを指定します

stringLog

[in] ログ文字列を指定します

n1/n2/n3/n1

[in] パラメータ(数値)を指定します

戻り値

なし

解説

ログを書き込みます

出力フォーマットイメージは以下の通りです

[レベル] [カテゴリ] ログ文字列 (n1,n2,n3,n4)

SetLogLevel で指定したレベル以下のログが出力されます

Dump

バイナリダンプを出力します

```
void Log::Dump(
        int level,                  ///<[in] Log level
        int category,               ///<[in] Log category
        const char* stringLog, ///<[in] Write string
        void* pData,                ///<[in] Log Data
        int length                  ///<[in] Data length
);
```

パラメータ

level

[in] ログレベルを指定します

category

[in] カテゴリを指定します

stringLog

[in] ログ文字列を指定します

pData

[in] Dump 出力するデータのアドレス

length

[in] Dump 出力するデータの長さ

戻り値

なし

解説

ログを書き込みます

出力フォーマットイメージは以下の通りです

[レベル] [カテゴリ] ログ文字列 ダンプ情報

SetLogLevel で指定したレベル以下のログが出力されます

SetLevelString

出力するログレベル文字列を設定します

```
void Log::SetLevelString(
        int level,          ///<[in] Log level
        char* levelString ///<[in] Log level string
);
```

パラメータ

level

[in] 設定するログレベル

category

[in] 出力するログレベルの文字列

戻り値

なし

解説

本関数で設定した文字列がログに出力されます

SetCategoryString

出力するカテゴリ文字列を設定します

```
void Log::SetCategoryString(
        int category,           ///<[in] Log category
        char* categoryString ///<[in] Category    string
);
```

パラメータ

category

[in] 設定するログカテゴリ

categoryString

[in] 出力するカテゴリの文字列

戻り値

なし

解説

本関数で設定した文字列がログに出力されます

SetLogLevel

出力するログレベルを設定します

```
void Log::SetLogLevel(
        int level ///<[in] Log level
);
```

パラメータ

level

[in] 設定するログレベル

戻り値

なし

解説

本関数で設定したログレベル以下のレベルが出力されます

SetLogOutput

出力先を設定します

```
void Log::SetLogOutput(
        StreamOutput* pOutput ///<[in] File stream
);
```

パラメータ

pOutput

[in] 出力先のクラス

戻り値

なし

解説

本関数で設定した出力さきに出力されます

参照

Pafee::StreamOutput

10. OS 抽象化

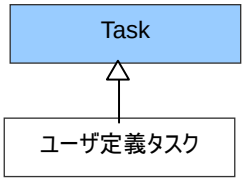
Windows,Linux,ITRON の API を抽象化できるクラスを提供します。 現状では PAFEE で用いられる API に機能を限定しています。 タスク、ミューテックス、セマフォ、シグナルクラスが提供されます。
ITRON(TOPPERS)の場合には静的APIしかサポートされていないので、Windows,Linux とは使い方が異なります。
既存のリソースにアタッチして使う形になります。

10.1 Task

Pafee::PafeeTask は汎用的なスレッド機能を提供します。ひとつのインスタンスでひとつのスレッドを生成します。複数のスレッドを使用する場合は、使用する個数分のインスタンスを生成する必要があります。

OS が提供する機能を使用して実現しているため、使用する OS の define を makefile に定義する必要があります。

タスククラスを使用するには、本タスククラスの派生クラス（ユーザ定義クラス）を作成し、タスクで実行する処理を記述する必要があります。



タスクを起動すると、「TaskProcedure」メソッドが実行されます。Task クラスの「TaskProcedure」メソッドは、仮想関数です。派生した「ユーザ定義タスク」で「TaskProcedure」メソッドをオーバーロードし、そのタスクで実行したい処理を記述してください。

「TaskProcedure」メソッドの引数は「void *」となっていますので、タスク実行時に引数を持たせる場合は、別途引数として渡す情報を含む構造体を宣言し、その構造体のアドレスを引数として実行してください。作成した派生クラスを実行するには、派生クラスのインスタンスを作成します。

以下に派生クラスの作成例を示します。

```
#include <stdio.h>

/*  引数情報構造体を定義します  */
typedef struct tagARGPARAM
{
      int value;
} ARGPARAM;

/*  派生クラスを作成します  */
class MyTaskClass : public PafeeTask
{
      void *TaskProcedure (void* param)
      {
            ARGPARAM* argParam;
            argParam = (ARGPARAM *)param;
            printf(“%d”, argParam ->value);
            return(argParam);
      }
}
```

TOPPERS/JSP 環境下でタスクを使用するためには、静的 API により、静的にタスクオブジェクトを生成する必要があります。そのため PAFEE では、静的 API で生成した各タスクからシステム起動時にタスクエントリ関数（C 関数として実装）をコールし、その後「Attach」を行うと、実行中のタスクエントリ関数から「TaskProcedure」メソッドを呼び出すような仕組みを実装しています。

以下に例として、ユーザ定義タスク TASK_TEST1 を追加する方法を説明します。

①ユーザが作成するソースファイルとコンフィギュレーションファイルの双方でインクルードする共通ヘッダファイルを作成し、
タスク実行時に引数として渡す情報を含む構造体の定義、値の宣言を行います。

（例）TaskTest1.h

```
#include "kernel_id.h"
#include "PafeeTaskConfig.h"    /*  構造体 TASK_ENTRY_PARAM  の定義が含まれています  */

/*  タスク起動時引数の構造体を定義します  */
typedef struct tagARGPARAM
{
      int value;
} ARGPARAM;

/*  タスク起動時引数の構造体を宣言します  */
ARGPARAM testParam1 = {1};

/*  タスク ID と引数構造体のアドレスから成る構造体を宣言します  */
/*  この構造体のアドレスを、タスクの拡張情報として静的 API に記述することで  */
/* PAFEE のタスク機能を使用することができます  */
TASK_ENTRY_PARAM EntryParam1 = {TASK_TEST1, &testParam1};
```

②コンフィギュレーションファイルに下記の記述を追加し、コンフィギュレータを実行します。（青字の内容は任意です）

※　タスク属性には、必ず「TA_ACT」を指定してください。指定しない場合、「TaskProcedure」が実行されなくなります。

※　タスク起動番地には、「TaskEntry」を指定してください。

```
INCLUDE("¥"PafeeTaskConfig.h¥"");
INCLUDE("¥"TaskTest1.h¥"");
CRE_TSK(TASK_TEST1, { TA_HLNG|TA_ACT, &EntryParam1,
            TaskEntry, MID_PRIORITY, STACK_SIZE, NULL });
```

必要条件

ヘッダー：　PafeeTask.h
　　PafeeTaskConfig.h（ITRON のみ）

define 値：

Windows　：PAFEE_OS_WINDOWS
Linux　　：PAFEE_OS_LINUX
ITRON　　：PAFEE_OS_ITRON

10.1.1 Task クラスメンバ

コンストラクタ

PafeeTask	コンストラクタ。

デストラクタ

~PafeeTask	デストラクタ。

操作

Create	タスクを生成し指定によっては起動します（Windows、Linux のみ）。
Attach	静的に生成したタスクをアタッチします(ITRON 専用)。
Start	タスクを起動します。
Stop	タスクを一時停止します。
Delete	タスクを削除します。
TaskProcedure	タスク関数です。派生クラスで実装します。
GetLastError	最後のエラーコードを取得します。

