ハム太と作るC++ライブラリ1

-STL編-

haseham

本書について

・「**STL**」は、C++でよく使う「**List**」などの標準的なﾃﾞｰﾀ構造を、
「ﾃﾝﾌﾟﾚｰﾄｸﾗｽ」として実装したﾗｲﾌﾞﾗﾘである。

(ﾃﾞｰﾀ構造というと難解に聞こえるが、これは「ｺﾝﾃﾅｸﾗｽ」のことである)

・「ﾃﾝﾌﾟﾚｰﾄｸﾗｽ」は、格納する値のﾃﾞｰﾀ型を抽象化したｸﾗｽであり、
例えば、**class Matrix44< T >** といったﾃﾝﾌﾟﾚｰﾄｸﾗｽを実装しておけば、

```
Matrix44< float > m1;
Matrix44< double > m2;
```

といった具合に、格納する値のﾃﾞｰﾀ型を指定するだけで
使いまわすことができるため、同じ処理を
何度も書かずに済み、修正も比較的容易である。

・しかしその一方で、読みにくいｺｰﾄﾞになりがちであり、
また、ﾃﾞﾊﾞｯｸ時のｴﾗｰｺｰﾄﾞも意味不明であることが多い。

・筆者もこれを書くにあたって、大変な苦労を強いられた。

・その手間を考えれば、少なからず多くの皆さんにとって
本書はきっと有意義なものとなるに違いない。

・ﾈｰﾐﾝｸﾞ規則については、ﾊﾝｶﾞﾘｱﾝ記法は使用せず、
Javaや**.NET**に慣れ親しんだ皆さんにもわかりやすいものとなっている。

・特にﾛｼﾞｶﾙな箇所は無いし、ﾃｽﾄは行なっているが、
ﾊﾞｸﾞが無いとは言い切れない。

・そのため、利用に際しては、自己責任でお願いしたい。

・「そもそも、**C/C++**の構文がわからない」という方は、
まず「構文編」を読んで頂きたい。↓

http://p.booklog.jp/book/74354

STLの概要

・「**STL**」には、様々なｺﾝﾃﾅｸﾗｽが含まれているが、
　そのすべてを網羅することは大変な手間であるし、
　「概観をわかりやすく伝える」という本書の意図に反する。

・込み入った話は、洋書の訳本を読んでいただければと思う。

【 本書で紹介するｺﾝﾃﾅｸﾗｽ 】

・「std::**vector**」 ... 可変長配列。宣言後に要素数を変更できる。
　また、あらかじめ多めに確保するように指定しておくことで、
　ﾘｻｲｽﾞ時のﾒﾓﾘの再確保を減らす事ができる。

・「std::**list**」 ... 双方向ﾘﾝｸﾄﾞﾘｽﾄ。各要素は、直前と直後の要素への参照を持つ。
　前後の要素を相互にﾘﾝｸさせることによって、鎖(くさり)のように連なっている。

・「stdext::**hash_map**」 ... ﾊｯｼｭ表。配列のような「要素番号」のみならず、
　「要素名」でも参照できるため、「辞書」ともいわれる。
　非常に便利である反面、大量のﾒﾓﾘを要するため、注意が必要である。
　「std::**map**」とは異なり、ﾊｯｼｭｱﾙｺﾞﾘｽﾞﾑで駆動するため、
　実行速度では有利である。

・「std::**stack**」 ... ｽﾀｯｸは、たまった書類のように、上へ上へと積み上げていく構造をしている。

・「std::**queue**」 ... ｷｭｰは、一方通行のﾄﾝﾈﾙに例えられることが多い。
　ﾄﾝﾈﾙの入口から、新規の要素を詰め込み、出口で取り出して削除する。

・「std::**deque**」 ... ﾃﾞｷｭｰでは、両端から要素を詰め込む。どちらもが入口であり、出口である。

・「std::**auto_ptr**」 ... **1**つのﾎﾟｲﾝﾀを格納し、使用されなくなった時点で自動的に解放する。

・「std::**allocator**」 ... 各ｺﾝﾃﾅｸﾗｽの内部で、ﾒﾓﾘの確保を行なっているｸﾗｽである。
派生させて、処理を上書きすることで、処理速度が向上する。

・「std::**iterator**」 ... ｲﾃﾚｰﾀｰは、**STL**におけるﾎﾟｲﾝﾀである。
ﾘｻｲｽﾞなどによって、要素のｱﾄﾞﾚｽが移動した場合は、無効となる。

・「std::**wstring**」 ... ﾜｲﾄﾞ文字配列。

・「std::**bitset**」 ... ﾋﾞｯﾄﾌﾗｸﾞ。

【vector】（可変長配列）

```
#pragma once

#ifndef _STL_VECTOR_H_
#define _STL_VECTOR_H_



// ｱﾛｹｰﾀｸﾗｽのﾍｯﾀﾞﾌｧｲﾙをｲﾝｸﾙｰﾄﾞしておく。
#include "STL_Allocator.h"

#include <vector>

using namespace std;

namespace STL
{

// ------------------------------------------------
// 【使用例】

// STL::CVector< int >* p_v = new STL::CVector< int >();
//
// p_v->Add( 111111 );
// p_v->Add( 222222 );
// p_v->Add( 333333 );
//
// int value0 = p_v->GetValue( 0 );
// int value1 = p_v->GetValue( 1 );
// int value2 = p_v->GetValue( 2 );
//
// delete p_v;

// std::vector< int, STL::CAllocator< int > > v1;
//
// v1.insert( v1.end(), 100 );
// v1.insert( v1.end(), 200 );
```

```
//
// std::vector< int, STL::CAllocator< int > > v2;
//
// v2.insert( v2.end(), 300 );
// v2.insert( v2.end(), 400 );
//
// v1 = v2; // v2 のすべての要素が、v1 にｺﾋﾟｰされる。

// ====================================================
// 可変長配列ｸﾗｽの定義 ( std::vectorのﾗｯﾊﾟｰ )

// ・指定要素へのｱｸｾｽや、両端への追加、削除は、定数時間で完了する。
// ・その反面、中間地点の要素を検索したり、挿入するには線形時間かかる。
// ・ｺﾝｽﾄﾗｸﾀでｷｬﾊﾟｼﾃｨを指定するよりも、
//  ﾃﾞﾌｫﾙﾄｺﾝｽﾄﾗｸﾀで宣言した後で、ｷｬﾊﾟｼﾃｨを設定した方が効率が良い。

// ※ﾘｻｲｽﾞを行うと、別のﾒﾓﾘ領域が確保され、
//   要素値が移動するので、ｱﾄﾞﾚｽが変わってしまう。
//  ( ﾎﾟｲﾝﾀ変数に格納している以前のｱﾄﾞﾚｽが、無効となるのと同じ )

// ※要素数は、拡張はできるが、縮小はできない。

// ※要素が存在しないindexやｲﾃﾚｰﾀｰを渡した場合はｺｹる。
//  ( 速度を優先するため、STL側でも範囲判定はあまり行なっていない )

template<typename TValue>
class CVector
{


protected: // publicでもOK。

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾒﾝﾊﾞ

std::vector< TValue , CAllocator< TValue > > vector; // ﾍﾞｸﾀｰ本体。
```

```
public:

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾍﾞｸﾀｰ本体の型

   typedef typename  std::vector< TValue , CAllocator< TValue > > InsideType;


// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｲﾃﾚｰﾀｰ

      // 通常のｲﾃﾚｰﾀｰ ( ﾗﾝﾀﾞﾑｱｸｾｽ )
typedef typename std::vector< TValue, CAllocator< TValue > >::iterator Iterator;

      // 逆順ｲﾃﾚｰﾀｰ
typedef typename std::vector< TValue, CAllocator< TValue > >::reverse_iterator RevIterator;


// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

CVector( void )
{

}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾃﾞｽﾄﾗｸﾀ

~CVector( void )
{

}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定値で初期化された指定数の要素】で初期化する。( assign() )
```

```
  inline void InitByValue( TValue value_, const size_t count_ )
  {
        vector.assign( count_, value_ );
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定範囲の要素値】で初期化する。                    ( assign() )

// dest_vector を、src_vector[5]、[6]、[7] で初期化する場合は、
//
// dest_vector.InitByRange( src_vector.GetItr( 5 ), src_vector.GetItr( 8 ) );

  inline void InitByRange( Iterator src_start_itr_, Iterator src_end_itr_ )
  {
        vector.assign( src_start_itr_ , src_end_itr_);
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定indexのｲﾃﾚｰﾀ】を返す。

// ※戻り値のｲﾃﾚｰﾀを、参照にしていないのには理由がある。
//   このｸﾗｽがﾛｰｶﾙ変数として宣言された場合、
//   ｽﾀｯｸ領域のｱﾄﾞﾚｽを返してしまうため、ｺｹるからである。

inline Iterator GetItr( const size_t index_ )
{
return ( vector.begin() + index_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾍﾞｸﾀｰ】を返す。

inline InsideType& GetVector( void )
{
        return vector;
}
```

```
// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾍﾞｸﾀｰ】を格納する。

inline void SetVector( InsideType& vector_ )
{
	vector = vector_;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【要素数】を返す。                                    ( size() )

inline size_tGetCount( void )
{
	return vector.size();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最大要素数】を返す。                                ( max_size() )

// ※これは「size_t」の最大値。    (「size_t」は、32bitｼｽﾃﾑでは「unsigned int」)

inline size_t GetMaxCount( void )
{
	return vector.max_size();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【要素数】を変更する。                                ( resize() )

// ・新しく追加された要素の値は、0ｸﾘｱされている。

inline void Resize( const size_t length_ )
{
	vector.resize( length_ );
}
```

```
// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【要素数】を変更し、【追加された要素】を、指定値で初期化する。( resize() )

inline void ResizeEx( const size_t length_, TValue value_ )
{
        vector.resize( length_, value_);
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【最初の要素の値】を返す。                    ( front() )

inline TValue GetFirstValue( void )
{
        return vector.front();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【最後の要素の値】を返す。                    ( back() )

inline TValue GetLastValue( void )
{
        return vector.back();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexの要素値】を返す。              ( 値 = vector[i] )

inline TValue GetValue( const size_t index_)
{
returnvector[ index_ ];

// vector.at( index_ ) の場合は、範囲外のindexに対して、out_of_range例外をｽﾛｰする。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexの要素】に【指定値】を格納する。    ( vector[i] = 値 )
```

```
voidSetValue( const size_t index_, TValue value_ )
{
        vector[ index_ ] = value_;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【要素】を末尾に追加する。                          ( push_back() )

inline void Add(TValue value_ )
{
return vector.push_back( value_ ); // 追加する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定ｲﾃﾚｰﾀｰの要素】に、【要素】を挿入し、【挿入位置のｲﾃﾚｰﾀｰ】を返す。( insert() )

// ※指定したｲﾃﾚｰﾀｰの要素に、指定値が格納される。
// ※指定するindexは、AddやResizeで拡張された範囲を超えるとｺｹる。

// ※逆順ｲﾃﾚｰﾀは受け付けない。

inline IteratorInsert( Iterator itr_, TValue value_ )
{
return vector.insert( itr_, value_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定ｲﾃﾚｰﾀｰの要素】に、【指定数の要素】を挿入する。( insert() )

// ※逆順ｲﾃﾚｰﾀは受け付けない。

inline void InsertByValue( Iterator itr_, TValue value_, const size_t count_ )
{
  vector.insert( itr_, count_, value_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
```

```
// 【指定index】に、【指定範囲の要素】を挿入する。　　(insert())

// ※逆順ｲﾃﾚｰﾀは受け付けない。

// vector1 の [1] に、vector2 の [5]、[6]、[7] を挿入する場合は、
//
// vector1.InsertByRange( vector1.GetItr(1), vector2.GetItr(5), vector2.GetItr(8) );

inline void InsertByRange( Iterator dest_itr_, Iterator src_start_itr_, Iterator src_end_itr_ )
{
  vector.insert( dest_itr_, src_start_itr_, src_end_itr_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定index】に、【要素】を挿入し、【挿入位置のｲﾃﾚｰﾀｰ】を返す。(insert())

// ※指定したindexの要素に、指定値が格納される。
// ※指定するindexは、AddやResizeで拡張された範囲を超えるとｺｹる。

inline Iterator InsertAt(const size_t index_, TValue value_ )
{
return vector.insert( vector.begin() + index_, value_); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定ｲﾃﾚｰﾀ-の指す要素】を削除し、【その次の要素のｲﾃﾚｰﾀｰ】を返す。(erase())

inline Iterator Remove( Iterator itr_ )
{
        return vector.erase( itr_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定範囲の要素】を削除し、【その次の要素のｲﾃﾚｰﾀｰ】を返す。(erase())

// ※逆順ｲﾃﾚｰﾀは受け付けない。
```

```
// vector1 の [5]、[6]、[7] を削除する場合は、
//
// vector1.RemoveByRange( vector1.GetItr(5), vector1.GetItr(8) );

inline Iterator RemoveByRange( Iterator start_itr_, Iterator end_itr_ )
{
        return vector.erase( start_itr_, end_itr_ ); // 削除する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexの要素】を削除する。                    ( erase() )

inline Iterator RemoveAt( const size_t index_ )
{
        return vector.erase( vector.begin() + index_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【最後の要素】を削除する。                         ( pop_back() )

inline void RemoveLast(void)
{
        vector.pop_back();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｸﾘｱする。                                         ( clear() )

inline void Clear(void)
{
        vector.clear();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【別のｲﾝｽﾀﾝｽ】と内容を入れ替える。                  ( swap() )

inline void Swap( CVector<TValue>& vector_ )
```

```
{
	vector.swap( vector_.GetVector() );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【内部的に確保される要素数】を返す。　　（capacity()）

//・未設定だと、要素数が設定されている。

inline size_t GetCapacity( void )
{
	return vector.capacity();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【内部的に確保される要素数】を変更する。　　（reserve()）

//※ﾍﾞｸﾀｰは、内部的にﾒﾓﾘ領域を多めに確保していて、この値は、その際の要素数。
//　この値を多めに見積もっておけば、要素数が拡張された際に、
//　再確保&ｺﾋﾟｰの回数が減ることが期待できる。
//・この値は、vector::capacity()で取得できる。
//・多すぎる場合は、vector::shrink_to_fit()で切り詰めることができる。

inline void SetCapacity( const size_t capacity_ )
{
	vector.reserve(capacity_ );
}

// ----------------------------------------------------
//【内部的に確保されている要素数】を切り詰める。（shrink_to_fit()）

//※さいきん追加されたらしく未定義。

//inline void FitCapacity(void)
//{
//	vector.shrink_to_fit();
//}
```

```
// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最初の要素のｲﾃﾚｰﾀ】 を返す。                          ( begin() )

// int value = *itr; // ふつうに最初の要素値が返る。

inline Iterator GetFirstItr( void )
{
        return vector.begin();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最後の要素のｲﾃﾚｰﾀ】 を返す。                            ( end() )

// int value = *itr;    // ※範囲外のﾎﾟｲﾝﾀなので、ｱｸｾｽ違反ｴﾗｰが出る。
// int value = *(itr-1); // ※最後の要素値。( なぜか逆方向に進めてしまう )

inline Iterator GetLastItr( void )
{
        return vector.end();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 逆順ﾙｰﾌﾟ時の 【最初の要素([count-1])のｲﾃﾚｰﾀ】 を返す。( rbegin() )

// for ( RevIterator itr = v.rbegin() ; itr != v1.rend(); itr++ )
// {
    //      int value =  *(itr) ;
// }

// 要素数が3の場合、
//
// int value = *( itr );    // 要素[2]の値が返る。
// int value = *( itr - 1 ); // 要素[1]の値が返る。
// int value = *( itr - 2 ); // 要素[0]の値が返る。

inline RevIterator GetFirstItr_Rev( void )
```

```
{
        return vector.rbegin();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 逆順ﾙｰﾌﾟ時の、【最後の要素([0])のｲﾃﾚｰﾀ】を返す。 ( rend() )

// 要素数が3の場合、
//
// int value = *( itr - 1 ); // 要素[0]の値が返る。
// int value = *( itr - 2 ); // 要素[1]の値が返る。
// int value = *( itr - 3 ); // 要素[2]の値が返る。

    // ※ *itr は構文ｴﾗｰ。
// ※ *(itr) は範囲外のﾎﾟｲﾝﾀなので、ｱｸｾｽ違反ｴﾗｰが出る。

inline RevIterator GetLastItr_Rev( void )
{
        return vector.rend();
}

// ----------------------------------------------
// ##################################################
// 演算子のｵｰﾊﾞｰﾛｰﾄﾞ
// ##################################################
// ----------------------------------------------<OK><ﾃｽﾄ済み>
//  TValue value = instance[ index_ ]

// ※ int v1 = (*p_vector)[0];  これはできるが、
//    (*p_vector)[0] = 10: これはできない。
//    *p1 = *p2; これはOK。

inline TValue operator[]( const size_t index_ )
{
return vector[ index_ ];
}
```

```
// ----------------------------------------------<OK><ﾃｽﾄ済み>
//  instance = CVector

inline void operator=( CVector<TValue>& vector_ )
{
vector = vector_.GetVector();
}

// ----------------------------------------------<OK><ﾃｽﾄ済み>
//  bool result = ( instance == CVector )

// ※要素値の総和が同値ならTRUE。

inline bool operator==( CVector<TValue>& vector_ )
{
return ( vector == vector_.GetVector() );
}

// ----------------------------------------------<OK><ﾃｽﾄ済み>
//  bool result = ( instance != CVector )

inline bool operator!=( CVector<TValue>& vector_ )
{
return ( vector != vector_.GetVector() );
}

// ------------------------------------------------

}; // end class

}; // end namespace

// ------------------------------------------------

#endif

// ------------------------------------------------
```

