

# CommonLibs

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

Yuri Zamyatin, 2020



<b>1 Namespace Index</b>	<b>1</b>
1.1 Namespace List	1
<b>2 Hierarchical Index</b>	<b>3</b>
2.1 Class Hierarchy	3
<b>3 Class Index</b>	<b>5</b>
3.1 Class List	5
<b>4 File Index</b>	<b>7</b>
4.1 File List	7
<b>5 Namespace Documentation</b>	<b>9</b>
5.1 Common Namespace Reference	9
5.1.1 Function Documentation	10
5.1.1.1 AreRawStringsEqual() [1/2]	10
5.1.1.2 AreRawStringsEqual() [2/2]	10
5.1.1.3 CopyRawString() [1/2]	10
5.1.1.4 CopyRawString() [2/2]	11
5.1.1.5 GetIteratorDistance()	11
5.1.1.6 GetRawStringLength() [1/2]	12
5.1.1.7 GetRawStringLength() [2/2]	12
<b>6 Class Documentation</b>	<b>13</b>
6.1 Common::CDoesNotExist Class Reference	13
6.1.1 Detailed Description	13
6.1.2 Constructor & Destructor Documentation	13
6.1.2.1 CDoesNotExist()	13
6.2 Common::CException Class Reference	14
6.2.1 Detailed Description	14
6.2.2 Constructor & Destructor Documentation	14
6.2.2.1 CException()	14
6.2.2.2 ~CException()	15
6.2.3 Member Function Documentation	15
6.2.3.1 GetMessage()	15
6.2.4 Member Data Documentation	15
6.2.4.1 Message	15
6.3 Common::COutOfRange Class Reference	15
6.3.1 Detailed Description	16
6.3.2 Constructor & Destructor Documentation	16
6.3.2.1 COutOfRange() [1/2]	16
6.3.2.2 COutOfRange() [2/2]	16
6.3.3 Member Function Documentation	17
6.3.3.1 GetExpectedRange()	17

6.3.3.2 GetRequestedIndex()	17
6.3.4 Member Data Documentation	18
6.3.4.1 ExpectedIndex	18
6.3.4.2 RequestedIndex	18
6.4 Common::TOptional< T > Class Template Reference	18
6.4.1 Detailed Description	18
6.4.2 Constructor & Destructor Documentation	19
6.4.2.1 TOptional() [1/2]	19
6.4.2.2 TOptional() [2/2]	19
6.4.3 Member Function Documentation	19
6.4.3.1 GetValue()	19
6.4.3.2 GetValueOr()	19
6.4.3.3 operator=()	20
6.4.3.4 SetValue()	20
6.4.3.5 ValueExists()	20
6.5 Common::TPair< T1, T2 > Class Template Reference	21
6.5.1 Detailed Description	21
6.5.2 Constructor & Destructor Documentation	21
6.5.2.1 TPair() [1/2]	21
6.5.2.2 TPair() [2/2]	21
6.5.3 Member Function Documentation	22
6.5.3.1 MakePair()	22
6.5.4 Member Data Documentation	22
6.5.4.1 First	22
6.5.4.2 Second	22
6.6 Common::TVector< T > Class Template Reference	23
6.6.1 Detailed Description	27
6.6.2 Member Typedef Documentation	27
6.6.2.1 CConstIterator	27
6.6.2.2 CConstReverselIterator	27
6.6.2.3 CIterator	27
6.6.2.4 CReverselIterator	28
6.6.2.5 CSafeConstIterator	28
6.6.2.6 CSafeConstReverselIterator	28
6.6.2.7 CSafeIterator	28
6.6.2.8 CSafeReverselIterator	28
6.6.3 Member Enumeration Documentation	28
6.6.3.1 EReservedCapacityRule	28
6.6.4 Constructor & Destructor Documentation	29
6.6.4.1 TVector() [1/8]	29
6.6.4.2 TVector() [2/8]	29
6.6.4.3 TVector() [3/8]	29

