CommonLibs

Custom types and utulites made to replace C++ Standard Library in projects, where STL is not allowed

Yuri Zamyatin, 2020

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 Common Namespace Reference

Classes

- class CDoesNotExist

  Represents "Element does not exist" error. Stores message.
- class CException

  Basic exception class. C-style message is required.
- class COutOfRange

  Represents "Out of Range" error. Can hold message, requested and expected indices.
- class TOptional

  Represents object that may not exist.
- class TPair

  Represents a pair of objects.
- class TVector

  Container representing array that can change its size.

Functions

- template<typename IteratorType >
  size_t GetIteratorDistance (IteratorType Begin, IteratorType End)

  Counts elements between two iterators. Range: [Begin: End)
- size_t GetRawStringLength (const char *const NullTermString)

  Calculates length of the C-string.
- size_t GetRawStringLength (const char *const NullTermString, size_t MaxLength)

  Calculates length of the C-string. Stops when null character is reached or MaxLength elements have been counted. Useful with malformatted input.
- void CopyRawString (const char *const NullTermStringFrom, char *const RawStringTo)

  Does the copy of C-style string (ended with '\0').
- void CopyRawString (const char *const NullTermStringFrom, char *const RawStringTo, size_t MaxLength)

  Does the copy of C-style string. Stops when null character is reached or MaxLength elements have been copied. Useful when string should be cut.
- bool AreRawStringsEqual (const char *const NullTermString1, const char *const NullTermString2)

  Checks whether two C-style strings are equal.
- bool AreRawStringsEqual (const char *const NullTermString1, const char *const NullTermString2, size_t MaxCompareLength)

  Checks whether two C-style strings are equal. Use this to compare N first elements, even if strings are not null-terminated.

5.1.1 Function Documentation

5.1.1.1 AreRawStringsEqual() [1/2]

```
bool Common::AreRawStringsEqual (
            const char *const NullTermString1,
            const char *const NullTermString2 )
```

Checks whether two C-style strings are equal.

Parameters

NullTermString1	First null-terminated string
NullTermString2	Second null-terminated string

Returns

true if characters before '\0' are the same false otherwise

5.1.1.2 AreRawStringsEqual() [2/2]

```
bool Common::AreRawStringsEqual (
            const char *const NullTermString1,
            const char *const NullTermString2,
            size_t MaxCompareLength )
```

Checks whether two C-style strings are equal. Use this to compare N first elements, even if strings are not null-terminated.

Parameters

NullTermString1	First string (whether null- terminated or limited with MaxCompareLength)
NullTermString2	Second string (whether null- terminated or limited with MaxCompareLength)
MaxComparedLength	Max amount of characters to compare; does not include trailing '\0'

Returns

true if characters before '\0' are the same false otherwise

5.1.1.3 CopyRawString() [1/2]

```
void Common::CopyRawString (
            const char *const NullTermStringFrom,
            char *const RawStringTo )
```

Does the copy of C-style string (ended with '\0').

Parameters

NullTermStringFrom	Source: char array that ends with '\0'
NullTermStringTo	Destination: Char array that is large enough to receive copied elements. May not end with '\0'

5.1.1.4 CopyRawString() [2/2]

```
void Common::CopyRawString (
            const char *const NullTermStringFrom,
            char *const RawStringTo,
            size_t MaxLength )
```

Does the copy of C-style string. Stops when null character is reached or MaxLength elements have been copied. Useful when string should be cut.

Parameters

NullTermStringFrom	Source: char array that ends with '\0' (or not, if you rely on MaxLength and buffer size)
NullTermStringTo	Destination: Char array that is large enough to receive copied elements. May not end with '\0'. After copying it gets '\0' anyway
MaxLength	Max amount of characters to copy; does not include trailing '\0'

5.1.1.5 GetIteratorDistance()

```
template<typename IteratorType >
size_t Common::GetIteratorDistance (
            IteratorType Begin,
            IteratorType End )
```

Counts elements between two iterators. Range: [Begin: End)

Template Parameters

IteratorType	Iterator with implemented ++, != and *

Parameters

Begin	Iterator referring to the first element
End	Iterator referring to the element after last one

Returns

Distance between iterators

Note

Begin must not be greater than end (negative results are not supported)

5.1.1.6 GetRawStringLength() [1/2]

```
size_t Common::GetRawStringLength (
            const char *const NullTermString )
```

Calculates length of the C-string.

Parameters

NullTermString	Char array that ends with '\0'

Returns

Number of actual letters in string ('\0' is not counted)

5.1.1.7 GetRawStringLength() [2/2]

```
size_t Common::GetRawStringLength (
            const char *const NullTermString,
            size_t MaxLength )
```

Calculates length of the C-string. Stops when null character is reached or MaxLength elements have been counted. Useful with malformatted input.

Parameters

NullTermString	Char array that ends with '\0' (or not, if you rely on MaxLength and buffer size)
MaxLength	Max amount of characters to count; does not include the trailing '\0'

Returns

Number of actual letters in string ('\0' is not counted)

Chapter 6

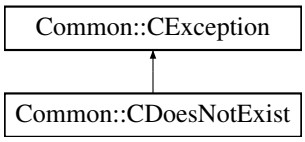
Class Documentation

6.1 Common::CDoesNotExist Class Reference

Represents "Element does not exist" error. Stores message.

`#include "CommonTypes/Exception.h"`

Inheritance diagram for Common::CDoesNotExist:



Public Member Functions

- CDoesNotExist (const char *const Message) noexcept
  Pass only message, if other properties cannot be specified.

Additional Inherited Members

6.1.1 Detailed Description

Represents "Element does not exist" error. Stores message.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 CDoesNotExist()

```
Common::CDoesNotExist::CDoesNotExist (
            const char *const Message )  [inline], [noexcept]
```

Pass only message, if other properties cannot be specified.

Parameters

Message	Description. Will be copied to the inner buffer

Note

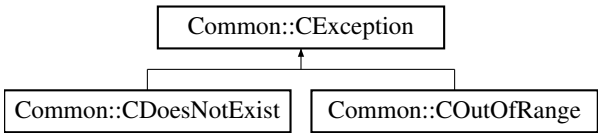
If length of message $>$ 40, first 40 symbols will be saved.

6.2 Common::CException Class Reference

Basic exception class. C-style message is required.

`#include "CommonTypes/Exception.h"`

Inheritance diagram for Common::CException:



Public Member Functions

- CException (const char ∗const Message) noexcept
  All exceptions must provide the message.
- virtual ∼CException ()
- virtual const char ∗ GetMessage () const noexcept
  Error message.