// std::vector<**bool**> は、特別仕様となっており、以下のﾒｿｯﾄﾞが使用できる。

// vector1.**flip()**;　　// すべての要素の論理値を、反転させる。
// vector1[i].**flip()**;　　// 要素[i]の論理値を、反転させる。
// vector1[i] = true;　　// 要素[i]に、**true**を代入する。
// vector1[i1] = vector1[index2];　// 要素[i1]に、要素[i2]を代入する。

// ・ﾋﾞｯﾄﾌﾗｸﾞを**bool**値として使用しているため、
//　ﾒﾓﾘｻｲｽﾞは、通常の**1/8**のｻｲｽﾞに抑えられている。

【allocator】 (ﾒﾓﾘ確保の最適化)

```
#pragma once

#ifndef _STL_ALLOCATOR_H_
#define _STL_ALLOCATOR_H_



#include<memory>

using namespace std;



namespace STL
{

// ====================================================
// ｱﾛｹｰﾀ - ｸﾗｽの定義 ( std::allocatorの派生ｸﾗｽ )

// ・ﾍﾞｸﾀｰなどのﾒﾓﾘを確保する処理には、汎用化するための処理が含まれている。
//  この部分を上書きして特化すると、ｲﾝｽﾀﾝｽ化時の実行速度が向上する。

// 詳細はこちら↓
//http://msdn.microsoft.com/ja-jp/library/vstudio/5fk3e8ek.aspx

// ※要素が存在しないindexやｲﾃﾚｰﾀｰを渡した場合はｺｹる。
//  ( 速度を優先するため、STL側でも範囲判定はあまり行なっていない )

template<typename T>
classCAllocator : public std::allocator<T>
{

public:

// ----------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ
```

```
CAllocator() : std::allocator<T>::allocator()
{

}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

CAllocator(const STL::CAllocator<T>& a_ ) : std::allocator<T>::allocator( a_ )
{

}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾒﾓﾘを動的に確保する。

// n:     確保する数
// hint:  確保する際のﾋﾝﾄ ( 使用される保証はない )
// 戻り値: 確保した領域の先頭ｱﾄﾞﾚｽ
T* allocate( size_t n, const void* hint = 0 )
{
      return new T[ sizeof( T ) * n ];

      // return (T*) operator new( n * sizeof(T) );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾒﾓﾘを解放する。

   // p: 解放する領域の先頭ﾎﾟｲﾝﾀ
   // n: 解放する領域のｻｲｽﾞ
   void deallocate( T* p, size_t n )
   {
    delete [] p;

       // ::operator delete(p);
   }
```

```
// ----------------------------------------------<OK><ﾃｽﾄ済み>
// 要素数の限界値を返す。 ( ※max_size() で返される値。これを越えてﾘｻｲｽﾞはできない )

size_t max_size() const
{
        return (size_t) -1 / sizeof( T );
}

// ----------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀの初期化処理

    //p:    初期化する領域の先頭ﾎﾟｲﾝﾀ
    //val:  初期値
   void construct( T* p, const T& val )
   {
          new ( (void*) p ) T( val );
   }

//----------------------------------------------<OK><ﾃｽﾄ済み>
// ﾃﾞｽﾄﾗｸﾀを呼ぶ。

   void destroy( T* p )
   {
         (p)->~T();
   }

// ----------------------------------------------

}; // end class

// ================================================

}; // end namespace

// ----------------------------------------------
```

```
#endif
```

【iterator】（STLのﾎﾟｲﾝﾀ）

```
// ------------------------------------
// 「ｲﾃﾚｰﾀ」 を使用したﾙｰﾌﾟ処理。

std::vector< int > vector1;

vector1.push_back( 0 ); // 要素0を追加する。
vector1.push_back( 1 ); // 要素1を追加する。
vector1.push_back( 2 ); // 要素2を追加する。

std::vector< int >::iterator itr = vector1.begin(); // 先頭ｲﾃﾚｰﾀを取得する。
// （「要素[0]のﾎﾟｲﾝﾀ」 だと考えればわかりやすい）

while ( itr != vector1.end() ) // 終端ｲﾃﾚｰﾀになるまで、ﾙｰﾌﾟを続ける。
{
   int value = *itr; // 値を取り出す時の書き方も、ﾎﾟｲﾝﾀと同じ。

    ++itr; // ｲﾃﾚｰﾀをｲﾝｸﾘﾒﾝﾄ(加算)している。
}
// ------------------------------------
// 「逆順ｲﾃﾚｰﾀ」を使用したﾙｰﾌﾟ処理。 （※終端の要素から始まる）

std::vector< int >::reverse_iterator itr = vector1.rbegin(); // 逆順ｲﾃﾚｰﾀを取得する。

while ( itr != vector1.rend() ) // 先頭ｲﾃﾚｰﾀになるまで、ﾙｰﾌﾟを続ける。
{
   int value = *itr; // 値を取り出す時の書き方も、ﾎﾟｲﾝﾀと同じ。

    ++itr; // 逆順ｲﾃﾚｰﾀをｲﾝｸﾘﾒﾝﾄしている。
}

// ------------------------------------
// 「先頭挿入ｲﾃﾚｰﾀ」による挿入処理。 （先頭に挿入(追加)できるのは、listとdequeだけ）

std::list< int > list1;
```

```
std::front_insert_iterator< std::list< int > > itr( list1 ); // 先頭挿入ｲﾃﾚｰﾀを作る。

*itr = 0; // 先頭に、0を挿入する。 →      [0]=0
*itr = 1; // 先頭に、1を挿入する。 →      [0]=1 [1]=0
*itr = 2; // 先頭に、2を挿入する。 →      [0]=2 [1]=1 [2]=0

// ------------------------------------
// ｲﾃﾚｰﾀｰを作らずに使えるﾍﾙﾊﾟｰ関数だと、こうなる。↓

std::front_inserter( list1 ) = 0; // 先頭に、0を挿入する。 →      [0]=0
std::front_inserter( list1 ) = 1; // 先頭に、1を挿入する。 →      [0]=1 [1]=0
std::front_inserter( list1 ) = 2; // 先頭に、2を挿入する。 →      [0]=2 [1]=1 [2]=0



// ------------------------------------
// 「末尾挿入ｲﾃﾚｰﾀ」による挿入処理。
//
//  ( 末尾に挿入(追加)できるのは、vector、list、stack、queue、deque など、
//      append()ﾒｿｯﾄﾞがあるｺﾝﾃﾅｸﾗｽなら、使用出来ます。 )

std::list< int > list1;

std::back_insert_iterator< std::list< int > > itr( list1 ); // 先頭挿入ｲﾃﾚｰﾀを作る。

*itr = 0; // 末尾に、0を挿入する。 →      [0]=0
*itr = 1; // 末尾に、1を挿入する。 →      [0]=0 [1]=1
*itr = 2; // 末尾に、2を挿入する。 →      [0]=0 [1]=1 [2]=2

// ------------------------------------
// ｲﾃﾚｰﾀｰを作らずに使えるﾍﾙﾊﾟｰ関数だと、こうなる。↓

std::back_inserter( list1 ) = 0; // 末尾に、0を挿入する。 →      [0]=0
std::back_inserter( list1 ) = 1; // 末尾に、1を挿入する。 →      [0]=1 [1]=0
std::back_inserter( list1 ) = 2; // 末尾に、2を挿入する。 →      [0]=2 [1]=1 [2]=0

// ------------------------------------
// 「挿入ｲﾃﾚｰﾀ」による挿入処理。
```

```
//
//  ( どこにでも挿入できる。 insert()ﾒｿｯﾄﾞが使えるｺﾝﾃﾅｸﾗｽなら使える。 )

std::list< int > list1;

 // 挿入ｲﾃﾚｰﾀを作る。 ( ｺﾝｽﾄﾗｸﾀの引数2で指定した要素位置に挿入する )
std::insert_iterator< std::list< int > > itr( list1, list1.begin()  );

*itr = 0; // 末尾に、0を挿入する。 →       [0]=0
*itr = 1; // 末尾に、1を挿入する。 →       [0]=0 [1]=1
*itr = 2; // 末尾に、2を挿入する。 →       [0]=0 [1]=1 [2]=2

// ------------------------------------
// ｲﾃﾚｰﾀｰを作らずに使えるﾍﾙﾊﾟｰ関数だと、こうなる。 ↓

std::inserter( list1, list1.end() ) = 0; // 末尾に、0を挿入する。 →       [0]=0
std::inserter( list1, list1.end() ) = 1; // 末尾に、1を挿入する。 →       [0]=1 [1]=0
std::inserter( list1, list1.end() ) = 2; // 末尾に、2を挿入する。 →       [0]=2 [1]=1 [2]=0

// ------------------------------------
// ######################################
// ｲﾃﾚｰﾀの種類
// ######################################
// ------------------------------------
//  【入力ｲﾃﾚｰﾀｰ】

++itr;          // 1歩進める。
T value = *itr; // 要素値を取得する。

// ------------------------------
//  【出力ｲﾃﾚｰﾀｰ】

++itr;          // 1歩進める。
*itr =  value; // 要素値を設定する。

// ------------------------------
//  【前方ｲﾃﾚｰﾀｰ】
```

```
++itr;              // 1歩進める。
T value = *itr; // 要素値を取得する。
*itr =  value;  // 要素値を設定する。

// ------------------------------
//  【双方向ｲﾃﾚｰﾀｰ】

++itr; // 1歩進める。
--itr; // 1歩戻す。
T value = *itr; // 要素値を取得する。
*itr =  value;  // 要素値を設定する。

// ------------------------------
//  【ﾗﾝﾀﾞﾑｱｸｾｽｲﾃﾚｰﾀｰ】

++itr; // 1歩進める。
--itr; // 1歩戻す。
itr += n; // n歩進める。
itr -= n; // n歩戻す。
T value = *itr; // 要素値を取得する。
*itr =  value;   // 要素値を設定する。

// ------------------------------
```

・「vector」 のｲﾃﾚｰﾀｰは、ﾗﾝﾀﾞﾑｱｸｾｽ。
「list」 (双方向ﾘｽﾄ) のｲﾃﾚｰﾀｰは、双方向。
「slist」 (単方向ﾘｽﾄ) だと、前方ｲﾃﾚｰﾀｰ。

