

---

# **librapid**

***Release v0.7.3***

**Toby Davis**

**Aug 04, 2023**



# CONTENTS

<b>1</b>	<b>What is LibRapid?</b>	<b>1</b>
1.1	Getting Started	1
1.1.1	Installation	1
1.1.2	Your First Program	2
1.1.3	Your First Program: Explained	2
1.2	CMake Integration	3
1.2.1	Installation	3
1.2.2	CMake Options	3
1.2.2.1	LIBRAPID_BUILD_EXAMPLES	3
1.2.2.2	LIBRAPID_BUILD_TESTS	4
1.2.2.3	LIBRAPID_CODE_COV	4
1.2.2.4	LIBRAPID_STRICT	4
1.2.2.5	LIBRAPID_QUIET	4
1.2.2.6	LIBRAPID_GET_FFTW	4
1.2.2.7	LIBRAPID_GET_BLAS	4
1.2.2.8	LIBRAPID_USE_OMP	5
1.2.2.9	LIBRAPID_USE_OPENCL	5
1.2.2.10	LIBRAPID_USE_CUDA	5
1.2.2.11	LIBRAPID_USE_MULTIPREC	6
1.2.2.12	LIBRAPID_OPTIMISE_SMALL_ARRAYS	6
1.2.2.13	LIBRAPID_FAST_MATH	6
1.2.2.14	LIBRAPID_NATIVE_ARCH	6
1.2.2.15	LIBRAPID_CUDA_FLOAT_VECTOR_WIDTH and LIBRAPID_CUDA_DOUBLE_VECTOR_WIDTH	6
1.2.2.16	LIBRAPID_NO_WINDOWS_H	7
1.2.2.17	LIBRAPID_MKL_CONFIG_PATH	7
1.3	API Reference	7
1.3.1	Topics and Usage Examples	7
1.3.1.1	Array Iterators	7
1.3.2	Arrays, Matrices and Linear Algebra	12
1.3.2.1	Linear Algebra	13
1.3.2.2	Array Class Listing	14
1.3.2.3	Array From Data	26
1.3.2.4	Pseudoconstructors	26
1.3.2.5	Array View	26
1.3.2.6	Array Operations	30
1.3.2.7	Size Type	81
1.3.2.8	Stride Tools	81
1.3.2.9	Storage	82
1.3.2.10	OpenCL Storage	91
1.3.2.11	CUDA Storage	91

1.3.3	Vectors . . . . .	100
1.3.3.1	Vector Listing . . . . .	100
1.3.4	Complex Numbers . . . . .	100
1.3.4.1	Complex Number Listing . . . . .	101
1.3.4.2	Complex Number Examples . . . . .	125
1.3.4.3	Complex Number Implementation Details . . . . .	125
1.3.5	Mathematics . . . . .	125
1.3.6	Multi-Precision Arithmetic . . . . .	125
1.3.6.1	Multi-Precision Listing . . . . .	125
1.4	Tutorials . . . . .	125
1.5	Performance and Benchmarks . . . . .	125
1.5.1	Lazy Evaluation . . . . .	126
1.5.1.1	Making Use of LibRapid's Lazy Evaluation . . . . .	126
1.5.2	Linear Algebra . . . . .	126
1.5.2.1	Solution . . . . .	127
1.5.2.2	Explanation . . . . .	127
1.6	Caution . . . . .	127
1.6.1	Array Referencing Issues . . . . .	127
<b>2</b>	<b>Why use LibRapid?</b>	<b>129</b>
2.1	A Small Example . . . . .	129
<b>3</b>	<b>Current Development Stage</b>	<b>131</b>
<b>4</b>	<b>Roadmap</b>	<b>133</b>
<b>5</b>	<b>Licencing</b>	<b>135</b>
	<b>Index</b>	<b>137</b>

## WHAT IS LIBRAPID?

LibRapid is a high performance Array library for C++. It supports a wide range of calculations and operations, useful classes and functions, and even supports CUDA! It uses SIMD instructions and multithreading where possible, achieving incredible performance on all operations.

**Getting Started** Write your first program with LibRapid.

**CMake Integration** See all available CMake options to make the most of LibRapid's features.

**API Reference** View LibRapid's API and documentation.

**Tutorials** Learn how to use some of LibRapid's features.

**Performance and Benchmarks** View LibRapid's benchmark results.

**Caution** **Learn about potential issues that may occur with LibRapid**

## 1.1 Getting Started

### 1.1.1 Installation

To use LibRapid in your CMake project, first clone the project:

```
git clone --recursive https://github.com/LibRapid/libRapid.git
```

**Warning:** Make sure to use the `--recursive` flag when cloning the repository. This will ensure that all submodules are cloned as well!

Make sure you have a structure similar to the following:

```
yourProject/  
  CMakeLists.txt  
  main.cpp  
  librapid/  
    CMakeLists.txt  
  ...  
  ...
```

Next, add the following to your `CMakeLists.txt`

```
add_subdirectory(librapid)
target_link_libraries(yourTarget PUBLIC librapid)
```

**Note:** If you are not familiar with CMake, I suggest you follow a quick tutorial on it just to get the hang of the basics. After that, check out the sample `CMakeLists.txt` file in the `examples` directory of the repository.

(`examples/templateCMakeLists.txt`)[<https://github.com/LibRapid/librapid/blob/master/examples/templateCMakeLists.txt>]

That's it! LibRapid will now be compiled and linked with your project!

## 1.1.2 Your First Program

```
1 #include <librapid>
2 namespace lrc = librapid;
3
4 int main() {
5     lrc::Array<int> myFirstArray = lrc::fromData({{1, 2, 3, 4},
6                                                  {5, 6, 7, 8}});
7
8     lrc::Array<int> mySecondArray = lrc::fromData({{8, 7, 6, 5},
9                                                    {4, 3, 2, 1}});
10
11     fmt::print("{}\n\n", myFirstArray);
12     fmt::print("{}\n", mySecondArray);
13
14     fmt::print("Sum of two Arrays:\n{}\n", myFirstArray + mySecondArray);
15     fmt::print("First row of my Array: {}\n", myFirstArray[0]);
16     fmt::print("First row of my Array: {}\n", myFirstArray[0] + mySecondArray[1]);
17
18     return 0;
19 }
```

## 1.1.3 Your First Program: Explained

```
1 #include <librapid>
2 namespace lrc = librapid;
```

The first line here allows you to use all of LibRapid's features in your file. The second line isn't required, but it makes your code shorter and quicker to type.

```
5 lrc::Array<int> myFirstArray = lrc::fromData({{1, 2, 3, 4},
6                                              {5, 6, 7, 8}});
7
8 lrc::Array<int> mySecondArray = lrc::fromData({{8, 7, 6, 5},
9                                              {4, 3, 2, 1}});
```

These lines create two `Array` instances from a list of values. Both arrays are 2-dimensional and have 2 rows and 4 columns.

```
11 fmt::print("{}\n\n", myFirstArray);
12 fmt::print("{}\n", mySecondArray);
```

Here, we print out the Arrays we just created. Try changing the numbers to see how the formatting changes!

```
14 fmt::print("Sum of two Arrays:\n{}\n", myFirstArray + mySecondArray);
```

This line performs a simple arithmetic operation on our Arrays and prints the result.

```
15 fmt::print("First row of my Array: {}\n", myFirstArray[0]);
16 fmt::print("First row of my Array: {}\n", myFirstArray[0] + mySecondArray[1]);
```

As you can see, Array instances can be indexed with the traditional square bracket notation. This means you can easily access sub-arrays of higher-dimensional array objects.

Now that you've seen how easy it is to use LibRapid, check out the rest of the documentation to learn more about the library's features! There are more example programs in the `examples` directory of the repository.

(examples/)[<https://github.com/LibRapid/librapid/tree/master/examples>]

## 1.2 CMake Integration

### 1.2.1 Installation

Link librapid like any other CMake library:

Clone the repository: `git clone --recursive https://github.com/LibRapid/libRapid.git`

Add the following to your `CMakeLists.txt`

```
add_subdirectory(librapid)
target_link_libraries(yourTarget PUBLIC librapid)
```

**Tip:** For a template `CMakeLists.txt` file, see the `examples` directory: `examples/CMakeLists.txt`

### 1.2.2 CMake Options

#### 1.2.2.1 LIBRAPID\_BUILD\_EXAMPLES

DEFAULT: OFF

Build the suite of example programs in the `examples` directory.

### 1.2.2.2 LIBRAPID\_BUILD\_TESTS

DEFAULT: OFF

Build LibRapid's unit tests.

### 1.2.2.3 LIBRAPID\_CODE\_COV

DEFAULT: OFF

Enable code coverage for LibRapid's unit tests.

### 1.2.2.4 LIBRAPID\_STRICT

DEFAULT: OFF

Enable strict compilation flags, turn on all warnings, and treat warnings as errors.

### 1.2.2.5 LIBRAPID\_QUIET

DEFAULT: OFF

Disable all warnings from LibRapid. This is useful if you are using LibRapid as a dependency and want a cleaner compilation output. Warnings should be minimal in the first place, but this option is provided just in case.

### 1.2.2.6 LIBRAPID\_GET\_FFTW

DEFAULT: OFF

Add FFTW as a dependency and link it with LibRapid. This is required for FFT support unless CUDA is enabled.

**Danger:** FFTW is licensed under the GPL, which is not compatible with LibRapid's MIT license. If you are using LibRapid as a dependency in an open source project, you may need to use LibRapid under a GPL license. If you forget, you'll *probably* be fine, but I can't guarantee anything. I'm not a lawyer, so don't take my word for it.

### 1.2.2.7 LIBRAPID\_GET\_BLAS

DEFAULT: OFF

Download a precompiled OpenBLAS build for your platform, and link it with LibRapid. This is useful if you don't (or can't) have BLAS installed on your system.

**Warning:** Always prefer to use your system's BLAS installation if possible.



### 1.2.2.8 LIBRAPID\_USE\_OMP

DEFAULT: ON

If OpenMP is found on the system, link LibRapid with it. This is required for multi-threading support and can significantly improve performance.

**Warning:** If this flag is enabled and OpenMP is not found installed on the system, the build will continue without OpenMP support.

### 1.2.2.9 LIBRAPID\_USE\_OPENCL

DEFAULT: ON

Search for OpenCL and link LibRapid with it. This is required for OpenCL support.

If this flag is enabled and OpenCL is not found installed on the system, the build will ↪ continue without OpenCL support.

**Danger:** If you are using OpenCL as a backend in your code, you must call `librapid::configureOpenCL()` before using any OpenCL arrays. This function will initialise the OpenCL context and queue, compile the OpenCL kernels and configure the OpenCL device for optimal performance. See the documentation for this function for more information.

### 1.2.2.10 LIBRAPID\_USE\_CUDA

DEFAULT: ON

Search for CUDA and link LibRapid with it. This is required for GPU support.

If this flag is enabled and CUDA is not found installed on the system, the build will ↪ continue without CUDA support.

**Danger:** LibRapid's CUDA support appears to only works on Windows, for some reason. I have no way of testing it on Linux or MacOS, so I can't guarantee that it will work. If you have experience in this area, please feel free to contact me and we can work together to get it working.

### 1.2.2.11 LIBRAPID\_USE\_MULTIPREC

DEFAULT: OFF

If MPIR and MPFR are found on the system, LibRapid will automatically link with them. If not, LibRapid will build custom, modified versions of these libraries. This is required for arbitrary precision support.

**Warning:** This can lead to longer build times and larger binaries.

### 1.2.2.12 LIBRAPID\_OPTIMISE\_SMALL\_ARRAYS

DEFAULT: OFF

Enabling this flag removes multithreading support for trivial array operations. For relatively small arrays (on the order of 1,000,000 elements), this can lead to a significant performance boost. For arrays larger than this, multithreading can be more efficient.

### 1.2.2.13 LIBRAPID\_FAST\_MATH

DEFAULT: OFF

Enabling this flag enables fast math mode for all LibRapid functions. This can lead to a significant performance boost, but may cause some functions to return slightly incorrect results due to lower precision operations being performed.

### 1.2.2.14 LIBRAPID\_NATIVE\_ARCH

DEFAULT: OFF

Enabling this flag compiles librapid with the most advanced instruction set available on the system. This can lead to significant performance boosts, but may cause the library to be incompatible with older systems.

Compiling with this flag may also cause the binaries to be incompatible with other CPU architectures, so be careful when distributing your programs.

### 1.2.2.15 LIBRAPID\_CUDA\_FLOAT\_VECTOR\_WIDTH and LIBRAPID\_CUDA\_DOUBLE\_VECTOR\_WIDTH

DEFAULT: 4

Set the default vector width for SIMD CUDA kernels. This must be in the range [1, 4]. Higher values will lead to better performance in most cases, but can increase register pressure which may lead to lower performance than expected. For optimal performance, you should try changing this value to suit your specific use case.

**Warning:** This setting requires CUDA support to be enabled.

### 1.2.2.16 LIBRAPID\_NO\_WINDOWS\_H

DEFAULT: OFF

Prevent the inclusion of `windows.h` in LibRapid's headers. Sometimes the macros and functions defined in this header can cause conflicts with other libraries, so this option is provided to prevent this.

**Danger:** It is not possible to fully remove `windows.h` when compiling with CUDA support on Windows, but many of the modules are still disabled. There is a possibility that conflicts will still arise, but I am yet to encounter any.

### 1.2.2.17 LIBRAPID\_MKL\_CONFIG\_PATH

DEFAULT: ""

If you have Intel's OneAPI Math Kernel Library installed on your system, you can provide the path to the `MKLConfig.cmake` file here. This will force LibRapid to link with MKL and ignore any other BLAS libraries. On systems with Intel CPUs, this can result in a significant performance boost.

## 1.3 API Reference

**Important:** This list is **INCOMPLETE!** If you think something is missing, try searching for it first. If you still can't find it, please open an issue on the [LibRapid GitHub repository](#).

Arrays, Matrices and Linear Algebra   Multidimensional arrays, matrices, linear algebra and more.

Machine Learning   Machine learning in LibRapid.

Vectors   Fixed-size vectors and supported operations.

Complex Numbers   Complex numbers and their operations.

Mathematics   General mathematical operations that work on most data types.

Multi-Precision Arithmetic   Arbitrary-precision integers, floating points and rationals.

Utilities   Utility functions and classes to support development.

### 1.3.1 Topics and Usage Examples

#### 1.3.1.1 Array Iterators

LibRapid provides many methods to iterate over the elements of an array. Each one has its own advantages and disadvantages, and the best one to use depends heavily upon the situation.