Protected Attributes

- char Message [41]

6.2.1 Detailed Description

Basic exception class. C-style message is required.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 CException()

```
Common::CException::CException (
            const char *const Message )  [inline], [noexcept]
```

All exceptions must provide the message.

Parameters

Message	Error description. Will be copied to an inner buffer

Note

If length of message $>$ 40, first 40 symbols will be saved.

6.2.2.2 ~CException()

```
virtual Common::CException::~CException ( )  [inline], [virtual]
```

6.2.3 Member Function Documentation

6.2.3.1 GetMessage()

```
virtual const char* Common::CException::GetMessage ( ) const  [inline], [virtual], [noexcept]
```

Error message.

Returns

C-style string with error description

6.2.4 Member Data Documentation

6.2.4.1 Message

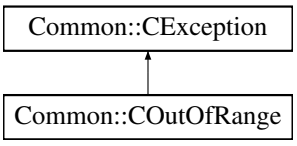
```
char Common::CException::Message[41]  [protected]
```

6.3 Common::COutOfRange Class Reference

Represents "Out of Range" error. Can hold message, requested and expected indices.

```
#include "CommonTypes/Exception.h"
```

Inheritance diagram for Common::COutOfRange:



Public Member Functions

- COutOfRange (const char *const Message) noexcept

  Pass only message, if other properties cannot be specified.
- COutOfRange (const char *const Message, int RequestedIndex, const TPair< size_t, size_t > &ExpectedIndex) noexcept

  Describes valid range and errored value. Contains message.
- const TPair< size_t, size_t > & GetExpectedRange () const noexcept

  Specifies valid range.
- int GetRequestedIndex () const noexcept

  Index that caused this exception.

Protected Attributes

- const int RequestedIndex = 0
- const TPair< size_t, size_t > ExpectedIndex = { 0,0 }

6.3.1 Detailed Description

Represents "Out of Range" error. Can hold message, requested and expected indices.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 COutOfRange() [1/2]

```
Common::COutOfRange::COutOfRange (
            const char *const Message )  [inline], [noexcept]
```

Pass only message, if other properties cannot be specified.

Parameters

Message	Description. Will be copied to an inner buffer

Note

Range will be set to empty [0: 0), requested index to 0.

If length of message > 40, first 40 symbols will be saved.

6.3.2.2 COutOfRange() [2/2]

```
Common::COutOfRange::COutOfRange (
            const char *const Message,
```

```
int RequestedIndex,
const TPair< size_t, size_t > & ExpectedIndex )  [inline], [noexcept]
```

Describes valid range and errored value. Contains message.

Parameters

Message	Description. Will be copied to an inner buffer
RequestedIndex	Errored index (out of range).
ExpectedIndex	Pair of Min and Max+1 indexes, that were available. Range: [First: Second)

Note

If length of message $>$ 40, first 40 symbols will be saved.

6.3.3 Member Function Documentation

6.3.3.1 GetExpectedRange()

```
const TPair<size_t, size_t>& Common::COutOfRange::GetExpectedRange ( ) const  [inline], [noexcept]
```

Specifies valid range.

Returns

Pair of Min and Max+1 indexes, that were available. Range: [First: Second)

Note

Returns 0 Index with [0: 0) range if constructed only with message.

6.3.3.2 GetRequestedIndex()

```
int Common::COutOfRange::GetRequestedIndex ( ) const  [inline], [noexcept]
```

Index that caused this exception.

Returns

Value of index, that is not in expected range

Note

Returns 0 Index with [0: 0) range if constructed only with message.

6.3.4 Member Data Documentation

6.3.4.1 ExpectedIndex

```
const TPair<size_t, size_t> Common::COutOfRange::ExpectedIndex = { 0,0 }  [protected]
```

6.3.4.2 RequestedIndex

```
const int Common::COutOfRange::RequestedIndex = 0  [protected]
```

6.4 Common::TOptional< T > Class Template Reference

Represents object that may not exist.

```
#include "CommonTypes/Optional.h"
```

Public Member Functions

- TOptional ()=default
  No object by default.
- TOptional (const T &Value) noexcept
  Initialize optional with existing value (copy).
- void SetValue (const T &Value) noexcept
  Set value to optional (copy).
- TOptional< T > & operator= (const T &Value) noexcept
  Set value to the optional (copy).
- bool ValueExists () const noexcept
  Check if optional has value.
- const T & GetValue () const
  Gets value if it exists or throws an exception.
- const T & GetValueOr (const T &OtherVariant) const noexcept
  Get value or passed value (if not possible)

6.4.1 Detailed Description

template<typename T>
class Common::TOptional< T >

Represents object that may not exist.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 TOptional() [1/2]

```
template<typename T >
Common::TOptional< T >::TOptional ( )  [default]
```

No object by default.

6.4.2.2 TOptional() [2/2]

```
template<typename T >
Common::TOptional< T >::TOptional (
            const T & Value )  [inline], [noexcept]
```

Initialize optional with existing value (copy).

Parameters

Value	Object to create copy from

6.4.3 Member Function Documentation

6.4.3.1 GetValue()

```
template<typename T >
const T& Common::TOptional< T >::GetValue ( ) const  [inline]
```

Gets value if it exists or throws an exception.

Returns

Optional's value

6.4.3.2 GetValueOr()

```
template<typename T >
const T& Common::TOptional< T >::GetValueOr (
            const T & OtherVariant ) const  [inline], [noexcept]
```

Get value or passed value (if not possible)

Parameters

OtherVariant	

Returns

Optional value or provided value

6.4.3.3 operator=()

```
template<typename T >
TOptional<T>& Common::TOptional< T >::operator= (
            const T & Value )  [inline], [noexcept]
```

Set value to the optional (copy).

Parameters

Value	Object to create copy from

Returns

Reference to this optional

6.4.3.4 SetValue()

```
template<typename T >
void Common::TOptional< T >::SetValue (
            const T & Value )  [inline], [noexcept]
```

Set value to optional (copy).

Parameters

Value	Object to create copy from

6.4.3.5 ValueExists()

```
template<typename T >
bool Common::TOptional< T >::ValueExists ( ) const  [inline], [noexcept]
```

Check if optional has value.

Returns

True if value exists

6.5 Common::TPair< T1, T2 > Class Template Reference

Represents a pair of objects.

`#include "CommonTypes/Pair.h"`

Public Member Functions

- TPair ()=default
  Initialize pair with type default values.
- TPair (const T1 &First, const T2 &Second) noexcept
  Creates a pair copying passed values.
- void MakePair (T1 First, T2 Second) noexcept
  Assigns two values to pair at once (copy).