6.6.4.4 TVector() [4/8]	30
6.6.4.5 TVector() [5/8]	30
6.6.4.6 TVector() [6/8]	31
6.6.4.7 TVector() [7/8]	31
6.6.4.8 TVector() [8/8]	31
6.6.4.9 ~TVector()	32
6.6.5 Member Function Documentation	32
6.6.5.1 Assign()	32
6.6.5.2 AutoAt()	32
6.6.5.3 Back() [1/2]	33
6.6.5.4 Back() [2/2]	33
6.6.5.5 Begin()	33
6.6.5.6 begin() [1/2]	34
6.6.5.7 begin() [2/2]	34
6.6.5.8 cbegin()	34
6.6.5.9 cend()	34
6.6.5.10 Clear()	34
6.6.5.11 ConstBegin()	35
6.6.5.12 ConstEnd()	35
6.6.5.13 ConstReverseBegin()	35
6.6.5.14 ConstReverseEnd()	35
6.6.5.15 crbegin()	36
6.6.5.16 crend()	36
6.6.5.17 End()	36
6.6.5.18 end() [1/2]	36
6.6.5.19 end() [2/2]	36
6.6.5.20 Erase()	36
6.6.5.21 EraseMultiple()	37
6.6.5.22 Front() [1/2]	37
6.6.5.23 Front() [2/2]	38
6.6.5.24 GetCapacity()	38
6.6.5.25 GetCapacityRule()	38
6.6.5.26 GetSize()	39
6.6.5.27 Insert() [1/2]	39
6.6.5.28 Insert() [2/2]	39
6.6.5.29 IsEmpty()	40
6.6.5.30 operator!=(())	40
6.6.5.31 operator=() [1/3]	40
6.6.5.32 operator=() [2/3]	41
6.6.5.33 operator=() [3/3]	41
6.6.5.34 operator==(())	42
6.6.5.35 operator[]() [1/2]	42

6.6.5.36 operator[]() [2/2]	42
6.6.5.37 Pop()	43
6.6.5.38 PopMultiple()	43
6.6.5.39 Push() [1/2]	43
6.6.5.40 Push() [2/2]	44
6.6.5.41 RawData() [1/2]	44
6.6.5.42 RawData() [2/2]	44
6.6.5.43 rbegin() [1/2]	45
6.6.5.44 rbegin() [2/2]	45
6.6.5.45 rend() [1/2]	45
6.6.5.46 rend() [2/2]	45
6.6.5.47 Reserve()	45
6.6.5.48 Resize()	46
6.6.5.49 ReverseBegin()	46
6.6.5.50 ReverseEnd()	46
6.6.5.51 SafeAt() [1/2]	47
6.6.5.52 SafeAt() [2/2]	47
6.6.5.53 SafeBack() [1/2]	47
6.6.5.54 SafeBack() [2/2]	47
6.6.5.55 SafeBegin()	47
6.6.5.56 SafeConstBegin()	48
6.6.5.57 SafeConstEnd()	48
6.6.5.58 SafeConstReverseBegin()	48
6.6.5.59 SafeConstReverseEnd()	48
6.6.5.60 SafeEnd()	49
6.6.5.61 SafeFront() [1/2]	49
6.6.5.62 SafeFront() [2/2]	49
6.6.5.63 SafePop()	49
6.6.5.64 SafeReverseBegin()	49
6.6.5.65 SafeReverseEnd()	50
6.6.5.66 SafeShift()	50
6.6.5.67 SetCapacityRule()	50
6.6.5.68 Shift()	50
6.6.5.69 ShiftMultiple()	51
6.6.5.70 ShrinkToFit()	51
6.6.5.71 Swap()	51
<b>7 File Documentation</b>	<b>53</b>
7.1 CommonTypes/Exception.h File Reference	53
7.2 CommonTypes/Optional.h File Reference	53
7.3 CommonTypes/Pair.h File Reference	54
7.4 CommonTypes/Private/Vector.tpp File Reference	54