## Implicit Iteration

This is the **simplest and easiest** way to iterate over an array, but is also the **slowest**. This method should only be used when performance is not a concern or when the array is known to be relatively small.

```
auto a = lrc::Array<int>(lrc::Shape({4, 5}));

for (auto val : a) {
    for (auto val2 : val) {
        val2 = lrc::randint(1, 10);
    }
}

for (const auto &val : a) {
    for (const auto &val2 : val) {
        fmt::print("{} ", val2);
    }
    fmt::print("\n");
}
```

**Warning:** Due to the way LibRapid works internally, the iterator type returned by `Array::begin()` and `Array::end()` makes use of the `ArrayView` class. Since this is *not a direct C++ reference* many IDEs will claim that the value is unused and will suggest removing it. **Do not remove it!** The `ArrayView` is still referencing the original array and your data will still be updated correctly :)

Keep in mind that this issue only comes up when you're using the non-const iterator, which is when you're assigning to the iterator.

I am currently looking into ways to fix this issue, but it is proving to be quite difficult...

## Subscript Iteration

This method of iterating over an array is slightly faster than implicit iteration, but is still slow compared to other methods. This involves using a `for` loop to iterate over each axis of the array and then using the `operator[]` to access the elements.

```
auto a = lrc::Array<int>(lrc::Shape({4, 5}));

for (auto i = 0; i < a.shape()[0]; i++) {
    for (auto j = 0; j < a.shape()[1]; j++) {
        a[i][j] = lrc::randint(1, 10);
    }
}

for (auto i = 0; i < a.shape()[0]; i++) {
    for (auto j = 0; j < a.shape()[1]; j++) {
        fmt::print("{} ", a[i][j]);
    }
    fmt::print("\n");
}
```

## Direct Iteration

This approach is the fastest safe way to iterate over an array. Again, using a `for` loop to iterate over each axis of the array, but this time using the `operator()` method to access the elements.

This method is *much faster* than using the `operator[]` method because no temporary `ArrayView` objects are created.

```
auto a = lrc::Array<int>(lrc::Shape({4, 5}));

for (auto i = 0; i < a.shape()[0]; i++) {
    for (auto j = 0; j < a.shape()[1]; j++) {
        a(i, j) = lrc::randint(1, 10);
    }
}

for (auto i = 0; i < a.shape()[0]; i++) {
    for (auto j = 0; j < a.shape()[1]; j++) {
        fmt::print("{} ", a(i, j));
    }
    fmt::print("\n");
}
```

## Direct Storage Access

LibRapid's array types have a `Storage` object which stores the actual data of the array. This object can be accessed via the `Array::storage()` method. This method is the fastest way to iterate over an array, but it is also the most dangerous, and you should *only use it if you know what you are doing*.

**Danger:** This method only works on `ArrayContainer` instances (Array types which own their own data). If you try to use this approach on any other datatype, such as an `ArrayView` or `Function`, your code will not compile because these types do not store their own data and hence do not have a `storage()` method.

Note also that this does not give any information about the shape of the array, so you must be careful to ensure that you are accessing the correct elements.

```
auto a = lrc::Array<int>(lrc::Shape({4, 5}));

for (auto i = 0; i < a.shape().size(); i++) {
    a.storage()[i] = lrc::randint(1, 10);
}

for (auto i = 0; i < a.shape().size(); i++) {
    fmt::print("{} ", a.storage()[i]);
}
```

**Warning:** The `Storage` object stores the data in row-major order, so you must be careful that you are accessing the correct elements.

For example, if you have a 3D array with shape `{2, 3, 4}`, the elements will be accessed in the following order:

```
(0, 0, 0)
(0, 0, 1)
(0, 1, 0)
(0, 1, 1)
(0, 2, 0)
(0, 2, 1)
(1, 0, 0)
(1, 0, 1)
(1, 1, 0)
(1, 1, 1)
(1, 2, 0)
(1, 2, 1)
```

## Benchmarks

These benchmarks were performed on a Ryzen 9 3950x CPU with 64GB of RAM. The code used is included below.

25000 × 25000 **array of floats**

### MSVC

```
Iterator Timer [    ITERATOR    ] -- Elapsed: 1.25978m | Average: 25.19570s
Iterator Timer [ FOR LOOP INDEXED ] -- Elapsed: 1.06851m | Average: 10.68511s
Iterator Timer [ FOR LOOP DIRECT ] -- Elapsed: 1.03243m | Average: 2.13607s
Iterator Timer [    STORAGE     ] -- Elapsed: 1.00972m | Average: 712.74672ms
```

### GCC (WSL2)

```
Iterator Timer [    ITERATOR    ] -- Elapsed: 1.30497m | Average: 26.09936s
Iterator Timer [ FOR LOOP INDEXED ] -- Elapsed: 1.00171m | Average: 12.02046s
Iterator Timer [ FOR LOOP DIRECT ] -- Elapsed: 1.00257m | Average: 222.79388ms
Iterator Timer [    STORAGE     ] -- Elapsed: 1.00265m | Average: 268.56730ms
```

1000 × 1000 **array of floats**

### MSVC

```
Iterator Timer [    ITERATOR    ] -- Elapsed: 20.03113s | Average: 60.51699ms
Iterator Timer [ FOR LOOP INDEXED ] -- Elapsed: 20.01374s | Average: 20.56911ms
Iterator Timer [ FOR LOOP DIRECT ] -- Elapsed: 20.00305s | Average: 3.65019ms
Iterator Timer [    STORAGE     ] -- Elapsed: 20.00049s | Average: 1.45257ms
```

## GCC (WSL2)

Iterator Timer [	ITERATOR	]	-- Elapsed: 20.03222s	Average: 75.30909ms
Iterator Timer [	FOR LOOP INDEXED	]	-- Elapsed: 20.00276s	Average: 23.67190ms
Iterator Timer [	FOR LOOP DIRECT	]	-- Elapsed: 20.00003s	Average: 62.70073us
Iterator Timer [	STORAGE	]	-- Elapsed: 20.00014s	Average: 242.00937us

100 × 100 array of floats

## MSVC

Iterator Timer [	ITERATOR	]	-- Elapsed: 10.00005s	Average: 594.18031us
Iterator Timer [	FOR LOOP INDEXED	]	-- Elapsed: 10.00007s	Average: 210.48345us
Iterator Timer [	FOR LOOP DIRECT	]	-- Elapsed: 10.00003s	Average: 14.38816us
Iterator Timer [	STORAGE	]	-- Elapsed: 10.00001s	Average: 14.94997us

## GCC (WSL2)

Iterator Timer [	ITERATOR	]	-- Elapsed: 10.00055s	Average: 621.22918us
Iterator Timer [	FOR LOOP INDEXED	]	-- Elapsed: 10.00001s	Average: 235.57702us
Iterator Timer [	FOR LOOP DIRECT	]	-- Elapsed: 10.00000s	Average: 650.03031ns
Iterator Timer [	STORAGE	]	-- Elapsed: 10.00000s	Average: 2.44980us

## Code

```
lrc::Shape benchShape({25000, 25000});

{
    auto a = lrc::Array<float>(benchShape);
    lrc::Timer iteratorTimer(fmt::format("Iterator Timer [ {:^16} ]", "ITERATOR"));
    iteratorTimer.setTargetTime(10);

    while (iteratorTimer.isRunning()) {
        for (auto val : a) {
            for (auto val2 : val) { val2 = 1; }
        }
    }

    fmt::print("{:.5f}\n", iteratorTimer);
}

{
    auto a = lrc::Array<float>(benchShape);
    lrc::Timer iteratorTimer(fmt::format("Iterator Timer [ {:^16} ]", "FOR LOOP INDEXED
↪"));
    iteratorTimer.setTargetTime(10);
}
```

(continues on next page)

(continued from previous page)

```

    while (iteratorTimer.isRunning()) {
        for (int64_t i = 0; i < a.shape()[0]; i++) {
            for (int64_t j = 0; j < a.shape()[1]; j++) { a[i][j] = 1; }
        }
    }

    fmt::print("{:.5f}\n", iteratorTimer);
}

{
    auto a = lrc::Array<float>(benchShape);
    lrc::Timer iteratorTimer(fmt::format("Iterator Timer [ {:^16} ]", "FOR LOOP DIRECT
↪"));
    iteratorTimer.setTargetTime(10);

    while (iteratorTimer.isRunning()) {
        for (int64_t i = 0; i < a.shape()[0]; i++) {
            for (int64_t j = 0; j < a.shape()[1]; j++) { a(i, j) = 1; }
        }
    }

    fmt::print("{:.5f}\n", iteratorTimer);
}

{
    auto a = lrc::Array<float>(benchShape);
    lrc::Timer iteratorTimer(fmt::format("Iterator Timer [ {:^16} ]", "STORAGE"));
    iteratorTimer.setTargetTime(10);

    while (iteratorTimer.isRunning()) {
        for (int64_t i = 0; i < a.shape().size(); i++) { a.storage()[i] = 1; }
    }

    fmt::print("{:.5f}\n", iteratorTimer);
}

```

### 1.3.2 Arrays, Matrices and Linear Algebra

The main feature of LibRapid is its high-performance array library. It provides an intuitive way to perform highly efficient operations on arrays and matrices in C++.



### 1.3.2.1 Linear Algebra

#### Level 1 (Vector-Vector)

#### Level 2 (Matrix-Vector)

#### GEMV

**Warning:** doxygenfile: Cannot find file “librapid/include/librapid/array/linalg/level3/gemv.hpp”

#### Level 3 (Matrix-Matrix)

#### GEMM

namespace **librapid**

namespace **linalg**

#### Functions

```
template<typename Int, typename Alpha, typename A, typename B, typename Beta, typename C>
void gemm(bool transA, bool transB, Int m, Int n, Int k, Alpha alpha, A *a, Int lda, B *b, Int ldb, Beta
        beta, C *c, Int ldc, backend::CPU backend = backend::CPU())
```

General matrix-matrix multiplication.

Computes  $C = \alpha OP_A(A) OP_B(B) + \beta C$  for matrices **A**, **B** and **C**.  $OP_A$  and  $OP_B$  are either the identity or the transpose operation.

##### Template Parameters

- **Int** – Integer type for matrix dimensions
- **Alpha** – Type of  $\alpha$
- **A** – Type of **A**
- **B** – Type of **B**
- **Beta** – Type of  $\beta$
- **C** – Type of **C**

##### Parameters

- **transA** – Whether to transpose **A** (determines  $OP_A$ )
- **transB** – Whether to transpose **B** (determines  $OP_B$ )
- **m** – Rows of **A** and **C**
- **n** – Columns of **B** and **C**
- **k** – Columns of **A** and rows of **B**
- **alpha** – Scalar  $\alpha$
- **a** – Pointer to **A**
- **lda** – Leading dimension of **A**
- **b** – Pointer to **B**
- **ldb** – Leading dimension of **B**
- **beta** – Scalar  $\beta$
- **c** – Pointer to **C**
- **ldc** – Leading dimension of **C**

- **backend** – Backend to use for computation

```
CuBLASGemmComputeType cublasGemmComputeType(cublasDataType_t a, cublasDataType_t b,  
                                              cublasDataType_t c)
```

```
template<typename Int, typename Alpha, typename A, typename B, typename Beta, typename C>  
void gemm(bool transA, bool transB, Int m, Int n, Int k, Alpha alpha, A *a, Int lda, B *b, Int ldb, Beta  
         beta, C *c, Int ldc, backend::CUDA)
```

```
struct CuBLASGemmComputeType
```

```
    #include <gemm.hpp>
```

### Public Members

```
cublasComputeType_t computeType
```

```
cublasDataType_t scaleType
```

## 1.3.2.2 Array Class Listing

### Defines

```
SINIT(SUB_TYPE)
```

```
SVEC(SUB_TYPE)
```

```
ARRAY_FROM_DATA_DEF(TYPE_INIT, TYPE_VEC)
```

```
template<typename ShapeType_, typename StorageType_>
```

```
struct TypeInfo<array::ArrayContainer<ShapeType_, StorageType_>>
```

```
    #include <arrayContainer.hpp>
```

### Public Types

```
using Scalar = typename TypeInfo<StorageType_>::Scalar
```

```
using Packet = std::false_type
```

```
using Backend = typename TypeInfo<StorageType_>::Backend
```

### Public Static Attributes

```

static constexpr detail::LibRapidType type = detail::LibRapidType::ArrayContainer

static constexpr int64_t packetWidth = 1

static constexpr bool supportsArithmetic = TypeInfo<Scalar>::supportsArithmetic

static constexpr bool supportsLogical = TypeInfo<Scalar>::supportsLogical

static constexpr bool supportsBinary = TypeInfo<Scalar>::supportsBinary

static constexpr bool allowVectorisation = TypeInfo<Scalar>::packetWidth > 1

static constexpr cudaDataType_t CudaType = TypeInfo<Scalar>::CudaType

static constexpr int64_t cudaPacketWidth = 1

static constexpr bool canAlign = false

static constexpr int64_t canMemcpy = false

template<typename SizeType, size_t dims, typename StorageScalar>
struct IsArrayContainer<array::ArrayContainer<Shape<SizeType, dims>, StorageScalar>> : public std::true_type
    #include <arrayContainer.hpp>
template<typename T>
struct IsArrayType<array::ArrayView<T>>
    #include <arrayContainer.hpp>

```

### Public Static Attributes

```

static constexpr bool val = true

namespace librapid

    namespace array

        template<typename ShapeType_, typename StorageType_>

            class ArrayContainer
                #include <arrayContainer.hpp>

```

## Public Types

```
using StorageType = StorageType_  
  
using ShapeType = ShapeType_  
  
using StrideType = Stride<size_t, 32>  
  
using SizeType = typename ShapeType::SizeType  
  
using Scalar = typename StorageType::Scalar  
  
using Packet = typename typetraits::TypeInfo<Scalar>::Packet  
  
using Backend = typename typetraits::TypeInfo<ArrayContainer>::Backend  
  
using Iterator = detail::ArrayIterator<ArrayView<ArrayContainer>>  
  
using DirectSubscriptType = typename detail::SubscriptType<StorageType>::Direct  
  
using DirectRefSubscriptType = typename detail::SubscriptType<StorageType>::Ref
```

## Public Functions

### **ArrayContainer**()

Default constructor.

```
template<typename T>  
ArrayContainer(const std::initializer_list<T> &data)
```

```
template<typename T>  
explicit ArrayContainer(const std::vector<T> &data)
```

```
explicit ArrayContainer(const ShapeType &shape)
```

Constructs an array container from a shape

**Parameters** **shape** – The shape of the array container

```
ArrayContainer(const ShapeType &shape, const Scalar &value)
```

Create an array container from a shape and a scalar value. The scalar value represents the value the memory is initialized with.