10.1.2 Task クラスメンバ詳細

PafeeTask

タスククラスを構築します。

```
PafeeTask::PafeeTask(
        void
);
```

パラメータ

なし

戻り値

なし

解説

タスククラスのインスタンスを作成します。

タスクはインスタンス毎に管理されます。複数のタスクを作成する場合は、作成する数と同数のインスタンスを作成します。

~PafeeTask

タスククラスを破棄します。

```
virtual PafeeTask::~PafeeTask(
        void
);
```

パラメータ

なし

戻り値

なし

解説

クラスを破棄します。

Create

タスクを生成し、指定によっては即座に起動します。
Windows,Linux のみ有効で、IRTON では使用できません。

```
bool PafeeTask::Create(
        TASK_ID* pTaskID,
        TASK_PRIORITY priority,
        TASK_SCHEDULING schedule = 0, // 0 : SCHED_OTHER(Linux)
        int stackSize = 0,
        void* pParam = NULL,
        bool isStartImmediately = true
);
```

パラメータ

pTaskID	生成するタスクの ID を格納するバッファのポインタ
priority	タスク優先順位。範囲は OS に依存します。 Windows： タイムスライスの比率を決定する要素になる。 Linux： スケジュリングポリシに FIFO,RR を選択した時に有効。
schedule	Windows： このパラメータは無視されます。常にラウンドロビンのスケジュリング。 Linux： FIFO,RR,OTHER が選択可能。
stackSize	タスクが使用できるスタックのサイズを指定します。
pParam	タスクの実行時引数を指定します。
isStartImmediately	Windows： 生成と同時に起動するか、しないかを選択できます。 Linux： このパラメータは無視されます。常に生成と同時に起動します。

戻り値

bool

true	:タスクの生成成功
false	:タスクの生成失敗

解説

タスクを生成し指定によっては起動します。

Windows,Linux のみサポートし、TOPPERS では使用でません。

Attach

静的に生成したタスクをアタッチします。
IRTON のみで使用可能で、Windows、Linux では使用できません。

```
bool PafeeTask::Attach(
        TASK_ID id
);
```

パラメータ

Id	静的に生成したタスクの ID

戻り値

bool

true	:タスクの関連付け成功
false	:タスクの関連付け失敗

解説

静的に生成したタスクをアタッチします。
ITRON のみで使用可能で、Windows、Linux では使用できません。

Start

タスクを起動します。
Windows 及び ITRON のみで使用可能で、Linux では使用できません。

```
bool PafeeTask::Start(
        void
);
```

パラメータ

なし

戻り値

bool

true　　:タスクの起動成功
false　　:タスクの起動失敗

解説

タスクを起動します。
Windows 及び ITRON のみで使用可能で、Linux では使用できません。Linux ではタスクの作成と同時にタスクが起動し、タスクが終了するまでタスクを停止することはできません。

Stop

タスクを一時停止します。
Windows 及び ITRON のみで使用可能で、Linux では使用できません。

```
bool PafeeTask::Stop(
        void
);
```

パラメータ

なし

戻り値

bool

true　　:タスクの一時停止成功
false　　:タスクの一時停止失敗

解説

タスクを一時停止します。
Windows 及び ITRON のみで使用可能で、Linux では使用できません。Linux ではタスクの作成と同時にタスクが起動し、タスクが終了するまでタスクを停止することはできません。

Delete

タスクを削除します。
Windows 及び Linux のみで使用可能で、ITRON では使用できません。

```
bool PafeeTask::Delete(
        void
);
```

パラメータ

なし

戻り値

bool

true　　:タスクの削除成功
false　　:タスクの削除失敗

解説

タスクを削除します。
Windows 及び Linux のみで使用可能で、ITRON では使用できません。

TaskProcedure

タスク実行関数。
派生クラスで実装します。

```
virtual void* PafeeTask::Delete(
        void *pParam
);
```

パラメータ

pParam	タスク実行時引数

戻り値

void *
戻り値

解説

タスク実行関数です。仮想関数で、派生クラスで実装します。本関数で実装したコードがタスクとして起動します。タスク起動時に引数が必要な場合は、別途必要なデータ型（構造体等）を定義し、パラメータ「pParam」を定義したデータ型にキャストして使用してください。

10.1.3 公開定数

TASK_ERROR

タスククラスで発生するエラーコードを定義します。

```
typedef enum tagTASK_ERROR{
        ERROR_NONE, /**< Error none */
        ERROR_NOHANDLE, /**< Function is called without handle */
        ERROR_HAS_HANDLE, /**< Allready has handle */
        ERROR_CANNOT_CREATE, /**< Can not create */
        ERROR_CANNOT_DELETE, /**< Can not delete */
        ERROR_CANNOT_START, /**< Can not start */
        ERROR_STILL_SUSPENDED, /**< Stopped more times than Started */
        ERROR_CANNOT_STOP, /**< Can not stop */
        ERROR_MAX /**< MAX */
} TASK_ERROR;
```

メンバ定数

PafeeTask::TASK_ERROR：

ERROR_NON	エラーなし
ERROR_NOHANDLE	タスクが作成されていません。 Create、または Attach を行っていない状態でスタート・ストップ等 を行った場合に発生します。
ERROR_HAS_HANDLE	すでにタスクが作成されています。 Create、または Attach を行った状態のタスクインスタンスに対し、再度 Create、または Attach を行った場合に発生します。
ERROR_CANNOT_CREATE	タスク作成失敗。 リソース不足やその他の原因でタスクの作成に失敗した場合に発生します。
ERROR_CANNOT_DELETE	タスク削除失敗。
ERROR_CANNOT_START	タスク起動失敗。
ERROR_STILL_SUSPENDED	タスク再開失敗。（Windows のみ） Start を行った際、事前にそれ以上の回数の Stop を行っているため タスク処理が再開されない場合に発生します。
ERROR_CANNOT_STOP	タスク停止失敗。
ERROR_MAX	エラー番号の個数

解説

エラー変数はインスタンス毎に管理されます。たとえ、他のシグナルでエラーコードが破棄された場合も、その他のインスタンスでは、最後に発生したエラーを呼び出すことができます。

10.2 Mutex

Pafee::PafeeMutex は汎用的なミューテックス機能を提供します。ひとつのインスタンスでひとつのミューテックスを使用します。複数のミューテックスを使用する場合は、使用する個数分のインスタンスを生成する必要があります。

OS が提供する機能を使用して実現しているため、使用する OS の define を makefile に定義する必要があります。

ITRON 環境下では、ミューテックスは実装されていないため、シグナルを利用してミューテックス機能を実現しています。

必要条件

ヘッダー：PafeeMutex.h

define 値：

Windows ：PAFEE_OS_WINDOWS

Linux ：PAFEE_OS_LINUX

ITRON ：PAFEE_OS_ITRON

10.2.1 Mutex クラスメンバ

コンストラクタ

PafeeMutex	コンストラクタ。

デストラクタ

~PafeeMutex	デストラクタ。

操作

Create	Mutex を生成します(Windows、Linux 専用)。
Attach	静的に生成された mutex への関連付けを行います(ITRON 専用)。
Delete	Mutex を破棄します。
Lock	Mutex をロックします。ロックできるまでスレッドはブロックされます。
AsyncLock	Mutex をロックします。ロックできなければすぐに処理を返します(非同期処理)。
TimedLock	Mutex をロックします。指定時間待ってもロックできなければ処理を返します タイムアウト時間の単位はプラットホームに依存します。
Unlock	Mutex をアンロックします。
GetLastError	最後のエラーコードを取得します。