---

7.5 CommonTypes/Private/VectorIterators.hpp File Reference . . . . .	54
7.6 CommonTypes/Vector.h File Reference . . . . .	54
7.7 CommonUtils/AdvancedIteration.h File Reference . . . . .	54
7.8 CommonUtils/Assert.h File Reference . . . . .	55
7.8.1 Macro Definition Documentation . . . . .	55
7.8.1.1 ASSERT . . . . .	55
7.9 CommonUtils/Private/AdvancedIteration.hpp File Reference . . . . .	55
7.10 CommonUtils/Private/RawString.cpp File Reference . . . . .	55
7.11 CommonUtils/RawString.h File Reference . . . . .	56
7.12 x64/Debug/CommonLibs.vcxproj.FileListAbsolute.txt File Reference . . . . .	56
<b>Index</b>	<b>57</b>





# Chapter 1

## Namespace Index

### 1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

<a href="#">Common</a>	.....	9
------------------------	-------	---



## Chapter 2

# Hierarchical Index

### 2.1 Class Hierarchy

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

Common::CException . . . . .	14
Common::CDoesNotExist . . . . .	13
Common::COutOfRange . . . . .	15
Common::TOptional< T > . . . . .	18
Common::TPair< T1, T2 > . . . . .	21
Common::TPair< size_t, size_t > . . . . .	21
Common::TVector< T > . . . . .	23



## Chapter 3

# Class Index

### 3.1 Class List

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

<a href="#">Common::CDoesNotExist</a>	
Represents "Element does not exist" error. Stores message . . . . .	13
<a href="#">Common::CException</a>	
Basic exception class. C-style message is required . . . . .	14
<a href="#">Common::COutOfRange</a>	
Represents "Out of Range" error. Can hold message, requested and expected indices . . . . .	15
<a href="#">Common::TOptional&lt; T &gt;</a>	
Represents object that may not exist . . . . .	18
<a href="#">Common::TPair&lt; T1, T2 &gt;</a>	
Represents a pair of objects . . . . .	21
<a href="#">Common::TVector&lt; T &gt;</a>	
Container representing array that can change its size . . . . .	23



## Chapter 4

# File Index

### 4.1 File List

Here is a list of all files with brief descriptions:

CommonTypes/ <a href="#">Exception.h</a> . . . . .	53
CommonTypes/ <a href="#">Optional.h</a> . . . . .	53
CommonTypes/ <a href="#">Pair.h</a> . . . . .	54
CommonTypes/ <a href="#">Vector.h</a> . . . . .	54
CommonTypes/Private/ <a href="#">Vector.hpp</a> . . . . .	54
CommonTypes/Private/ <a href="#">VectorIterators.hpp</a> . . . . .	54
CommonUtils/ <a href="#">AdvancedIteration.h</a> . . . . .	54
CommonUtils/ <a href="#">Assert.h</a> . . . . .	55
CommonUtils/ <a href="#">RawString.h</a> . . . . .	56
CommonUtils/Private/ <a href="#">AdvancedIteration.hpp</a> . . . . .	55
CommonUtils/Private/ <a href="#">RawString.cpp</a> . . . . .	55





## 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 malformed 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

<i>NullTermString1</i>	First null-terminated string
<i>NullTermString2</i>	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

<i>NullTermString1</i>	First string (whether null- terminated or limited with MaxCompareLength)
<i>NullTermString2</i>	Second string (whether null- terminated or limited with MaxCompareLength)
<i>MaxComparedLength</i>	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

<i>NullTermStringFrom</i>	Source: char array that ends with '\0'
<i>NullTermStringTo</i>	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

<i>NullTermStringFrom</i>	Source: char array that ends with '\0' (or not, if you rely on MaxLength and buffer size)
<i>NullTermStringTo</i>	Destination: Char array that is large enough to receive copied elements. May not end with '\0'. After copying it gets '\0' anyway
<i>MaxLength</i>	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

<i>IteratorType</i>	Iterator with implemented ++, != and *
---------------------	--

#### Parameters

<i>Begin</i>	Iterator referring to the first element
<i>End</i>	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**

<i>NullTermString</i>	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 malformed input.