**Parameters**

- **shape** – The shape of the array container
- **value** – The value to initialize the memory with

```
explicit ArrayContainer(const Scalar &value)
```

Allows for a fixed-size array to be constructed with a fill value

**Parameters** **value** – The value to fill the array with

explicit **ArrayContainer**(ShapeType &&shape)

Construct an array container from a shape, which is moved, not copied.

**Parameters** **shape** – The shape of the array container

**ArrayContainer**(const ArrayContainer &other) = default

Reference an existing array container.

This constructor does not copy the data, but instead references the data of the input array container.

This means that the input array container must outlive the constructed array container. Please use `ArrayContainer::copy()` if you want to copy the data.

**Parameters** **other** – The array container to reference

**ArrayContainer**(ArrayContainer &&other) noexcept = default

Construct an array container from a temporary array container.

**Parameters** **other** – The array container to move.

template<typename **TransposeType**>

**ArrayContainer**(const Transpose<TransposeType> &trans)

template<typename **ShapeTypeA**, typename **StorageTypeA**, typename **ShapeTypeB**, typename **StorageTypeB**, typename **Alpha**, typename **Beta**>

**ArrayContainer**(const *linalg*::ArrayMultiply<ShapeTypeA, StorageTypeA, ShapeTypeB, StorageTypeB, Alpha, Beta> &multiply)

template<typename **desc**, typename **Functor\_**, typename ...**Args**>

**ArrayContainer** &**assign**(const detail::Function<desc, Functor\_, Args...> &function)

template<typename **desc**, typename **Functor\_**, typename...

**Args**> **ArrayContainer** (const detail::Function< desc, Functor\_, Args...

> &function) **LIBRAPID\_RELEASE\_NOEXCEPT**

Construct an array container from a function object. This will assign the result of the function to the array container, evaluating it accordingly.

**Template Parameters**

- **desc** – The assignment descriptor
- **Functor\_** – The function type
- **Args** – The argument types of the function

**Parameters** **function** – The function to assign

**ArrayContainer** &**operator**=(const ArrayContainer &other) = default

Reference an existing array container.

This assignment operator does not copy the data, but instead references the data of the input array container. This means that the input array container must outlive the constructed array container.

Please use `ArrayContainer::copy()` if you want to copy the data.

**Parameters** **other** – The array container to reference

**ArrayContainer** &**operator**=(const Scalar &value)

**ArrayContainer** &**operator**=(ArrayContainer &&other) noexcept = default

Assign a temporary array container to this array container.

**Parameters** **other** – The array container to move.

**Returns** A reference to this array container.

template<typename **desc**, typename **Functor\_**, typename ...**Args**>

**ArrayContainer** &**operator**=(const detail::Function<desc, Functor\_, Args...> &function)

Assign a function object to this array container. This will assign the result of the function to the array container, evaluating it accordingly.

**Template Parameters**

- **Functor\_** – The function type
- **Args** – The argument types of the function

**Parameters** **function** – The function to assign

**Returns** A reference to this array container.

```
template<typename TransposeType>
```

```
ArrayContainer &operator=(const Transpose<TransposeType> &transpose)
```

```
template<typename ShapeTypeA, typename StorageTypeA, typename ShapeTypeB, typename  
StorageTypeB, typename Alpha, typename Beta>
```

```
ArrayContainer &operator=(const linalg::ArrayMultiply<ShapeTypeA, StorageTypeA,  
ShapeTypeB, StorageTypeB, Alpha, Beta> &multiply)
```

```
template<typename T>
```

```
detail::CommaInitializer<ArrayContainer> operator<<(const T &value)
```

Allow ArrayContainer objects to be initialized with a comma separated list of values. This makes use of the CommaInitializer class

**Template Parameters** **T** – The type of the values

**Parameters** **value** – The value to set in the Array object

**Returns** The comma initializer object

```
ArrayContainer copy() const
```

```
auto operator[] (int64_t index) const
```

Access a sub-array of this ArrayContainer instance. The sub-array will reference the same memory as this ArrayContainer instance.

**See also:**

*ArrayView*

**Parameters** **index** – The index of the sub-array

**Returns** A reference to the sub-array (*ArrayView*)

```
auto operator[] (int64_t index)
```

```
template<typename ...Indices>
```

```
DirectSubscriptType operator() (Indices... indices) const
```

```
template<typename ...Indices>
```

```
DirectRefSubscriptType operator() (Indices... indices)
```

```
Scalar get() const
```

```
ShapeType::SizeType ndim() const noexcept
```

Return the number of dimensions of the ArrayContainer object

**Returns** Number of dimensions of the ArrayContainer

```
const ShapeType &shape() const noexcept
```

Return the shape of the array container. This is an immutable reference.

**Returns** The shape of the array container.

```
const StorageType &storage() const noexcept
```

Return the StorageType object of the ArrayContainer

**Returns** The StorageType object of the ArrayContainer

StorageType &**storage**() noexcept

Return the StorageType object of the ArrayContainer

**Returns** The StorageType object of the ArrayContainer

Packet **packet**(size\_t index) const

Return a Packet object from the array's storage at a specific index.

**Parameters** **index** – The index to get the packet from

**Returns** A Packet object from the array's storage at a specific index

Scalar **scalar**(size\_t index) const

Return a Scalar from the array's storage at a specific index.

**Parameters** **index** – The index to get the scalar from

**Returns** A Scalar from the array's storage at a specific index

void **writePacket**(size\_t index, const Packet &value)

Write a Packet object to the array's storage at a specific index

**Parameters**

- **index** – The index to write the packet to
- **value** – The value to write to the array's storage

void **write**(size\_t index, const Scalar &value)

Write a Scalar to the array's storage at a specific index

**Parameters**

- **index** – The index to write the scalar to
- **value** – The value to write to the array's storage

template<typename T>

ArrayContainer &**operator+=(**const T &other)

template<typename T>

ArrayContainer &**operator-=**(const T &other)

template<typename T>

ArrayContainer &**operator\*=**(const T &other)

template<typename T>

ArrayContainer &**operator/=**(const T &other)

template<typename T>

ArrayContainer &**operator%=(**const T &other)

template<typename T>

ArrayContainer &**operator&=(**const T &other)

template<typename T>

ArrayContainer &**operator|=(**const T &other)

template<typename T>

ArrayContainer &**operator^=(**const T &other)

template<typename T>

ArrayContainer &**operator<<=(**const T &other)

template<typename T>

ArrayContainer &**operator>>=(**const T &other)

Iterator **begin**() const noexcept

Return an iterator to the beginning of the array container.

**Returns** Iterator

Iterator **end**() const noexcept

Return an iterator to the end of the array container.

**Returns** Iterator

Iterator **begin**()

Return an iterator to the beginning of the array container.

**Returns** Iterator

Iterator **end**()

Return an iterator to the end of the array container.

**Returns** Iterator

std::string **str**(const std::string &format = "{}") const

Return a string representation of the array container \format The format to use for the string representation

**Returns** A string representation of the array container

template<typename **desc**, typename **Functor\_**, typename ...**Args**>

auto **assign**(const detail::Function<desc, Functor\_, Args...> &function) -> ArrayContainer&

template<typename **desc**, typename **Functor\_**, typename ...**Args**>

auto **operator**=(const detail::Function<desc, Functor\_, Args...> &function) -> ArrayContainer&

template<typename **TransposeType**>

auto **operator**=(const Transpose<TransposeType> &transpose) -> ArrayContainer&

template<typename **ShapeTypeA**, typename **StorageTypeA**, typename **ShapeTypeB**, typename **StorageTypeB**, typename **Alpha**, typename **Beta**>

auto **operator**=(const *linalg*::ArrayMultiply<ShapeTypeA, StorageTypeA, ShapeTypeB, StorageTypeB, Alpha, Beta> &arrayMultiply) -> ArrayContainer&

template<typename **T**>

auto **operator**<<(const T &value) -> detail::CommaInitializer<ArrayContainer>

template<typename ...**Indices**>

auto **operator**() (Indices... indices) const -> DirectSubscriptType

template<typename ...**Indices**>

auto **operator**() (Indices... indices) -> DirectRefSubscriptType

template<typename **T**>

auto **operator**+=(const T &value) -> ArrayContainer&

template<typename **T**>

auto **operator**--=(const T &value) -> ArrayContainer&

template<typename **T**>

auto **operator**\*=(const T &value) -> ArrayContainer&

template<typename **T**>

auto **operator**/=(const T &value) -> ArrayContainer&

template<typename **T**>



```

auto operator%=(const T &value) -> ArrayContainer&

template<typename T>
auto operator&=(const T &value) -> ArrayContainer&

template<typename T>
auto operator|=(const T &value) -> ArrayContainer&

template<typename T>
auto operator^=(const T &value) -> ArrayContainer&

template<typename T>
auto operator<<=(const T &value) -> ArrayContainer&

template<typename T>
auto operator>>=(const T &value) -> ArrayContainer&

```

### Public Static Functions

```
static auto fromData(const std::initializer_list<Scalar> &data) -> ArrayContainer
```

Create an array from a list of values (possibly multi-dimensional)

Create a new array from a potentially nested list of values. It is possible to specify the data type of the Array with the `Scalar` template parameter. If no type is specified, the type will be inferred from the data. The backend on which the Array is created can also be specified with the `Backend` template parameter. If no backend is specified, the Array will be created on the CPU.

#### Template Parameters

- **Scalar** – The type of the Array
- **Backend** – The backend on which the Array is created

**Parameters** `data` – The data from which the Array is created

**Returns** The created Array

```
static auto fromData(const std::vector<Scalar> &data) -> ArrayContainer
```

```
static auto fromData(const std::initializer_list<std::initializer_list<Scalar>> &data) ->
    ArrayContainer
```

```
static auto fromData(const std::vector<std::vector<Scalar>> &data) -> ArrayContainer
```

```
static auto fromData(const std::initializer_list<std::initializer_list<std::initializer_list<Scalar>>>
    &data) -> ArrayContainer
```

```
static auto fromData(const std::vector<std::vector<std::vector<Scalar>>> &data) ->
    ArrayContainer
```

```
static auto fromData(const
    std::initializer_list<std::initializer_list<std::initializer_list<std::initializer_list<Scalar>>>>
    &data) -> ArrayContainer
```

```
static auto fromData(const std::vector<std::vector<std::vector<std::vector<Scalar>>>> &data) ->
    ArrayContainer
```

```
static auto fromData(const
    std::initializer_list<std::initializer_list<std::initializer_list<std::initializer_list<std::initializer_list<Scalar>>>>>
    &data) -> ArrayContainer
```

```
static auto fromData(const std::vector<std::vector<std::vector<std::vector<std::vector<Scalar>>>>>
                    &data) -> ArrayContainer

static auto fromData(const
                    std::initializer_list<std::initializer_list<std::initializer_list<std::initializer_list<std::initializer_list<Scalar>>>>>
                    &data) -> ArrayContainer

static auto fromData(const
                    std::vector<std::vector<std::vector<std::vector<std::vector<std::vector<Scalar>>>>>
                    &data) -> ArrayContainer

static auto fromData(const
                    std::initializer_list<std::initializer_list<std::initializer_list<std::initializer_list<std::initializer_list<Scalar>>>>>
                    &data) -> ArrayContainer

static auto fromData(const
                    std::vector<std::vector<std::vector<std::vector<std::vector<std::vector<std::vector<Scalar>>>>>
                    &data) -> ArrayContainer

static auto fromData(const
                    std::initializer_list<std::initializer_list<std::initializer_list<std::initializer_list<std::initializer_list<Scalar>>>>>
                    &data) -> ArrayContainer

static auto fromData(const
                    std::vector<std::vector<std::vector<std::vector<std::vector<std::vector<std::vector<std::vector<Scalar>>>>>
                    &data) -> ArrayContainer
```

## Private Members

ShapeType **m\_shape**

StorageType **m\_storage**

namespace **detail**

template<typename T>

struct **SubscriptType**

*#include <arrayContainer.hpp>*

## Public Types

using **Scalar** = T

using **Direct** = const Scalar&

using **Ref** = Scalar&

template<typename T>

```
struct SubscriptType<Storage<T>>
    #include <arrayContainer.hpp>
```

### Public Types

```
using Scalar = T
```

```
using Direct = const Scalar&
```

```
using Ref = Scalar&
```

```
template<typename T, size_t... Dims>
struct SubscriptType<FixedStorage<T, Dims...>>
    #include <arrayContainer.hpp>
```

### Public Types

```
using Scalar = T
```

```
using Direct = const Scalar&
```

```
using Ref = Scalar&
```

```
template<typename T>
struct SubscriptType<CudaStorage<T>>
    #include <arrayContainer.hpp>
```

### Public Types

```
using Scalar = T
```

```
using Direct = const detail::CudaRef<Scalar>
```

```
using Ref = detail::CudaRef<Scalar>
```

```
template<typename T>
struct IsArrayType
    #include <arrayContainer.hpp>
```

### Public Static Attributes

```
static constexpr bool val = false  
  
template<typename T>  
  
struct IsArrayType<ArrayRef<T>>  
    #include <arrayContainer.hpp>
```

### Public Static Attributes

```
static constexpr bool val = true  
  
template<typename ...T>  
  
struct IsArrayType<FunctionRef<T...>>  
    #include <arrayContainer.hpp>
```

### Public Static Attributes

```
static constexpr bool val = true  
  
template<typename T> ArrayView< T > >  
    #include <arrayContainer.hpp>
```

### Public Static Attributes

```
static constexpr bool val = true  
  
template<typename First, typename ...Types>  
  
struct ContainsArrayType  
    #include <arrayContainer.hpp>
```

### Public Static Functions

```
static inline constexpr auto evaluator()
```

## Public Static Attributes

```
static constexpr bool val = evaluator()
```

```
namespace typetraits
```

## Functions

```
LIBRAPID_DEFINE_AS_TYPE (typename SizeType COMMA size_t dims COMMA typename StorageScalar,  
array::ArrayContainer< Shape< SizeType COMMA dims > COMMA StorageScalar >)
```

```
template<typename ShapeType_, typename StorageType_> ArrayContainer< ShapeType_,  
StorageType_ > >
```

```
#include <arrayContainer.hpp>
```

## Public Types

```
using Scalar = typename TypeInfo<StorageType_>::Scalar
```

```
using Packet = std::false_type
```

```
using Backend = typename TypeInfo<StorageType_>::Backend
```