【list】（双方向ﾘﾝｸﾄﾞﾘｽﾄ）

```
#pragma once

#ifndef _STL_LIST_H_
#define _STL_LIST_H_



#include <list>
#include <algorithm>
#include <iterator>

using namespace std;



namespace STL
{

// ====================================================
// 双方向ﾘｽﾄｸﾗｽの定義（std::listのﾗｯﾊﾟｰ）

// ・挿入と削除は速いが、ｱｸｾｽは遅い。（先頭ｱｲﾃﾑから順に、ﾘﾝｸをたどってる）
// ・挿入や削除によってindexが移動しても、そのまま使える。（ｲﾃﾚｰﾀは常に同じ要素を指している）

// ※要素が存在しないindexやｲﾃﾚｰﾀｰを渡した場合はｺｹる。
//  （速度を優先するため、STL側でも範囲判定はあまり行なっていない）

template<typename T>
classCList
{



protected: // publicでもOK。

// ----------------------------------------------------
// ﾃﾞｰﾀﾒﾝﾊﾞ
```

```
std::list<T> list; // ﾘｽﾄ本体。


public:

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾘｽﾄ本体の型

    typedef typename std::list<T> InsideType; // ﾘｽﾄ本体の型


// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｲﾃﾚｰﾀｰ

typedef typename std::list<T>::iteratorIterator; // 通常のｲﾃﾚｰﾀｰ (ﾗﾝﾀﾞﾑｱｸｾｽ)

typedef typename std::list<T>::reverse_iterator RevIterator; // 逆順ｲﾃﾚｰﾀｰ


// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

  CList( void )
  {

  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾃﾞｽﾄﾗｸﾀ

  ~CList( void )
  {

  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
```

```
// 【指定値で初期化された指定数の要素】で初期化する。( assign() )

  inline void InitByValue( T value_, const size_t count_ )
  {
        list.assign( count_, value_ );
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定範囲の要素値】で初期化する。        ( assign() )

// dest_list を、src_list[5]、[6]、[7] で初期化する場合は、
//
// dest_list.InitByRange( src_list.GetItr(5), src_list.GetItr(8) );

  inline void InitByRange( Iterator src_start_itr_, Iterator src_end_itr_ )
  {
        list.assign( src_start_itr_, src_end_itr_ );
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾘｽﾄ】を返す。

inline InsideType& GetList( void )
{
        return list;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾘｽﾄ】を格納する。

  inline void SetList( InsideType& list_ )
  {
        list = list_;
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【要素数】を返す。                                    ( size() )
```

```
inline size_t GetCount( void )
{
	return list.size();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【最大要素数】を返す。                                ( max_size() )

// ※「size_t」の最大値。(「size_t」は、32bitｼｽﾃﾑでは「unsigned int」)

inline size_t GetMaxCount( void )
{
	return list.max_size();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【要素数】を変更する。                                ( resize() )

inline void Resize( const size_t length_ )
{
	list.resize( length_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【要素数】を変更し、新規要素を指定値で初期化する。    ( resize() )

inline void ResizeEx( const size_t length_, T value_ )
{
	 list.resize( length_, value_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【最初の要素の値】を返す。                            ( front() )

inline T GetFirstValue( void )
{
```

```
        return list.front();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【最後の要素の値】を返す。                                ( back() )

inline T GetLastValue( void )
{
        return list.back();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexのｲﾃﾚｰﾀ】を返す。                              ( advance() )

inline Iterator GetItr( const size_t index_ )
{

    std::list<T>::iterator itr = list.begin();

std::advance( itr, index_ ); // ｲﾃﾚｰﾀｰに指定indexを加算する。

return itr;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexの要素値】を返す。                            ( advance() )

inline T GetValue( const size_t index_ )
{

    std::list<T>::iterator itr = list.begin();

std::advance( itr, index_ ); // ｲﾃﾚｰﾀｰに指定indexを加算する。

return *itr;
}
```

```
// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexの要素値】を変更する。                    ( advance() )

inline void SetValue( const size_t index_, T value_ )
{

      std::list<T>::iterator itr = list.begin();

std::advance( itr, index_ ); // ｲﾃﾚｰﾀｰに指定indexを加算する。

*itr = value_;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【要素】を先頭に追加する。                          ( push_front() )

inline void AddFirst( T value_ )
{
return list.push_front( value_ ); // 追加する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【要素】を末尾に追加する。                          ( push_back() )

inline void AddLast( T value_ )
{
return list.push_back( value_ ); // 追加する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定ｲﾃﾚｰﾀの要素】を、【別のﾘｽﾄ】に置換する。        ( splice() )

inline void ReplaceByList( Iterator dest_itr_, STL::CList<T>& src_list_ )
{
list.splice( dest_itr_, src_list_.GetList() ); // 挿入する。
}
```

```
// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定ｲﾃﾚｰﾀの要素】 を、 【別のﾘｽﾄの指定ｲﾃﾚｰﾀの指す要素】 に置換する。 ( splice() )

// dest_list[1] に src_list[5] を挿入する場合は、
//
// dest_list.ReplaceByListItem( dest_list.GetItr(1), src_list, src_list.GetItr(5) );

inline void ReplaceByListItem( Iterator dest_itr_, STL::CList<T>& src_list_, Iterator src_itr_ )
{
list.splice( dest_itr_, src_list_.GetList(), src_itr_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定ｲﾃﾚｰﾀの要素】 を、 【別のﾘｽﾄ中の指定範囲の要素】 に置換する。 ( splice() )

// ・ｺﾋﾟｰ元のﾘｽﾄは、 同じﾘｽﾄでもいい。

// list[1]に、[5][6][7]を挿入する場合は、
//
// list2.ReplaceByRange( list2.GetItr(1), list1, list1.GetItr(5), list1.GetItr(8) );

inline void ReplaceByRange(
                        Iterator dest_itr_,
    STL::CList<T>& src_list_,
                                Iterator src_start_itr_,
    Iterator src_end_itr_
 )
{
list.splice( dest_itr_, src_list_.GetList(), src_start_itr_, src_end_itr_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定ｲﾃﾚｰﾀｰの要素】 に、 【要素】 を挿入し、 【挿入位置のｲﾃﾚｰﾀｰ】 を返す。 ( insert() )

// ※逆順ｲﾃﾚｰﾀは受け付けない。

inline Iterator Insert( Iterator itr_, T value_ )
```

```
{
return list.insert( itr_, value_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定ｲﾃﾚｰﾀｰの要素】 に、 【指定数の要素】 を挿入する。 ( insert() )

// ※逆順ｲﾃﾚｰﾀは受け付けない。

inline void InsertByValue( Iterator itr_, T value_, const size_t count_ )
{
list.insert( itr_, count_, value_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 指定indexに 【指定範囲の要素】 を挿入する。              ( insert() )

// ※逆順ｲﾃﾚｰﾀは受け付けない。

// list1 の [1] に、 list2 の [5]、[6]、[7] を挿入する場合は、
//
// list1.InsertByRange( list1.GetItr(1), list2.GetItr(5), list2.GetItr(8) );

inline void InsertByRange( Iterator dest_itr_, Iterator src_start_itr_, Iterator src_end_itr_ )
{
list.insert( dest_itr_, src_start_itr_, src_end_itr_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定index】 に 【要素】 を挿入し、 【挿入位置のｲﾃﾚｰﾀｰ】 を返す。 ( insert() )

inline Iterator InsertAt( const size_t index_, T value_ )
{

        std::list<T>::iterator itr = list.begin();

std::advance( itr, index_ ); // ｲﾃﾚｰﾀｰに指定indexを加算する。
```

```
return list.insert( itr, value_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定ｲﾃﾚｰﾀ-の指す要素】を削除し、【その次の要素のｲﾃﾚｰﾀｰ】を返す。( erase() )

// ※逆順ｲﾃﾚｰﾀは受け付けない。

inline Iterator Remove( Iterator itr_ )
{
        return list.erase( itr_ ); // 削除する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定範囲の要素】を削除し、【その次の要素のｲﾃﾚｰﾀｰ】を返す。( erase() )

// ※逆順ｲﾃﾚｰﾀは受け付けない。

// list1 の [5]、[6]、[7] を削除する場合は、
//
// list1.RemoveByRange( list1.GetItr(5), list1.GetItr(8) );

inline Iterator RemoveByRange( Iterator start_itr_, Iterator end_itr_ )
{
        return list.erase( start_itr_, end_itr_ ); // 削除する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【要素】を削除し、【その次の要素のｲﾃﾚｰﾀｰ】を返す。    ( erase() )

inline Iterator RemoveAt( const size_t index_ )
{

    std::list<T>::iterator itr = list.begin();

std::advance( itr, index_ ); // ｲﾃﾚｰﾀｰに指定indexを加算する。
```

```
		return list.erase( itr ); // 削除する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最初の要素】を削除する。                          ( pop_front() )

inline void RemoveFirst( void )
{
		list.pop_front();
}
// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最後の要素】を削除する。                          ( pop_back() )

inline void RemoveLast( void )
{
		list.pop_back();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｸﾘｱする。                                          ( clear() )

inline void Clear( void )
{
		list.clear();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 昇順にｿｰﾄする。 ( C標準ﾗｲﾌﾞﾗﾘのqsort関数よりも速い ) ( sort() )

inline void Sort( void )
{
		list.sort();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 降順(逆順)にｿｰﾄする。                               ( sort() )
```

```
inline void Sort_Reverse( void )
{
	list.sort( std::greater<T>( ) );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【並び順】を逆にする。                                    ( reverse() )

inline void Reverse( void )
{
		list.reverse();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【値が重複する要素】を削除する。                          ( unique() )

// ※重複する値は、連続していなければいけない。( つまり、事前にｿｰﾄしておかないといけない )

inline void Unique( void )
{
		 list.unique();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【別のﾘｽﾄ】を混ぜてｿｰﾄする。                              ( merge() )

// ※両方の要素値がｿｰﾄされていなければならない。( 逆順ではﾀﾞﾒ )

inline void Merge( STL::CList<T>& list_ )
{
		list.merge( list_.GetList() );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【最初の要素のｲﾃﾚｰﾀ】を返す。                              ( begin() )
```

```
inline Iterator GetFirstItr(void)
{
        return list.begin();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最後の要素のｲﾃﾚｰﾀ】 を返す。                          ( end() )

inline Iterator GetLastItr(void)
{
        return list.end();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 逆順ﾙｰﾌﾟ時の 【最初の要素([count-1])のｲﾃﾚｰﾀ】 を返す。( rbegin() )

inline RevIterator GetFirstItr_Rev(void)
{
        return list.rbegin();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 逆順ﾙｰﾌﾟ時の 【最後の要素([0])のｲﾃﾚｰﾀ】 を返す。      ( rend() )

inline RevIterator GetLastItr_Rev(void)
{
         return list.rend();
}

// ------------------------------------------------
// ################################################
// 演算子のｵｰﾊﾞｰﾛｰﾄﾞ
// ################################################
// ----------------------------------------------<OK><ﾃｽﾄ済み>
//  T value = instance[ index_ ]

// begin_itr++; // list[0] 加算されていない。
```

```
// ++begin_itr; // list[1] 加算されている。

        // itr += 1;    // 構文ｴﾗｰ。
        // *( itr + 1); // 構文ｴﾗｰ。

// というわけで、 advance()でｲﾝｸﾘﾒﾝﾄするしか無い。

// int value = list[i]; // 値は取れるが、
// list[i] = 10;         // 格納はできない。

inline T operator[]( const size_t index_ )
{

    std::list<T>::iterator itr = list.begin();

std::advance( itr, index_ ); // ｲﾃﾚｰﾀｰに指定indexを加算する。

return *itr;
}

 // ------------------------------------------------<OK><ﾃｽﾄ済み>
 //  instance = CList

inline void operator=( CList<T>& list_ )
{
list = list_.GetList();
}

 // ------------------------------------------------<OK><ﾃｽﾄ済み>
 //  bool result = ( instance == CList )

inline bool operator==( CList<T>& list_ )
{
return ( list == list_.GetList() );
}

 // ------------------------------------------------<OK><ﾃｽﾄ済み>
```

```
 //  bool result = ( instance != CList )

inline bool operator!=( CList<T>& list_ )
{
return ( list != list_.GetList() );
}

// ------------------------------------------------------

}; // end class

}; // end namespace

// ------------------------------------------------------


#endif
```