**Parameters**

<i>NullTermString</i>	Char array that ends with '\0' (or not, if you rely on <i>MaxLength</i> and buffer size)
<i>MaxLength</i>	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

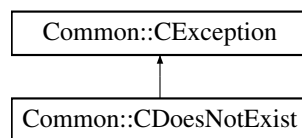
# Class Documentation

### 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

<i>Message</i>	Description. Will be copied to the inner buffer
----------------	---

## Note

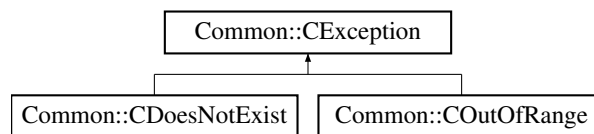
If length of message > 40, first 40 symbols will be saved.

## 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**

<i>Message</i>	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**

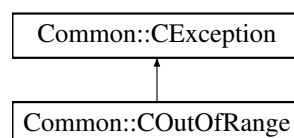
```
char Common::CException::Message[41] [protected]
```

**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

<i>Message</i>	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

<i>Message</i>	Description. Will be copied to an inner buffer
<i>RequestedIndex</i>	Errored index (out of range).
<i>ExpectedIndex</i>	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

<i>Value</i>	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**

<i>OtherVariant</i>	
---------------------	--

**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**

<i>Value</i>	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**

<i>Value</i>	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**

<i>First</i>	First value in pair
<i>Second</i>	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**

<i>First</i>	First value in pair
<i>Second</i>	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 TReverselIterator< T \*, T & > [CReverselIterator](#)  
*Reverse iterator. Increment is actually decrement, etc.*
- typedef TReverselIterator< const T \*, const T & > [CConstReverselIterator](#)  
*Version of TReverselIterator for const values.*
- typedef TSafelIterator< T \*, T & > [CSafelIterator](#)  
*Iterator that does bounds checking and throws OutOfRange().*
- typedef TSafelIterator< const T \*, const T & > [CSafeConstIterator](#)  
*Version of TSafelIterator for const values.*
- typedef TSafeReverselIterator< T \*, T & > [CSafeReverselIterator](#)  
*Reverse iterator that can throw OutOfRange().*
- typedef TSafeReverselIterator< const T \*, const T & > [CSafeConstReverselIterator](#)  
*Version of TSafeReverselIterator 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)`  
*Assignment operator makes a copy of another vector.*
- `TVector< T > & operator= (TVector< T > &&Other) noexcept`  
*Move assignment.*
- `T & operator[] (size_t Index)`  
*Index operator that provides 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 vector. 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).*
- [CReverselIterator ReverseBegin](#) ()  
*Reverse iterator pointing to the first elem.*
- [CConstReverselIterator 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).*
- [CSafeReverselIterator SafeReverseBegin](#) ()  
*Safe reverse iterator pointing to the first elem.*

- [CUnsafeConstReverseIterator UnsafeConstReverseBegin \(\)](#) 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.*
- [CUnsafeReverseIterator UnsafeReverseEnd \(\)](#) 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 compatibility)*
- [CConstIterator begin \(\)](#) const  
*ConstBegin() alias (overloaded, for compatibility)*
- [CConstIterator cbegin \(\)](#) const  
*ConstBegin() alias (for compatibility)*
- [CReverseIterator rbegin \(\)](#)  
*ReverseBegin() alias (for compatibility)*
- [CUnsafeReverseIterator rbegin \(\)](#) const  
*ConstReverseBegin() alias (overloaded, for compatibility)*
- [CConstReverseIterator crbegin \(\)](#) const  
*ConstReverseBegin() alias (for compatibility)*
- [CIterator end \(\)](#)  
*End() alias (for compatibility)*
- [CConstIterator end \(\)](#) const  
*ConstEnd() alias (overloaded, for compatibility)*
- [CConstIterator cend \(\)](#) const  
*ConstEnd() alias (for compatibility)*
- [CReverseIterator rend \(\)](#)  
*ReverseEnd() alias (for compatibility)*
- [CUnsafeReverseIterator rend \(\)](#) const  
*ConstReverseEnd() alias (overloaded, for compatibility)*
- [CConstReverseIterator crend \(\)](#) const  
*ConstReverseEnd() alias (for compatibility)*