## Public Static Attributes

```
static constexpr detail::LibRapidType type = detail::LibRapidType::ArrayContainer
```

```
static constexpr int64_t packetWidth = 1
```

```
static constexpr bool supportsArithmetic = TypeInfo<Scalar>::supportsArithmetic
```

```
static constexpr bool supportsLogical = TypeInfo<Scalar>::supportsLogical
```

```
static constexpr bool supportsBinary = TypeInfo<Scalar>::supportsBinary
```

```
static constexpr bool allowVectorisation = TypeInfo<Scalar>::packetWidth > 1
```

```
static constexpr cudaDataType_t CudaType = TypeInfo<Scalar>::CudaType
```

```
static constexpr int64_t cudaPacketWidth = 1
```

```
static constexpr bool canAlign = false

static constexpr int64_t canMemcpy = false

template<typename T>

struct IsArrayContainer : public std::false_type
    #include <arrayContainer.hpp> Evaluates as true if the input type is an ArrayContainer instance
    Template Parameters T – Input type

    template<typename SizeType, size_t dims,
    typename StorageScalar> ArrayContainer< Shape< SizeType, dims >,
    StorageScalar > > : public std::true_type
        #include <arrayContainer.hpp>
```

### 1.3.2.3 Array From Data

#### Defines

**HIGHER\_DIMENSIONAL\_FROM\_DATA**(TYPE)

**SINIT**(SUB\_TYPE)

**SVEC**(SUB\_TYPE)

namespace **librapid**

### 1.3.2.4 Pseudoconstructors

**Warning:** doxygenfile: Cannot find file “librapid/include/librapid/array/pseudoconstructors.hpp”

### 1.3.2.5 Array View

```
template<typename T>

struct TypeInfo<array::ArrayView<T>>
    #include <arrayView.hpp>
```

#### Public Types

using **Scalar** = typename TypeInfo<std::decay\_t<T>>::Scalar

using **Backend** = typename TypeInfo<std::decay\_t<T>>::Backend

## Public Static Attributes

```
static constexpr detail::LibRapidType type = detail::LibRapidType::ArrayView
```

```
static constexpr bool allowVectorisation = false
```

```
namespace librapid
```

```
namespace array
```

```
template<typename T>
```

```
class ArrayView
```

*#include <arrayView.hpp>* An intermediate type to represent a slice or view of an array.

**Template Parameters** **T** – The type of the array.

## Public Types

```
using BaseType = typename std::decay_t<T>
```

```
using Scalar = typename typetraits::TypeInfo<BaseType>::Scalar
```

```
using Reference = BaseType&
```

```
using ConstReference = const BaseType&
```

```
using Backend = typename typetraits::TypeInfo<BaseType>::Backend
```

```
using ArrayType = Array<Scalar, Backend>
```

```
using StrideType = typename ArrayType::StrideType
```

```
using ShapeType = typename ArrayType::ShapeType
```

```
using Iterator = detail::ArrayIterator<ArrayView>
```

## Public Functions

```
ArrayView() = delete
```

Default constructor should never be used.

```
explicit ArrayView(T &array)
```

Copy an *ArrayView* object

**Parameters** **array** – The array to copy

explicit **ArrayView**(T &&array) = delete

Copy an *ArrayView* object (not const)

**Parameters** **array** – The array to copy

**ArrayView**(const ArrayView &other) = default

Copy an *ArrayView* object (const)

**Parameters** **other** – The array to copy

**ArrayView**(ArrayView &&other) = default

Constructs an *ArrayView* from a temporary instance

**Parameters** **other** – The *ArrayView* to move

ArrayView &**operator**=(const ArrayView &other) = default

Assigns another *ArrayView* object to this *ArrayView*.

**Parameters** **other** – The *ArrayView* to assign.

**Returns** A reference to this

ArrayView &**operator**=(const Scalar &scalar)

Assigns a temporary *ArrayView* to this *ArrayView*.

**Parameters**

- **other** – The *ArrayView* to move.
- **scalar** – The scalar value to assign

**Returns** A reference to this *ArrayView*. Assign a scalar value to this *ArrayView*. This function should only be used to assign to a zero-dimensional “scalar” *ArrayView*, and will throw an error if used incorrectly.

**Returns** A reference to this

template<typename **RefType**>

ArrayView &**operator**=(const ArrayRef<RefType> &other)

const ArrayView<T> **operator**[](int64\_t index) const

Access a sub-array of this *ArrayView*.

**Parameters** **index** – The index of the sub-array.

**Returns** An *ArrayView* from this

ArrayView<T> **operator**[](int64\_t index)

template<typename **CAST** = Scalar>

**CAST** **get**() const

Since even scalars are represented as an *ArrayView* object, it can be difficult to operate on them directly. This allows you to extract the scalar value stored by a zero-dimensional *ArrayView* object

**Template Parameters** **CAST** – Type to cast to

**Returns** The scalar represented by the *ArrayView* object

template<typename **CAST**>

explicit **operator** **CAST**() const

Same functionality as “get”, except slightly less robust for user-defined types.

**Template Parameters** **CAST** – Type to cast to

**Returns** The scalar represented by the *ArrayView* object

ShapeType **shape**() const

Access the underlying shape of this *ArrayView*

**Returns** Shape object

StrideType **stride**() const

Access the stride of this *ArrayView*

**Returns** *Stride* object



int64\_t **offset**() const

Access the offset of this *ArrayView*. This is the offset, in elements, from the referenced Array's first element.

**Returns** Offset

void **setShape**(const ShapeType &shape)

Set the Shape of this *ArrayView* to something else. Intended for internal use only.

**Parameters** **shape** – The new shape of this *ArrayView*

void **setStride**(const StrideType &stride)

Set the *Stride* of this *ArrayView* to something else. Intended for internal use only.

**Parameters** **stride** – The new stride of this *ArrayView*

void **setOffset**(const int64\_t &offset)

Set the offset of this *ArrayView* object. Intended for internal use only.

**Parameters** **offset** – The new offset of this *ArrayView*

int64\_t **ndim**() const

Returns the number of dimensions of this *ArrayView*

**Returns** Number of dimensions

auto **scalar**(int64\_t index) const

Return the Scalar at a given index in this *ArrayView*. This is intended for use internally, but can be used externally too.

**Parameters** **index** – The index of the Scalar to return

**Returns** Scalar at the given index

ArrayType **eval**() const

Evaluate the contents of this *ArrayView* object and return an Array instance from it. Depending on your use case, this may result in more performant code, but the new Array will not reference the original data in the *ArrayView*.

**Returns** A new Array instance

Iterator **begin**() const

Iterator **end**() const

std::string **str**(const std::string &format = "{}") const

Cast an *ArrayView* to a std::string, aligning items down the columns. A format string can also be specified, which will be used to format the items to strings

**Parameters** **format** – The format string

**Returns** A std::string representation of this *ArrayView*

template<typename **RefType**>

ArrayView<T> &**operator**=(const ArrayRef<RefType> &other)

## Private Members

T &**m\_ref**

ShapeType **m\_shape**

StrideType **m\_stride**

```
int64_t m_offset = 0
```

```
namespace typetraits
```

### Functions

```
LIBRAPID_DEFINE_AS_TYPE(typename T, array::ArrayView<T>)
```

```
template<typename T> ArrayView< T > >
```

```
    #include <arrayView.hpp>
```

### Public Types

```
using Scalar = typename TypeInfo<std::decay_t<T>>>::Scalar
```

```
using Backend = typename TypeInfo<std::decay_t<T>>>::Backend
```

### Public Static Attributes

```
static constexpr detail::LibRapidType type = detail::LibRapidType::ArrayView
```

```
static constexpr bool allowVectorisation = false
```

## 1.3.2.6 Array Operations

### Defines

```
LIBRAPID_BINARY_FUNCTOR(NAME_, OP_)
```

```
LIBRAPID_BINARY_COMPARISON_FUNCTOR(NAME_, OP_)
```

```
LIBRAPID_UNARY_KERNEL_GETTER
```

```
LIBRAPID_BINARY_KERNEL_GETTER
```

```
LIBRAPID_UNARY_SHAPE_EXTRACTOR
```

```
LIBRAPID_BINARY_SHAPE_EXTRACTOR
```

```
LIBRAPID_UNARY_FUNCTOR(NAME, OP)
```

```
IS_ARRAY_OP
```

**IS\_ARRAY\_OP\_ARRAY**

**IS\_ARRAY\_OP\_WITH\_SCALAR**

template<typename **ShapeType**, typename **StorageType**>

struct **DescriptorExtractor**<array::ArrayContainer<ShapeType, StorageType>>

*#include <operations.hpp>* Extracts the Descriptor type of an ArrayContainer object. In this case, the Descriptor is Trivial

#### Template Parameters

- **ShapeType** – The shape type of the ArrayContainer
- **StorageType** – The storage type of the ArrayContainer

#### Public Types

using **Type** = ::librapid::detail::descriptor::Trivial

template<typename **T**>

struct **DescriptorExtractor**<array::ArrayView<**T**>>

*#include <operations.hpp>* Extracts the Descriptor type of an ArrayView object

**Template Parameters** **T** – The Array type of the ArrayView

#### Public Types

using **Type** = ::librapid::detail::descriptor::Trivial

template<typename **Descriptor**, typename **Functor**, typename ...**Args**>

struct **DescriptorExtractor**<::librapid::detail::Function<Descriptor, Functor, Args...>>

*#include <operations.hpp>* Extracts the Descriptor type of a Function object

#### Template Parameters

- **Descriptor** – The descriptor of the Function
- **Functor** – The functor type of the Function
- **Args** – The argument types of the Function

#### Public Types

using **Type** = Descriptor

template<>

struct **TypeInfo**<::librapid::detail::Plus>

*#include <operations.hpp>*

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "plus"

static constexpr const char *filename = "arithmetic"

static constexpr const char *kernelName = "addArrays"

static constexpr const char *kernelNameScalarRhs = "addArraysScalarRhs"

static constexpr const char *kernelNameScalarLhs = "addArraysScalarLhs"

template<>
struct TypeInfo<::librapid::detail::Minus>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "minus"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "subArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "subArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "subArraysScalarLhs"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Multiply>
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
```

```
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)
```

```
template<typename ...Args>
```

```
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename First, typename Second>
```

```
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)
```

```
template<typename ...Args>
```

```
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "multiply"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "mulArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "mulArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "mulArraysScalarLhs"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Divide>
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "divide"

static constexpr const char *filename = "arithmetic"

static constexpr const char *kernelName = "divArrays"

static constexpr const char *kernelNameScalarRhs = "divArraysScalarRhs"

static constexpr const char *kernelNameScalarLhs = "divArraysScalarLhs"

template<>
struct TypeInfo<::librapid::detail::LessThan>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "less than"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "lessThanArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "lessThanArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "lessThanArraysScalarLhs"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::GreaterThan>
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
```

```
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)
```

```
template<typename ...Args>
```

```
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename First, typename Second>
```

```
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)
```

```
template<typename ...Args>
```

```
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "greater than"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "greaterThanArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "greaterThanArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "greaterThanArraysScalarLhs"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::LessThanEqual>
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "less than or equal"

static constexpr const char *filename = "arithmetic"

static constexpr const char *kernelName = "lessThanEqualArrays"

static constexpr const char *kernelNameScalarRhs = "lessThanEqualArraysScalarRhs"

static constexpr const char *kernelNameScalarLhs = "lessThanEqualArraysScalarLhs"

template<>
struct TypeInfo<::librapid::detail::GreaterThanOrEqual>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```



### Public Static Attributes

```
static constexpr const char *name = "greater than or equal"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "greaterThanEqualArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "greaterThanEqualArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "greaterThanEqualArraysScalarLhs"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::ElementWiseEqual>
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
```

```
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)
```

```
template<typename ...Args>
```

```
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename First, typename Second>
```

```
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)
```

```
template<typename ...Args>
```

```
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "element wise equal"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "elementWiseEqualArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "elementWiseEqualArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "elementWiseEqualArraysScalarLhs"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::ElementWiseNotEqual>
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "element wise not equal"

static constexpr const char *filename = "arithmetic"

static constexpr const char *kernelName = "elementWiseNotEqualArrays"

static constexpr const char *kernelNameScalarRhs = "elementWiseNotEqualArraysScalarRhs"

static constexpr const char *kernelNameScalarLhs = "elementWiseNotEqualArraysScalarLhs"

template<>
struct TypeInfo<::librapid::detail::Neg>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "negate"

static constexpr const char *filename = "negate"

static constexpr const char *kernelName = "negateArrays"

template<>
```

```
struct TypeInfo<::librapid::detail::Sin>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "sin"

static constexpr const char *filename = "trigonometry"

static constexpr const char *kernelName = "sinArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Cos>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "cos"

static constexpr const char *filename = "trigonometry"

static constexpr const char *kernelName = "cosArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Tan>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "tan"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "tanArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Asin>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "arcsin"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "asinArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Acos>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "arcs"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "acosArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Atan>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "arctan"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "atanArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Sinh>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "hyperbolic sine"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "sinhArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Cosh>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "hyperbolic cosine"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "coshArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Tanh>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "hyperbolic tangent"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "tanhArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Exp>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "exponent"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "expArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Log>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "logarithm"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "logArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Log2>  
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "logarithm base 2"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "log2Arrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Log10>  
    #include <operations.hpp>
```



### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "logarithm base 10"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "log10Arrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Sqrt>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "square root"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "sqrtArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Cbrt>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "cube root"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "cbrtArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Abs>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "absolute value"
```

```
static constexpr const char *filename = "abs"
```

```
static constexpr const char *kernelName = "absArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Floor>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "floor"
```

```
static constexpr const char *filename = "floorCeilRound"
```

```
static constexpr const char *kernelName = "floorArrays"
```

```
template<>
```

```
struct TypeInfo<::librapid::detail::Ceil>
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "ceiling"
```

```
static constexpr const char *filename = "floorCeilRound"
```

```
static constexpr const char *kernelName = "ceilArrays"
```

```
namespace librapid
```

## Functions

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto sin (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descript
detail::Sin, VAL >
```

Calculate the sine of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \sin(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Sine function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto cos (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descript
detail::Cos, VAL >
```

Calculate the cosine of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \cos(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Cosine function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto tan (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descript
detail::Tan, VAL >
```

Calculate the tangent of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \tan(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Tangent function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto asin (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descript
detail::Asin, VAL >
```

Calculate the arcsine of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \sin^{-1}(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Arcsine function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto acos (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descript
detail::Acos, VAL >
```