10.2.2 クラスメンバ詳細

PafeeMutex

ミューテックスクラスを構築します。

```
PafeeMutex::PafeeMutex(
        void
);
```

パラメータ

なし

戻り値

なし

解説

ミューテックスのインスタンスを作成します。

ミューテックスはインスタンス毎に管理されます。複数のミューテックスを作成する場合は、作成する数と同数のインスタンスを作成します。

~PafeeMutex

ミューテックスクラスを破棄します。

```
PafeeMutex::~PafeeMutex(
        void
);
```

パラメータ

なし

戻り値

なし

解説

ミューテックスクラスを破棄します。

破棄するとき、ミューテックスの破棄を行います。他のスレッドでミューテックスを使用している状態で破棄を行った場合、ミューテックスの開放を正常に行うことができなくなることがあります。手動で破棄する場合は確実に全てのスレッドでミューテックスが開放されていることを確認し、破棄を行ってください。

Create

ミューテックスを生成します。Windows 及び Linux のみサポートし、ITORON では使用することができません。

```
bool PafeeMutex::Create(
        void
);
```

パラメータ

なし

戻り値

bool：

true	生成成功
false	生成失敗

解説

ミューテックスを生成します。

すでにミューテックスが生成されている場合は、ミューテックスの生成は失敗します。

Attach

静的に生成したミューテックスの関連付けを行います。ITRON のみサポートし、Windows 及び Linux では使用することができません。

```
bool PafeeMutex::Attach(
        ID          id
);
```

パラメータ

id

[in] 静的に生成したミューテックスの ID

戻り値

bool:

true	関連付け成功
false	関連付け失敗

解説

静的ミューテックスの関連付けを行います。

すでにミューテックスが関連付けされている場合は、ミューテックスの関連付けは失敗します。

Delete

ミューテックスを破棄します。

```
bool PafeeMutex::Delete(
        void
);
```

パラメータ

なし

戻り値

bool:

true	破棄成功
false	破棄失敗

解説

ミューテックスの破棄を行います。

Windows、および Linux 環境下では、他のスレッドでミューテックを使用している場合、破棄は失敗します。

ITRON 環境下においては、破棄はいかなる条件でも成功しますが、実際のミューテックスの破棄は行われず、関連付けの解除のみが行われます。

Lock

ミューテックスをロックします。ロックできるまで処理はブロックされます(同期処理)。

```
bool PafeeMutex::Lock(
        void
);
```

パラメータ

なし

戻り値

bool:

true	ロック成功
false	ロック失敗

解説

ミューテックスのロックを行います。

ロックできるまで処理はブロックされますが、エラーが発生した場合はロックに失敗し、処理を返します。

AsyncLock

ミューテックスをロックします。ロックできなければすぐに処理を返します(非同期処理)。

```
bool PafeeMutex::AsyncLock(
        void
);
```

パラメータ

なし

戻り値

bool:

true	ロック成功
false	ロック失敗

解説

ミューテックスのロックを行います。

ミューテックスのロックを試みます。ロックできる場合はミューテックスのロックを行い、処理を返します。ロックできなければ、すぐに処理を返します。エラーが発生した場合もロックに失敗し、処理を返します。

TimedLock

ミューテックスをロックします。指定時間待ってもロックできなければ処理を返します。
タイムアウト時間の単位はプラットフォームに依存します。

```
bool PafeeMutex::TimedLock(
        unsigned long       time        ///< [in] timeout time
);
```

パラメータ

time

ミューテックスのロックを試みるタイムアウト間隔。入力単位はプラットフォームに依存します。

戻り値

bool:

true	ロック成功
false	ロック失敗

解説

ミューテックスのロックを試みます。ロックできる場合はミューテックスのロックを行い、処理を返します。指定時間待ってもロックできなければ、処理を返します。エラーが発生した場合もロックに失敗し、処理を返します。

UnLock

ミューテックスをアンロックします。

```
bool PafeeMutex::UnLock(
        void
);
```

パラメータ

なし

戻り値

bool：

true	アンロック成功
false	アンロック失敗

解説

ロックされているミューテックスをアンロックします。アンロックに失敗した場合はエラーを返します。ロックされていないミューテックスのアンロックを行った場合もエラーを返します。

GetLastError

最後のエラーコードを取得します。

```
PafeeMutex::MUTEX_ERROR PafeeMutex::GetLastError(
        void
);
```

パラメータ

なし

戻り値

PafeeMutex::MUTEX_ERROR：

ERROR_NON	エラーなし
ERROR_NOMUTEX	ミューテックスが作成されていません
ERROR_HAS_MUTEX	すでにミューテックスが作成されています
ERROR_CANNOT_CREATE	ミューテックス作成失敗
ERROR_CANNOT_DELETE	ミューテックス削除失敗
ERROR_CANNOT_LOCK	ロック失敗（タイムアウト以外の要因）
ERROR_TIME_OUT	ロック失敗（タイムアウト）
ERROR_CANNOT_UNLOCK	アンロック失敗
ERROR_OHTER	その他エラー
ERROR_MAX	エラー番号の個数

解説

最後に発生したエラーコードを返します。エラーコードは各インスタンスごとに管理されます。エラーコードの詳細は、10.2.3 公開定数を参照してください。

10.2.3 公開定数

MUTEX_ERROR

ミューテックスクラスで発生するエラーコードを定義します。

```
typedef enum PafeeMutex::MUTEX_ERROR{
        ERROR_NONE, /**< Error none */
        ERROR_NOMUTEX, /**< handle, id is not exist */
        ERROR_HAS_MUTEX, /**< Allready exist handle */
        ERROR_CANNOT_CREATE, /**< Mutex can not create */
        ERROR_CANNOT_DELETE, /**< Mutex can not delete */
        ERROR_CANNOT_LOCK, /**< Mutex can not get */
        ERROR_TIME_OUT, /**< Mutex can not get for time out */
        ERROR_CANNOT_UNLOCK, /**< Error none */
        ERROR_OHTER, /**< Unknown error */
        ERROR_MAX /**< Error none */
} MUTEX_ERROR;
```

メンバ定数

PafeeMutex::MUTEX_ERROR:

ERROR_NON	エラーなし
ERROR_NOMUTEX	ミューテックスが作成されていません。 Create、または Attach を行っていない状態でロック・アンロック・削除を行った場合に発生します。
ERROR_HAS_MUTEX	すでにミューテックスが作成されています。 Create、または Attach を行った状態のミューテックスに対し、再度 Create、または Attach を行った場合に発生します。
ERROR_CANNOT_CREATE	ミューテックス作成失敗。 リソース不足やその他の原因でミューテックスの作成に失敗した場合に発生します。
ERROR_CANNOT_DELETE	ミューテックス削除失敗。 他のスレッドで使用中のインスタンスのミューテックスを削除しようとした場合に発生します。
ERROR_CANNOT_LOCK	ロック失敗（タイムアウト以外の要因）。 AsyncLock 使用時、他のスレッドがすでにロックしている場合等に発生します。
ERROR_TIME_OUT	ロック失敗（タイムアウト）。 一定時間ロックを試みたが、ミューテックスのロックを行えなかった場合に発生します。
ERROR_CANNOT_UNLOCK	アンロック失敗。
ERROR_OHTER	その他エラー。
ERROR_MAX	エラー番号の個数

解説

エラー変数はインスタンス毎に管理されます。たとえ、他のミューテックスでエラーコードが破棄された場合も、その他のインスタンスでは、最後に発生したエラーを呼び出すことができます。

10.3 Semaphore

Pafee::PafeeSemaphore は汎用的なセマフォ機能を提供します。ひとつのインスタンスでひとつのセマフォを使用します。複数のセマフォを使用する場合は、使用する個数分のインスタンスを生成する必要があります。

OS が提供する機能を使用して実現しているため、使用する OS の define を makefile に定義する必要があります。

必要条件

ヘッダー：PafeeSemaphore.h

define 値：

Windows ：PAFEE_OS_WINDOWS

Linux ：PAFEE_OS_LINUX

ITRON ：PAFEE_OS_ITRON

10.3.1 Semaphore クラスメンバ

コンストラクタ

PafeeSemaphore	コンストラクタ。

デストラクタ

~PafeeSemaphore	デストラクタ。

操作

Create	セマフォを生成します(Windows、Linux 専用)。
Attach	静的に生成されたセマフォへの関連付けを行います(ITRON 専用)。
Delete	セマフォを破棄します。
Take	セマフォを獲得します。獲得できるまでスレッドはブロックされます。
AsyncTake	セマフォを獲得します。獲得できなければすぐに処理を返します(非同期処理)。
TimedTake	セマフォを獲得します。指定時間待っても獲得できなければ処理を返します タイムアウト時間の単位はプラットホームに依存します。
Release	セマフォを返却します。
GetLastError	最後のエラーコードを取得します。

10.3.2 クラスメンバ詳細

PafeeSemaphore

セマフォクラスを構築します。

```
PafeeSemaphore::PafeeSemaphore(
        void
);
```

パラメータ

なし

戻り値

なし

解説

セマフォのインスタンスを作成します。

セマフォはインスタンス毎に管理されます。複数のセマフォを作成する場合は、作成する数と同数のインスタンスを作成します。

~PafeeSemaphore

セマフォクラスを破棄します。

```
PafeeSemaphore::~PafeeSemaphore(
        void
);
```

パラメータ

なし

戻り値

なし

解説

セマフォクラスを破棄します。

破棄するとき、セマフォの破棄を行います。他のスレッドでセマフォを使用している状態で破棄を行った場合、セマフォスの開放を正常に行うことができなくなることがあります。手動で破棄する場合は確実に全てのスレッドでセマフォが開放されていることを確認し、破棄を行ってください。

Create

セマフォを生成します。Windows 及び Linux のみサポートし、ITORON では使用することができません。

```
bool PafeeSemaphore::Create(
        int maxCount ///< [in] Max count of semaphore
);
```

パラメータ

maxCount:

セマフォの最大カウント数

戻り値

bool:

true	生成成功
false	生成失敗

解説

セマフォを生成します。

すでにセマフォが生成されている場合は、セマフォの生成は失敗します。

Attach

静的に生成したセマフォの関連付けを行います。ITRON のみサポートし、Windows 及び Linux では使用することができません。

```
bool PafeeSemaphore::Attach(
        ID          id
);
```

パラメータ

id

[in] 静的に生成したセマフォの ID

戻り値

bool:

true	関連付け成功
false	関連付け失敗

解説

静的セマフォの関連付けを行います。

すでにセマフォが関連付けされている場合は、セマフォの関連付けは失敗します。

動的にセマフォを作成することはできません。

Delete

セマフォを破棄します。

```
bool PafeeSemaphore::Delete(
        void
);
```

パラメータ

なし

戻り値

bool：

true	破棄成功
false	破棄失敗

解説

セマフォの破棄を行います。

Windows、および Linux 環境下では、他のスレッドでセマフォを使用している場合、破棄は失敗します。

ITRON 環境下においては、破棄はいかなる条件でも成功しますが、実際のセマフォの破棄は行われず、関連付けの解除のみが行われます。

Take

セマフォを獲得します。獲得できるまで処理はブロックされます(同期処理)。

```
bool PafeeSemaphore::Take(
        void
);
```

パラメータ

なし

戻り値

bool:

true	獲得成功
false	獲得失敗

解説

セマフォの獲得を行います。

獲得できるまで処理はブロックされますが、エラーが発生した場合は獲得に失敗し、処理を返します。

AsyncTake

セマフォを獲得します。獲得できなければすぐに処理を返します(非同期処理)。

```
bool PafeeSemaphore::AsyncTake(
        void
);
```

パラメータ

なし

戻り値

bool:

true	獲得成功
false	獲得失敗

解説

セマフォの獲得を行います。

セマフォの獲得を試みます。獲得できる場合はセマフォの獲得を行い、処理を返します。獲得できなければ、すぐに処理を返します。エラーが発生した場合も獲得に失敗し、処理を返します。

TimedTake

セマフォを獲得します。指定時間待っても獲得できなければ処理を返します。
タイムアウト時間の単位はプラットフォームに依存します。

```
bool PafeeSemaphore::TimedTake(
        unsigned long       time        ///< [in] timeout time
);
```

パラメータ

time

セマフォの獲得を試みるタイムアウト間隔。入力単位はプラットフォームに依存します。

戻り値

bool:

true	獲得成功
false	獲得失敗

解説

セマフォの獲得を試みます。獲得できる場合はセマフォの獲得を行い、処理を返します。指定時間待っても獲得できなければ、処理を返します。エラーが発生した場合も獲得に失敗し、処理を返します。

Release

セマフォを開放します。

```
bool PafeeSemaphore::Release(
        void
);
```

パラメータ

なし

戻り値

bool：

true	開放成功
false	開放失敗

解説

獲得しているセマフォを開放します。開放に失敗した場合はエラーを返します。獲得されていないセマフォの開放を行った場合もエラーを返します。

GetLastError

最後のエラーコードを取得します。

```
PafeeSemaphore::SEM_ERROR PafeeSemaphore::GetLastError(
        void
);
```

パラメータ

なし

戻り値

PafeeSemapfore::SEM_ERROR:

ERROR_NON	エラーなし
ERROR_NOHANDLE	セマフォが作成されていません
ERROR_HAS_HANDLE	すでにセマフォが作成されています
ERROR_CANNOT_CREATE	セマフォ作成失敗
ERROR_CANNOT_DELETE	セマフォ削除失敗
ERROR_CANNOT_GET	セマフォの獲得失敗（タイムアウト以外の要因）
ERROR_TIME_OUT	セマフォの獲得失敗（タイムアウト）
ERROR_CANNOT_RELEASE	セマフォの開放失敗
ERROR_MAX	エラー番号の個数