## 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 CReverseliterator

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

Reverse iterator. Increment is actually decrement, etc.

#### 6.6.2.5 CSafeConstliterator

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

Version of TSafeliterator for const values.

#### 6.6.2.6 CSafeConstReverseliterator

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

Version of TSafeReverseliterator for const values.

#### 6.6.2.7 CSafeliterator

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

Iterator that does bounds checking and throws OutOfRange().

#### 6.6.2.8 CSafeReverseliterator

```
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 <a href="#">ShrinkToFit()</a> will be called

## 6.6.4 Constructor &amp; 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

<i>CapacityRule</i>	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

<i>Size</i>	Number of elements to allocate
<i>DefaultValue</i>	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

<i>Size</i>	Number of elements in original array
<i>Array</i>	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

<i>ValuesList</i>	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**

<i>IteratorType</i>	Iterator that implements ++, != and *
---------------------	---------------------------------------

**Parameters**

<i>Begin</i>	Iterator referring to the beginning of container
<i>End</i>	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**

<i>Other</i>	vector to copy
--------------	----------------

**6.6.4.8 TVector()** [8/8]

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

Move constructor.

**Parameters**

<i>Other</i>	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

<i>IteratorType</i>	Iterator that implements ++, != and *
---------------------	---------------------------------------

##### Parameters

<i>Begin</i>	Iterator referring to the beginning of container
<i>End</i>	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

<i>Index</i>	aka offset value
<i>DefaultValue</i>	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 compatibility)

#### 6.6.5.7 begin() [2/2]

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

[ConstBegin\(\)](#) alias (overloaded, for compatibility)

#### 6.6.5.8 cbegin()

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

[ConstBegin\(\)](#) alias (for compatibility)

#### 6.6.5.9 cend()

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

[ConstEnd\(\)](#) alias (for compatibility)

#### 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 compatibility)

**6.6.5.16 crend()**

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

[ConstReverseEnd\(\)](#) alias (for compatibility)

**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 compatibility)

**6.6.5.19 end() [2/2]**

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

[ConstEnd\(\)](#) alias (overloaded, for compatibility)

**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

<i>Position</i>	Position of element to be removed
<i>bAllowAutoShrink</i>	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

<i>PositionFrom</i>	Starting index for erase
<i>PositionTo</i>	End point for erase. Element with this index will also be removed
<i>bAllowAutoShrink</i>	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**

<i>Position</i>	Index where to insert
<i>Value</i>	Value to insert
<i>FillOnResizeWith</i>	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

<i>IteratorType</i>	Iterator with implemented ++, != and *
---------------------	--

### Parameters

<i>Position</i>	Index of the first inserted element
<i>Begin</i>	Iterator referring to the first element
<i>End</i>	Iterator referring to the element after last one
<i>FillOnResizeWith</i>	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

<i>ValuesList</i>	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

<i>Other</i>	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

<i>Other</i>	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

<i>Other</i>	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 provides access to the element.

##### Parameters

<i>Index</i>	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**

<i>bAllowAutoShrink</i>	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**

<i>ElementsCount</i>	Number of elements to be removed
<i>bAllowAutoShrink</i>	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

<i>Value</i>	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

<i>IteratorType</i>	Iterator with implemented ++, != and *
---------------------	--

## Parameters

<i>Begin</i>	Iterator referring to the first element
<i>End</i>	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 compatibility)

**6.6.5.44 rbegin()** [2/2]

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

[ConstReverseBegin\(\)](#) alias (overloaded, for compatibility)

**6.6.5.45 rend()** [1/2]

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

[ReverseEnd\(\)](#) alias (for compatibility)

**6.6.5.46 rend()** [2/2]

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

[ConstReverseEnd\(\)](#) alias (overloaded, for compatibility)

**6.6.5.47 Reserve()**

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

Allocates elements internally for the future use.

**Parameters**

<i>NewCapacity</i>	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**

<i>NewSize</i>	New size of vector. If <code>NewSize &lt; Size</code> , deletes last elements. Otherwise, creates new with passed value.
<i>DefaultValue</i>	Optional. Value to initialize added elements
<i>bAllowAutoShrink</i>	Optional. Enables auto shrink according to the capacity rule. See: <a href="#">EReservedCapacityRule</a>

**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**

CSafeliterator 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

CSafeliterator 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

CSafeReverseliterator 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 vector. Reallocation on removal happens only if bool bAllowAutoShrink was passed with supported operation.

##### Parameters

<i>CapacityRule</i>	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

<i>bAllowAutoShrink</i>	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

<i>ElementsCount</i>	Number of elements to be removed
<i>bAllowAutoShrink</i>	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**

<i>Vector1</i>	First vector
<i>Vector2</i>	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.hpp File Reference

## 7.5 CommonTypes/Private/VectorIterators.hpp 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.hpp"
#include "Private/Vector.hpp"
```