Calculate the arccosine of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \cos^{-1}(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Arccosine function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto atan (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descrip
detail::Atan, VAL >
```

Calculate the arctangent of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \tan^{-1}(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Arctangent function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto sinh (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descrip
detail::Sinh, VAL >
```

Calculate the hyperbolic sine of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \sinh(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Hyperbolic sine function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto cosh (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descrip
detail::Cosh, VAL >
```

Calculate the hyperbolic cosine of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \cosh(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Hyperbolic cosine function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto tanh (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descrip
detail::Tanh, VAL >
```

Calculate the hyperbolic tangent of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \tanh(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Hyperbolic tangent function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto exp (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descrip
detail::Exp, VAL >
```

Raise  $e$  to the power of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = e^{A_i}$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Exponential function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto log (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descript
detail::Log, VAL >
```

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \ln(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Natural logarithm function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto log10 (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descrip
detail::Log10, VAL >
```

Compute the base 10 logarithm of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \log_{10}(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Base 10 logarithm function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto log2 (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descrip
detail::Log2, VAL >
```

Compute the base 2 logarithm of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \log_2(A_i)$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Base 2 logarithm function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto sqrt (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descrip
detail::Sqrt, VAL >
```

Compute the square root of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \sqrt{A_i}$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Square root function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto cbrt (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descript
detail::Cbrt, VAL >
```

Compute the cube root of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \sqrt[3]{A_i}$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Cube root function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto abs (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descript
detail::Abs, VAL >
```

Compute the absolute value of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = |A_i|$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Absolute value function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto floor (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descript
detail::Floor, VAL >
```

Compute the floor of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \lfloor A_i \rfloor$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Floor function object

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto ceil (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::Descript
detail::Ceil, VAL >
```

Compute the ceiling of each element in the array.

$$R = \{R_0, R_1, R_2, \dots\} \text{ where } R_i = \lceil A_i \rceil$$

**Template Parameters** **VAL** – Type of the input

**Parameters** **val** – The input array or function

**Returns** Ceiling function object

namespace **array**

## Functions

```
template<class LHS, class RHS,
typename std::enable_if_t< detail::isArrayOpArray< LHS, RHS >(),
int > = 0> auto operator+ (LHS &&lhs,
RHS &&rhs) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::DescriptorType_t< LHS,
RHS >, detail::Plus, LHS, RHS >
```

Element-wise array addition.

Performs element-wise addition on two arrays. They must both be the same size and of the same data type.

### Template Parameters

- **LHS** – Type of the LHS element
- **RHS** – Type of the RHS element

### Parameters

- **lhs** – The first array
- **rhs** – The second array

**Returns** The element-wise sum of the two arrays

```
template<class LHS, class RHS,
typename std::enable_if_t< detail::isArrayOpArray< LHS, RHS >(),
int > = 0> auto operator- (LHS &&lhs,
RHS &&rhs) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::DescriptorType_t< LHS,
RHS >, detail::Minus, LHS, RHS >
```

Element-wise array subtraction.

Performs element-wise subtraction on two arrays. They must both be the same size and of the same data type.

### Template Parameters

- **LHS** – Type of the LHS element
- **RHS** – Type of the RHS element

### Parameters

- **lhs** – The first array
- **rhs** – The second array

**Returns** The element-wise difference of the two arrays

```
template<class LHS, class RHS,
typename std::enable_if_t< detail::isArrayOpArray< LHS, RHS >(),
int > = 0> auto operator* (LHS &&lhs,
RHS &&rhs) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::DescriptorType_t< LHS,
RHS >, detail::Multiply, LHS, RHS >
```

Element-wise array multiplication.

Performs element-wise multiplication on two arrays. They must both be the same size and of the same data type.

### Template Parameters

- **LHS** – Type of the LHS element
- **RHS** – Type of the RHS element

### Parameters

- **lhs** – The first array
- **rhs** – The second array

**Returns** The element-wise product of the two arrays



```
template<class LHS, class RHS,
typename std::enable_if_t< detail::isArrayOpArray< LHS, RHS >(),
int > = 0> auto operator/ (LHS &&lhs,
RHS &&rhs) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::DescriptorType_t< LHS,
RHS >, detail::Divide, LHS, RHS >
```

Element-wise array division.

Performs element-wise division on two arrays. They must both be the same size and of the same data type.

**Template Parameters**

- **LHS** – Type of the LHS element
- **RHS** – Type of the RHS element

**Parameters**

- **lhs** – The first array
- **rhs** – The second array

**Returns** The element-wise division of the two arrays

```
template<class LHS, class RHS,
typename std::enable_if_t< detail::isArrayOpArray< LHS, RHS >(),
int > = 0> auto operator< (LHS &&lhs,
RHS &&rhs) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::DescriptorType_t< LHS,
RHS >, detail::LessThan, LHS, RHS >
```

Element-wise array comparison, checking whether  $a < b$  for all  $a, b$  in input arrays.

Performs an element-wise comparison on two arrays, checking if the first value is less than the second. They must both be the same size and of the same data type.

**Template Parameters**

- **LHS** – Type of the LHS element
- **RHS** – Type of the RHS element

**Parameters**

- **lhs** – The first array
- **rhs** – The second array

**Returns** The element-wise comparison of the two arrays

```
template<class LHS, class RHS,
typename std::enable_if_t< detail::isArrayOpArray< LHS, RHS >(),
int > = 0> auto operator> (LHS &&lhs,
RHS &&rhs) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::DescriptorType_t< LHS,
RHS >, detail::GreaterThan, LHS, RHS >
```

Element-wise array comparison, checking whether  $a > b$  for all  $a, b$  in input arrays.

Performs an element-wise comparison on two arrays, checking if the first value is greater than the second. They must both be the same size and of the same data type.

**Template Parameters**

- **LHS** – Type of the LHS element
- **RHS** – Type of the RHS element

**Parameters**

- **lhs** – The first array
- **rhs** – The second array

**Returns** The element-wise comparison of the two arrays

```
template<class LHS, class RHS,
typename std::enable_if_t< detail::isArrayOpArray< LHS, RHS >(),
int > = 0> auto operator<= (LHS &&lhs,
RHS &&rhs) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::DescriptorType_t< LHS,
RHS >, detail::LessThanEqual, LHS, RHS >
```

Element-wise array comparison, checking whether  $a \leq b$  for all  $a, b$  in input arrays.

Performs an element-wise comparison on two arrays, checking if the first value is less than or equal to the second. They must both be the same size and of the same data type.

**Template Parameters**

- **LHS** – Type of the LHS element
- **RHS** – Type of the RHS element

**Parameters**

- **lhs** – The first array
- **rhs** – The second array

**Returns** The element-wise comparison of the two arrays

```
template<class LHS, class RHS,
typename std::enable_if_t< detail::isArrayOpArray< LHS, RHS >(),
int > = 0> auto operator>= (LHS &&lhs,
RHS &&rhs) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::DescriptorType_t< LHS,
RHS >, detail::GreaterThanEqual, LHS, RHS >
```

Element-wise array comparison, checking whether  $a \geq b$  for all  $a, b$  in input arrays.

Performs an element-wise comparison on two arrays, checking if the first value is greater than or equal to the second. They must both be the same size and of the same data type.

**Template Parameters**

- **LHS** – Type of the LHS element
- **RHS** – Type of the RHS element

**Parameters**

- **lhs** – The first array
- **rhs** – The second array

**Returns** The element-wise comparison of the two arrays

```
template<class LHS, class RHS,
typename std::enable_if_t< detail::isArrayOpArray< LHS, RHS >(),
int > = 0> auto operator== (LHS &&lhs,
RHS &&rhs) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::DescriptorType_t< LHS,
RHS >, detail::ElementWiseEqual, LHS, RHS >
```

Element-wise array comparison, checking whether  $a == b$  for all  $a, b$  in input arrays.

Performs an element-wise comparison on two arrays, checking if the first value is equal to the second. They must both be the same size and of the same data type.

**Template Parameters**

- **LHS** – Type of the LHS element
- **RHS** – Type of the RHS element

**Parameters**

- **lhs** – The first array
- **rhs** – The second array

**Returns** The element-wise comparison of the two arrays

```
template<class LHS, class RHS,
typename std::enable_if_t< detail::isArrayOpArray< LHS, RHS >(),
int > = 0> auto operator!= (LHS &&lhs,
RHS &&rhs) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::DescriptorType_t< LHS,
RHS >, detail::ElementWiseNotEqual, LHS, RHS >
```

Element-wise array comparison, checking whether  $a \neq b$  for all  $a, b$  in input arrays.

Performs an element-wise comparison on two arrays, checking if the first value is not equal to the second. They must both be the same size and of the same data type.

**Template Parameters**

- **LHS** – Type of the LHS element
- **RHS** – Type of the RHS element

**Parameters**

- **lhs** – The first array
- **rhs** – The second array

**Returns** The element-wise comparison of the two arrays

```
template<class VAL, typename std::enable_if_t< detail::isArrayOp< VAL >(),
int > = 0> auto operator- (VAL &&val) LIBRAPID_RELEASE_NOEXCEPT -> detail::Function< typetraits::
detail::Neg, VAL >
```

Negate each element in the array.

**Template Parameters** **VAL** – Type to negate

**Parameters** **val** – The input array or function

**Returns** Negation function object

namespace **detail**

## Functions

```
template<typename desc, typename Functor, typename ...Args>
auto makeFunction(Args&&... args)
```

Construct a new function object with the given functor type and arguments.

**Template Parameters**

- **desc** – Functor descriptor
- **Functor** – Function type
- **Args** – Argument types

**Parameters** **args** – Arguments passed to the function (forwarded)

**Returns** A new Function instance

```
template<typename VAL>
constexpr bool isArrayOp()
```

```
template<typename LHS, typename RHS>
constexpr bool isArrayOpArray()
```

```
template<typename LHS, typename RHS>
constexpr bool isArrayOpWithScalar()
```

struct **Plus**

*#include <operations.hpp>*

### Public Functions

```
template<typename T, typename V>  
inline auto operator() (const T &lhs, const V &rhs) const  
  
template<typename Packet>  
inline auto packet(const Packet &lhs, const Packet &rhs) const
```

struct **Minus**

```
#include <operations.hpp>
```

### Public Functions

```
template<typename T, typename V>  
inline auto operator() (const T &lhs, const V &rhs) const  
  
template<typename Packet>  
inline auto packet(const Packet &lhs, const Packet &rhs) const
```

struct **Multiply**

```
#include <operations.hpp>
```

### Public Functions

```
template<typename T, typename V>  
inline auto operator() (const T &lhs, const V &rhs) const  
  
template<typename Packet>  
inline auto packet(const Packet &lhs, const Packet &rhs) const
```

struct **Divide**

```
#include <operations.hpp>
```

### Public Functions

```
template<typename T, typename V>  
inline auto operator() (const T &lhs, const V &rhs) const  
  
template<typename Packet>  
inline auto packet(const Packet &lhs, const Packet &rhs) const
```

struct **LessThan**

```
#include <operations.hpp>
```

### Public Functions

```
template<typename T, typename V>
inline auto operator() (const T &lhs, const V &rhs) const

template<typename Packet>
inline auto packet(const Packet &lhs, const Packet &rhs) const
```

```
struct GreaterThan
```

```
#include <operations.hpp>
```

### Public Functions

```
template<typename T, typename V>
inline auto operator() (const T &lhs, const V &rhs) const

template<typename Packet>
inline auto packet(const Packet &lhs, const Packet &rhs) const
```

```
struct LessThanEqual
```

```
#include <operations.hpp>
```

### Public Functions

```
template<typename T, typename V>
inline auto operator() (const T &lhs, const V &rhs) const

template<typename Packet>
inline auto packet(const Packet &lhs, const Packet &rhs) const
```

```
struct GreaterThanEqual
```

```
#include <operations.hpp>
```

### Public Functions

```
template<typename T, typename V>
inline auto operator() (const T &lhs, const V &rhs) const

template<typename Packet>
inline auto packet(const Packet &lhs, const Packet &rhs) const
```

```
struct ElementWiseEqual
```

```
#include <operations.hpp>
```

### Public Functions

```
template<typename T, typename V>  
inline auto operator() (const T &lhs, const V &rhs) const  
  
template<typename Packet>  
inline auto packet(const Packet &lhs, const Packet &rhs) const
```

```
struct ElementWiseNotEqual  
    #include <operations.hpp>
```

### Public Functions

```
template<typename T, typename V>  
inline auto operator() (const T &lhs, const V &rhs) const  
  
template<typename Packet>  
inline auto packet(const Packet &lhs, const Packet &rhs) const
```