解説

最後に発生したエラーコードを返します。エラーコードは各インスタンスごとに管理されます。10.3.3 公開定数を参照してください。

10.3.3 公開定数

SEM_ERROR

セマフォクラスで発生するエラーコードを定義します。

```
typedef enum PafeeSemaphore::SEM_ERROR{
        ERROR_NONE, /**< Error none */
        ERROR_NOHANDLE, /**< function is called without handle */
        ERROR_HAS_HANDLE, /**< Allready has handle */
        ERROR_CANNOT_CREATE, /**< Can not create */
        ERROR_CANNOT_DELETE, /**< Can not delete */
        ERROR_CANNOT_GET, /**< Can not get */
        ERROR_TIME_OUT, /**< Can not get because of time out */
        ERROR_CANNOT_RELEASE, /**< Can not release */
        ERROR_MAX /**< MAX */
} SEM_ERROR;
```

メンバ定数

PafeeSemaphore::SEM_ERROR:

ERROR_NON	エラーなし
ERROR_NOHANDLE	セマフォが作成されていません。 Create、または Attach を行っていない状態で獲得・開放・削除を行った場合に発生します。
ERROR_HAS_HANDLE	すでにセマフォが作成されています。 Create、または Attach を行った状態のセマフォインスタンスに対し、再度 Create、または Attach を行った場合に発生します。
ERROR_CANNOT_CREATE	セマフォ作成失敗。 リソース不足やその他の原因でセマフォの作成に失敗した場合に発生します。
ERROR_CANNOT_DELETE	セマフォ削除失敗。 他のスレッドで使用中のインスタンスのセマフォを削除しようとした場合に発生します。
ERROR_CANNOT_GET	獲得失敗（タイムアウト以外の要因）。 AsyncLock 使用時、他のスレッドがすでに獲得している場合等に発生します。
ERROR_TIME_OUT	獲得失敗（タイムアウト）。 一定時間獲得を試みたが、セマフォの獲得を行えなかった場合に発生します。
ERROR_CANNOT_RELEASE	開放失敗。
ERROR_MAX	エラー番号の個数

解説

エラー変数はインスタンス毎に管理されます。たとえ、他のセマフォでエラーコードが破棄された場合も、その他のインスタンスでは、最後に発生したエラーを呼び出すことができます。

10.4 Signal

Pafee::PafeeSignal は汎用的なシグナル機能を提供します。ひとつのインスタンスでひとつのシグナルを使用します。複数のシグナルを使用する場合は、使用する個数分のインスタンスを生成する必要があります。

OS が提供する機能を使用して実現しているため、使用する OS の define を makefile に定義する必要があります。

シグナルを手動でリセットする機能はありません。待機関数を抜けるとシグナルは自動的にリセットされます。ptherad の pthread_cond_broadcast のような複数スレッドの待機状態を解除させるような機能はありません。ITORN の様にビットパターンでイベントを待ち受けるような機能はありません。シグナル待ち状態になる前にシグナルされた場合、そのシグナルは記憶され、次回のウェイトはすぐに解け、処理を返します。

必要条件

ヘッダー:PafeeSignal.h

define 値:

Windows :PAFEE_OS_WINDOWS

Linux :PAFEE_OS_LINUX

ITRON :PAFEE_OS_ITRON

10.4.1 Signal クラスメンバ

コンストラクタ

PafeeSignal	コンストラクタ。

デストラクタ

~PafeeSignal	デストラクタ。

操作

Create	シグナルを生成します(Windows、Linux 専用)。
Attach	静的に生成されたシグナルへの関連付けを行います(ITRON 専用)。
Delete	シグナルを破棄します。
Wait	シグナルを待ちます。シグナルするまでスレッドはブロックされます。
AsyncWait	シグナルを待ちます。シグナルしなければすぐに処理を返します(非同期処理)。
TimedWait	シグナルを待ちます。指定時間待ってもシグナルしなければ処理を返します タイムアウト時間の単位はプラットホームに依存します。
Send	シグナルさせます。
GetLastError	最後のエラーコードを取得します。

10.4.2 クラスメンバ詳細

PafeeSignal

シグナルクラスを構築します。

```
PafeeSignal::PafeeSignal(
        void
);
```

パラメータ

なし

戻り値

なし

解説

シグナルのインスタンスを作成します。

シグナルはインスタンス毎に管理されます。複数のシグナルを作成する場合は、作成する数と同数のインスタンスを作成します。

~PafeeSignal

シグナルクラスを破棄します。

```
PafeeSignal::~PafeeSignal(
        void
);
```

パラメータ

なし

戻り値

なし

解説

シグナルクラスを破棄します。

Create

シグナルを生成します。Windows 及び Linux のみサポートし、ITORON では使用することができません。

```
bool PafeeSignal::Create(
        void
);
```

パラメータ

なし

戻り値

bool：

true	生成成功
false	生成失敗

解説

シグナルを生成します。

すでにシグナルが生成されている場合は、シグナルの生成は失敗します。

Attach

静的に生成したシグナルの関連付けを行います。ITRON のみサポートし、Windows 及び Linux では使用することができません。

```
bool PafeeSignal::Attach(
        ID          id
);
```

パラメータ

id

[in] 静的に生成したシグナルの ID

戻り値

bool:

true	関連付け成功
false	関連付け失敗

解説

静的シグナルの関連付けを行います。

すでにシグナルが関連付けされている場合は、シグナルの関連付けは失敗します。

動的にシグナルを作成することはできません。

Delete

シグナルを破棄します。

```
bool PafeeSignal::Delete(
        void
);
```

パラメータ

なし

戻り値

bool:

true	破棄成功
false	破棄失敗

解説

シグナルの破棄を行います。

Windows、および Linux 環境下では、他のスレッドでシグナルを使用している場合、破棄は失敗します。

ITRON 環境下においては、破棄はいかなる条件でも成功しますが、実際のシグナルの破棄は行われず、関連付けの解除のみが行われます。

Wait

シグナルを待ちます。獲得できるまで処理はブロックされます(同期処理)。

```
bool PafeeSignal::Wait(
        void
);
```

パラメータ

なし

戻り値

bool:

true	獲得成功
false	獲得失敗

解説

シグナルの獲得を行います。

獲得できるまで処理はブロックされますが、エラーが発生した場合は獲得に失敗し、処理を返します。

本メソッドがコールされる前にシグナルした場合、そのシグナルは記憶され、本関数は即座に処理を返します。

AsyncWait

シグナルを獲得します。獲得できなければすぐに処理を返します(非同期処理)。

```
bool PafeeSignal::AsyncWait(
        void
);
```

パラメータ

なし

戻り値

bool:

true	獲得成功
false	獲得失敗

解説

シグナルの獲得を行います。

シグナルの獲得を試みます。獲得できる場合はシグナルの獲得を行い、処理を返します。獲得できなければ、すぐに処理を返します。エラーが発生した場合も獲得に失敗し、処理を返します。

本メソッドがコールされる前にシグナルした場合、そのシグナルは記憶され、本関数は即座に処理を返します。

TimedWait

シグナルを獲得します。指定時間待っても獲得できなければ処理を返します。
タイムアウト時間の単位はプラットフォームに依存します。

```
bool PafeeSignal::TimedWait(
        unsigned long       time        ///< [in] timeout time
);
```