【hash_map】 (ﾊｯｼｭﾃｰﾌﾞﾙ)

```
#pragma once

#ifndef _STL_HASH_MAP_H_
#define _STL_HASH_MAP_H_


#include <hash_map>

using namespace std;
using namespace stdext;


namespace STL
{

// ======================================================
// ﾊｯｼｭﾏｯﾌﾟｸﾗｽの定義 ( hash_mapのﾗｯﾊﾟｰ )

// ・ﾊｯｼｭｱﾙｺﾞﾘｽﾞﾑを使用しているため、std::map よりも実行速度が速い。

template<typename TKey,typename TValue>
class  CHashMap
{


protected: // publicでもOK。


// ------------------------------------------------
// ﾒﾝﾊﾞ

hash_map<TKey,TValue> hash; // ﾊｯｼｭﾏｯﾌﾟ本体。


public:
```

```
// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾊｯｼｭﾏｯﾌﾟ本体の型

    typedef typename hash_map<TKey,TValue> InsideType; // ﾊｯｼｭﾏｯﾌﾟ本体の型

    typedef typename std::pair<TKey,TValue> KeyPairType; // ｷｰﾍﾟｱの型

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｲﾃﾚｰﾀｰ

typedef typename hash_map<TKey,TValue>::iterator Iterator; // 通常のｲﾃﾚｰﾀｰ (ﾗﾝﾀﾞﾑｱｸｾｽ)

typedef typename hash_map<TKey,TValue>::reverse_iterator RevIterator; // 逆順ｲﾃﾚｰﾀｰ

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

  CHashMap( void )
  {

  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾃﾞｽﾄﾗｸﾀ

~CHashMap( void )
{

}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定ｷｰに対応する要素のｲﾃﾚｰﾀｰ】を返す。   ( find() )

// STL::CHashMap<std::string,int>::Iterator itr = hash1.GetItr( std::string("BBB") );

inline Iterator GetItr( TKey& key_ )
```

```
        {

return  hash.find( key_ );

// if ( itr == hash.end() ) // 見つからなかった。

}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾊｯｼｭﾏｯﾌﾟ】 を返す。

inline InsideType& GetHashMap( void )
{
            return hash;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾊｯｼｭﾏｯﾌﾟ】 を格納する。

  inline void SetHashMap( InsideType& hash_ )
  {
        hash = hash_;
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【要素数】 を返す。                                    ( size() )

inline size_t GetCount( void )
{
            return hash.size();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定ｷｰに対応する要素数】 を返す。                    ( count() )

inline size_t GetCountByKey( TKey key_ )
{
```

```
        return hash.count( key_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【最大要素数】を返す。                                ( max_size() )

// ※「size_t」の最大値。(「size_t」は、32bitｼｽﾃﾑでは「unsigned int」)

inline  size_t GetMaxCount( void )
{
        return hash.max_size();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定したｷｰ】が存在するなら、TRUEを返す。              ( find() )

inline bool ContainsKey( TKey key_ )
{
        return ( hash.find( key_ ) != hash.end() );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定ｷｰに対応する値】を返す。          ( 値 = hash_map[ key ] )

inline TValue GetValue( TKey key_ )
{
return hash[ key_ ];

// ※存在しないｷｰを指定すると、ﾃﾞﾌｫﾙﾄ値で初期化されたｷｰﾍﾟｱが追加され、その値が返る。

// ・hash.at( key_ ) の場合は、未登録のｷｰに対して、out_of_range例外をｽﾛｰする。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定ｷｰに対応する値】を変更する。      ( hash_map[key] = 値 )

inline void SetValue( TKey key_, TValue value_ )
```

```
{
 hash[ key_ ] = value_;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【ｷｰﾍﾟｱ】を変更する。 (無い場合は、追加される)          ( insert() )

inline void Set( TKey key_, TValue value_ )
{
hash_map<TKey,TValue>::iterator itr = hash.find( key_ ); // 一致するｷｰが有るか、探す。

if ( itr != hash.end() ) // ｷｰが登録済みなら、
{
itr->second = value_; // 値を入れ替える。
}
else // ｷｰが未登録なら、
{
        hash.insert( hash_map<TKey,TValue>::value_type( key_, value_ ) ); // 登録する。
}
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【ｷｰﾍﾟｱ】を追加する。                                  ( insert() )

// hash1.Add( std::string("AAA"), 1000 );

inline void Add( TKey key_, TValue value_ )
{
    hash.insert( hash_map<TKey,TValue>::value_type( key_, value_ ) ); // 登録する。

    // ・暗黙的に行われるﾀｲﾌﾟｷｬｽﾄを回避するために、value_typeを使っている。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【ｷｰﾍﾟｱ】を削除する。                                  ( erace() )

// ・存在しないｷｰが渡されてもｺｹない。
```

```
inline void RemoveAt( TKey key_ )
{
        hash.erase( key_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｸﾘｱする。                                          ( clear() )

inline void Clear( void )
{
        hash.clear();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【要素数】が0ならTRUEを返す。                        ( empty() )

inline bool IsEmpty( void )
{
        return hash.empty();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最初の要素のｲﾃﾚｰﾀ】を返す。                        ( begin() )

inline Iterator GetFirstItr( void )
{
        return hash.begin();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最後の要素のｲﾃﾚｰﾀ】を返す。                        ( end() )

inline Iterator GetLastItr( void )
{
        return hash.end();
}
```

```
// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最初の要素の逆順ｲﾃﾚｰﾀ】を返す。                    ( rbegin() )

inline RevIterator GetFirstItr_Rev( void )
{
        return hash.rbegin();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最後の要素の逆順ｲﾃﾚｰﾀ】を返す。                    ( rend() )

inline RevIterator GetLastItr_Rev( void )
{
        return hash.rend();
}

 // ------------------------------------------------
 // ################################################
 // 演算子のｵｰﾊﾞｰﾛｰﾄﾞ
 // ################################################
 // ------------------------------------------------<OK><ﾃｽﾄ済み>
 //  TValue value = instance[ key_ ]

inline TValue operator[]( TKey key_ )
    {
return hash[ key_ ];
    }

 // ------------------------------------------------<OK><ﾃｽﾄ済み>
 //  instance = CHashMap

inline void operator=( CHashMap<TKey,TValue>& hash_ )
    {
hash = hash_.GetHashMap();
    }
```

```
 // ------------------------------------------------<OK><ﾃｽﾄ済み>
 //  bool result = ( instance == CHashMap )

inline bool operator==( CHashMap<TKey,TValue>& hash_ )
	{
return ( hash == hash_.GetHashMap() );
	}

 // ------------------------------------------------<OK><ﾃｽﾄ済み>
 //  bool result = ( instance != CHashMap )

inline bool operator!=( CHashMap<TKey,TValue>& hash_ )
	{
return ( hash != hash_.GetHashMap() );
	}

// ------------------------------------------------

}; // end class

}; // end namespace

// ------------------------------------------------

#endif
```

【stack】(積み上げるｺﾝﾃﾅ)

```
#pragma once

#ifndef _STL_STACK_H_
#define _STL_STACK_H_


#include <stack>


using namespace std;


namespace STL
{

// ====================================================
// ｽﾀｯｸｸﾗｽの定義 ( std::stackのﾗｯﾊﾟｰ )

// ・ｽﾀｯｸは、要素を積み上げていくｺﾝﾃﾅ。
// ・最上段の要素に対して、Add()ﾒｿｯﾄﾞで要素を積み、Remove()で削除する。
// ・値も、最上段の要素のものしか取得できない。( GetValue()で取得する )

//  [2] add →       [1][0]
//  [2] ← get&remove [1][0]

template<typename T>
class CStack
{


protected: // publicでもOK。


// ----------------------------------------------------
// ﾃﾞｰﾀﾒﾝﾊﾞ
```

```
std::stack<T> stack; // ｽﾀｯｸ 本体。


public:

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｽﾀｯｸ本体の型

    typedef typename std::stack<T> InsideType; // ｽﾀｯｸ本体の型


// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

  CStack( void )
  {

  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾃﾞｽﾄﾗｸﾀ

~CStack( void )
{

}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ｽﾀｯｸ】 を返す。

  inline InsideType& GetStack( void )
  {
        return stack;
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
```

```
// 【ｽﾀｯｸ】 を格納する。

  inline void SetStack( InsideType& stack_ )
  {
        stack = stack_;
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【要素数】 を返す。                                    ( size() )

inline size_t GetCount( void )
{
        return stack.size();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【要素数】 が0ならTRUEを返す。                          ( empty() )

inline bool IsEmpty( void )
{
        return stack.empty();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 最上段に 【要素】 を追加する。                          ( push() )

// [3] add→  [2][1][0]

inline void Add( T value_ )
{
        stack.push( value_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最上段の要素】 を削除する。                          ( pop() )

// [2] ←remove [1][0]
```

```
inline void Remove( void )
{
	stack.pop();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最上段の要素値】を返す。                                ( top() )

// [2] ←get    [2][1][0]

inline T GetValue( void )
{
	return stack.top();
}

// ----------------------------------------------------

}; // end class

}; // end namespace

// ----------------------------------------------------

#endif
```

【queue】 (ﾄﾝﾈﾙに詰め込むｺﾝﾃﾅ)

```
#pragma once

#ifndef _STL_QUEUE_H_
#define _STL_QUEUE_H_


#include <queue>

using namespace std;


namespace STL
{

// ======================================================
// ｷｭｰｸﾗｽの定義 ( std::queueのﾗｯﾊﾟｰ )

// ・ｷｭｰは、ﾄﾝﾈﾙに車を詰めていくﾀｲﾌﾟのｺﾝﾃﾅ。
// ・新しい要素を末尾に追加し、先頭の要素を取り出しては削除していく。

//  [3] add→ [2][1] remove→ [0]

// push ( 押しこむ ) → pop ( 取り出す )

template<typename T>
class CQueue
{


protected: // publicでもOK。


// ------------------------------------------------
// ﾃﾞｰﾀﾒﾝﾊﾞ
```

```
std::queue<T> queue; // ｷｭｰ 本体。


public:


// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｷｭｰ本体の型

    typedef typename std::queue<T> InsideType; // ｷｭｰ本体の型


// ---------------------------------------------------
// ｺﾝｽﾄﾗｸﾀ

   CQueue( void )
   {

   }

// ---------------------------------------------------
// ﾃﾞｽﾄﾗｸﾀ

   ~CQueue( void )
   {

   }

// ---------------------------------------------------
// 【ｷｭｰ】 を返す。

inline InsideType& GetQueue( void )
{
            return queue;
}

// ---------------------------------------------------
```

```
// 【ｷｭｰ】 を格納する。

  inline void SetQueue( InsideType& queue_ )
  {
        queue = queue_;
  }

// -----------------------------------------------------
// 【要素数】 を返す。                                    ( size() )

inline size_t GetCount( void )
{
            return queue.size();
}

// -----------------------------------------------------
// 【要素数】 が0ならTRUEを返す。                          ( empty() )

inline bool IsEmpty( void )
{
             return queue.empty();
}

// -----------------------------------------------------
// 末尾に 【要素】 を追加する。                            ( push() )

// [3] add→ [2][1][0]

inline void Add( T value_ )
{
             queue.push( value_ );
}

// -----------------------------------------------------
// 先頭の 【要素】 を削除する。( 値を処理した後、削除する )   ( pop() )

// [2][1] remove→ [0]
```

```
inline void Remove( void )
{
        queue.pop();
}

// ----------------------------------------------------
// 【要素】を返す。( 削除はしない )                    ( front() )

// [2][1][0] get→ [0]

inline T GetFrontValue( void )
{
        return queue.front();
}

// ----------------------------------------------------
// 終端の【要素】を返す。( 取り出して削除はしない )      ( back() )

// [2] ←get [2][1][0]

inline T GetBackValue( void )
{
        return queue.back();
}

// ----------------------------------------------------

}; // end class

}; // end namespace

// ----------------------------------------------------


#endif
```