```
struct Neg  
    #include <operations.hpp>
```

### Public Functions

```
template<typename T>  
inline auto operator() (const T &arg) const  
  
template<typename Packet>  
inline auto packet(const Packet &arg) const
```

```
struct Sin  
    #include <operations.hpp>
```

### Public Functions

```
template<typename T>  
inline auto operator() (const T &arg) const  
  
template<typename Packet>  
inline auto packet(const Packet &arg) const
```

```
struct Cos  
    #include <operations.hpp>
```

**Public Functions**

```
template<typename T>
inline auto operator()(const T &arg) const
```

```
template<typename Packet>
inline auto packet(const Packet &arg) const
```

```
struct Tan
```

```
#include <operations.hpp>
```

**Public Functions**

```
template<typename T>
inline auto operator()(const T &arg) const
```

```
template<typename Packet>
inline auto packet(const Packet &arg) const
```

```
struct Asin
```

```
#include <operations.hpp>
```

**Public Functions**

```
template<typename T>
inline auto operator()(const T &arg) const
```

```
template<typename Packet>
inline auto packet(const Packet &arg) const
```

```
struct Acos
```

```
#include <operations.hpp>
```

**Public Functions**

```
template<typename T>
inline auto operator()(const T &arg) const
```

```
template<typename Packet>
inline auto packet(const Packet &arg) const
```

```
struct Atan
```

```
#include <operations.hpp>
```

### Public Functions

```
template<typename T>  
inline auto operator()(const T &arg) const
```

```
template<typename Packet>  
inline auto packet(const Packet &arg) const
```

```
struct Sinh
```

```
    #include <operations.hpp>
```

### Public Functions

```
template<typename T>  
inline auto operator()(const T &arg) const
```

```
template<typename Packet>  
inline auto packet(const Packet &arg) const
```

```
struct Cosh
```

```
    #include <operations.hpp>
```

### Public Functions

```
template<typename T>  
inline auto operator()(const T &arg) const
```

```
template<typename Packet>  
inline auto packet(const Packet &arg) const
```

```
struct Tanh
```

```
    #include <operations.hpp>
```

### Public Functions

```
template<typename T>  
inline auto operator()(const T &arg) const
```

```
template<typename Packet>  
inline auto packet(const Packet &arg) const
```

```
struct Exp
```

```
    #include <operations.hpp>
```



**Public Functions**

```
template<typename T>
inline auto operator()(const T &arg) const
```

```
template<typename Packet>
inline auto packet(const Packet &arg) const
```

```
struct Log
```

```
#include <operations.hpp>
```

**Public Functions**

```
template<typename T>
inline auto operator()(const T &arg) const
```

```
template<typename Packet>
inline auto packet(const Packet &arg) const
```

```
struct Log2
```

```
#include <operations.hpp>
```

**Public Functions**

```
template<typename T>
inline auto operator()(const T &arg) const
```

```
template<typename Packet>
inline auto packet(const Packet &arg) const
```

```
struct Log10
```

```
#include <operations.hpp>
```

**Public Functions**

```
template<typename T>
inline auto operator()(const T &arg) const
```

```
template<typename Packet>
inline auto packet(const Packet &arg) const
```

```
struct Sqrt
```

```
#include <operations.hpp>
```

### Public Functions

```
template<typename T>  
inline auto operator() (const T &arg) const
```

```
template<typename Packet>  
inline auto packet(const Packet &arg) const
```

```
struct Cbrt
```

```
    #include <operations.hpp>
```

### Public Functions

```
template<typename T>  
inline auto operator() (const T &arg) const
```

```
template<typename Packet>  
inline auto packet(const Packet &arg) const
```

```
struct Abs
```

```
    #include <operations.hpp>
```

### Public Functions

```
template<typename T>  
inline auto operator() (const T &arg) const
```

```
template<typename Packet>  
inline auto packet(const Packet &arg) const
```

```
struct Floor
```

```
    #include <operations.hpp>
```

### Public Functions

```
template<typename T>  
inline auto operator() (const T &arg) const
```

```
template<typename Packet>  
inline auto packet(const Packet &arg) const
```

```
struct Ceil
```

```
    #include <operations.hpp>
```

## Public Functions

```
template<typename T>
inline auto operator() (const T &arg) const
```

```
template<typename Packet>
inline auto packet(const Packet &arg) const
```

namespace **typetraits**

## Typedefs

```
template<typename ...Args>
```

```
using DescriptorType_t = typename DescriptorType<Args...>::Type
```

A simplification of the *DescriptorType* to reduce code size

**See also:**

*DescriptorType*

**Template Parameters** **Args** – Input types

```
template<typename Descriptor1, typename Descriptor2>
```

```
struct DescriptorMerger
```

*#include <operations.hpp>* Merge together two Descriptor types. Two trivial operations will result in another trivial operation, while any other combination will result in a Combined operation.

**Template Parameters**

- **Descriptor1** – The first descriptor
- **Descriptor2** – The second descriptor

## Public Types

```
using Type = ::librapid::detail::descriptor::Combined
```

```
template<typename Descriptor1>
```

```
struct DescriptorMerger<Descriptor1, Descriptor1>
```

*#include <operations.hpp>*

## Public Types

```
using Type = Descriptor1
```

```
template<typename T>
```

```
struct DescriptorExtractor
```

*#include <operations.hpp>* Extracts the Descriptor type of the provided type.

**Template Parameters** **T** – The type to extract the descriptor from

## Public Types

using **Type** = ::librapid::detail::descriptor::Trivial

```
template<typename ShapeType, typename StorageType> ArrayContainer< ShapeType,  
StorageType > >
```

*#include <operations.hpp>* Extracts the Descriptor type of an ArrayContainer object. In this case, the Descriptor is Trivial

### Template Parameters

- **ShapeType** – The shape type of the ArrayContainer
- **StorageType** – The storage type of the ArrayContainer

## Public Types

using **Type** = ::librapid::detail::descriptor::Trivial

```
template<typename T> ArrayView< T > >
```

*#include <operations.hpp>* Extracts the Descriptor type of an ArrayView object

**Template Parameters** **T** – The Array type of the ArrayView

## Public Types

using **Type** = ::librapid::detail::descriptor::Trivial

```
template<typename Descriptor, typename Functor, typename...  
Args> Function< Descriptor, Functor, Args... > >
```

*#include <operations.hpp>* Extracts the Descriptor type of a Function object

### Template Parameters

- **Descriptor** – The descriptor of the Function
- **Functor** – The functor type of the Function
- **Args** – The argument types of the Function

## Public Types

using **Type** = Descriptor

```
template<typename First, typename ...Rest>
```

```
struct DescriptorType
```

*#include <operations.hpp>*

Return the combined Descriptor type of the provided types

Allows a number of Descriptor types to be merged together into a single Descriptor type. The Descriptors used are extracted from the of the provided types.

### Template Parameters

- **First** – The first type to merge
- **Rest** – The remaining types

- **First** – The first type to merge
- **Rest** – The remaining types

### Public Types

```
using FirstType = std::decay_t<First>
```

```
using FirstDescriptor = typename DescriptorExtractor<FirstType>::Type
```

```
using RestDescriptor = decltype(impl::descriptorExtractor<Rest...>())
```

```
using Type = typename DescriptorMerger<FirstDescriptor, RestDescriptor>::Type
```

```
template<> Plus >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
```

```
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)
```

```
template<typename ...Args>
```

```
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename First, typename Second>
```

```
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)
```

```
template<typename ...Args>
```

```
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "plus"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "addArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "addArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "addArraysScalarLhs"
```

```
template<> Minus >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "minus"

static constexpr const char *filename = "arithmetic"

static constexpr const char *kernelName = "subArrays"

static constexpr const char *kernelNameScalarRhs = "subArraysScalarRhs"

static constexpr const char *kernelNameScalarLhs = "subArraysScalarLhs"
```

```
template<> Multiply >
#include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "multiply"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "mulArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "mulArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "mulArraysScalarLhs"
```

```
template<> Divide >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
```

```
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)
```

```
template<typename ...Args>
```

```
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename First, typename Second>
```

```
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)
```

```
template<typename ...Args>
```

```
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "divide"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "divArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "divArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "divArraysScalarLhs"
```

```
template<> LessThan >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "less than"

static constexpr const char *filename = "arithmetic"

static constexpr const char *kernelName = "lessThanArrays"

static constexpr const char *kernelNameScalarRhs = "lessThanArraysScalarRhs"

static constexpr const char *kernelNameScalarLhs = "lessThanArraysScalarLhs"
```

```
template<> GreaterThan >
#include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```



### Public Static Attributes

```
static constexpr const char *name = "greater than"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "greaterThanArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "greaterThanArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "greaterThanArraysScalarLhs"
```

```
template<> LessThanEqual >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
```

```
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)
```

```
template<typename ...Args>
```

```
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename First, typename Second>
```

```
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)
```

```
template<typename ...Args>
```

```
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "less than or equal"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "lessThanEqualArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "lessThanEqualArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "lessThanEqualArraysScalarLhs"
```

```
template<> GreaterThanEqual >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "greater than or equal"

static constexpr const char *filename = "arithmetic"

static constexpr const char *kernelName = "greaterThanEqualArrays"

static constexpr const char *kernelNameScalarRhs = "greaterThanEqualArraysScalarRhs"

static constexpr const char *kernelNameScalarLhs = "greaterThanEqualArraysScalarLhs"
```

```
template<> ElementWiseEqual >
#include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)

template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename First, typename Second>
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "element wise equal"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "elementWiseEqualArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "elementWiseEqualArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "elementWiseEqualArraysScalarLhs"
```

```
template<> ElementWiseNotEqual >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename T1, typename T2>
```

```
static inline constexpr const char *getKernelNameImpl(std::tuple<T1, T2> args)
```

```
template<typename ...Args>
```

```
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename First, typename Second>
```

```
static inline auto getShapeImpl(const std::tuple<First, Second> &tup)
```

```
template<typename ...Args>
```

```
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "element wise not equal"
```

```
static constexpr const char *filename = "arithmetic"
```

```
static constexpr const char *kernelName = "elementWiseNotEqualArrays"
```

```
static constexpr const char *kernelNameScalarRhs = "elementWiseNotEqualArraysScalarRhs"
```

```
static constexpr const char *kernelNameScalarLhs = "elementWiseNotEqualArraysScalarLhs"
```

```
template<> Neg >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "negate"
```

```
static constexpr const char *filename = "negate"
```

```
static constexpr const char *kernelName = "negateArrays"
```

```
template<> Sin >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "sin"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "sinArrays"
```

```
template<> Cos >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "cos"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "cosArrays"
```

```
template<> Tan >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "tan"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "tanArrays"
```

```
template<> Asin >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "arcsin"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "asinArrays"
```

```
template<> Acos >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "arcsin"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "acosArrays"
```

```
template<> Atan >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "arctan"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "atanArrays"
```

```
template<> Sinh >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "hyperbolic sine"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "sinhArrays"
```

```
template<> Cosh >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "hyperbolic cosine"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "coshArrays"
```

```
template<> Tanh >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "hyperbolic tangent"
```

```
static constexpr const char *filename = "trigonometry"
```

```
static constexpr const char *kernelName = "tanhArrays"
```

```
template<> Exp >
```

```
    #include <operations.hpp>
```



### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "exponent"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "expArrays"
```

```
template<> Log >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "logarithm"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "logArrays"
```

```
template<> Log2 >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "logarithm base 2"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "log2Arrays"
```

```
template<> Log10 >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "logarithm base 10"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "log10Arrays"
```

```
template<> Sqrt >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "square root"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "sqrtArrays"
```

```
template<> Cbrt >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "cube root"
```

```
static constexpr const char *filename = "expLogPow"
```

```
static constexpr const char *kernelName = "cbrtArrays"
```

```
template<> Abs >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "absolute value"
```

```
static constexpr const char *filename = "abs"
```

```
static constexpr const char *kernelName = "absArrays"
```

```
template<> Floor >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>  
static inline constexpr const char *getKernelName(std::tuple<Args...> args)
```

```
template<typename ...Args>  
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "floor"
```

```
static constexpr const char *filename = "floorCeilRound"
```

```
static constexpr const char *kernelName = "floorArrays"
```

```
template<> Ceil >
```

```
    #include <operations.hpp>
```

### Public Static Functions

```
template<typename ...Args>
static inline constexpr const char *getKernelName(std::tuple<Args...> args)

template<typename ...Args>
static inline auto getShape(const std::tuple<Args...> &args)
```

### Public Static Attributes

```
static constexpr const char *name = "ceiling"

static constexpr const char *filename = "floorCeilRound"

static constexpr const char *kernelName = "ceilArrays"
```

```
namespace impl
```

### Functions

```
template<typename ...Rest>
constexpr auto descriptorExtractor()
```

A constexpr function which supports the *DescriptorType* for multi-type inputs

**Template Parameters** Rest –  
**Returns**

#### 1.3.2.7 Size Type

**Warning:** doxygenfile: Cannot find file “librapid/include/librapid/array/sizeType.hpp”

#### 1.3.2.8 Stride Tools

```
namespace librapid
```

```
template<typename T = size_t, size_t N = 32>
```

```
class Stride : public librapid::Shape<size_t, 32>
```

*#include <strideTools.hpp>* A *Stride* is a vector of integers that describes the distance between elements in each dimension of an ArrayContainer object. This can be used to access elements in a non-trivial order, or to access a sub-array of an ArrayContainer object. The *Stride* class inherits from the Shape class.

**See also:**

Shape

#### Template Parameters

- **T** – The type of the *Stride*. Must be an integer type.

- **N** – The number of dimensions in the *Stride*.

## Public Functions

**Stride**() = default

Default Constructor.

**Stride**(const Shape<T, N> &shape)

Construct a *Stride* from a Shape object. This will assume that the data represented by the Shape object is a contiguous block of memory, and will calculate the corresponding strides based on this.

**Parameters** **shape** –

**Stride**(const Stride &other) = default

Copy a *Stride* object

**Parameters** **other** – The *Stride* object to copy.

**Stride**(Stride &&other) noexcept = default

Move a *Stride* object

**Parameters** **other** – The *Stride* object to move.

Stride &**operator**=(const Stride &other) = default

Assign a *Stride* object to this *Stride* object.

**Parameters** **other** – The *Stride* object to assign.

Stride &**operator**=(Stride &&other) noexcept = default

Move a *Stride* object to this *Stride* object.

**Parameters** **other** – The *Stride* object to move.

namespace **typetraits**

## Functions

**LIBRAPID\_DEFINE\_AS\_TYPE** (typename T COMMA size\_t N, Stride< T COMMA N >)

### 1.3.2.9 Storage

namespace **librapid**

template<typename **Scalar\_**>

class **Storage**

*#include <storage.hpp>*

## Public Types