パラメータ

time

シグナルの獲得を試みるタイムアウト間隔。入力単位はプラットフォームに依存します。

戻り値

bool:

true	獲得成功
false	獲得失敗

解説

シグナルの獲得を試みます。獲得できる場合はシグナルの獲得を行い、処理を返します。指定時間待っても獲得できなければ、処理を返します。エラーが発生した場合も獲得に失敗し、処理を返します。

本メソッドがコールされる前にシグナルした場合、そのシグナルは記憶され、本関数は即座に処理を返します。

Send

シグナルさせます。

```
bool PafeeSignal::Send(
        void
);
```

パラメータ

なし

戻り値

bool：

true	シグナル成功
false	シグナル失敗

解説

シグナルさせます。待機しているスレッドが存在しない場合、シグナルは記憶され、次回シグナル待機が即座に終了し、処理を返します。

GetLastError

最後のエラーコードを取得します。

```
PafeeSignal::SIGNAL_ERROR PafeeSignal::GetLastError(
        void
);
```

パラメータ

なし

戻り値

PafeeSemapfore::SIGNAL_ERROR：

ERROR_NON	エラーなし
ERROR_NOSIGNAL	シグナルが作成されていません
ERROR_HAS_SIGNAL	すでにシグナルが作成されています
ERROR_CANNOT_CREATE	シグナル作成失敗
ERROR_CANNOT_DELETE	シグナル削除失敗
ERROR_CANNOT_SIGNAL	シグナル失敗（タイムアウト以外の要因）
ERROR_TIME_OUT	シグナル失敗（タイムアウト）
ERROR_OHTER	その他のエラー
ERROR_MAX	エラー番号の個数

解説

最後に発生したエラーコードを返します。エラーコードは各インスタンスごとに管理されます。エラーコードの詳細は、10.4.3 公開定数を参照してください。

10.4.3 公開定数

SIGNAL_ERROR

シグナルクラスで発生するエラーコードを定義します。

```
typedef enum tagSIGNAL_ERROR{
        ERROR_NONE, /**< Error none */
        ERROR_NOSIGNAL, /**< handle, id is not exist */
        ERROR_HAS_SIGNAL, /**< Allready exist m_handle */
        ERROR_CANNOT_CREATE, /**< Signal can not create */
        ERROR_CANNOT_DELETE, /**< Signal can not delete */
        ERROR_CANNOT_SIGNAL, /**< Signal can not get */
        ERROR_TIME_OUT, /**< Signal can not get for time out */
        ERROR_OHTER, /**< Unknown error */
        ERROR_MAX /**< Error none */
} SIGNAL_ERROR;
```

メンバ定数

PafeeSignal::SIGNAL_ERROR：

ERROR_NON	エラーなし
ERROR_NOHANDLE	シグナルが作成されていません。 Create、または Attach を行っていない状態で獲得・開放・削除を行った場合に発生します。
ERROR_HAS_HANDLE	すでにシグナルが作成されています。 Create、または Attach を行った状態のシグナルインスタンスに対し、再度 Create、または Attach を行った場合に発生します。
ERROR_CANNOT_CREATE	シグナル作成失敗。 リソース不足やその他の原因でシグナルの作成に失敗した場合に発生します。
ERROR_CANNOT_DELETE	シグナル削除失敗。 他のスレッドで使用中のインスタンスのシグナルを削除しようとした場合に発生します。
ERROR_CANNOT_SIGNAL	シグナル失敗（タイムアウト以外の要因）。 AsyncLock 使用時、シグナルしていない場合等に発生します。
ERROR_TIME_OUT	シグナル失敗（タイムアウト）。 一定時間シグナルを待ったが、シグナルしなかった場合に発生します。
ERROR_OTHERS	上記以外のその他のエラー。
ERROR_MAX	エラー番号の個数

解説

エラー変数はインスタンス毎に管理されます。たとえ、他のシグナルでエラーコードが破棄された場合も、その他のインスタンスでは、最後に発生したエラーを呼び出すことができます。

11. C標準ライブラリサブセット

フリースタンディング環境ではサポートされているC 標準ライブラリおよびC++標準ライブラリが極めて限定的である場合が多くあります。 そのような環境で本フレームワークを提供するために、必要なC標準ライブラリ関数を独自実装しています。

独自に実装された標準ライブラリサブセットと、開発環境下で用意されている標準ライブラリは define を用いて使用する関数を切り替えることができます。

必要条件

ヘッダー: pafee_stdio.h（stdio.h のサブセット）
pafee_stdlib.h（stdlib.h のサブセット）
pafee_string.h（string.h のサブセット）

define 値:

stdio	:PAFEE_LIB_STDIO_ENABLE（stdio.h のサブセット使用）
stdlib	:PAFEE_LIB_STDLIB_ENABLE（stdlib.h のサブセット使用）
string	:PAFEE_LIB_STRING_ENABLE（string.h のサブセット使用）

11.1 stdio

stdio では「sptinrf」と「snprintf」を実装します。標準関数と本フレームワークで提供する関数は、define の有無で使用する関数を選択することができます。この場合の関数の書式は、本フレームワークの書式となります。

たとえば、「sprintf」関数を使用する場合、ソースコードでは「pafee_sprintf」を用いて開発を行うと、make ファイルに「PAFEE_LIB_STDIO_ENABLE」を定義した場合は、本フレームワークの関数「pafee_sprintf」が使用され、定義しなかった場合は、標準ライブラリの「sprintf」が使用されます。

必要条件

ヘッダー：PafeeStdio.h

define 値：

PAFEE_LIB_STDIO_ENABLE

11.1.1 サブセット関数一覧

関数一覧

pafee_sprintf	機能を限定した sprintf 関数を提供します。
pafee_snprintf	機能を限定した snprintf 関数を提供します。

11.1.2 サブセット関数詳細

pafee_sprintf

機能を限定した sprintf 関数を提供します。

```
int pafee_sprintf(
        char *convertString,
        const char *pFormat,
        ...
);
```

パラメータ

convertString	変換した文字列を格納するﾊﾞｯﾌｧのポインタ
pFormat	変換する文字列のﾌｫｰﾏｯﾄ
…	可変引数

戻り値

int

作成した文字列長

解説

sprintf の機能を限定した関数を提供します。

本関数でサポートする書式は以下の通りです。

変換文字:	d,x,X,c,s,p,%
ﾌｨｰﾙﾄﾞ幅:	数値
エスケープ文字:	¥t,¥n,¥r

pafee_snprintf

機能を限定した snprintf 関数を提供します。

```
int pafee_snprintf(
        char *convertString,
        const unsigned int maxSize,
        const char *pFormat,
        …
);
```

パラメータ

convertString	変換した文字列を格納するバッファのポインタ
maxSize	変換した文字列を格納するバッファサイズ
pFormat	変換する文字列のフォーマット
…	可変引数