### 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.hpp"
```

## 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 malformed 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 malformed 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

- ~CException
  - Common::CException, [15](#)
- ~TVector
  - Common::TVector< T >, [32](#)
- AreRawStringsEqual
  - Common, [10](#)
- ASSERT
  - Assert.h, [55](#)
- Assert.h
  - ASSERT, [55](#)
- Assign
  - Common::TVector< T >, [32](#)
- AutoAt
  - Common::TVector< T >, [32](#)
- Back
  - Common::TVector< T >, [33](#)
- Begin
  - Common::TVector< T >, [33](#)
- begin
  - Common::TVector< T >, [33](#), [34](#)
- cbegin
  - Common::TVector< T >, [34](#)
- CConstIterator
  - Common::TVector< T >, [27](#)
- CConstReverselIterator
  - Common::TVector< T >, [27](#)
- CDoesNotExist
  - Common::CDoesNotExist, [13](#)
- cend
  - Common::TVector< T >, [34](#)
- CException
  - Common::CException, [14](#)
- Clterator
  - Common::TVector< T >, [27](#)
- Clear
  - Common::TVector< T >, [34](#)
- Common, [9](#)
  - AreRawStringsEqual, [10](#)
  - CopyRawString, [10](#), [11](#)
  - GetIteratorDistance, [11](#)
  - GetRawStringLength, [12](#)
- Common::CDoesNotExist, [13](#)
  - CDoesNotExist, [13](#)
- Common::CException, [14](#)
  - ~CException, [15](#)
  - CException, [14](#)
  - GetMessage, [15](#)
  - Message, [15](#)
- Common::COutOfRange, [15](#)
  - COutOfRange, [16](#)
  - ExpectedIndex, [18](#)
  - GetExpectedRange, [17](#)
  - GetRequestedIndex, [17](#)
  - RequestedIndex, [18](#)
- Common::TOptional< T >, [18](#)
  - GetValue, [19](#)
  - GetValueOr, [19](#)
  - operator=, [20](#)
  - SetValue, [20](#)
  - TOptional, [19](#)
  - ValueExists, [20](#)
- Common::TPair< T1, T2 >, [21](#)
  - First, [22](#)
  - MakePair, [22](#)
  - Second, [22](#)
  - TPair, [21](#)
- Common::TVector< T >, [23](#)
  - ~TVector, [32](#)
  - Assign, [32](#)
  - AutoAt, [32](#)
  - Back, [33](#)
  - Begin, [33](#)
  - begin, [33](#), [34](#)
  - cbegin, [34](#)
  - CConstIterator, [27](#)
  - CConstReverselIterator, [27](#)
  - cend, [34](#)
  - Clterator, [27](#)
  - Clear, [34](#)
  - ConstBegin, [34](#)
  - ConstEnd, [35](#)
  - ConstReverseBegin, [35](#)
  - ConstReverseEnd, [35](#)
  - crbegin, [35](#)
  - crend, [36](#)
  - CReverselIterator, [27](#)
  - CSafeConstIterator, [28](#)
  - CSafeConstReverselIterator, [28](#)
  - CSafelIterator, [28](#)
  - CSafeReverselIterator, [28](#)
  - End, [36](#)
  - end, [36](#)
  - Erase, [36](#)
  - EraseMultiple, [37](#)