Public Attributes

- T1 First = T1()
  First value in pair.
- T2 Second = T2()
  Second value in pair.

6.5.1 Detailed Description

template<typename T1, typename T2>
class Common::TPair< T1, T2 >

Represents a pair of objects.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 TPair() [1/2]

```
template<typename T1 , typename T2 >
Common::TPair< T1, T2 >::TPair ( )  [default]
```

Initialize pair with type default values.

6.5.2.2 TPair() [2/2]

```
template<typename T1 , typename T2 >
Common::TPair< T1, T2 >::TPair (
            const T1 & First,
            const T2 & Second )  [inline], [noexcept]
```

Creates a pair copying passed values.

Parameters

First	First value in pair
Second	Second value in pair

6.5.3 Member Function Documentation

6.5.3.1 MakePair()

```
template<typename T1 , typename T2 >
void Common::TPair< T1, T2 >::MakePair (
            T1 First,
            T2 Second )  [inline], [noexcept]
```

Assigns two values to pair at once (copy).

Parameters

First	First value in pair
Second	Second value in pair

6.5.4 Member Data Documentation

6.5.4.1 First

```
template<typename T1 , typename T2 >
T1 Common::TPair< T1, T2 >::First = T1()
```

First value in pair.

6.5.4.2 Second

```
template<typename T1 , typename T2 >
T2 Common::TPair< T1, T2 >::Second = T2()
```

Second value in pair.

6.6 Common::TVector< T > Class Template Reference

Container representing array that can change its size.

`#include "CommonTypes/Vector.h"`

Public Types

- enum EReservedCapacityRule : uint8_t { EReservedCapacityRule::Exponential, EReservedCapacityRule::Linear, EReservedCapacityRule::NeverReserve }

  Used to define how methods deal with Capacity.
- typedef TIterator< T *, T & > CIterator

  Iterator. Implemented op-s: ++, +=, +, –, -=, -, ==, !=, =.
- typedef TIterator< const T *, const T & > CConstIterator

  Version of CIterator for const values.
- typedef TReverseIterator< T *, T & > CReverseIterator

  Reverse iterator. Increment is actually decrement, etc.
- typedef TReverseIterator< const T *, const T & > CConstReverseIterator

  Version of TReverseIterator for const values.
- typedef TSafeIterator< T *, T & > CSafeIterator

  Iterator that does bounds checking and throws OutOfRange().
- typedef TSafeIterator< const T *, const T & > CSafeConstIterator

  Version of TSafeIterator for const values.
- typedef TSafeReverseIterator< T *, T & > CSafeReverseIterator

  Reverse iterator that can throw OutOfRange().
- typedef TSafeReverseIterator< const T *, const T & > CSafeConstReverseIterator

  Version of TSafeReverseIterator for const values.

Public Member Functions

- TVector ()=default

  Empty vector with no heap allocation.
- TVector (EReservedCapacityRule CapacityRule) noexcept

  Creates empty vector with Capacity preset predefined.
- TVector (size_t Size, const T &DefaultValue={})

  Vector with pre-allocated elements.
- TVector (size_t Size, const T *const Array)

  Vector constructed from raw dynamic array (copy).
- TVector (const std::initializer_list< T > &ValuesList)

  Modern C++ initialization syntax: name = {...}.
- template<typename IteratorType >
  TVector (IteratorType Begin, IteratorType End, typename std::enable_if<!std::is_integral< IteratorType >::value >::type *=0)

  Constructor to get values from another container.
- TVector (const TVector< T > &Other)

  Initialize by copying another TVector.
- TVector (TVector< T > &&Other) noexcept

  Move constructor.
- ~TVector ()

- template<typename IteratorType >
  void Assign (IteratorType Begin, IteratorType End, bool bAllowAutoShrink=false)

  Allows to copy values from another container.
- TVector< T > & operator= (const std::initializer_list< T > &ValuesList)

  Allows assignment with = {...} style.
- TVector< T > & operator= (const TVector< T > &Other)

  Assingment operator makes a copy of another vector.
- TVector< T > & operator= (TVector< T > &&Other) noexcept

  Move assignment.
- T & operator[ ] (size_t Index)

  Index operator that prodives access to the element.
- const T & operator[ ] (size_t Index) const

  Index operator for const vectors.
- T & SafeAt (size_t Index)

  [] with range check.
- const T & SafeAt (size_t Index) const
- T & AutoAt (size_t Index, const T &DefaultValue={})

  If element does not exist, this will resize vector and fill newly created elements with provided value.
- T * RawData () noexcept

  Provides access to the internal buffer.
- const T * RawData () const noexcept

  RawData() for const vectors.
- bool operator== (const TVector< T > &Other) const noexcept

  Checks if two vectors contain the same values by calling equality operator recursively.
- bool operator!= (const TVector< T > &Other) const noexcept

  Opposite to operator ==.
- void Push (const T &Value)

  Adds one element to the end of vector.
- template<typename IteratorType >
  void Push (IteratorType Begin, IteratorType End)

  Adds elements to the end, uses iterators.
- void Insert (size_t Position, const T &Value=T{}, const T &FillOnResizeWith=T{})

  Inserts one elements to the specified position.
- template<typename IteratorType >
  void Insert (size_t Position, IteratorType Begin, IteratorType End, const T &FillOnResizeWith=T{}, typename std::enable_if<!std::is_integral< IteratorType >::value >::type *=0)

  Inserts range of elements, starting at Position.
- T Pop (bool bAllowAutoShrink=false)

  Removes one element from the end of vector.
- T SafePop (bool bAllowAutoShrink=false)

  Pop() with range check.
- void PopMultiple (size_t ElementsCount, bool bAllowAutoShrink=false)

  Removes N elements from the end of vector.
- T Shift (bool bAllowAutoShrink=false)

  Removes one element from the beginning of vector.
- T SafeShift (bool bAllowAutoShrink=false)

  Shift() with range check.
- void ShiftMultiple (size_t ElementsCount, bool bAllowAutoShrink=false)

  Removes N elements from the beginning of vector.
- void Erase (size_t Position, bool bAllowAutoShrink=false)

  Removes element with specified position.