【deque】(両方から詰め込むｷｭｰ)

```
#pragma once

#ifndef _STL_DEQUE_H_
#define _STL_DEQUE_H_


// ｱﾛｹｰﾀｸﾗｽのﾍｯﾀﾞﾌｧｲﾙをｲﾝｸﾙｰﾄﾞしておく。
#include "STL_Allocator.h"

#include <deque>

using namespace std;


namespace STL
{

// ------------------------------------------------
//【使用例】

// ================================================
// ﾃﾞｷｭｰｸﾗｽの定義 ( std::dequeのﾗｯﾊﾟｰ )

//・ｷｭｰとの違いは、両端に値を追加できること。
//・ﾒｿｯﾄﾞの構成は、std::vector とほぼ同じ。( ｷｬﾊﾟｼﾃｨの取得･設定ができない )

//※追加、削除、ﾘｻｲｽﾞなどを行うと、要素のｱﾄﾞﾚｽが変更されるため、以前のｲﾃﾚｰﾀが使用できなくなる。


template<typename TValue>
class CDeque
{
```

```
protected: // publicでもOK。

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾒﾝﾊﾞ

std::deque<TValue,CAllocator<TValue>> deque; // ﾃﾞｷｭｰ本体。


public:

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾃﾞｷｭｰ本体の型

    typedef typename std::deque<TValue,CAllocator<TValue>> InsideType; // ﾃﾞｷｭｰ本体の型


// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｲﾃﾚｰﾀｰ

        // 通常のｲﾃﾚｰﾀｰ (ﾗﾝﾀﾞﾑｱｸｾｽ)
typedef typename std::deque<TValue,CAllocator<TValue>>::iterator Iterator;

       // 逆順ｲﾃﾚｰﾀｰ (ﾗﾝﾀﾞﾑｱｸｾｽ)
typedef typename std::deque<TValue,CAllocator<TValue>>::reverse_iterator RevIterator;


// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

CDeque( void )
{

}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾃﾞｽﾄﾗｸﾀ
```

```
~CDeque(void )
{

}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定値で初期化された指定数の要素】で初期化する。( assign() )

  inline void InitByValue( TValue value_, const size_t count_ )
  {
        deque.assign( count_, value_ );
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定範囲の要素値】で初期化する。        ( assign() )

// dest_deque を、src_deque[5]、[6]、[7] で初期化する場合は、
//
// dest_deque.InitRange( src_deque.GetItr(5), src_deque.GetItr(8) );

  inline void InitByRange( Iterator src_start_itr_, Iterator src_end_itr_ )
  {
        deque.assign( src_start_itr_, src_end_itr_ );
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexのｲﾃﾚｰﾀ】を返す。

// ※戻り値のｲﾃﾚｰﾀを、参照にしていないのには理由がある。
//   このｸﾗｽがﾛｰｶﾙ変数として宣言された場合、
//   ｽﾀｯｸ領域のｱﾄﾞﾚｽを返してしまうため、ｺｹるからである。

inline Iterator GetItr( const size_t index_ )
{
return ( deque.begin() + index_ );
}
```

```
// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【ﾃﾞｷｭｰ】を返す。

inline InsideType& GetDeque( void )
{
        return deque;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【ﾃﾞｷｭｰ】を格納する。

inline void SetDeque( InsideType& deque_ )
{
        deque = deque_;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【要素数】を返す。                                    ( size() )

inline size_t GetCount( void )
{
        return deque.size();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【最大要素数】を返す。                                ( max_size() )

// ※「size_t」の最大値。(「size_t」は、32bitｼｽﾃﾑでは「unsigned int」)

inline size_t GetMaxCount( void )
{
        return deque.max_size();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【要素数】を変更する。                                ( resize() )
```

```
// ・新しく追加された要素の値は、0ｸﾘｱされている。

inline void Resize( const size_t length_ )
{
        deque.resize( length_ );
}

// ------------------------------------------------------<OK><ﾃｽﾄ済み>
//【要素数】を変更し、【追加された要素】を、【指定値】で初期化する。( resize() )

inline void ResizeEx( const unsigned int length_, TValue value_ )
{
        deque.resize( length_, value_ );
}

// ------------------------------------------------------<OK><ﾃｽﾄ済み>
//【最初の要素の値】を返す。                              ( front() )

inline TValue GetFirstValue( void )
{
        return deque.front();
}

// ------------------------------------------------------<OK><ﾃｽﾄ済み>
//【最後の要素の値】を返す。                              ( back() )

inline TValue GetLastValue( void )
{
        return deque.back();
}

// ------------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexの要素値】を返す。                          ( 値 = deque[i] )

inline TValue GetValue( const size_t index_ )
{
return deque[ index_ ];
```

```
// deque.at( index_ ) の場合は、範囲外のindexに対して、out_of_range例外をｽﾛｰする。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexに要素値】を格納する。                    ( deque[i] = 値 )

inline void SetValue( const size_t index_, TValue value_ )
{
        deque[ index_ ] = value_;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【要素】を末尾に追加する。                           ( push_back() )

inline void Add( TValue value_ )
{
return deque.push_back( value_ ); // 追加する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定ｲﾃﾚｰﾀｰの要素】に、【要素】を挿入し、【挿入位置のｲﾃﾚｰﾀｰ】を返す。( insert() )

// ※指定したｲﾃﾚｰﾀｰの要素に、指定値が格納される。
// ※指定するindexは、AddやResizeで拡張された範囲を超えるとｺｹる。

// ※逆順ｲﾃﾚｰﾀは受け付けない。

inline Iterator Insert( Iterator itr_, TValue value_ )
{
return deque.insert( itr_, value_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定ｲﾃﾚｰﾀｰの要素】に、【指定数の要素】を挿入する。( insert() )

// ※逆順ｲﾃﾚｰﾀは受け付けない。
```

```
inline void InsertByValue( Iterator itr_, TValue value_, const size_t count_ )
{
deque.insert( itr_, count_, value_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定index】に、【指定範囲の要素】を挿入する。( insert() )

// ※逆順ｲﾃﾚｰﾀは受け付けない。

// deque1 の [1] に、deque2 の [5]、[6]、[7] を挿入する場合は、
//
// deque1.InsertByRange( deque1.GetItr(1), deque2.GetItr(5), deque2.GetItr(8) );

inline void InsertByRange( Iterator dest_itr_, Iterator src_start_itr_, Iterator src_end_itr_ )
{
deque.insert( dest_itr_, src_start_itr_, src_end_itr_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定index】に、【要素】を挿入し、【挿入位置のｲﾃﾚｰﾀｰ】を返す。( insert() )

// ※指定したindexの要素に、指定値が格納される。
// ※指定するindexは、AddやResizeで拡張された範囲を超えるとｺｹる。

inline Iterator InsertAt( const size_t index_, TValue value_ )
{
return deque.insert( deque.begin() + index_, value_ ); // 挿入する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定ｲﾃﾚｰﾀ-の指す要素】を削除し、【その次の要素のｲﾃﾚｰﾀｰ】を返す。( erase() )

inline Iterator Remove( Iterator itr_ )
{
        return deque.erase( itr_ );
```

```
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定範囲の要素】を削除し、【その次の要素のｲﾃﾚｰﾀｰ】を返す。( erase() )

// ※逆順ｲﾃﾚｰﾀは受け付けない。

// deque1 の [5]、[6]、[7] を削除する場合は、
//
// deque1.RemoveByRange( deque1.GetItr(5), deque1.GetItr(8) );

inline Iterator RemoveByRange( Iterator start_itr_, Iterator end_itr_ )
{
        return deque.erase( start_itr_, end_itr_ ); // 削除する。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexの要素】を削除する。                    ( erase() )

inline Iterator RemoveAt( const size_t index_ )
{
        return deque.erase( deque.begin() + index_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【最後の要素】を削除する。                         ( pop_back() )

inline void RemoveLast( void )
{
        deque.pop_back();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｸﾘｱする。                                         ( clear() )

inline void Clear( void )
{
```

```
        deque.clear();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最初の要素のｲﾃﾚｰﾀ】を返す。                    ( begin() )

// int value = *itr; // ふつうに最初の要素の値が返る。

inline Iterator GetFirstItr( void )
{
        return deque.begin();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【最後の要素のｲﾃﾚｰﾀ】を返す。                      ( end() )

// int value = *itr;    // ※範囲外のﾎﾟｲﾝﾀなのでｱｸｾｽ違反ｴﾗｰが出る。
// int value = *(itr-1); // ※これが最後の要素の値。( なぜか逆方向に進めてしまう )

inline Iterator GetLastItr( void )
{
        return deque.end();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 逆順ﾙｰﾌﾟ時の、【最初の要素([count-1])のｲﾃﾚｰﾀ】を返す。( rbegin() )

// for ( RevIterator itr = deque1.rbegin() ; itr != deque1.rend(); itr++ )
// {
    //      int value = *(itr);
// }

// 要素数が3の場合、
//
// int value = *( itr );     // 要素[2]の値が返る。
// int value = *( itr - 1 ); // 要素[1]の値が返る。
// int value = *( itr - 2 ); // 要素[0]の値が返る。
```

```
inline RevIterator GetFirstItr_Rev( void )
{
        return deque.rbegin();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 逆順ﾙｰﾌﾟ時の、【最後の要素([0])のｲﾃﾚｰﾀ】を返す。    ( rend() )

// 要素数が3の場合、
//
// int value = *( itr - 1 ); // 要素[0]の値が返る。
// int value = *( itr - 2 ); // 要素[1]の値が返る。
// int value = *( itr - 3 ); // 要素[2]の値が返る。

    // ※ *itr は構文ｴﾗｰ。
// ※ *(itr) は範囲外のﾎﾟｲﾝﾀなのでｱｸｾｽ違反ｴﾗｰが出る。

inline RevIterator GetLastItr_Rev( void )
{
        return deque.rend();
}

// ------------------------------------------------
// ################################################
// 演算子のｵｰﾊﾞｰﾛｰﾄﾞ
// ################################################
// ------------------------------------------------<OK><ﾃｽﾄ済み>
//  TValue value = instance[ index_ ]

// ※ int v1 = (*p_deque)[0];  これはできるが、(*p_deque)[0] = 10: これはできない。  *p1 = *p2; これはOK。

inline TValue operator[]( const size_t index_ )
{
return deque[ index_ ];
}
```

```
// ------------------------------------------------<OK><ﾃｽﾄ済み>
//  instance = CDeque

inline void operator=( CDeque<TValue>& deque_ )
{
deque = deque_.GetDeque();
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//  bool result = ( instance == CDeque )

// ※要素値の総和が同値ならTRUE。

inline bool operator==( CDeque<TValue>& deque_ )
{
return ( deque == deque_.GetDeque() );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//  bool result = ( instance != CDeque )

inline bool operator!=( CDeque<TValue>& deque_ )
{
return ( deque != deque_.GetDeque() );
}

// ------------------------------------------------

}; // end class

}; // end namespace

// ------------------------------------------------


#endif
```

【auto_ptr】 (自動解放ﾎﾟｲﾝﾀ)

```
#pragma once

#ifndef _STL_AUTO_PTR_H_
#define _STL_AUTO_PTR_H_



#include <memory>

using namespace std;



namespace STL
{

// ======================================================
// 自動ﾎﾟｲﾝﾀｸﾗｽの定義 ( std::auto_ptrのﾗｯﾊﾟｰ )

// ・ｺﾝｽﾄﾗｸﾀで渡したﾎﾟｲﾝﾀを、自動的に解放してくれる。      std::auto_ptr<CObject> ap1( new CObject() );
// ・->演算子(上書き)を使って、ﾎﾟｲﾝﾀのﾒﾝﾊﾞにｱｸｾｽできる。  CObject* p_obj = ap1->GetValue();

// ※配列のﾎﾟｲﾝﾀは、格納できない。
// ※同じﾎﾟｲﾝﾀを、複数のｵｰﾄﾎﾟｲﾝﾀに格納することはできない。
// ※ｵｰﾄﾎﾟｲﾝﾀを、ｺﾝﾃﾅの要素にはできない。

// std::auto_ptr     ... ﾛｰｶﾙ変数として使用する場合は軽くていい。
// boost::shared_ptr ... ｺﾝﾃﾅを使う場合や関数に渡す場合に使用する。

// ※要素が存在しないindexやｲﾃﾚｰﾀｰを渡した場合はｺｹる。
//  ( 速度を優先するため、STL側でも範囲判定はあまり行なっていない )

template<typename T>
class CAutoPtr
{
```

```
protected: // publicでもOK。

// ------------------------------------------------
// ﾃﾞｰﾀﾒﾝﾊﾞ

std::auto_ptr<T> ptr; // 自動ﾎﾟｲﾝﾀ本体。

public:

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// ｵｰﾄﾎﾟｲﾝﾀ本体の型

    typedef typename std::auto_ptr<T> InsideType; // ｵｰﾄﾎﾟｲﾝﾀ本体の型


// ------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

CAutoPtr() : ptr()
{

}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

CAutoPtr( T* p_ ) : ptr( p_ )
{

}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾃﾞｽﾄﾗｸﾀ

~CAutoPtr( void )
{
```

```
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ｵｰﾄﾎﾟｲﾝﾀ】 を返す。

inline InsideType& GetAutoPtr( void )
{
        return ptr;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ｵｰﾄﾎﾟｲﾝﾀ】 を格納する。

inline void SetAutoPtr( InsideType& ptr_ )
{
        ptr = ptr_;
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾎﾟｲﾝﾀ】 を返す。                                    ( get() )

inline T* GetPtr( void )
{
        return ptr.get();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾎﾟｲﾝﾀ】 を変更する。                                ( reset() )

inline void SetPtr( T* p_ = NULL )
{
        ptr.reset( p_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾎﾟｲﾝﾀ】 を明示的に解放する。                        ( release() )
```

```
inline void Release( void )
{
        ptr.release();
}

// ---------------------------------------------
// #############################################
// 演算子のｵｰﾊﾞｰﾛｰﾄﾞ
// #############################################
// ---------------------------------------------<OK><ﾃｽﾄ済み>
//  T* p = ap1.get(); // auto_ptrにも -> がｵｰﾊﾞｰﾛｰﾄﾞされているので、
//                    // それを使ってﾎﾟｲﾝﾀのﾒﾝﾊﾞにｱｸｾｽすることもできる。

inline T* operator->()
{
return ptr.get();
}

// ---------------------------------------------<OK><ﾃｽﾄ済み>
//  T value = *ap1; // ﾎﾟｲﾝﾀの値を取り出す。

inline T operator*()
{
return *ptr;
}

// ---------------------------------------------<OK><ﾃｽﾄ済み>
//  instance = ap2

inline void operator=( CAutoPtr<T>& ptr_ )
{
ptr = ptr_.GetAutoPtr();
}

// -------------------------------------------------
```

```
}; // end class

}; // end namespace

// ------------------------------------------------

#endif
```