  - EReservedCapacityRule, [28](#)
  - Exponential, [29](#)

- Front, [37, 38](#)
- GetCapacity, [38](#)
- GetCapacityRule, [38](#)
- GetSize, [38](#)
- Insert, [39](#)
- IsEmpty, [40](#)
- Linear, [29](#)
- NeverReserve, [29](#)
- operator!=, [40](#)
- operator=, [40, 41](#)
- operator==, [41](#)
- operator[], [42](#)
- Pop, [42](#)
- PopMultiple, [43](#)
- Push, [43, 44](#)
- RawData, [44](#)
- rbegin, [44, 45](#)
- rend, [45](#)
- Reserve, [45](#)
- Resize, [46](#)
- ReverseBegin, [46](#)
- ReverseEnd, [46](#)
- SafeAt, [46, 47](#)
- SafeBack, [47](#)
- SafeBegin, [47](#)
- SafeConstBegin, [47](#)
- SafeConstEnd, [48](#)
- SafeConstReverseBegin, [48](#)
- SafeConstReverseEnd, [48](#)
- SafeEnd, [48](#)
- SafeFront, [49](#)
- SafePop, [49](#)
- SafeReverseBegin, [49](#)
- SafeReverseEnd, [49](#)
- SafeShift, [50](#)
- SetCapacityRule, [50](#)
- Shift, [50](#)
- ShiftMultiple, [51](#)
- ShrinkToFit, [51](#)
- Swap, [51](#)
- TVector, [29–31](#)
- CommonTypes/Exception.h, [53](#)
- CommonTypes/Optional.h, [53](#)
- CommonTypes/Pair.h, [54](#)
- CommonTypes/Private/Vector.tpp, [54](#)
- CommonTypes/Private/VectorIterators.tpp, [54](#)
- CommonTypes/Vector.h, [54](#)
- CommonUtils/AdvancedIteration.h, [54](#)
- CommonUtils/Assert.h, [55](#)
- CommonUtils/Private/AdvancedIteration.tpp, [55](#)
- CommonUtils/Private/RawString.cpp, [55](#)
- CommonUtils/RawString.h, [56](#)
- ConstBegin
  - Common::TVector< T >, [34](#)
- ConstEnd
  - Common::TVector< T >, [35](#)
- ConstReverseBegin
  - Common::TVector< T >, [35](#)
- ConstReverseEnd
  - Common::TVector< T >, [35](#)
- CopyRawString
  - Common, [10, 11](#)
- COutOfRange
  - Common::COutOfRange, [16](#)
- crbegin
  - Common::TVector< T >, [35](#)
- crend
  - Common::TVector< T >, [36](#)
- CReverselIterator
  - Common::TVector< T >, [27](#)
- CSafeConstIterator
  - Common::TVector< T >, [28](#)
- CSafeConstReverselIterator
  - Common::TVector< T >, [28](#)
- CSafeIterator
  - Common::TVector< T >, [28](#)
- CSafeReverselIterator
  - Common::TVector< T >, [28](#)
- End
  - Common::TVector< T >, [36](#)
- end
  - Common::TVector< T >, [36](#)
- Erase
  - Common::TVector< T >, [36](#)
- EraseMultiple
  - Common::TVector< T >, [37](#)
- EReservedCapacityRule
  - Common::TVector< T >, [28](#)
- ExpectedIndex
  - Common::COutOfRange, [18](#)
- Exponential
  - Common::TVector< T >, [29](#)
- First
  - Common::TPair< T1, T2 >, [22](#)
- Front
  - Common::TVector< T >, [37, 38](#)
- GetCapacity
  - Common::TVector< T >, [38](#)
- GetCapacityRule
  - Common::TVector< T >, [38](#)