- void EraseMultiple (size_t PositionFrom, size_t PositionTo, bool bAllowAutoShrink=false)

  Removes range of elements from vector.
- void Reserve (size_t NewCapacity)

  Allocates elements internally for the future use.
- void Resize (size_t NewSize, const T &DefaultValue={}, bool bAllowAutoShrink=false)

  Changes size of vector.
- void ShrinkToFit ()

  Clears memory that was reserved for future use.
- void Clear ()

  Removes all elements from the vector.
- size_t GetSize () const noexcept

  Size is number of elements that you can use.
- size_t GetCapacity () const noexcept

  Capacity is Size + reserved space for the future use.
- bool IsEmpty () const noexcept

  Simple check if size of this vector equals 0.
- void SetCapacityRule (EReservedCapacityRule CapacityRule) noexcept

  Set enum value that will describe how re-allocation works when elements are added / removed from vertor. Reallocation on removal happens only if bool bAllowAutoShrink was passed with supported operation.
- EReservedCapacityRule GetCapacityRule () const noexcept

  Returns Capacity rule that is currently applied. It affects how elements are allocated & destructed.
- T & Front ()

  Provides access to the first element.
- const T & Front () const

  Front() for const vectors.
- T & SafeFront ()

  Front() with range check.
- const T & SafeFront () const

  SafeFront() for const vectors.
- T & Back ()

  Provides access to the last element.
- const T & Back () const

  Back() for const vectors.
- T & SafeBack ()

  Back() with range check.
- const T & SafeBack () const

  SafeBack() for const vectors.
- CIterator Begin ()

  Iterator pointing to the first elem.
- CConstIterator ConstBegin () const

  Iterator pointing to the first elem (const).
- CReverseIterator ReverseBegin ()

  Reverse iterator pointing to the first elem.
- CConstReverseIterator ConstReverseBegin () const

  Reverse iterator pointing to the first elem (const).
- CSafeIterator SafeBegin ()

  Safe iterator pointing to the first elem.
- CSafeConstIterator SafeConstBegin () const

  Safe iterator pointing to the first elem (const).
- CSafeReverseIterator SafeReverseBegin ()

  Safe reverse iterator pointing to the first elem.

- CSafeConstReverseIterator SafeConstReverseBegin () const

  Safe reverse iterator pointing to the first elem (const).
- CIterator End ()

  Iterator pointing to the elem after last.
- CConstIterator ConstEnd () const

  Iterator pointing to the elem after last (const).
- CReverseIterator ReverseEnd ()

  Reverse iterator pointing to the elem after last.
- CConstReverseIterator ConstReverseEnd () const

  Reverse iterator pointing to the elem after last (const).
- CSafeIterator SafeEnd ()

  Safe iterator pointing to the elem after last.
- CSafeConstIterator SafeConstEnd () const

  Safe iterator pointing to the elem after last (const).
- CSafeReverseIterator SafeReverseEnd ()

  Safe reverse iterator pointing to the elem after last.
- CSafeConstReverseIterator SafeConstReverseEnd () const

  Safe reverse it. pointing to elem after last (const).
- CIterator begin ()

  Begin() alias (for compatibity)
- CConstIterator begin () const

  ConstBegin() alias (overloaded, for compatibity)
- CConstIterator cbegin () const

  ConstBegin() alias (for compatibity)
- CReverseIterator rbegin ()

  ReverseBegin() alias (for compatibity)
- CConstReverseIterator rbegin () const

  ConstReverseBegin() alias (overloaded, for compatibity)
- CConstReverseIterator crbegin () const

  ConstReverseBegin() alias (for compatibity)
- CIterator end ()

  End() alias (for compatibity)
- CConstIterator end () const

  ConstEnd() alias (overloaded, for compatibity)
- CConstIterator cend () const

  ConstEnd() alias (for compatibity)
- CReverseIterator rend ()

  ReverseEnd() alias (for compatibity)
- CConstReverseIterator rend () const

  ConstReverseEnd() alias (overloaded, for compatibity)
- CConstReverseIterator crend () const

  ConstReverseEnd() alias (for compatibity)

Static Public Member Functions

- static void Swap (TVector< T > &Vector1, TVector< T > &Vector2) noexcept

  Swaps two vectors internally without deep copy.

6.6.1 Detailed Description

template<typename T>
class Common::TVector< T >

Container representing array that can change its size.

Fast in accessing elements as offsets are used, but not very effective in terms of adding and removing elements. To minimize those drawbacks, there are memory reserving rules that you can manage.

Note

If the logical result of operation may not be determined, there are two methods: one that throws an exception ("Safe" prefix) and another one that does assertion in debug mode. Custom exception is COutOfRange, derived from CException

May throw std::bad_alloc if allocation fails. Allocation failure in assignment operations will cause the vector to be cleared (as you intended to remove old elements anyway). If allocation fails in insertions, resize, push/pop, etc. - vector will stay in the previous state.

6.6.2 Member Typedef Documentation

6.6.2.1 CConstIterator

```
template<typename T >
typedef TIterator<const T*, const T&> Common::TVector< T >::CConstIterator
```

Version of CIterator for const values.

6.6.2.2 CConstReverseIterator

```
template<typename T >
typedef TReverseIterator<const T*, const T&> Common::TVector< T >::CConstReverseIterator
```

Version of TReverseIterator for const values.

6.6.2.3 CIterator

```
template<typename T >
typedef TIterator<T*, T&> Common::TVector< T >::CIterator
```

Iterator. Implemented op-s: ++, +=, +, –, -=, -, ==, !=, =.

6.6.2.4 CReverseIterator

```
template<typename T >
typedef TReverseIterator<T*, T&> Common::TVector< T >::CReverseIterator
```

Reverse iterator. Increment is actually decrement, etc.

6.6.2.5 CSafeConstIterator

```
template<typename T >
typedef TSafeIterator<const T*, const T&> Common::TVector< T >::CSafeConstIterator
```

Version of TSafeIterator for const values.

6.6.2.6 CSafeConstReverseIterator

```
template<typename T >
typedef TSafeReverseIterator<const T*, const T&> Common::TVector< T >::CSafeConstReverseIterator
```

Version of TSafeReverseIterator for const values.

6.6.2.7 CSafeIterator

```
template<typename T >
typedef TSafeIterator<T*, T&> Common::TVector< T >::CSafeIterator
```

Iterator that does bounds checking and throws OutOfRange().

6.6.2.8 CSafeReverseIterator

```
template<typename T >
typedef TSafeReverseIterator<T*, T&> Common::TVector< T >::CSafeReverseIterator
```

Reverse iterator that can throw OutOfRange().

6.6.3 Member Enumeration Documentation

6.6.3.1 EReservedCapacityRule

```
template<typename T >
enum Common::TVector::EReservedCapacityRule :  uint8_t  [strong]
```

Used to define how methods deal with Capacity.