【**wstring**】 (ﾜｲﾄﾞ文字配列) **1**

```
#pragma once

#ifndef _STL_STRINGW_H_
#define _STL_STRINGW_H_

#include <string>

using namespace std;

namespace STL
{

// ---------------------------------------------------------------------
// NotFound定数 ( 検索した際に、条件に一致する位置が見つからなかった時に返る値。)

#define NotFound 4294967295 // std::wstring::npos

 // ------------------------------------------------------<OK><ﾃｽﾄ済み>
// ｲﾃﾚｰﾀｰ

typedef std::wstring::iterator Iterator; // 通常のｲﾃﾚｰﾀｰ (ﾗﾝﾀﾞﾑｱｸｾｽ)
 typedef std::wstring::const_iterator ConstIterator; // 固定ｲﾃﾚｰﾀｰ
 typedef std::wstring::reverse_iterator RevIterator; // 逆順ｲﾃﾚｰﾀｰ (ﾗﾝﾀﾞﾑｱｸｾｽ)
  typedef std::wstring::const_reverse_iterator ConstRevIterator; // 逆順固定ｲﾃﾚｰﾀｰ
 // ---------------------------------------------------------------------
// ﾜｲﾄﾞ文字列ｸﾗｽの定義 ( std::wstringのﾗｯﾊﾟｰ )

class CStringW
 {

// ------------------------------------------------
// ﾃﾞｰﾀﾒﾝﾊﾞ

  protected:
```

```
 std::wstring text; // ﾜｲﾄﾞ文字列の本体。

 // ﾏﾙﾁﾊﾞｲﾄを使う場合は、ここを std::string にする。

  public:

// ------------------------------------------------
// ｺﾝｽﾄﾗｸﾀ

CStringW(void) : text()
{

}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

CStringW( CStringW& text_ )
{
   text_.GetString( text );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

CStringW( const std::wstring& text_ ) : text( text_ )
{

}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

CStringW( const WCHAR* p_text_ ) : text( p_text_ )
{

}
```

```
// ----------------------------------------------<OK><ﾃｽﾄ済み>
// ﾃﾞｽﾄﾗｸﾀ

~CStringW(void)
{

}

// ----------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾜｲﾄﾞ文字列】を返す。

inline void GetString( std::wstring& text_ )
{
   text_ = text;
}

// ----------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾜｲﾄﾞ文字列】を変更する。

inline void SetString( const std::wstring& text_ )
{
   text = text_;
}

// ----------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定indexの1文字】を返す。                    ( at() )

inline WCHAR GetChar( const size_t index_ )
{
return text.at( index_ );
}

// ----------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾎﾟｲﾝﾀ】を返す。                              ( c_str() )

// ※返される文字列は、終端に必ずNULL文字が付加されている。
```

```
//   ・文字数0の場合、p_result[0]は、NULL文字。
//   ・p_result[ length ] は、NULL文字。

inline const WCHAR* GetPtr(void)
{
return text.c_str(); // data() と同じ。
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexのｲﾃﾚｰﾀ】 を返す。

// ※戻り値のｲﾃﾚｰﾀを、参照にしていないのには理由がある。
//   このｸﾗｽがﾛｰｶﾙ変数として宣言された場合、
//   ﾛｰｶﾙのｱﾄﾞﾚｽを返してしまうためｺｹるからである。

inline ConstIterator GetItr( const size_t index_ )
{
    return ( text.begin() + index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字数】 を返す。                          ( length() or size() )

// ※終端のNULL文字を含まない。

inline size_t GetLength(void)
{
     return text.length();
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【最大文字数】 を返す。                          ( max_size() )

inline size_t GetMaxLength(void)
{
       return text.max_size();
}
```

```
// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を初期化する。                              ( assign() )

// ※ std::wstring::value_type は、wchar_t 。

// STL::CStringW text1;
// text1.Init( L"あいうえお", 3 ); // text1 は、"あいう" で初期化される。

inline std::wstring& Init( const WCHAR* p_text_ )
{
        return text.assign( p_text_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を初期化する。                              ( append() )

// STL::CStringW text1;
// text1.Init( L"あいうえお", 3 ); // text1 は、"あいう" で初期化される。

inline std::wstring& Init( const WCHAR* p_text_ , const size_t length_ )
{
       return text.assign( p_text_, length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】で初期化する。                              ( append() )

// STL::CStringW text1;
// std::wstring text2( L"かきくけこ" );
       // text1.InitByString( text2 ); // text1 は、"かきくけこ" で初期化される。

inline void InitByString( const std::wstring& text_ )
{
       text.assign( text_ );
}
```

```
// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】で初期化する。                              ( append() )

// STL::CStringW text1;
// std::wstring text2( L"かきくけこ" );
      // text1.InitByString( text2, 1, 2 ); // text1 は、"きく" で初期化される。

inline std::wstring& InitByString(
                              const std::wstring& text_,       // この文字列で初期化される。
       const size_t           start_index_, // 部分文字列の開始位置。
 const size_t          length_       // 部分文字列の文字数。
    )
{
     return text.assign( text_, start_index_, length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定した文字】で初期化する。                        ( append() )

// STL::CStringW text1;
// text1.InitByChar( L'あ', 3 ); // text1 は、"あああ" で初期化される。

inline std::wstring& InitByChar( const WCHAR char_, const size_t count_ )
{
      return text.assign( count_, char_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定範囲の文字列】で初期化する。( ﾎﾟｲﾝﾀ版 )  ( append() )

 // STL::CStringW text1;
// WCHAR src[] = L"かきくけこ";
      // text1.InitByRange( src + 1, src + 3 ); // text1 は、"きく" で初期化される。

inline std::wstring& InitByRange( const WCHAR* p_src_start_, const WCHAR* p_src_end_ )
{
     return text.assign( p_src_start_, p_src_end_ );
```

```
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定範囲の文字列】で初期化する。( ｲﾃﾚｰﾀ版 )  ( append() )

// STL::CStringW text1;
// std::wstring src( L"かきくけこ");
// std::wstring::iterator itr = src.begin();
      // text1.InitByRange( itr + 1, itr + 3 ); // text1 は、"きく" で初期化される。

inline std::wstring& InitByRange( const Iterator src_start_itr_, const Iterator src_end_itr_ )
{
    return text.assign( src_start_itr_, src_end_itr_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定文字列】と比較する。                    ( compare() )

// ※この比較では、大文字･小文字は区別される。
//  ( ( L"Uma" == L"uma" ) は false )

inline bool Compare( const std::wstring& text_ )
{
       return ( text.compare( text_ ) == 0 );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定範囲】と【指定文字列】を比較する。        ( compare() )

// 引数1,2で指定された範囲の部分文字列と、指定された文字列を比較する。

inline bool Compare(
       const size_t         index_,   // 比較する部分のｵﾌｾｯﾄ。( この文字列の )
                  const size_t         length_, // 比較する部分の文字数。( この文字列の )
        const std::wstring& text_     // 比較する文字列。
   )
{
```

```
        return ( text.compare( index_, length_, text_ ) == 0 );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定範囲】と【指定文字列の指定範囲】を比較する。( compare() )

inline bool Compare(
        const size_t        index_,       // 比較する部分のｵﾌｾｯﾄ。( この文字列の )
    const size_t        length_,      // 比較する部分の文字数。( この文字列の )
     const std::wstring& text_,        // 比較する文字列。
        const size_t        text_offset_, // 比較する部分のｵﾌｾｯﾄ。( 指定文字列の )
const size_t        text_length_ // 比較する部分の文字数。( 指定文字列の )
    )
{
            return ( text.compare( index_, length_, text_, text_offset_, text_length_ ) == 0 );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定文字列】と比較する。                    ( compare() )

inline bool Compare( const WCHAR* p_text_ )
{
            return ( text.compare( p_text_ ) == 0 );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定範囲】と【指定文字列】を比較する。        ( compare() )

inline bool Compare(
        const size_t    index_,  // 比較する部分のｵﾌｾｯﾄ。( この文字列の )
        const size_t    length_, // 比較する部分の文字数。( この文字列の )
    const WCHAR* p_text_  // 比較する文字列。
    )
{
            return ( text.compare( index_, length_, p_text_ ) == 0 );
}
```

```
// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定範囲】と【指定文字列の指定範囲】を比較する。( compare() )

inline bool Compare(
        const size_t index_,       // 比較する部分のｵﾌｾｯﾄ。( この文字列の )
    const size_t length_,      // 比較する部分の文字数。( この文字列の )
     const WCHAR* p_text_,  // 比較する文字列。
 const size_t text_length_ // 比較する部分の文字数。( 指定文字列の )
    )
{
            return ( text.compare( index_, length_, p_text_, text_length_ ) == 0 );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定範囲の部分文字列】を切り出して返す。       ( substr() )

// ※【戻り値】を参照にすると、【ﾛｰｶﾙのｱﾄﾞﾚｽ】が返る。

inline std::wstring SubString( const size_t start_index_, const size_t length_ )
{
return text.substr( start_index_, length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を追加する。                          ( append() )

// ※ std::wstring::value_type は、wchar_t 。

inline std::wstring& Add( const WCHAR* p_text_ )
{
return text.append( p_text_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を追加する。                          ( append() )

inline std::wstring& Add( const WCHAR* p_text_ , const size_t length_ )
```

```
{
		return text.append( p_text_, length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を追加する。                                ( append() )

inline std::wstring& Add( const std::wstring& text_ )
{
		return text.append( text_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を追加する。                                ( append() )

inline std::wstring& Add( const std::wstring& text_ , const size_t start_index_, const size_t length_ )
{
		return text.append( text_, start_index_, length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字】を末尾に追加する。                          ( push_back() )

inline void AddChar( const WCHAR char_ )
{
text.push_back( char_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定した文字】を、【指定した文字数】だけ追加する。( append() )

inline std::wstring& AddChar( const WCHAR char_, const size_t count_ )
{
		return text.append( count_, char_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
```

```
// 【指定範囲の文字列】を追加する。( ﾎﾟｲﾝﾀ版 )　　( append() )

// WCHAR src_text[] = L"かきくけこ";
// std::wstring added_text = t1.Add( src_text + 1, src_text + 3 ); // "きく" が追加される。

inline std::wstring& Add( const WCHAR* p_src_start_, const WCHAR* p_src_end_ )
{
        return text.append( p_src_start_, p_src_end_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定範囲の文字列】を追加する。( ｲﾃﾚｰﾀ版 )　( append() )

// std::wstring src_text( L"かきくけこ");
// std::wstring::iterator itr = src_text.begin();
// std::wstring added_text = t1.Add( itr + 1, itr + 3 ); // "きく" が追加される。

inline std::wstring& Add( const Iterator src_start_itr_, const Iterator src_end_itr_ )
{
        return text.append( src_start_itr_, src_end_itr_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を挿入する。                          ( insert() )

inline std::wstring& Insert( const size_t index_, const WCHAR* p_text_ )
{
return text.insert( index_, p_text_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を挿入する。                          ( insert() )

    // text1.Insert( 1, L"かきくけこ", 3 );
    // 2文字目に、"かきく" が挿入される。( 引数3は、挿入する文字数 )

inline std::wstring& Insert( const size_t index_, const WCHAR* p_text_, const size_t text_length_ )
```

```
{
        return text.insert( index_, p_text_, text_length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を挿入する。                              ( insert() )

inline std::wstring& Insert( const size_t index_, const std::wstring& text_ )
{
        return text.insert( index_, text_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を挿入する。                              ( insert() )

    // std::wstring src( L"かきくけこ" );
    // text1.InsertByRange( 1, src, 1, 2 ); // 2文字目に、"きく" を挿入する。

inline std::wstring& InsertByRange(
                        const size_t        index_,
                        const std::wstring& text_,
    const size_t        text_offset_,
    const size_t        text_length_
  )
{
        return text.insert( index_, text_, text_offset_, text_length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を挿入する。                              ( insert() )

    // ※【指定したｲﾃﾚｰﾀの指す位置までの文字列】で初期化される。
    //
    //   text1.Insert( text1.GetFirstItr() + 2 ); // 2文字目までの文字列になる。

    // ※他のｲﾝｽﾀﾝｽのｲﾃﾚｰﾀｰを渡すとｺｹる。
    // ※begin()で取得したｲﾃﾚｰﾀをそのまま渡すとｺｹる。( 1文字目は、begin() + 1 )
```

```
	//・使い道がよくわからない。おそらく使うことはない。

inline Iterator Insert( Iterator itr_ )
{
		return text.insert( itr_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字】を挿入する。                                ( insert() )

//・【ｲﾃﾚｰﾀで指定した位置】に、【指定した文字】を挿入する。

inline Iterator InsertByChar( Iterator itr_, WCHAR char_ )
{
		return text.insert( itr_, char_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を挿入する。                              ( insert() )

//・上記に加えて、さらに【挿入する文字数】を指定できる。

inline void InsertByChar( Iterator itr_, WCHAR char_, const size_t count_ )
{
		text.insert( itr_, count_, char_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を挿入する。                              ( insert() )

// text1.InsertByChar( 1, L'か', 3 );
//
//・2文字目から、'か' が 3文字、挿入される。( "あかかかいうえお" )

inline std::wstring& InsertByChar(
				const size_t index_,
```

```
                                const WCHAR char_,
  const size_t count_
      )
{
          return text.insert( index_, count_, char_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を挿入する。                              ( insert() )

inline void InsertByRange( Iterator itr_, const WCHAR* p_src_start_, const WCHAR* p_src_end_ )
{
          text.insert( itr_, p_src_start_, p_src_end_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を挿入する。                              ( insert() )

inline void InsertByRange(
                                    Iterator dest_start_itr_,
                                const Iterator src_start_itr_,
                                const Iterator src_end_itr_
                               )
{
          text.insert( dest_start_itr_, src_start_itr_, src_end_itr_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を挿入する。                              ( insert() )

// ・TInputIteratorには、読み取り専用のｲﾃﾚｰﾀ型を指定する。

// STL::stringW t1( L"あいうえお" );
// std::string  t2(  "かきくけこ" );
// t1.InsertByRange<std::string::iterator>( t1.GetItr(1), t2.begin(), t2.end() );
//
// → "あきいうえお" とはならず、stringの箇所が文字化けしている。
```

```
// ( 要するに、挿入は成功したが、文字ｺｰﾄﾞが異なるため、文字化けしている、 )

template<typename TInputIterator>
inline void InsertByRangeEx( Iterator itr_, TInputIterator src_start_itr_, TInputIterator src_end_itr_ )
{
        text.insert( itr_, src_start_itr_, src_end_itr_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列の指定部分】を削除して返す。          ( erase() )

// text1.RemoveAt( 1, 2 );
    // 2文字目以降の2文字が削除される。( "あいうえお" → "あえお" )

inline std::wstring& RemoveAt( const size_t index_, const size_t length_ )
{
        return text.erase( index_, length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexの1文字】を削除し、【次の文字のｲﾃﾚｰﾀｰ】を返す。( erase() )

// text1.Remove( text1.GetItr( 1 ) ); // 2文字目が削除される。
//
//・2文字目が削除されたので、3文字目が2文字目に移動する。
//  戻り値のｲﾃﾚｰﾀｰは、この文字を指している。

inline Iterator RemoveChar( Iterator itr_ )
{
        return text.erase( itr_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列の指定範囲】を削除し、【次の文字のｲﾃﾚｰﾀｰ】を返す。( erase() )

// text1.RemoveByRange( text1.GetItr(1), text1.GetItr(2)  );
    // ・2文字目から3文字目の範囲 ( 2文字目 ) が削除される。
```

```
inline Iterator RemoveByRange( Iterator start_itr_, Iterator end_itr_  )
{
        return text.erase( start_itr_, end_itr_ );
}
```