- GetExpectedRange
  - Common::COutOfRange, [17](#)
- GetIteratorDistance
  - Common, [11](#)
- GetMessage
  - Common::CException, [15](#)
- GetRawStringLength
  - Common, [12](#)
- GetRequestedIndex
  - Common::COutOfRange, [17](#)
- GetSize
  - Common::TVector< T >, [38](#)
- GetValue
  - Common::TOptional< T >, [19](#)

- GetValueOr
    - Common::TOptional< T >, 19
  - Insert
    - Common::TVector< T >, 39
  - IsEmpty
    - Common::TVector< T >, 40
  - Linear
    - Common::TVector< T >, 29
  - MakePair
    - Common::TPair< T1, T2 >, 22
  - Message
    - Common::CException, 15
  - NeverReserve
    - Common::TVector< T >, 29
  - operator!=
    - Common::TVector< T >, 40
  - operator=
    - Common::TOptional< T >, 20
    - Common::TVector< T >, 40, 41
  - operator==
    - Common::TVector< T >, 41
  - operator[]
    - Common::TVector< T >, 42
  - Pop
    - Common::TVector< T >, 42
  - PopMultiple
    - Common::TVector< T >, 43
  - Push
    - Common::TVector< T >, 43, 44
  - RawData
    - Common::TVector< T >, 44
  - rbegin
    - Common::TVector< T >, 44, 45
  - rend
    - Common::TVector< T >, 45
  - RequestedIndex
    - Common::COutOfRange, 18
  - Reserve
    - Common::TVector< T >, 45
  - Resize
    - Common::TVector< T >, 46
  - ReverseBegin
    - Common::TVector< T >, 46
  - ReverseEnd
    - Common::TVector< T >, 46
  - SafeAt
    - Common::TVector< T >, 46, 47
  - SafeBack
    - Common::TVector< T >, 47
  - SafeBegin
    - Common::TVector< T >, 47
  - SafeConstBegin
    - Common::TVector< T >, 47
  - SafeConstEnd
    - Common::TVector< T >, 48
  - SafeConstReverseBegin
    - Common::TVector< T >, 48
  - SafeConstReverseEnd
    - Common::TVector< T >, 48
  - SafeEnd
    - Common::TVector< T >, 48
  - SafeFront
    - Common::TVector< T >, 49
  - SafePop
    - Common::TVector< T >, 49
  - SafeReverseBegin
    - Common::TVector< T >, 49
  - SafeReverseEnd
    - Common::TVector< T >, 49
  - SafeShift
    - Common::TVector< T >, 50
  - Second
    - Common::TPair< T1, T2 >, 22
  - SetCapacityRule
    - Common::TVector< T >, 50
  - SetValue
    - Common::TOptional< T >, 20
  - Shift
    - Common::TVector< T >, 50
  - ShiftMultiple
    - Common::TVector< T >, 51
  - ShrinkToFit
    - Common::TVector< T >, 51
  - Swap
    - Common::TVector< T >, 51
  - TOptional
    - Common::TOptional< T >, 19
  - TPair
    - Common::TPair< T1, T2 >, 21
  - TVector
    - Common::TVector< T >, 29–31
  - ValueExists
    - Common::TOptional< T >, 20
- x64/Debug/CommonLibs.vcxproj.FileListAbsolute.txt, 56