Enumerator

Exponential	[ADD] allocates (NewSize-1)∗2 if capacity exceed and vector is not small enough [DEL] allocates 2∗Size if Capacity > 4 ∗ Size and vector is not small enough
Linear	[ADD] allocates NewSize + 3 + 32 / sizeof(T) if capacity exceed [DEL] allocates Size + 4 + 32 / sizeof(T) if size exceeds capacity by more than 2∗(8 + 64 / sizeof(T))
NeverReserve	[ADD, DEL] memory is never reserved automatically; if true bAllowAutoShrink was passed into another method, then ShrinkToFit() will be called

6.6.4 Constructor & Destructor Documentation

6.6.4.1 TVector() [1/8]

```
template<typename T >
Common::TVector< T >::TVector ( )  [default]
```

Empty vector with no heap allocation.

6.6.4.2 TVector() [2/8]

```
template<typename T >
Common::TVector< T >::TVector (
            EReservedCapacityRule CapacityRule )  [noexcept]
```

Creates empty vector with Capacity preset predefined.

Parameters

CapacityRule	Describes how memory will be reserved

See also

EReservedCapacityRule for more info about presets.

6.6.4.3 TVector() [3/8]

```
template<typename T >
Common::TVector< T >::TVector (
            size_t Size,
            const T & DefaultValue = {} )  [explicit]
```

Vector with pre-allocated elements.

Parameters

Size	Number of elements to allocate
DefaultValue	Optional. Value to initialize with

See also

Use Reserve() on empty vector to avoid initialization.

6.6.4.4 TVector() [4/8]

```
template<typename T >
Common::TVector< T >::TVector (
            size_t Size,
            const T *const Array )
```

Vector constructed from raw dynamic array (copy).

Parameters

Size	Number of elements in original array
Array	Pointer to heap with C-style array

Note

Array[0] to Array[Size-1] must exist and have the same type as vector.

Raw array is not changed.

6.6.4.5 TVector() [5/8]

```
template<typename T >
Common::TVector< T >::TVector (
            const std::initializer_list< T > & ValuesList )
```

Modern C++ initialization syntax: name = {...}.

Parameters

ValuesList	Initializer list

6.6.4.6 TVector() [6/8]

```
template<typename T >
template<typename IteratorType >
Common::TVector< T >::TVector (
            IteratorType Begin,
            IteratorType End,
            typename std::enable_if<!std::is_integral< IteratorType >::value >::type *  = 0
)
```

Constructor to get values from another container.

Template Parameters

IteratorType	Iterator that implements ++, != and *

Parameters

Begin	Iterator referring to the beginning of container
End	Iterator referring to the end of container

6.6.4.7 TVector() [7/8]

```
template<typename T >
Common::TVector< T >::TVector (
            const TVector< T > & Other )
```

Initialize by copying another TVector.

Parameters

Other	vector to copy

6.6.4.8 TVector() [8/8]

```
template<typename T >
Common::TVector< T >::TVector (
            TVector< T > && Other )  [noexcept]
```

Move constructor.

Parameters

Other	Temporary object to get data from

6.6.4.9 ~TVector()

```
template<typename T >
Common::TVector< T >::~TVector ( )
```

6.6.5 Member Function Documentation

6.6.5.1 Assign()

```
template<typename T >
template<typename IteratorType >
void Common::TVector< T >::Assign (
            IteratorType Begin,
            IteratorType End,
            bool bAllowAutoShrink = false )
```

Allows to copy values from another container.

Template Parameters

IteratorType	Iterator that implements ++, != and *

Parameters

Begin	Iterator referring to the beginning of container
End	Iterator referring to the end of container

6.6.5.2 AutoAt()

```
template<typename T >
T& Common::TVector< T >::AutoAt (
            size_t Index,
            const T & DefaultValue = {} )
```

If element does not exist, this will resize vector and fill newly created elements with provided value.

Parameters

Index	aka offset value
DefaultValue	Value to initialize added elements

Returns

Reference to the requested element

See also

Use operator [] if you are sure, that element exists

6.6.5.3 Back() [1/2]

```
template<typename T >
T& Common::TVector< T >::Back ( )
```

Provides access to the last element.

Returns

Reference to the last element

Note

Vector must not be empty.

6.6.5.4 Back() [2/2]

```
template<typename T >
const T& Common::TVector< T >::Back ( ) const  [inline]
```

Back() for const vectors.

6.6.5.5 Begin()

```
template<typename T >
CIterator Common::TVector< T >::Begin ( )
```

Iterator pointing to the first elem.

Returns

CIterator iterator

6.6.5.6 begin() [1/2]

```
template<typename T >
CIterator Common::TVector< T >::begin ( )  [inline]
```

Begin() alias (for compatibity)

6.6.5.7 begin() [2/2]

```
template<typename T >
CConstIterator Common::TVector< T >::begin ( ) const  [inline]
```

ConstBegin() alias (overloaded, for compatibity)

6.6.5.8 cbegin()

```
template<typename T >
CConstIterator Common::TVector< T >::cbegin ( ) const  [inline]
```

ConstBegin() alias (for compatibity)

6.6.5.9 cend()

```
template<typename T >
CConstIterator Common::TVector< T >::cend ( ) const  [inline]
```

ConstEnd() alias (for compatibity)

6.6.5.10 Clear()

```
template<typename T >
void Common::TVector< T >::Clear ( )
```

Removes all elements from the vector.

6.6.5.11 ConstBegin()

```
template<typename T >
CConstIterator Common::TVector< T >::ConstBegin ( ) const
```

Iterator pointing to the first elem (const).

Returns

CConstIterator iterator

6.6.5.12 ConstEnd()

```
template<typename T >
CConstIterator Common::TVector< T >::ConstEnd ( ) const
```

Iterator pointing to the elem after last (const).

Returns

CConstIterator iterator

6.6.5.13 ConstReverseBegin()

```
template<typename T >
CConstReverseIterator Common::TVector< T >::ConstReverseBegin ( ) const
```

Reverse iterator pointing to the first elem (const).

Returns

CConstReverseIterator iterator

6.6.5.14 ConstReverseEnd()

```
template<typename T >
CConstReverseIterator Common::TVector< T >::ConstReverseEnd ( ) const
```

Reverse iterator pointing to the elem after last (const).

Returns

CConstReverseIterator iterator

6.6.5.15 crbegin()

```
template<typename T >
CConstReverseIterator Common::TVector< T >::crbegin ( ) const  [inline]
```

ConstReverseBegin() alias (for compatibity)

6.6.5.16 crend()

```
template<typename T >
CConstReverseIterator Common::TVector< T >::crend ( ) const  [inline]
```

ConstReverseEnd() alias (for compatibity)

6.6.5.17 End()

```
template<typename T >
CIterator Common::TVector< T >::End ( )
```

Iterator pointing to the elem after last.