【wstring】 (ﾜｲﾄﾞ文字配列) 2

```
// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】 をｸﾘｱする。                                ( clear() )

inline void Clear(void)
{
        text.clear();
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// L"" なら、 true を返す。                            ( empty() )

inline bool IsEmpty(void)
{
        return text.empty();
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列の指定範囲】 をｺﾋﾟｰする。                    ( substr() )

// ※戻り値が参照でないのは、ﾛｰｶﾙのｱﾄﾞﾚｽを含んだ参照を返すためである。
//  ( ここを参照にすると、ｺﾝﾊﾟｲﾙ時に警告が表示されるが、動作はする )

inline std::wstring Substring( const size_t index_, const size_t length_ )
{
 return text.substr( index_, length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】 をｺﾋﾟｰする。   ( ｺﾋﾟｰした文字数を返す ) ( _Copy_s() )

// ・copy()が使いものにならないため、安全性の高い_Copy_s()を使用している。
// ( この手の_s系ﾒｿｯﾄﾞで、ｺﾋﾟｰ先とｺﾋﾟｰ元の要素数を指定しているのは、
      //  ﾊﾞｯﾌｧｵｰﾊﾞｰﾗﾝを防止するためである。 )

inline WCHAR* Copy(void)
```

```
{
	size_t src_length = text.length(); // ｺﾋﾟｰ元の文字数を取得しておく。

		WCHAR* p_dest = new WCHAR[ src_length + 1 ];
		// ｺﾋﾟｰ先の文字列ﾊﾞｯﾌｧを確保する。( +1 は、終端のNULL文字 )

 size_t writed_length = text._Copy_s( p_dest, src_length + 1, src_length, 0 ); // ｺﾋﾟｰする。
 //
 // p1: ｺﾋﾟｰ先の文字列ﾊﾞｯﾌｧへのﾎﾟｲﾝﾀ。
 // p2: ｺﾋﾟｰ先の文字列ﾊﾞｯﾌｧの要素数。( 文字数 )
 // p3: ｺﾋﾟｰ元の文字数。
 // p4: ｺﾋﾟｰを開始するｵﾌｾｯﾄ位置。( ｺﾋﾟｰ元のindex )
 //
		// 戻り値: ｺﾋﾟｰされた文字数が返る。

		 p_dest[ src_length ] = NULL; // 終端にNULL文字を入れる。

 return p_dest;
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を交換する。                                ( swap() )

inline void Swap( CStringW& text_ )
{
    std::wstring t1;
    text_.GetString( t1 );
    text.swap( t1 );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字数】を変更する。                                ( resize() )

// ・追加された要素は、0ｸﾘｱされている。
// ( "あいうえお" → "あいうえお\0\0\0\0\0" )

// ・一般的には、NULL文字までが文字列とみなされるが、
```

```
// lengthで返る文字数には、増加分も含まれている。
// ( ただし、NULL文字なので、表示はされない )

inline void Resize( const size_t new_length_ ) // 新しい文字数。
{
        text.resize( new_length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字数】を変更する。                          ( resize() )

// text.ResizeEx( 10, L'無' ); // → "あいうえお 無無無無無"

inline void ResizeEx(
            const size_t new_length_,    // 新しい文字数。
    const WCHAR  default_char_ // 新規要素は、この文字で初期化される。
  )
{
        text.resize( new_length_, default_char_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を置換する。                          ( replace() )

// 引数1～2の指定範囲を、指定文字列に置き換える。
    // ( ※一般的な、「検索→置換」とは別物 )

inline std::wstring& Replace(
  const size_t index_,        // 置換範囲のｵﾌｾｯﾄ位置。 (置換先)
  const size_t length_,       // 置換範囲の文字数。     (置換先)
                    const WCHAR* p_src_text_  // 置き換える文字列。     (置換元)
)
{
        return text.replace( index_, length_, p_src_text_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
```

```
// 【文字列】を置換する。                                ( replace() )

inline std::wstring& Replace(
  const size_t        index_,  // 置換範囲のｵﾌｾｯﾄ位置。  (置換先)
  const size_t        length_, // 置換範囲の文字数。      (置換先)
                      const std::wstring& src_text_ // 置き換える文字列。      (置換元)
 )
{
            return text.replace( index_, length_, src_text_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を置換する。                                ( replace() )

// STL::stringW text1( L"あいうえお" );
// std::wstring text2( L"かきくけこ" );
// text1.Replace( text1.GetItr(1), text1.GetItr(2), text2 ); // → "あかきくけこうえお"

inline std::wstring& Replace(
  const Iterator      start_itr_,  // 置換範囲の始点ｲﾃﾚｰﾀｰ。 (置換先)
  const Iterator      end_itr_,    // 置換範囲の終点ｲﾃﾚｰﾀｰ。 (置換先)
   const std::wstring& src_text_ // 置き換える文字列。          (置換元)
 )
{
             return text.replace( start_itr_, end_itr_, src_text_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を置換する。                                ( replace() )

// STL::stringW text1( L"あいうえお" );
// text1.Replace( 1,2, L'■', 3 ); // → "あ■■■えお"

inline std::wstring& ReplaceByChar(
 const size_t index_,           // 置換範囲のｵﾌｾｯﾄ位置。    (置換先)
 const size_t length_,          // 置換範囲の文字数。        (置換先)
 const WCHAR  src_char_,        // 置き換える文字。          (置換元)
```

```
 const size_t src_char_count_   // 置き換える文字の文字数。 (置換元)
   )
{
        return text.replace( index_, length_, src_char_count_, src_char_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【指定文字】に置換する。                          ( replace() )

// STL::stringW text1( L"あいうえお" );
// text1.Replace( text1.GetItr(1), text1.GetItr(2), L'■' ); // → "あ■■■うえお"

inline std::wstring& ReplaceByChar(
      const Iterator    start_itr_,       // 置換範囲の始点ｲﾃﾚｰﾀｰ。 (置換先)
      const Iterator    end_itr_,         // 置換範囲の終点ｲﾃﾚｰﾀｰ。 (置換先)
      const WCHAR    src_char_,       // 置き換える文字。          (置換元)
      const size_t      src_char_count_ // 置き換える文字の文字数。  (置換元)
    )
{
        return text.replace( start_itr_, end_itr_, src_char_count_, src_char_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を置換する。                            ( replace() )

// 指定範囲を、指定文字列の先頭から指定文字数の部分文字列で置き換える。

// STL::CStringW text1( L"あいうえお" );
// std::wstring  text2( L"かきくけこ" );
// text1.Replace( 1, 2, L"かきくけこ", 2 ); // → "あ かきえお"

inline std::wstring& ReplaceByRange(
 const size_t index_,          // 置換範囲のｵﾌｾｯﾄ位置。 (置換先)
 const size_t length_,         // 置換範囲の文字数。     (置換先)
                    const WCHAR* p_src_text_, // 置き換える文字列。      (置換元)
      const size_t src_length_     // 置換範囲の文字数。     (置換元)
  )
```

```
{
		return text.replace( index_, length_, p_src_text_, src_length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を置換する。                              ( replace() )

// STL::CStringW text1( L"あいうえお" );
// std::wstring  text2( L"かきくけこ" );
// text1.Replace( 1, 2, text2, 1, 2 ); // → "あきくえお"

inline std::wstring& ReplaceByRange(
    const size_t        index_,      // 置換範囲のｵﾌｾｯﾄ位置。 (置換先)
    const size_t        length_,     // 置換範囲の文字数。       (置換先)
		const std::wstring& src_text_,   // 置き換える文字列。         (置換元)
    const size_t        src_offset_, // 置換範囲のｵﾌｾｯﾄ位置。 (置換元)
    const size_t        src_length_ // 置換範囲の文字数。        (置換元)
    )
{
		return text.replace( index_, length_, src_text_, src_offset_, src_length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を置換する。                              ( replace() )

// STL::stringW text1( L"あいうえお" );
// text1.Replace( text1.GetItr(1), text1.GetItr(2), L"かきくけこ" ); // → "あ かきくけこうえお"

inline std::wstring& ReplaceByRange(
const Iterator     start_itr_,  // 置換範囲の始点ｲﾃﾚｰﾀｰ。 (置換先)
const Iterator     end_itr_,    // 置換範囲の終点ｲﾃﾚｰﾀｰ。 (置換先)
const WCHAR*   p_src_text_ // 置き換える文字列。         (置換元)
    )
{
		return text.replace( start_itr_, end_itr_, p_src_text_ );
}
```

```
// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を置換する。                                ( replace() )

// STL::stringW text1( L"あいうえお" );
// text1.Replace( text1.GetItr(1), text1.GetItr(2), L"かきくけこ" ); // → "あ かきうえお"

inline std::wstring& ReplaceByRange(
    const Iterator start_itr_,      // 置換範囲の始点ｲﾃﾚｰﾀｰ。        (置換先)
    const Iterator end_itr_,        // 置換範囲の終点ｲﾃﾚｰﾀｰ。        (置換先)
    const WCHAR*   p_src_text_, // 置き換える文字列。                (置換元)
    const size_t   src_length_      // 置換範囲の先頭からの文字数。(置換元)
  )
{
        return text.replace( start_itr_, end_itr_, p_src_text_, src_length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を置換する。                                ( replace() )

// STL::stringW text1( L"あいうえお" );
// WCHAR src[] = L"らりるれろ";
// t1.ReplaceByRange( t1.GetItr(1), t1.GetItr(2), src + 1, src + 3 ); // → "あ らりうえお"

inline std::wstring& ReplaceByRange(
    Iterator          start_itr_,     // 置換範囲の始点ｲﾃﾚｰﾀｰ。 (置換先)
    Iterator          end_itr_,       // 置換範囲の終点ｲﾃﾚｰﾀｰ。 (置換先)
    const WCHAR*   p_src_start_,  // 置換範囲の始点ﾎﾟｲﾝﾀ。      (置換元)
    const WCHAR*   p_src_end_    // 置換範囲の終点ﾎﾟｲﾝﾀ。      (置換元)
    )
{
        return text.replace( start_itr_, end_itr_, p_src_start_, p_src_end_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を置換する。                                ( replace() )

// STL::stringW t1( L"あいうえお" );
```

```
// STL::stringW t2( L"かきくけこ" );
// t1.ReplaceByRange( t1.GetItr(1), t1.GetItr(2), t2.GetItr(1), t2.GetItr(2) ); //  → "あきうえお"

inline std::wstring& ReplaceByRange(
    Iterator        start_itr_,     // 置換範囲の始点ｲﾃﾚｰﾀｰ。 (置換先)
    Iterator        end_itr_,       // 置換範囲の終点ｲﾃﾚｰﾀｰ。 (置換先)
    const Iterator src_start_itr_, // 置換範囲の始点ｲﾃﾚｰﾀｰ。 (置換元)
    const Iterator src_end_itr_    // 置換範囲の終点ｲﾃﾚｰﾀｰ。 (置換元)
    )
{
            return text.replace( start_itr_, end_itr_, src_start_itr_, src_end_itr_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// 【文字列】を置換する。                              ( replace() )

// STL::stringW t1( L"あいうえお" );
// STL::stringW t2( L"かきくけこ" );
// t1.ReplaceByRange<std::wstring::iterator>( t1.GetItr(1), t1.GetItr(2), t2.GetItr(1), t2.GetItr(2) );
      //  → "あきうえお"

template<typename TInputIterator>
inline std::wstring& ReplaceByRange(
    Iterator        start_itr_,         // 置換範囲の始点ｲﾃﾚｰﾀｰ。 (置換先)
    Iterator        end_itr_,           // 置換範囲の終点ｲﾃﾚｰﾀｰ。 (置換先)
    TInputIterator src_start_itr_,  // 置換範囲の始点ｲﾃﾚｰﾀｰ。 (置換元)
    TInputIterator src_end_itr_     // 置換範囲の終点ｲﾃﾚｰﾀｰ。 (置換元)
    )
{
            return text.replace( start_itr_, end_itr_, src_start_itr_, src_end_itr_ );
}
```

```
// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字】を検索し、【最初に見つかった箇所の先頭index】を返す。( find() )

// STL::stringW t1( L"あいうえお" );
//
// size_t result = text1.IndexOf( L'う', 0 ); // 要素[0]から検索する。
//
// if ( result == wstring::npos ) // 該当indexが無い場合は、trueになる。
// else // 今回は、 要素[2]が 'う' なので、2 が返る。

inline size_t IndexOf( const WCHAR char_, const size_t start_index_ = 0 )
{
        return text.find( char_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列】を検索し、【最初に見つかった箇所の先頭index】を返す。( find() )

// size_t result = text1.IndexOf( L"えお", 0 ); // 3 が返る。

inline size_t IndexOf( const WCHAR* p_text_, const size_t start_index_ = 0 )
{
        return text.find( p_text_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列】を検索し、【最初に見つかった箇所の先頭index】を返す。( find() )

// size_t result = text1.IndexOf( L"えお", 0, 4 ); // 4文字目まで判定する。( 結果は npos )
//
// ※ 4文字目の要素[3] には、'え' があるが、
//    5文字目の要素[4] の 'お' が範囲外で判定されないため、nposを返す。

inline size_t IndexOf( const WCHAR* p_text_, const size_t start_index_, const size_t length_ )
```

```
{
	return text.find( p_text_, start_index_, length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列】を検索し、【最初に見つかった箇所の先頭index】を返す。( find() )

inline size_t IndexOf( const std::wstring& text_, const size_t start_index_ = 0 )
{
	return text.find( text_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字以外】を検索し、【最初に見つかった箇所の先頭index】を返す。( find_first_not_of() )

inline size_t NotIndexOf( const WCHAR char_, const size_t start_index_  )
{
	  return text.find_first_not_of( char_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列以外】を検索し、【最初に見つかった箇所の先頭index】を返す。( find_first_not_of() )

inline size_t NotIndexOf( const WCHAR* p_text_, const size_t start_index_  )
{
	return text.find_first_not_of( p_text_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列以外】を検索し、【最初に見つかった箇所の先頭index】を返す。( find_first_not_of() )

inline size_t NotIndexOf( const WCHAR* p_text_, const size_t start_index_, const size_t length_ )
{
return text.find_first_not_of( p_text_, start_index_, length_ );
```

```
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列以外】を検索し、【最初に見つかった箇所の先頭index】を返す。( find_first_not_of() )

inline size_t NotIndexOf( const std::wstring& text_, const size_t start_index_ )
{
return text.find_first_not_of( text_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字】を検索し、【最後に見つかった箇所の先頭index】を返す。( find_last_of() )

inline size_t LastIndexOf( const WCHAR char_, const size_t start_index_ )
{
return text.find_last_of( char_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列】を検索し、【最後に見つかった箇所の先頭index】を返す。( find_last_of() )

inline size_t LastIndexOf( const WCHAR* p_text_, const size_t start_index_ )
{
return text.find_last_of( p_text_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列】を検索し、【最後に見つかった箇所の先頭index】を返す。( find_last_of() )

inline size_t LastIndexOf( const WCHAR* p_text_, const size_t start_index_, const size_t length_ )
{
return text.find_last_of( p_text_, start_index_, length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列】を検索し、【最後に見つかった箇所の先頭index】を返す。( find_last_of() )
```

```
inline size_t LastIndexOf( const std::wstring& text_, const size_t start_index_  )
{
return text.find_last_of( text_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字以外】を検索し、【最後に見つかった箇所の先頭index】を返す。( find_last_not_of() )

inline size_t NotLastIndexOf( const WCHAR char_, const size_t start_index_  )
{
return text.find_last_not_of( char_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列以外】を検索し、【最後に見つかった箇所の先頭index】を返す。( find_last_not_of() )

inline size_t NotLastIndexOf( const WCHAR* p_text_, const size_t start_index_  )
{
return text.find_last_not_of( p_text_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列以外】を検索し、【最後に見つかった箇所の先頭index】を返す。( find_last_not_of() )

inline size_t NotLastIndexOf( const WCHAR* p_text_, const size_t start_index_, const size_t length_  )
{
return text.find_last_not_of( p_text_, start_index_, length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列以外】を検索し、【最後に見つかった箇所の先頭index】を返す。( find_last_not_of() )
```

```
inline size_t NotLastIndexOf( const std::wstring text_, const size_t start_index_ )
{
return text.find_last_not_of( text_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字以外】を逆順に検索し、【最初に見つかった位置】を返す。( rfind() )

inline size_t IndexOf_Reverse( const WCHAR char_, const size_t start_index_ )
{
return text.rfind( char_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列以外】を逆順に検索し、【最初に見つかった位置】を返す。( rfind() )

inline size_t IndexOf_Reverse( const WCHAR* p_text_, const size_t start_index_ )
{
return text.rfind( p_text_, start_index_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列以外】を逆順に検索し、【最初に見つかった位置】を返す。( rfind() )

inline size_t IndexOf_Reverse( const WCHAR* p_text_, const size_t start_index_, const size_t
length_ )
{
return text.rfind( p_text_, start_index_, length_ );
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定文字列以外】を逆順に検索し、【最初に見つかった位置】を返す。( rfind() )

inline size_t IndexOf_Reverse( const std::wstring& text_, const size_t start_index_ )
{
return text.rfind( text_, start_index_ );
```

```
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【内部的に確保される要素数】を返す。                    ( capacity() )

//・未設定の状態だと、要素数が設定されている。

inline size_t GetCapacity(void)
{
      return text.capacity();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【内部的に確保される要素数】を変更する。                ( reserve() )

// ※wstringは、内部的にﾒﾓﾘ領域を多めに確保していて、この値は、その際の要素数。
//  多めに見積もっておけば、要素数が拡張された際の、再確保&ｺﾋﾟｰをしなくて済む。
//
//  ・この値は、wstring::capacity()で取得できる。
//  ・多すぎる場合は、wstring::shrink_to_fit()で切り詰めることができる。

inline void SetCapacity( const size_t capacity_ )
{
    text.reserve( capacity_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【先頭のｲﾃﾚｰﾀ】を返す。                              ( begin() )

inline ConstIterator GetFirstItr(void)
{
     return text.begin();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【末尾のｲﾃﾚｰﾀ】を返す。                              ( end() )
```

```
inline ConstIterator GetLastItr(void)
{
	return text.end();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 逆順ﾙｰﾌﾟ時の 【最初の要素([count-1])のｲﾃﾚｰﾀ】 を返す。( rbegin() )

inline ConstRevIterator GetFirstItr_Rev(void)
{
		return text.rbegin();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 逆順ﾙｰﾌﾟ時の 【最後の要素([0])のｲﾃﾚｰﾀ】 を返す。	( rend() )

inline ConstRevIterator GetLastItr_Rev(void)
{
		return text.rend();
}

 // ------------------------------------------------
 // ################################################
 // 演算子のｵｰﾊﾞｰﾛｰﾄﾞ
 // ################################################
 // ------------------------------------------------
 //  CString = instance + CString

inline std::wstring& operator+( std::wstring& text_ )
	{
		text += text_;

 return text;
	}

 // ------------------------------------------------
 //  instance += CString
```

```
inline void operator+=( std::wstring& text_ )
	{
		text += text_;
	}

 // ------------------------------------------------
 //  instance = CString

inline void operator=( std::wstring& text_ )
	{
	text = text_;
	}

 // ------------------------------------------------
 //  bool result = ( instance == CString )

inline bool operator==( std::wstring& text_ )
	{
	return (text == text_);
	}

// ------------------------------------------------
// ################################################
// static 関数
// ################################################
// ------------------------------------------------
//【ﾜｲﾄﾞ文字列】を、【Int32値】に変換して返す。( stol() )

//static inline int ToInt32(
//		const std::wstring text_,		// 変換する文字列。
//	const size_t	start_index_ = 0, // ｵﾌｾｯﾄ位置。
//	const int		base_ = 10	// 基数。規定値は10。( 10進数表記 )
//	)
//  {
//		return std::stol( text_, start_index_, base_ );
//  }
```

```
//// ------------------------------------------------
////【ﾜｲﾄﾞ文字列】を、【UInt32値】に変換して返す。( stoul() )

//static inline unsigned int ToUInt32(
//                          const std::wstring text_,             // 変換する文字列。
// const size_t         start_index_ = 0, // ｵﾌｾｯﾄ位置。
// const int            base_ = 10       // 基数。規定値は10。( 10進数表記 )
//   )
//   {
//       return std::stoul( text_, start_index_, base_ );
//   }

//// ------------------------------------------------
////【ﾜｲﾄﾞ文字列】を、【Int64値】に変換して返す。( stoll() )

//static inline __int64 ToInt64(
//                        const std::wstring text_,             // 変換する文字列。
//    const size_t        start_index_ = 0, // ｵﾌｾｯﾄ位置。
//    const int           base_ = 10       // 基数。規定値は10。( 10進数表記 )
// )
//   {
//       return std::stoll( text_, start_index_, base_ );
//   }

//// ------------------------------------------------
////【ﾜｲﾄﾞ文字列】を、【UInt64値】に変換して返す。( stoull() )

//static inline unsigned __int64 ToUInt64(
//                           const std::wstring text_,             // 変換する文字列。
//     const size_t        start_index_ = 0, // ｵﾌｾｯﾄ位置。
//     const int           base_ = 10       // 基数。規定値は10。( 10進数表記 )
//)
//   {
//       return std::stoull( text_, start_index_, base_ );
//   }
```

```
//// ------------------------------------------------
////【ﾜｲﾄﾞ文字列】を、【float値】に変換して返す。( stof() )

//static inline float ToFloat(
//                        const std::wstring text_,           // 変換する文字列。
//   const size_t        start_index_ = 0 // ｵﾌｾｯﾄ位置。
//      )
//   {
//         return std::stof( text_, start_index_ );
//   }

//// ------------------------------------------------
////【ﾜｲﾄﾞ文字列】を、【double値】に変換して返す。( stod() )

//// ・「long double」版のstold()には、「long」が付いているが、doubleと同じ64bit。

//static inline double ToDouble(
//                        const std::wstring text_,           // 変換する文字列。
//   const size_t        start_index_ = 0 // ｵﾌｾｯﾄ位置。
//     )
//   {
//         return std::stod( text_, start_index_ );
//   }

//// ------------------------------------------------
////【double値】を、【ﾜｲﾄﾞ文字列】に変換して返す。( to_wstring() )

//static inline std::wstring FromDouble( long double value_ )
//   {
//         return std::to_wstring( value_ );
//   }

//// ------------------------------------------------
////【long long値】を、ﾜｲﾄﾞ文字列に変換して返す。( to_wstring() )

//static inline std::wstring FromInt64( __int64 value_ )
//   {
```

```
//          return std::to_wstring( value_ );
//   }

//// ------------------------------------------------
////【unsigned long long】値を、ﾜｲﾄﾞ文字列に変換して返す。( to_wstring() )

//static inline std::wstring FromUInt64( unsigned __int64 value_ )
//   {
//          return std::to_wstring( value_ );
//   }

 // ------------------------------------------------

  }; // end class

}; // end namespace

// ------------------------------------------------

#endif
```