```
using Scalar = Scalar_

using RawPointer = Scalar*

using ConstRawPointer = const Scalar*

using Pointer = std::shared_ptr<Scalar>

using ConstPointer = std::shared_ptr<const Scalar>

using Reference = Scalar&

using ConstReference = const Scalar&

using SizeType = size_t

using DifferenceType = ptrdiff_t

using Iterator = RawPointer

using ConstIterator = ConstRawPointer

using ReverseIterator = std::reverse_iterator<Iterator>

using ConstReverseIterator = std::reverse_iterator<ConstIterator>
```

## Public Functions

**Storage()** = default

Default constructor.

explicit **Storage**(SizeType size)

Create a *Storage* object with size elements

**Parameters** **size** – Number of elements to allocate

explicit **Storage**(Scalar \*begin, Scalar \*end, bool ownsData)

**Storage**(SizeType size, ConstReference value)

Create a *Storage* object with size elements, each initialized to value.

**Parameters**

- **size** – Number of elements to allocate
- **value** – Value to initialize each element to

**Storage**(const Storage &other)

Create a *Storage* object from another *Storage* object. Additionally a custom allocator can be used. The data is **NOT** copied; it is referenced by the new *Storage* object. For a deep copy, use the *copy()* method.

**Parameters** *other* – *Storage* object to copy

**Storage**(Storage &&other) noexcept

Move a *Storage* object into this object.

**Parameters** *other* – *Storage* object to move

template<typename V>

**Storage**(const std::initializer\_list<V> &list)

Create a *Storage* object from an std::initializer\_list

**Template Parameters** *V* – Type of the elements in the initializer list

**Parameters**

- **list** – Initializer list to copy
- **alloc** – Allocator to use

template<typename V>

explicit **Storage**(const std::vector<V> &vec)

Create a *Storage* object from a std::vector

**Template Parameters** *V* – Type of the elements in the vector

**Parameters** *vec* – Vector to copy

Storage &**operator**=(const Storage &other)

Assignment operator for a *Storage* object

**Parameters** *other* – *Storage* object to copy

**Returns** \*this

**Storage & operator= (Storage &&other) LIBRAPID\_RELEASE\_NOEXCEPT**

Move assignment operator for a *Storage* object

**Parameters** *other* – *Storage* object to move

**Returns** \*this

**~Storage()**

Free a *Storage* object.

void **set**(const Storage &other)

Set this storage object to reference the same data as *other*.

**Parameters** *other* – *Storage* object to reference

Storage **toHostStorage()** const

Return a *Storage* object on the host with the same data as this *Storage* object (mainly for use with CUDA or OpenCL)

**Returns**

Storage **toHostStorageUnsafe()** const

Same as *toHostStorage()* but does not necessarily copy the data.

**Returns** *Storage* object on the host

Storage **copy()** const

Create a deep copy of this *Storage* object.

**Returns** Deep copy of this *Storage* object



void **resize**(SizeType newSize)  
 Resize a *Storage* object to size elements. Existing elements are preserved.  
**Parameters** **size** – New size of the *Storage* object

void **resize**(SizeType newSize, int)  
 Resize a *Storage* object to size elements. Existing elements are not preserved  
**Parameters** **size** – New size of the *Storage* object

SizeType **size**() const noexcept  
 Return the number of elements in the *Storage* object  
**Returns**

ConstReference **operator[]**(SizeType index) const  
 Const access to the element at index **index**  
**Parameters** **index** – Index of the element to access  
**Returns** Const reference to the element at index **index**

Reference **operator[]**(SizeType index)  
 Access to the element at index **index**  
**Parameters** **index** – Index of the element to access  
**Returns** Reference to the element at index **index**

Pointer **data**() const noexcept

RawPointer **begin**() noexcept

RawPointer **end**() noexcept

ConstIterator **begin**() const noexcept

ConstIterator **end**() const noexcept

ConstIterator **cbegin**() const noexcept

ConstIterator **cend**() const noexcept

ReverseIterator **rbegin**() noexcept

ReverseIterator **rend**() noexcept

ConstReverseIterator **rbegin**() const noexcept

ConstReverseIterator **rend**() const noexcept

ConstReverseIterator **crbegin**() const noexcept

ConstReverseIterator **crend**() const noexcept

template<typename V>  
 auto **fromData**(const std::initializer\_list<V> &list) -> Storage

template<typename V>  
 auto **fromData**(const std::vector<V> &vec) -> Storage

template<typename ShapeType>  
 auto **defaultShape**() -> ShapeType

## Public Static Functions

```
template<typename V>
static Storage fromData(const std::initializer_list<V> &vec)
```

```
template<typename V>
static Storage fromData(const std::vector<V> &vec)
```

```
template<typename ShapeType>
static ShapeType defaultShape()
```

## Private Functions

```
template<typename P>
void initData(P begin, P end)
```

Copy data from begin to end into this *Storage* object

**Template Parameters** **P** – Pointer type

**Parameters**

- **begin** – Beginning of data to copy
- **end** – End of data to copy

```
template<typename P>
void initData(P begin, SizeType size)
```

```
void resizeImpl(SizeType newSize, int)
```

Resize the *Storage* Object to **newSize** elements, retaining existing data.

**Parameters** **newSize** – New size of the *Storage* object

```
void resizeImpl(SizeType newSize)
```

Resize the *Storage* object to **newSize** elements. Note this does not initialize the new elements or maintain existing data.

**Parameters** **newSize** – New size of the *Storage* object

## Private Members

Pointer **m\_begin** = nullptr

SizeType **m\_size** = 0

bool **m\_ownsData** = true

```
template<typename Scalar_, size_t... Size_>
```

```
class FixedStorage
```

```
    #include <storage.hpp>
```

## Public Types

using **Scalar** = Scalar\_

using **Pointer** = Scalar\*

using **ConstPointer** = const Scalar\*

using **Reference** = Scalar&

using **ConstReference** = const Scalar&

using **SizeType** = size\_t

using **DifferenceType** = ptrdiff\_t

using **Iterator** = typename std::array<Scalar, product<Size\_...>()>::iterator

using **ConstIterator** = typename std::array<Scalar, product<Size\_...>()>::const\_iterator

using **ReverseIterator** = std::reverse\_iterator<Iterator>

using **ConstReverseIterator** = std::reverse\_iterator<ConstIterator>

## Public Functions

### **FixedStorage()**

Default constructor.

explicit **FixedStorage**(const Scalar &value)

Create a *FixedStorage* object filled with value

**Parameters** **value** – Value to fill the *FixedStorage* object with

**FixedStorage**(const FixedStorage &other)

Create a *FixedStorage* object from another *FixedStorage* object

**Parameters** **other** – *FixedStorage* object to copy

**FixedStorage**(FixedStorage &&other) noexcept

Move constructor for a *FixedStorage* object

**Parameters** **other** – *FixedStorage* object to move

explicit **FixedStorage**(const std::initializer\_list<Scalar> &list)

Create a *FixedStorage* object from a std::initializer\_list

**Template Parameters** **V** – Type of the elements in the initializer list

**Parameters** **list** – Initializer list to copy

explicit **FixedStorage**(const std::vector<Scalar> &vec)

Create a *FixedStorage* object from a std::vector

**Template Parameters** **V** – Type of the elements in the vector

**Parameters** **vec** – Vector to copy

**FixedStorage** &**operator**=(const **FixedStorage** &other)

Assignment operator for a *FixedStorage* object

**Parameters** **other** – *FixedStorage* object to copy

**Returns** \*this

**FixedStorage** &**operator**=(**FixedStorage** &&other) noexcept

Move assignment operator for a *FixedStorage* object

**Parameters** **other** – *FixedStorage* object to move

**Returns** \*this

**~FixedStorage**() = default

Free a *FixedStorage* object.

void **resize**(SizeType newSize)

Resize a *Storage* object to size elements. Existing elements are preserved.

**Parameters** **size** – New size of the *Storage* object

void **resize**(SizeType newSize, int)

Resize a *Storage* object to size elements. Existing elements are not preserved

**Parameters** **size** – New size of the *Storage* object

SizeType **size**() const noexcept

Return the number of elements in the *FixedStorage* object

**Returns** Number of elements in the *FixedStorage* object

**FixedStorage** **copy**() const

Create a copy of the *FixedStorage* object.

**Returns** Copy of the *FixedStorage* object

ConstReference **operator**[](SizeType index) const

Const access to the element at index index

**Parameters** **index** – Index of the element to access

**Returns** Const reference to the element at index index

Reference **operator**[](SizeType index)

Access to the element at index index

**Parameters** **index** – Index of the element to access

**Returns** Reference to the element at index index

Pointer **data**() const noexcept

Iterator **begin**() noexcept

Iterator **end**() noexcept

ConstIterator **begin**() const noexcept

ConstIterator **end**() const noexcept

ConstIterator **cbegin**() const noexcept

ConstIterator **cend**() const noexcept

```
ReverseIterator rbegin() noexcept
ReverseIterator rend() noexcept
ConstReverseIterator rbegin() const noexcept
ConstReverseIterator rend() const noexcept
ConstReverseIterator crbegin() const noexcept
ConstReverseIterator crend() const noexcept
template<typename ShapeType>
auto defaultShape() -> ShapeType
```

### Public Static Functions

```
template<typename ShapeType>
static ShapeType defaultShape()
```

### Public Static Attributes

```
static constexpr SizeType Size = product<Size_...>()
```

### Private Members

```
std::array<Scalar, Size> m_data
```

namespace **detail**

### Functions

```
template<typename T>
void safeDeallocate(T *ptr, size_t size)
```

Safely deallocate memory for `size` elements, using an `std::allocator alloc`. If the object cannot be trivially destroyed, the destructor will be called on each element of the data, ensuring that it is safe to free the allocated memory.

**Template Parameters** **A** – The allocator type  
**Parameters**

- **alloc** – The allocator object
- **ptr** – The pointer to free
- **size** – The number of elements of type `in` in the memory block

```
template<typename T>
std::shared_ptr<T> safeAllocate(size_t size)
```

Safely allocate memory for `size` elements using the allocator `alloc`. If the data can be trivially default constructed, then the constructor is not called and no data is initialized. Otherwise, the correct default constructor will be called for each element in the data, making sure the returned pointer is safe to use.

**See also:**

*[safeDeallocate](#)*

**Template Parameters** **A** – The allocator type to use  
**Parameters**

- **alloc** – The allocator object to use
- **size** – Number of elements to allocate

**Returns** Pointer to the first element

```
template<typename T>
std::shared_ptr<T> safePointerCopy(T *ptr, size_t size, bool ownsData)
```

Safely copy a pointer to a shared pointer. If **ownsData** is true, then the shared pointer will be initialized with a custom deleter that will call **safeDeallocate** on the pointer. Otherwise, the shared pointer will be initialized with a no-op deleter.

**Template Parameters** **T** – Type of the pointer

**Parameters**

- **ptr** – Raw pointer to copy
- **ownsData** – Whether the shared pointer should own the data

**Returns** Shared pointer to the data

```
template<typename T>
std::shared_ptr<T> safePointerCopy(const std::shared_ptr<T> &ptr, size_t size, bool ownsData =
                                     true)
```

namespace **typetraits**

## Functions

**LIBRAPID\_DEFINE\_AS\_TYPE**(typename Scalar, Storage<Scalar>)

```
template<typename Scalar_>
struct TypeInfo<Storage<Scalar_>>
    #include <storage.hpp>
```

## Public Types

using **Scalar** = **Scalar\_**

using **Backend** = backend::CPU

## Public Static Attributes

```
static constexpr bool isLibRapidType = true

template<typename Scalar_, size_t... Dims>
struct TypeInfo<FixedStorage<Scalar_, Dims...>>
    #include <storage.hpp>
```

## Public Types

```
using Scalar = Scalar_
```

```
using Backend = backend::CPU
```

## Public Static Attributes

```
static constexpr bool isLibRapidType = true
```

```
template<typename T>
```

```
struct IsStorage : public std::false_type
```

```
    #include <storage.hpp>
```

```
template<typename Scalar>
```

```
struct IsStorage<Storage<Scalar>> : public std::true_type
```

```
    #include <storage.hpp>
```

```
template<typename T>
```

```
struct IsFixedStorage : public std::false_type
```

```
    #include <storage.hpp>
```

```
template<typename Scalar, size_t... Size>
```

```
struct IsFixedStorage<FixedStorage<Scalar, Size...>> : public std::true_type
```

```
    #include <storage.hpp>
```

### 1.3.2.10 OpenCL Storage

### 1.3.2.11 CUDA Storage

## Defines

```
CUDA_REF_OPERATOR(OP)
```

```
CUDA_REF_OPERATOR_NO_ASSIGN(OP)
```

```
namespace librapid
```

```
    template<typename Scalar_>
```

```
    class CudaStorage
```

```
        #include <cudaStorage.hpp>
```

## Public Types

```
using Scalar = Scalar_  
  
using Pointer = std::shared_ptr<Scalar>  
  
using ConstPointer = const std::shared_ptr<Scalar>  
  
using Reference = Scalar&  
  
using ConstReference = const Scalar&  
  
using DifferenceType = std::ptrdiff_t  
  
using SizeType = std::size_t
```

## Public Functions

**CudaStorage()** = default

Default constructor; initializes with nullptr.

explicit **CudaStorage**(SizeType size)

Create a *CudaStorage* object with `elements`. The data is not initialized.

**Parameters** **size** – Number of elements

**CudaStorage**(SizeType size, ConstReference value)

Create a *CudaStorage* object with `elements`. The data is initialized to `value`.

**Parameters**

- **size** – Number of elements
- **value** – Value to fill with

**CudaStorage**(Scalar \*begin, SizeType size, bool independent)

**CudaStorage**(const CudaStorage &other)

Create a new *CudaStorage* object from an existing one.

**Parameters** **other** – The *CudaStorage* to copy

**CudaStorage**(CudaStorage &&other) noexcept

Create a new *CudaStorage* object from a temporary one, moving the data

**Parameters** **other** – The array to move

**CudaStorage**(const std::initializer\_list<Scalar> &list)

Create a *CudaStorage* object from an std::initializer\_list

**Parameters** **list** – Initializer list of elements

explicit **CudaStorage**(const std::vector<Scalar> &vec)

Create a *CudaStorage* object from an std::vector of values

**Parameters** **vec** – The vector to fill with



CudaStorage &**operator**=(const CudaStorage &other)

Assignment operator for a *CudaStorage* object

**Parameters** **other** – *CudaStorage* object to copy

**Returns** \*this

CudaStorage &**operator**=(CudaStorage &&other) noexcept

Move assignment operator for a *CudaStorage* object

**Parameters** **other** – *CudaStorage* object to move

**Returns** \*this

~CudaStorage()

Free a *CudaStorage* object.

void **set**(const CudaStorage &other)

Set this *CudaStorage* object to reference the same data as **other**.

**Parameters** **other** – *CudaStorage* object to reference

CudaStorage **copy**() const

Create a deep copy of this *CudaStorage* object.

**Returns** Deep copy of this *CudaStorage* object

void **resize**(SizeType newSize)

Resize a *CudaStorage* object to **size** elements. Existing elements are preserved where possible.

**See also:**

*resize(SizeType, int)*

**Parameters** **size** – Number of elements

void **resize**(SizeType newSize, int)

Resize a *CudaStorage* object to **size** elements. Existing elements are not preserved. This method of resizing is faster and more efficient than the version which preserves the original data, but of course, this has the drawback that data will be lost.

**Parameters** **size** – Number of elements

SizeType **size**() const noexcept

Return the number of elements in the *CudaStorage* object.

**Returns** The number of elements

detail::CudaRef<Scalar> **operator**[] (SizeType index) const

detail::CudaRef<Scalar> **operator**[] (SizeType index)

Pointer **data**() const noexcept

Return the underlying pointer to the data

**Returns** The underlying pointer to the data

Pointer **begin**() const noexcept

Returns the pointer to the first element of the *CudaStorage* object

**Returns** Pointer to the first element of the *CudaStorage* object

Pointer **end**() const noexcept

Returns the pointer to the last element of the *CudaStorage* object

**Returns** A pointer to the last element of the *CudaStorage*

template<typename **V**>