Returns

CIterator iterator

6.6.5.18 end() [1/2]

```
template<typename T >
CIterator Common::TVector< T >::end ( )  [inline]
```

End() alias (for compatibity)

6.6.5.19 end() [2/2]

```
template<typename T >
CConstIterator Common::TVector< T >::end ( ) const  [inline]
```

ConstEnd() alias (overloaded, for compatibity)

6.6.5.20 Erase()

```
template<typename T >
void Common::TVector< T >::Erase (
            size_t Position,
            bool bAllowAutoShrink = false )
```

Removes element with specified position.

Parameters

Position	Position of element to be removed
bAllowAutoShrink	Enables auto shrink according to the capacity rule. See: EReservedCapacityRule

Note

If Position is greater than max index, does nothing.

Warning

This method removes one element. To remove multiple, use EraseMultiple(). Your code with such a mistake will be compiled because of optional bool param.

6.6.5.21 EraseMultiple()

```
template<typename T >
void Common::TVector< T >::EraseMultiple (
            size_t PositionFrom,
            size_t PositionTo,
            bool bAllowAutoShrink = false )
```

Removes range of elements from vector.

Parameters

PositionFrom	Starting index for erase
PositionTo	End point for erase. Element with this index will also be removed
bAllowAutoShrink	Optional. Enables auto shrink according to the capacity rule. See: EReservedCapacityRule

Note

Ignores elements at unavailable positions.

Warning

This method removes multiple elements. To remove one, use Erase(). Your code with such a mistake will be compiled because of optional bool param.

6.6.5.22 Front() [1/2]

```
template<typename T >
T& Common::TVector< T >::Front ( )
```

Provides access to the first element.

Returns

Reference to the first element

Note

Vector must not be empty.

6.6.5.23 Front() [2/2]

```
template<typename T >
const T& Common::TVector< T >::Front ( ) const  [inline]
```

Front() for const vectors.

6.6.5.24 GetCapacity()

```
template<typename T >
size_t Common::TVector< T >::GetCapacity ( ) const  [noexcept]
```

Capacity is Size + reserved space for the future use.

Returns

Capacity of vector

6.6.5.25 GetCapacityRule()

```
template<typename T >
EReservedCapacityRule Common::TVector< T >::GetCapacityRule ( ) const  [noexcept]
```

Returns Capacity rule that is currently applied. It affects how elements are allocated & destructed.

Returns

EReservedCapacityRule Current capacity rule

See also

EReservedCapacityRule for more info about presets.

6.6.5.26 GetSize()

```
template<typename T >
size_t Common::TVector< T >::GetSize ( ) const  [noexcept]
```

Size is number of elements that you can use.

Returns

Size of vector

6.6.5.27 Insert() [1/2]

```
template<typename T >
void Common::TVector< T >::Insert (
            size_t Position,
            const T & Value = T{},
            const T & FillOnResizeWith = T{} )
```

Inserts one elements to the specified position.

Parameters

Position	Index where to insert
Value	Value to insert
FillOnResizeWith	If position is larger than vector size, elements that appear on resize will be initialized with this value

Note

If Position is greater than max index, vector is resized.

6.6.5.28 Insert() [2/2]

```
template<typename T >
template<typename IteratorType >
void Common::TVector< T >::Insert (
            size_t Position,
            IteratorType Begin,
            IteratorType End,
            const T & FillOnResizeWith = T{},
            typename std::enable_if<!std::is_integral< IteratorType >::value >::type *  = 0
)
```

Inserts range of elements, starting at Position.

Template Parameters

IteratorType	Iterator with implemented ++, != and *

Parameters

Position	Index of the first inserted element
Begin	Iterator referring to the first element
End	Iterator referring to the element after last one
FillOnResizeWith	If position is larger than vector size, elements that appear on resize will be initialized with this value

Note

If Position is greater than max index, vector is resized.

6.6.5.29 IsEmpty()

```
template<typename T >
bool Common::TVector< T >::IsEmpty ( ) const  [noexcept]
```

Simple check if size of this vector equals 0.

Returns

True if empty, false if not

6.6.5.30 operator"!=()

```
template<typename T >
bool Common::TVector< T >::operator!= (
            const TVector< T > & Other ) const  [noexcept]
```

Opposite to operator ==.

6.6.5.31 operator=() [1/3]

```
template<typename T >
TVector<T>& Common::TVector< T >::operator= (
            const std::initializer_list< T > & ValuesList )
```

Allows assignment with = {...} style.

Parameters

ValuesList	Initializer list

Returns

Reference to this vector

6.6.5.32 operator=() [2/3]

```
template<typename T >
TVector<T>& Common::TVector< T >::operator= (
            const TVector< T > & Other )
```

Assingment operator makes a copy of another vector.

Parameters

Other	vector

Returns

Reference to this vector

6.6.5.33 operator=() [3/3]

```
template<typename T >
TVector<T>& Common::TVector< T >::operator= (
            TVector< T > && Other )  [noexcept]
```

Move assignment.

Parameters

Other	Temporary object to get data from

Returns

Reference to this vector

6.6.5.34 operator==()

```
template<typename T >
bool Common::TVector< T >::operator== (
            const TVector< T > & Other ) const  [noexcept]
```

Checks if two vectors contain the same values by calling equality operator recursively.

Parameters

Other	Other vector to compare

Returns

True if sizes and values are equal, false otherwise

Note

Containing elements must implement == operator

6.6.5.35 operator[]() [1/2]

```
template<typename T >
T& Common::TVector< T >::operator[] (
            size_t Index )
```

Index operator that prodives access to the element.

Parameters

Index	aka offset value

Returns

Reference to the requested element

Note

Element with requested index must exist in vector.

6.6.5.36 operator[]() [2/2]

```
template<typename T >
const T& Common::TVector< T >::operator[] (
            size_t Index ) const  [inline]
```

Index operator for const vectors.

6.6.5.37 Pop()

```
template<typename T >
T Common::TVector< T >::Pop (
            bool bAllowAutoShrink = false )
```

Removes one element from the end of vector.

Parameters

bAllowAutoShrink	Optional. Enables auto shrink according to the capacity rule. See: EReservedCapacityRule

Returns

Removed element (by value)

Note

Vector must not be empty.

6.6.5.38 PopMultiple()

```
template<typename T >
void Common::TVector< T >::PopMultiple (
            size_t ElementsCount,
            bool bAllowAutoShrink = false )
```

Removes N elements from the end of vector.