戻り値

int

作成した文字列長

解説

snprintf の機能を限定した関数を提供します。

本関数でサポートする書式は以下の通りです。

変換文字:	d,x,X,c,s,p,%
フィールド幅:	数値
エスケープ文字:	¥t,¥n,¥r

11.2 stdlib

stdlib では「atoi」、「atol」、「strtol」および「strtoul」を実装します。標準関数と本フレームワークで提供する関数は、define の有無で使用する関数を選択することができます。この場合の関数の書式は、本フレームワークの書式となります。

たとえば、「atoi」関数を使用する場合、ソースコードでは「pafee_atoi」を用いて開発を行うと、make ファイルに「PAFEE_LIB_STDLIB_ENABLE」を定義した場合は、本フレームワークの関数「pafee_atoi」が使用され、定義しなかった場合は、標準ライブラリの「atoi」が使用されます。

必要条件

ヘッダー：PafeeStdlib.h

define 値：

PAFEE_LIB_STDLIB_ENABLE

11.2.1 stdlib サブセット関数一覧

関数一覧

pafee_atoi	atoi 関数と同等の機能を提供します。
pafee_atol	atol 関数と同等の機能を提供します。
pafee_strtol	機能を限定した strtol 関数を提供します。
pafee_strtoul	機能を限定した strtoul 関数を提供します。

pafee_atoi

atoi 関数と同等の機能を提供します。

```
int pafee_atoi(
        const char *convertString
);
```

パラメータ

convertString　　変換する文字列を格納したバッファのポインタ

戻り値

int
　　変換した数値

解説

atoi 関数と同等の機能を提供します。

pafee_atol

atol 関数と同等の機能を提供します。

```
long pafee_atol(
        const char *convertString
);
```

パラメータ

convertString　　変換する文字列を格納したバッファのポインタ

戻り値

long
　　変換した数値

解説

atol 関数と同等の機能を提供します。

pafee_strtol

機能を限定した strtol 関数を提供します。

```
long pafee_strtol(
        const char * restrictString,
        char ** restrictEnd,
        int base
);
```

パラメータ

restrictString	変換する文字列を格納したバッファのポインタ
restrictEnd	変換した文字列の最終アドレスを格納するバッファのポインタ
base	基数

戻り値

long

変換した数値

解説

機能を限定した strtol 関数を提供します。
本フレームワークで提供する strtol 関数では、10 進数および 16 進数の変換のみをサポートします。

pafee_strtoul

機能を限定した strtoul 関数を提供します。

```
unsignd long pafee_strtol(
        const char * restrictString,
        char ** restrictEnd,
        int base
);
```

パラメータ

restrictString	変換する文字列を格納したバッファのポインタ
restrictEnd	変換した文字列の最終アドレスを格納するバッファのポインタ
base	基数

戻り値

long

変換した数値

解説

機能を限定した strtoul 関数を提供します。
本フレームワークで提供する strtoul 関数では、10 進数および 16 進数の変換のみをサポートします。

11.3 string

string では「memcmp」、「memcpy」、「memmove」、「memset」、「strcat」、「strncat」、「strcpy」、「strncpy」、「strlen」、「strcmp」、「strncmp」、「strstr」、「strrstr」、「strchr」および「strrchr」を実装します。標準関数と本フレームワークで提供する関数は、define の有無で使用する関数を選択することができます。この場合の関数の書式は、本フレームワークの書式となります。

たとえば、「memcpy」関数を使用する場合、ソースコードでは「pafee_memcpy」を用いて開発を行うと、make ファイルに「PAFEE_LIB_STRING_ENABLE」を定義した場合は、本フレームワークの関数「pafee_memcpy 」が使用され、定義しなかった場合は、標準ライブラリの「memcpy」が使用されます。

必要条件

ヘッダー：PafeeString.h

define 値：

PAFEE_LIB_STRING_ENABLE

11.3.1 string サブセット関数一覧

関数一覧

pafee_memcmp	memcmp 関数と同等の機能を提供します。
pafee_memcpy	memcpy 関数と同等の機能を提供します。
pafee_memmove	memmove 関数と同等の機能を提供します。
pafee_memset	memset 関数と同等の機能を提供します。
pafee_strcat	strcat 関数と同等の機能を提供します。
pafee_strncat	strncat 関数と同等の機能を提供します。
pafee_strcpy	strcpy 関数と同等の機能を提供します。
pafee_strncpy	strncpy 関数と同等の機能を提供します。
pafee_strlen	strlen 関数と同等の機能を提供します。
pafee_strcmp	strcmp 関数と同等の機能を提供します。
pafee_strstr	strstr 関数と同等の機能を提供します。
pafee_strchr	strchr 関数と同等の機能を提供します。
pafee_strrchr	strrchr 関数と同等の機能を提供します。

pafee_memcmp

memcmp 関数と同等の機能を提供します。

```
int pafee_memcmp(
        const void *pBuffer1,
        const void *pBuffer2,
        size_t count
);
```

パラメータ

pBuffer1	比較対象のバッファのポインタ
pBuffer2	もう一つの比較対象のバッファのポインタ
count	比較する長さ

戻り値

int

0	:比較対象のバッファの内容が同じ
0 以外	:比較対象のバッファの内容が異なる

解説

memcmp 関数と同等の機能を提供します（比較対象データ「pBuffer1」ともう一つの比較対象データ「pBuffer2」の内容を比較し、二つの内容が同じであれば 0 を、異なれば 0 以外を返します）。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_memcmp」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_memcmp」関数は C 標準ライブラリの「memcmp」関数が使用されます。

pafee_memcpy

memcpy 関数と同等の機能を提供します。

```
char *pafee_memcpy(
        char *pTarget,
        const char *pSource,
        size_t count
);
```

パラメータ

pTarget	コピーしたデータを格納するバッファのポインタ
pSource	コピーするデータを格納したバッファのポインタ
count	コピーするデータ長

戻り値

void *

コピーしたデータを格納するバッファのポインタを返します。

解説

memcpy 関数と同等の機能を提供します（データ「pSource」を指定されたバッファ「pTarget」に指定サイズ「count」分コピーします。コピー元データとコピー先データのバッファが重なっている場合、データは保証されません。）。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_memcpy」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_memcpy」関数は C 標準ライブラリの「memcpy」関数が使用されます。

pafee_memmove

memmove 関数と同等の機能を提供します。

```
char *pafee_memmove(
        char *pTarget,
        const char *pSource,
        size_t count
);
```

パラメータ

pTarget	コピーしたデータを格納するバッファのポインタ
pSource	コピーするデータを格納したバッファのポインタ
count	コピーするデータ長

戻り値

void *

コピーしたデータを格納するバッファのポインタを返します。

解説

memove 関数と同等の機能を提供します（データ「pSource」を指定されたバッファ「pTarget」に指定サイズ「count」分コピーします。コピー元データとコピー先データのバッファが重なっている場合も、データは保証されます。）。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_memmove」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_memmove」関数は C 標準ライブラリの「memmove」関数が使用されます。

pafee_memset

memset 関数と同等の機能を提供します。

```
void *pafee_memcmp(
        void *pTarget,
        int value,
        unsigned int count
);
```