【bitset】（ﾋﾞｯﾄﾌﾗｸﾞ）

```
#pragma once

#ifndef _STL_BITSET_H_
#define _STL_BITSET_H_


#include <bitset>


using namespace std;


namespace STL
{

// ======================================================
// ﾋﾞｯﾄｾｯﾄｸﾗｽの定義（std::bitsetのﾗｯﾊﾟｰ）

// ・「ﾋﾞｯﾄｾｯﾄ」は、ﾋﾞｯﾄﾌﾗｸﾞを扱いやすくしたｺﾝﾃﾅ。
//  （ﾋﾞｯﾄ操作に慣れた人には、物足りない内容であり、通常ならﾏｸﾛで事足りる。）

// ・ﾃﾝﾌﾟﾚｰﾄ引数には、使用するﾋﾞｯﾄ数を指定する。
//  （たとえば、std::bitset<8>とした場合には、8bitﾌﾗｸﾞとなり、
//    0～7までの8つのﾋﾞｯﾄを、bool型のﾌﾗｸﾞ配列のように使用することができる。）

template<typename T>
class CBitSet
{


protected: // publicでもOK。


// ------------------------------------------------
// ﾃﾞｰﾀﾒﾝﾊﾞ
```

```
std::bitset<T> bits; // ﾋﾞｯﾄｾｯﾄ 本体。

public:

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

CBitSet(void) : bits()
{

}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ｺﾝｽﾄﾗｸﾀ

CBitSet( const unsigned __int64 value_ ) : bits( value_ )
{

}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// ﾃﾞｽﾄﾗｸﾀ

~CBitSet(void)
{

}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
// 【ﾋﾞｯﾄｾｯﾄ】 を返す。

  inline std::bitset<T>& GetBitSet(void)
  {
        return bits;
```

```
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【ﾋﾞｯﾄｾｯﾄ】 を格納する。

  inline void SetBitSet( std::bitset<T>& bits_ )
  {
        bits = bits_;
  }

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexのﾋﾞｯﾄ】 を、 ONにする。                    ( set() )

inline std::bitset<T>& SetOnAt( const size_t index_ )
{
        return bits.set( index_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexのﾋﾞｯﾄ】 を、 OFFにする。                   ( reset() )

inline std::bitset<T>& SetOffAt( const size_t index_ )
{
        return bits.reset( index_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexのﾋﾞｯﾄ】 の、ﾌﾗｸﾞ値 ( 0か1 ) を返す。        ( test() )

inline bool GetValue( const size_t index_ )
{
        return bits.test( index_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexのﾋﾞｯﾄ】 に、ﾌﾗｸﾞ値 ( 0か1 ) を設定する。      ( set() )
```

```
inline std::bitset<T>& SetValue( const size_t index_, const bool value_ )
{
	return bits.set( index_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【すべてのﾋﾞｯﾄ】を、OFFにする。                          ( reset() )

inline std::bitset<T>& SetOffAll(void)
{
	 return bits.reset();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【指定indexのﾋﾞｯﾄ】を、反転する。                          ( flip() )

inline std::bitset<T>& FlipAt( const size_t index_ )
{
	 return bits.flip( index_ );
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【すべてのﾋﾞｯﾄ】を、反転する。                          ( flip() )

inline std::bitset<T>& FlipAll(void)
{
	return bits.flip();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【いずれかのﾋﾞｯﾄ】がONなら、TRUEを返す。                ( any() )

inline bool IsAnyOn(void)
{
	return bits.any();
}
```

```
// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【すべてのﾋﾞｯﾄ】がONなら、TRUEを返す。                    ( all() )

inline bool IsAllOn(void)
{
        return bits.all();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【すべてのﾋﾞｯﾄ】がOFFなら、TRUEを返す。                  ( none() )

inline bool IsAllOff(void)
{
        return bits.none();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【ONになっているﾋﾞｯﾄの数】を返す。                      ( count() )

inline size_t GetOnCount(void)
{
        return bits.count();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【ﾋﾞｯﾄの数】を返す。                                  ( size() )

inline size_t GetBitCount(void)
{
        return bits.size();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【符号なし32bit整数値】を返す。                        ( to_ulong() )

inline unsigned int ToUInt32(void)
{
```

```
        return bits.to_ulong();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【符号なし64bit整数値】を返す。                    ( to_ullong() )

inline unsigned __int64 ToUInt64(void)
{
        return bits.to_ullong();
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//【文字列】を返す。                ( to_string() )

// "00101010" といった文字列が返る。

inline std::string ToString(void)
{
        return bits.to_string();
}



// ------------------------------------------------
// ################################################
// 演算子のｵｰﾊﾞｰﾛｰﾄﾞ
// ################################################
// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//  指定index_ のﾋﾞｯﾄ値を返す。

inline bool operator[]( const size_t index_ )
{
return bits[ index_ ];
}

// ----------------------------------------------------<OK><ﾃｽﾄ済み>
//  指定index_ の参照を返す。
```

```
inline std::bitset::reference operator[]( const size_t index_ )
{
return bits[ index_ ];
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//  instance = CBitSet

inline CBitSet<T>& operator=( const CBitSet<T>& bits_ )
{
bits = bits_.GetBitSet();
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//  instance |= CBitSet

inline CBitSet<T>& operator|=( const CBitSet<T>& bits_ )
{
bits |= bits_.GetBitSet();
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//  instance ^= CBitSet

inline CBitSet<T>& operator^=( const CBitSet<T>& bits_ )
{
bits ^= bits_.GetBitSet();
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//  ~instance

inline CBitSet<T>& operator~()
{
return ~bits;
}
```

```
// ------------------------------------------------<OK><ﾃｽﾄ済み>
// instance << shift_bit_count_

inline CBitSet<T> operator<<( const size_t shift_bit_count_ )
{
return ( bits <<  shift_bit_count_);
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// instance >> shift_bit_count_

inline CBitSet<T> operator>>( const size_t shift_bit_count_ )
{
return ( bits >>  shift_bit_count_);
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// instance <<= shift_bit_count_

inline CBitSet<T>& operator<<=( const size_t shift_bit_count_ )
{
return ( bits <<= shift_bit_count_);
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
// instance >>= shift_bit_count_

inline CBitSet<T>& operator>>=( const size_t shift_bit_count_ )
{
return ( bits >>= shift_bit_count_);
}

// ------------------------------------------------<OK><ﾃｽﾄ済み>
//  bool result = ( instance == CBitSet )

inline bool operator==( CBitSet<T>& bits_ )
{
```

```
return ( bits == bits_.GetBitSet() );
}


// ------------------------------------------------<OK><ﾃｽﾄ済み>
//  bool result = ( instance != CBitSet )

inline bool operator!=( CBitSet<T>& bits_ )
{
return ( bits != bits_.GetBitSet() );
}

// ------------------------------------------------

}; // end class

}; // end namespace

// ------------------------------------------------

#endif
```

奥付

「ハム太と作るC++ライブラリ1 STL編」

http://p.booklog.jp/book/74234

著者 : haseham
著者プロフィール：http://p.booklog.jp/users/haseham/profile

感想はこちらのコメントへ
http://p.booklog.jp/book/74234

ブクログ本棚へ入れる
http://booklog.jp/item/3/74234

電子書籍プラットフォーム：ブクログのパブー（http://p.booklog.jp/）
運営会社：株式会社ブクログ