Parameters

ElementsCount	Number of elements to be removed
bAllowAutoShrink	Optional. Enables auto shrink according to the capacity rule. See: EReservedCapacityRule

Note

If ElementsCount >= Size, clears vector

6.6.5.39 Push() [1/2]

```
template<typename T >
void Common::TVector< T >::Push (
            const T & Value )
```

Adds one element to the end of vector.

Parameters

Value	Element to add

See also

Call ShrinkToFit() to clear reserved memory, Reserve() to increase it

6.6.5.40 Push() [2/2]

```
template<typename T >
template<typename IteratorType >
void Common::TVector< T >::Push (
            IteratorType Begin,
            IteratorType End )
```

Adds elements to the end, uses iterators.

Template Parameters

IteratorType	Iterator with implemented ++, != and *

Parameters

Begin	Iterator referring to the first element
End	Iterator referring to the element after last one

6.6.5.41 RawData() [1/2]

```
template<typename T >
const T* Common::TVector< T >::RawData ( ) const  [inline], [noexcept]
```

RawData() for const vectors.

6.6.5.42 RawData() [2/2]

```
template<typename T >
T* Common::TVector< T >::RawData ( )  [noexcept]
```

Provides access to the internal buffer.

Returns

Pointer to c-style heap array

6.6.5.43 rbegin() [1/2]

```
template<typename T >
CReverseIterator Common::TVector< T >::rbegin ( )  [inline]
```

ReverseBegin() alias (for compatibity)

6.6.5.44 rbegin() [2/2]

```
template<typename T >
CConstReverseIterator Common::TVector< T >::rbegin ( ) const  [inline]
```

ConstReverseBegin() alias (overloaded, for compatibity)

6.6.5.45 rend() [1/2]

```
template<typename T >
CReverseIterator Common::TVector< T >::rend ( )  [inline]
```

ReverseEnd() alias (for compatibity)

6.6.5.46 rend() [2/2]

```
template<typename T >
CConstReverseIterator Common::TVector< T >::rend ( ) const  [inline]
```

ConstReverseEnd() alias (overloaded, for compatibity)

6.6.5.47 Reserve()

```
template<typename T >
void Common::TVector< T >::Reserve (
            size_t NewCapacity )
```

Allocates elements internally for the future use.

Parameters

NewCapacity	If greater than size, will update internal capacity

See also

Call ShrinkToFit() to clear reserved memory.

Note

Passing bAllowAutoShrink to methods may cause shrink.

6.6.5.48 Resize()

```
template<typename T >
void Common::TVector< T >::Resize (
            size_t NewSize,
            const T & DefaultValue = {},
            bool bAllowAutoShrink = false )
```

Changes size of vector.

Parameters

NewSize	New size of vector. If NewSize < Size, deletes last elements. Otherwise, creates new with passed value.
DefaultValue	Optional. Value to initialize added elements
bAllowAutoShrink	Optional. Enables auto shrink according to the capacity rule. See: EReservedCapacityRule

6.6.5.49 ReverseBegin()

```
template<typename T >
CReverseIterator Common::TVector< T >::ReverseBegin ( )
```

Reverse iterator pointing to the first elem.

Returns

CReverseIterator iterator

6.6.5.50 ReverseEnd()

```
template<typename T >
CReverseIterator Common::TVector< T >::ReverseEnd ( )
```

Reverse iterator pointing to the elem after last.

Returns

CReverseIterator iterator

6.6.5.51 SafeAt() [1/2]

```
template<typename T >
T& Common::TVector< T >::SafeAt (
            size_t Index )
```

[] with range check.

6.6.5.52 SafeAt() [2/2]

```
template<typename T >
const T& Common::TVector< T >::SafeAt (
            size_t Index ) const  [inline]
```

6.6.5.53 SafeBack() [1/2]

```
template<typename T >
T& Common::TVector< T >::SafeBack ( )
```

Back() with range check.

6.6.5.54 SafeBack() [2/2]

```
template<typename T >
const T& Common::TVector< T >::SafeBack ( ) const  [inline]
```

SafeBack() for const vectors.

6.6.5.55 SafeBegin()

```
template<typename T >
CSafeIterator Common::TVector< T >::SafeBegin ( )
```

Safe iterator pointing to the first elem.

Returns

CSafeIterator iterator

6.6.5.56 SafeConstBegin()

```
template<typename T >
CSafeConstIterator Common::TVector< T >::SafeConstBegin ( ) const
```

Safe iterator pointing to the first elem (const).

Returns

CSafeConstIterator iterator

6.6.5.57 SafeConstEnd()

```
template<typename T >
CSafeConstIterator Common::TVector< T >::SafeConstEnd ( ) const
```

Safe iterator pointing to the elem after last (const).

Returns

CSafeConstIterator iterator

6.6.5.58 SafeConstReverseBegin()

```
template<typename T >
CSafeConstReverseIterator Common::TVector< T >::SafeConstReverseBegin ( ) const
```

Safe reverse iterator pointing to the first elem (const).

Returns

CSafeConstReverseIterator

6.6.5.59 SafeConstReverseEnd()

```
template<typename T >
CSafeConstReverseIterator Common::TVector< T >::SafeConstReverseEnd ( ) const
```

Safe reverse it. pointing to elem after last (const).

Returns

CSafeConstReverseIterator

6.6.5.60 SafeEnd()

```
template<typename T >
CSafeIterator Common::TVector< T >::SafeEnd ( )
```

Safe iterator pointing to the elem after last.

Returns

CSafeIterator iterator

6.6.5.61 SafeFront() [1/2]

```
template<typename T >
T& Common::TVector< T >::SafeFront ( )
```

Front() with range check.

6.6.5.62 SafeFront() [2/2]

```
template<typename T >
const T& Common::TVector< T >::SafeFront ( ) const  [inline]
```

SafeFront() for const vectors.

6.6.5.63 SafePop()

```
template<typename T >
T Common::TVector< T >::SafePop (
            bool bAllowAutoShrink = false )
```

Pop() with range check.

6.6.5.64 SafeReverseBegin()

```
template<typename T >
CSafeReverseIterator Common::TVector< T >::SafeReverseBegin ( )
```

Safe reverse iterator pointing to the first elem.

Returns

CSafeReverseIterator iterator

6.6.5.65 SafeReverseEnd()

```
template<typename T >
CSafeReverseIterator Common::TVector< T >::SafeReverseEnd ( )
```

Safe reverse iterator pointing to the elem after last.

Returns

CSafeReverseIterator iterator

6.6.5.66 SafeShift()

```
template<typename T >
T Common::TVector< T >::SafeShift (
            bool bAllowAutoShrink = false )
```

Shift() with range check.