パラメータ

pTarget	値を設定するバッファのポインタ
value	バッファに設定する値
count	設定するバッファの長さ

戻り値

void *

値を設定するバッファのポインタ

解説

memset 関数と同等の機能を提供します（値を設定するバッファ「pTarget」に指定された値「value」を指定されたサイズ「count」BYTE 分設定します）。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_memset」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_memset」関数は C 標準ライブラリの「memset」関数が使用されます。

pafee_strcat

strcat 関数と同等の機能を提供します。

```
char *pafee_strcat(
        char *pTarget,
        const char *pSource,
);
```

パラメータ

pTarget	追加した文字列を格納するバッファのポインタ
pSource	追加する文字列を格納したバッファのポインタ

戻り値

char *

追加した文字列を格納するバッファのポインタを返します。

解説

strncpy 関数と同等の機能を提供します（文字列「pSource」を指定されたバッファ「pTarget」に追加します)。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_strcat」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_strcat」関数は C 標準ライブラリの「strcat」関数が使用されます。

pafee_strncat

strncat 関数と同等の機能を提供します。

```
char *pafee_strncat(
        char *pTarget,
        const char *pSource,
        size_t count
);
```

パラメータ

pTarget	追加した文字列を格納するバッファのポインタ
pSource	追加する文字列を格納したバッファのポインタ
count	追加する文字列長

戻り値

char *

追加した文字列を格納するバッファのポインタを返します。

解説

strncpy 関数と同等の機能を提供します（文字列「pSource」を指定されたバッファ「pTarget」に指定サイズ「count」分追加します。追加する文字列「pString」が count より小さい場合は、「pString」分のみ追加されます)。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_strncat」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_strncat」関数は C 標準ライブラリの「strncat」関数が使用されます。

pafee_strcpy

strcpy 関数と同等の機能を提供します。

```
char *pafee_strcpy(
        char *pTarget,
        const char *pSource,
);
```

パラメータ

pTarget	コピーした文字列を格納するバッファのポインタ
pSource	コピーする文字列を格納したバッファのポインタ

戻り値

char *

コピーした文字列を格納するバッファのポインタを返します。

解説

strcpy 関数と同等の機能を提供します（文字列「pSource」を指定されたバッファ「pTarget」にコピーします)。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_strcpy」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_strcpy」関数は C 標準ライブラリの「strcpy」関数が使用されます。

pafee_strncpy

strncpy 関数と同等の機能を提供します。

```
char *pafee_strncpy(
        char *pTarget,
        const char *pSource,
        size_t count
);
```

パラメータ

pTarget	コピーした文字列を格納するバッファのポインタ
pSource	コピーする文字列を格納したバッファのポインタ
count	コピーする文字列長

戻り値

char *

コピーした文字列を格納するバッファのポインタを返します。

解説

strncpy 関数と同等の機能を提供します（文字列「pSource」を指定されたバッファ「pTarget」に指定サイズ「count」分コピーします。コピーする文字列「pString」が count より小さい場合は、「pString」分のみコピーされます)。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_strncpy」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_strncpy」関数は C 標準ライブラリの「strncpy」関数が使用されます。

pafee_strlen

strlen 関数と同等の機能を提供します。

```
size_t pafee_strlen(
        const char *pString
);
```

パラメータ

pBuffer　　長さを求める文字列を格納したﾊﾞｯﾌｧのポインタ

戻り値

size_t
　　文字列長

解説

strlen 関数と同等の機能を提供します（文字列「pString」の長さを返します）。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_strlen」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_strlen」関数は C 標準ライブラリの「strlen」関数が使用されます。

pafee_strcmp

strcmp 関数と同等の機能を提供します。

```
int pafee_strcmp(
        const char *pBuffer1,
        const char *pBuffer2
);
```

パラメータ

pBuffer1	比較対象文字列を格納したバッファのポインタ
pBuffer2	もう一つの比較対象文字列を格納したバッファのポインタ

戻り値

int

0	:比較対象文字列の内容が一致
0 以外	:比較対象文字列の内容が異なる

解説

strcmp 関数と同等の機能を提供します（比較対象文字列「pString1」ともう一つの比較対象文字列「pString2」の内容を比較し、二つの内容が同じであれば 0 を、異なれば 0 以外を返します）。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_strcmp」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_strcmp」関数は C 標準ライブラリの「strcmp」関数が使用されます。

pafee_strstr

strstr 関数と同等の機能を提供します。

```
char *pafee_strstr(
        const char *pString,
        const char *pSearch
);
```

パラメータ

pString	検索されるする文字列を格納したバッファのポインタ
pSearch	検索する文字列を格納したバッファのポインタ

戻り値

char *

NULL	:検索する文字列が検索される文字列中に存在しない
NULL 以外	:検索する文字列を示すポインタ

解説

strstr 関数と同等の機能を提供します（検索する文字列「pSearch」を検索される文字列「pString」の先頭から検索し、最初に「pSearch」が現れた位置を返します）。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_strstr」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_strstr」関数はＣ標準ライブラリの「strstr」関数が使用されます。

pafee_strchr

strchr 関数と同等の機能を提供します。

```
char *pafee_strchr(
        const char *pString,
        int find
);
```

パラメータ

pString	検索されるする文字列を格納したバッファのポインタ
find	検索する文字

戻り値

char *

NULL	:検索する文字が検索される文字列中に存在しない
NULL 以外	:検索する文字を示すポインタ

解説

strchr 関数と同等の機能を提供します（検索する文字「find」を検索される文字列「pString」の先頭から検索し、最初に「find」が現れた位置を返します）。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_strchr」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_strstr」関数は C 標準ライブラリの「strchr」関数が使用されます。

pafee_strrchr

strrchr 関数と同等の機能を提供します。

```
char *pafee_strrchr(
        const char *pString,
        int find
);
```

パラメータ

pString	検索されるする文字列を格納したﾊﾞｯﾌｧのポインタ
find	検索する文字

戻り値

char *

NULL	:検索する文字が検索される文字列中に存在しない
NULL 以外	:検索する文字を示すポインタ

解説

strrchr 関数と同等の機能を提供します（検索する文字「find」を検索される文字列「pString」の末尾から検索し、最初に「find」が現れた位置を返します）。
「PAFEE_LIB_STRING_ENABLE」マクロを定義することで、本フレームワークの「pafee_strrchr」関数を使用します。
「PAFEE_LIB_STRING_ENABLE」マクロを定義しない場合は、「pafee_strrstr」関数は C 標準ライブラリの「strrchr」関数が使用されます。

pafee_strrstr

strrstr 関数と同等の機能を提供します。

```
char *pafee_strrstr(
        const char *pString,
        const char *pSearch
);
```

パラメータ

pString	検索されるする文字列を格納したﾊﾞｯﾌｧのポインタ
pSearch	検索する文字列を格納したﾊﾞｯﾌｧのポインタ

戻り値

char *

NULL	:検索する文字列が検索される文字列中に存在しない
NULL 以外	:検索する文字列の先頭を示すポインタ

解説

strrstr 関数と同等の機能を提供します（検索する文字列「pSearch」を検索される文字列「pString」の末尾から検索し、最初に「pSearch」が現れた位置を返します）。
strrstr 関数は非標準関数です。
pafee_strrstr 関数は他の PafeeString.h ﾍｯﾀﾞの関数とは異なり、「PAFEE_LIB_STRING_ENABLE」ﾏｸﾛによる pafee_strrstr 関数と strrstr 関数のｽｲｯﾁは行いません。