6.6.5.67 SetCapacityRule()

```
template<typename T >
void Common::TVector< T >::SetCapacityRule (
            EReservedCapacityRule CapacityRule )  [noexcept]
```

Set enum value that will describe how re-allocation works when elements are added / removed from vertor. Reallocation on removal happens only if bool bAllowAutoShrink was passed with supported operation.

Parameters

CapacityRule	Preset value from EReservedCapacityRule

Note

If bAllowAutoShrink was passed with operation, size that you have manually reserved may also be deallocated

6.6.5.68 Shift()

```
template<typename T >
T Common::TVector< T >::Shift (
            bool bAllowAutoShrink = false )
```

Removes one element from the beginning of vector.

Parameters

bAllowAutoShrink	Optional. Enables auto shrink according to the capacity rule. See: EReservedCapacityRule

Returns

Removed element (by value)

6.6.5.69 ShiftMultiple()

```
template<typename T >
void Common::TVector< T >::ShiftMultiple (
            size_t ElementsCount,
            bool bAllowAutoShrink = false )
```

Removes N elements from the beginning of vector.

Parameters

ElementsCount	Number of elements to be removed
bAllowAutoShrink	Enables auto shrink according to the capacity rule. See: EReservedCapacityRule

Note

If ElementsCount >= Size, clears vector

6.6.5.70 ShrinkToFit()

```
template<typename T >
void Common::TVector< T >::ShrinkToFit ( )
```

Clears memory that was reserved for future use.

See also

Methods that can reserve memory: push(), reserve(), etc.

6.6.5.71 Swap()

```
template<typename T >
static void Common::TVector< T >::Swap (
            TVector< T > & Vector1,
            TVector< T > & Vector2 )  [static], [noexcept]
```

Swaps two vectors internally without deep copy.

Parameters

Vector1	First vector
Vector2	Second vector

Chapter 7

File Documentation

7.1 CommonTypes/Exception.h File Reference

```
#include "Pair.h"
#include "../CommonUtils/RawString.h"
```

Classes

- class Common::CException

  Basic exception class. C-style message is required.
- class Common::COutOfRange

  Represents "Out of Range" error. Can hold message, requested and expected indices.
- class Common::CDoesNotExist

  Represents "Element does not exist" error. Stores message.

Namespaces

- Common

7.2 CommonTypes/Optional.h File Reference

```
#include "Exception.h"
```

Classes

- class Common::TOptional< T >

  Represents object that may not exist.

Namespaces

- Common

7.3 CommonTypes/Pair.h File Reference

Classes

- class Common::TPair< T1, T2 >

  Represents a pair of objects.

Namespaces

- Common

7.4 CommonTypes/Private/Vector.tpp File Reference

7.5 CommonTypes/Private/VectorIterators.tpp File Reference

7.6 CommonTypes/Vector.h File Reference

```
#include <initializer_list>
#include <type_traits>
#include <new>
#include "Exception.h"
#include "../CommonUtils/Assert.h"
#include "../CommonUtils/AdvancedIteration.h"
#include "Private/VectorIterators.tpp"
#include "Private/Vector.tpp"
```

Classes

- class Common::TVector< T >

  Container representing array that can change its size.

Namespaces

- Common

7.7 CommonUtils/AdvancedIteration.h File Reference

```
#include "Private/AdvancedIteration.tpp"
```

Namespaces

- Common

Functions

- template<typename IteratorType >
  size_t Common::GetIteratorDistance (IteratorType Begin, IteratorType End)

  Counts elements between two iterators. Range: [Begin: End)

7.8 CommonUtils/Assert.h File Reference

```
#include <iostream>
```

Macros

- #define ASSERT(Condition, Message)

7.8.1 Macro Definition Documentation

7.8.1.1 ASSERT

```
#define ASSERT(
               Condition,
               Message )
```

Value:

```
    do { \
        if (!(Condition)) { \
            std::cerr « "ASSERTION (" #Condition ") FAILED in " « __FILE__ \
                « ", line " « __LINE__ « ": " « (Message) « std::endl; \
            std::abort(); \
        } \
    } while (false)
```

7.9 CommonUtils/Private/AdvancedIteration.tpp File Reference

7.10 CommonUtils/Private/RawString.cpp File Reference

```
#include "../RawString.h"
```

Namespaces

- Common

Functions

- size_t Common::GetRawStringLength (const char *const NullTermString)

  Calculates length of the C-string.
- size_t Common::GetRawStringLength (const char *const NullTermString, size_t MaxLength)

  Calculates length of the C-string. Stops when null character is reached or MaxLength elements have been counted. Useful with malformatted input.
- void Common::CopyRawString (const char *const NullTermStringFrom, char *const RawStringTo)

  Does the copy of C-style string (ended with '\0').
- void Common::CopyRawString (const char *const NullTermStringFrom, char *const RawStringTo, size_t MaxLength)

  Does the copy of C-style string. Stops when null character is reached or MaxLength elements have been copied. Useful when string should be cut.
- bool Common::AreRawStringsEqual (const char *const NullTermString1, const char *const NullTermString2)

  Checks whether two C-style strings are equal.
- bool Common::AreRawStringsEqual (const char *const NullTermString1, const char *const NullTermString2, size_t MaxCompareLength)

  Checks whether two C-style strings are equal. Use this to compare N first elements, even if strings are not null-terminated.

7.11 CommonUtils/RawString.h File Reference

Namespaces

- Common

Functions

- size_t Common::GetRawStringLength (const char *const NullTermString)

  Calculates length of the C-string.
- size_t Common::GetRawStringLength (const char *const NullTermString, size_t MaxLength)

  Calculates length of the C-string. Stops when null character is reached or MaxLength elements have been counted. Useful with malformatted input.
- void Common::CopyRawString (const char *const NullTermStringFrom, char *const RawStringTo)

  Does the copy of C-style string (ended with '\0').
- void Common::CopyRawString (const char *const NullTermStringFrom, char *const RawStringTo, size_t MaxLength)

  Does the copy of C-style string. Stops when null character is reached or MaxLength elements have been copied. Useful when string should be cut.
- bool Common::AreRawStringsEqual (const char *const NullTermString1, const char *const NullTermString2)

  Checks whether two C-style strings are equal.
- bool Common::AreRawStringsEqual (const char *const NullTermString1, const char *const NullTermString2, size_t MaxCompareLength)

  Checks whether two C-style strings are equal. Use this to compare N first elements, even if strings are not null-terminated.

7.12 x64/Debug/CommonLibs.vcxproj.FileListAbsolute.txt File Reference

Index