```
auto fromData(const std::initializer_list<V> &list) -> CudaStorage
```

```
template<typename V>
```

```
auto fromData(const std::vector<V> &vec) -> CudaStorage
```

### Public Static Functions

```
template<typename ShapeType>
```

```
static ShapeType defaultShape()
```

```
template<typename V>
```

```
static CudaStorage fromData(const std::initializer_list<V> &vec)
```

```
template<typename V>
```

```
static CudaStorage fromData(const std::vector<V> &vec)
```

### Private Functions

```
template<typename P>
```

```
void initData(P begin, P end)
```

**Template Parameters** **P** – Pointer type

**Parameters**

- **begin** – Beginning of data to copy
- **end** – End of data to copy

```
void resizeImpl(SizeType newSize, int)
```

Resize the *Storage* Object to **newSize** elements, retaining existing data.

**Parameters** **newSize** – New size of the *Storage* object

```
void resizeImpl(SizeType newSize)
```

Resize the *Storage* object to **newSize** elements. Note this does not initialize the new elements or maintain existing data.

**Parameters** **newSize** – New size of the *Storage* object

### Private Members

Pointer **m\_begin** = nullptr

size\_t **m\_size**

bool **m\_ownsData** = true

namespace **detail**

## Functions

**template<typename T> T \*\_\_restrict cudaSafeAllocate (size\_t size)**

Safely allocate memory for size elements of type on the GPU using CUDA.

**See also:**

*safeAllocate*

**Template Parameters** T – Scalar type

**Parameters** size – Number of elements to allocate

**Returns** GPU pointer

**template<typename T> void cudaSafeDeallocate (T \*\_\_restrict data)**

Safely free memory for size elements of type on the GPU using CUDA.

**See also:**

*safeAllocate*

**Template Parameters** T – Scalar type

**Parameters** data – The data to deallocate

**Returns** GPU pointer

template<typename T>  
std::shared\_ptr<T> **cudaSharedPtrAllocate**(size\_t size)

template<typename LHS, typename RHS>  
auto **operator+**(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>  
auto **operator+**(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>  
auto **operator+**(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>  
auto **operator+=**(CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>  
auto **operator+=**(CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>  
auto **operator-**(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>  
auto **operator-**(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>  
auto **operator-**(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>  
auto **operator--**(CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>  
auto **operator--**(CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>

```
auto operator*(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator*(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator*(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator*+=(CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator*+=(CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator*/(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator*/(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator*/(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator/=(CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator/=(CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator%(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator%(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator%(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator%=(CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator%=(CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator^ (const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator^ (const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator^ (const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator^=(CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
```

```

auto operator^=(CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator&(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator&(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator&(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator&=(CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator&=(CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator|(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator|(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator|(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator|=(CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator|=(CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator<<(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator<<(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator<<(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator<<=(CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator<<=(CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator>>(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator>>(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator>>(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>

```

```
auto operator>>=(CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator>>=(CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator==(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator==(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator==(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator!=(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator!=(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator!=(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator<(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator<(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator<(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator>(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator>(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator>(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator<=(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator<=(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator<=(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
auto operator>=(const CudaRef<LHS> &lhs, const RHS &rhs)

template<typename LHS, typename RHS>
auto operator>=(const LHS &lhs, const CudaRef<RHS> &rhs)

template<typename LHS, typename RHS>
```

```

auto operator+=(const CudaRef<LHS> &lhs, const CudaRef<RHS> &rhs)

template<typename T>
std::shared_ptr<T> safePointerCopyCuda(T *ptr, bool ownsData = true)

template<typename T>
std::shared_ptr<T> safePointerCopyCuda(std::shared_ptr<T> ptr, bool ownsData = true)

template<typename T>
class CudaRef
    #include <cudaStorage.hpp>

```

### Public Types

```
using PtrType = std::shared_ptr<T>
```

### Public Functions

```

inline CudaRef(const PtrType &ptr, size_t offset)

inline CudaRef &operator=(const T &val)

inline T get() const

template<typename CAST>
inline operator CAST() const

inline std::string str(const std::string &format = "{}") const

```

### Private Members

```

std::shared_ptr<T> m_ptr

size_t m_offset

```

```
namespace typetraits
```

### Functions

```

LIBRAPID_DEFINE_AS_TYPE(typename Scalar_, CudaStorage<Scalar_>)

template<typename Scalar_>

struct TypeInfo<CudaStorage<Scalar_>>
    #include <cudaStorage.hpp>

```

### Public Types

```
using Scalar = Scalar_
```

```
using Backend = backend::CUDA
```

### Public Static Attributes

```
static constexpr bool isLibRapidType = true
```

```
template<typename T>
```

```
struct IsCudaStorage : public std::false_type
```

```
    #include <cudaStorage.hpp>
```

```
template<typename Scalar>
```

```
struct IsCudaStorage<CudaStorage<Scalar>> : public std::true_type
```

```
    #include <cudaStorage.hpp>
```

## 1.3.3 Vectors

LibRapid provides a highly optimised fixed-size vector library which supports all primitive types as well as user-defined ones (assuming they implement the required operations).

### 1.3.3.1 Vector Listing

**Warning:** doxygenclass: Cannot find class “librapid::GenericVector” in doxygen xml output for project “librapid” from directory: ../xml

## 1.3.4 Complex Numbers

**Documentation** View the API and documentation for complex numbers.

**Examples** See some examples of LibRapid’s complex number library in action

**Implementation Details** Learn about the implementation of complex numbers in LibRapid



### 1.3.4.1 Complex Number Listing

namespace **librapid**

#### Functions

```
template<typename T>
auto operator-(const Complex<T> &other) -> Complex<T>
```

Negate a complex number.

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **other** – *Complex* number to negate

**Returns** Negated complex number

```
template<typename L, typename R>
auto operator+(const Complex<L> &left, const Complex<R> &right)
```

Add two complex numbers.

Add two complex numbers together, returning the result

**Template Parameters**

- **L** – Scalar type of LHS
- **R** – Scalar type of RHS

**Parameters**

- **left** – LHS complex number
- **right** – RHS complex number

**Returns** Sum of LHS and RHS

```
template<typename T, typename R>
auto operator+(const Complex<T> &left, const R &right)
```

Add a complex number and a scalar.

Add a real number to the real component of a complex number, returning the result

**Template Parameters**

- **T** – Scalar type of the complex number
- **R** – Type of the real number

**Parameters**

- **left** – LHS complex number
- **right** – RHS scalar

**Returns** Sum of LHS and RHS

```
template<typename R, typename T>
auto operator+(const R &left, const Complex<T> &right)
```

Add a scalar to a complex number.

Add a real number to the real component of a complex number, returning the result

**Template Parameters**

- **R** – Type of the real number
- **T** – Scalar type of the complex number

**Parameters**

- **left** – LHS scalar
- **right** – RHS complex number

**Returns** Sum of LHS and RHS

template<typename **L**, typename **R**>

auto **operator-** (const Complex<**L**> &left, const Complex<**R**> &right)

Subtract a complex number from another complex number.

Subtract the real and imaginary components of the RHS complex number from the corresponding components of the LHS complex number, returning the result

**Template Parameters**

- **L** – Scalar type of the LHS complex number
- **R** – Scalar type of the RHS complex number

**Parameters**

- **left** – LHS complex number
- **right** – RHS complex number

**Returns** Difference of LHS and RHS

template<typename **T**, typename **R**>

auto **operator-** (const Complex<**T**> &left, const **R** &right)

Subtract a scalar from a complex number.

Subtract a real number from the real component of a complex number, returning the result

**Template Parameters**

- **T** – Scalar type of the complex number
- **R** – Type of the real number

**Parameters**

- **left** – LHS complex number
- **right** – RHS scalar

**Returns** Difference of LHS and RHS

template<typename **T**, typename **R**>

auto **operator-** (const **R** &left, const Complex<**T**> &right)

Subtract a complex number from a scalar.

Subtract the real and imaginary components of the RHS complex number from a real number, returning the result

**Template Parameters**

- **T** – Scalar type of the complex number
- **R** – Type of the real number

**Parameters**

- **left** – LHS scalar
- **right** – RHS complex number

**Returns** Difference of LHS and RHS

```
template<typename L, typename R>
auto operator*(const Complex<L> &left, const Complex<R> &right)
```

Multiply two complex numbers.

Multiply the LHS and RHS complex numbers, returning the result

#### Template Parameters

- **L** – Scalar type of the LHS complex number
- **R** – Scalar type of the RHS complex number

#### Parameters

- **left** – LHS complex number
- **right** – RHS complex number

**Returns** Product of LHS and RHS

```
template<typename T, typename R>
auto operator*(const Complex<T> &left, const R &right)
```

Multiply a complex number by a scalar.

Multiply the real and imaginary components of a complex number by a real number, returning the result

#### Template Parameters

- **T** – Scalar type of the complex number
- **R** – Type of the real number

#### Parameters

- **left** – LHS complex number
- **right** – RHS scalar

**Returns** Product of LHS and RHS

```
template<typename T, typename R>
auto operator*(const R &left, const Complex<T> &right)
```

Multiply a scalar by a complex number.

Multiply a real number by the real and imaginary components of a complex number, returning the result

#### Template Parameters

- **T** – Scalar type of the complex number
- **R** – Type of the real number

#### Parameters

- **left** – LHS scalar
- **right** – RHS complex number

**Returns** Product of LHS and RHS

```
template<typename L, typename R>
```

auto **operator/**(const Complex<*L*> &left, const Complex<*R*> &right)

Divide two complex numbers.

Divide the LHS complex number by the RHS complex number, returning the result

**Template Parameters**

- **L** – Scalar type of the LHS complex number
- **R** – Scalar type of the RHS complex number

**Parameters**

- **left** – LHS complex number
- **right** – RHS complex number

**Returns** Quotient of LHS and RHS

template<typename **T**, typename **R**>

auto **operator/**(const Complex<*T*> &left, const *R* &right)

Divide a complex number by a scalar.

Divide the real and imaginary components of a complex number by a real number, returning the result

**Template Parameters**

- **T** – Scalar type of the complex number
- **R** – Type of the real number

**Parameters**

- **left** – LHS complex number
- **right** – RHS scalar

**Returns** Quotient of LHS and RHS

template<typename **T**, typename **R**>

auto **operator/**(const *R* &left, const Complex<*T*> &right)

Divide a scalar by a complex number.

Divide a real number by the real and imaginary components of a complex number, returning the result

**Template Parameters**

- **T** – Scalar type of the complex number
- **R** – Type of the real number

**Parameters**

- **left** – LHS scalar
- **right** – RHS complex number

**Returns** Quotient of LHS and RHS

template<typename **L**, typename **R**>

constexpr bool **operator==**(const Complex<*L*> &left, const Complex<*R*> &right)

Equality comparison of two complex numbers.

**Template Parameters**

- **L** – Scalar type of LHS complex number
- **R** – Scalar type of RHS complex number

**Parameters**

- **left** – LHS complex number
- **right** – RHS complex number

**Returns** true if equal, false otherwise

```
template<typename T>
constexpr bool operator==(const Complex<T> &left, T &right)
```

Equality comparison of complex number and scalar.

Compares the real component of the complex number to the scalar, and the imaginary component to zero. Returns true if and only if both comparisons are true.

**Template Parameters** **T** – Scalar type of complex number

**Parameters**

- **left** – LHS complex number
- **right** – RHS scalar

**Returns** true if equal, false otherwise

```
template<typename T>
constexpr bool operator==(const T &left, const Complex<T> &right)
```

Equality comparison of scalar and complex number.

Compares the real component of the complex number to the scalar, and the imaginary component to zero. Returns true if and only if both comparisons are true.

**Template Parameters** **T** – Scalar type of complex number

**Parameters**

- **left** – LHS scalar
- **right** – RHS complex number

**Returns** true if equal, false otherwise

```
template<typename T>
constexpr bool operator!=(const Complex<T> &left, const Complex<T> &right)
```

Inequality comparison of two complex numbers.

**Template Parameters** **T** – Scalar type of complex number

**Parameters**

- **left** – LHS complex number
- **right** – RHS complex number

**Returns** true if equal, false otherwise

```
template<typename T>
constexpr bool operator!=(const Complex<T> &left, T &right)
```

Inequality comparison of complex number and scalar.

**See also:**

*`operator==(const Complex<T> &, T &)`*

**Template Parameters** **T** – Scalar type of complex number

**Parameters**

- **left** – LHS complex number
- **right** – RHS scalar

**Returns** true if equal, false otherwise

```
template<typename T>  
constexpr bool operator!=(const T &left, const Complex<T> &right)  
    Inequality comparison of scalar and complex number.
```

**See also:**

*operator==(const T &, const Complex<T> &)*

**Template Parameters** **T** – Scalar type of complex number

**Parameters**

- **left** – LHS scalar
- **right** – RHS complex number

**Returns** true if equal, false otherwise

```
template<typename T>  
T real(const Complex<T> &val)
```

Return  $\text{Re}(z)$ .

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **val** – *Complex* number

**Returns** Real component of the complex number

```
template<typename T>  
T imag(const Complex<T> &val)
```

Return  $\text{Im}(z)$ .

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **val** – *Complex* number

**Returns** Imaginary component of the complex number

```
template<typename T>  
Complex<T> sqrt(const Complex<T> &val)
```

Return  $\sqrt{z}$ .

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **val** – *Complex* number

**Returns** Square root of the complex number

```
template<typename T>  
T abs(const Complex<T> &val)
```

Return  $\sqrt{\text{Re}(z)^2 + \text{Im}(z)^2}$ .

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **val** – *Complex* number

**Returns** Absolute value of the complex number

```
template<typename T>
Complex<T> conj(const Complex<T> &val)
    Returns  $z^*$ .
```

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **val** – *Complex* number

**Returns** *Complex* conjugate of the complex number

```
template<typename T>
Complex<T> acos(const Complex<T> &other)
```

Compute the complex arc cosine of a complex number.

This function computes the complex arc cosine of the input complex number,  $z = \text{acos}(z)$

The algorithm handles NaN and infinity values, and avoids overflow.

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **other** – Input complex number

**Returns** *Complex* arc cosine of the input complex number

```
template<typename T>
Complex<T> acosh(const Complex<T> &other)
```

Compute the complex hyperbolic arc cosine of a complex number.

This function computes the complex area hyperbolic cosine of the input complex number,  $z = \text{acosh}(z)$

The algorithm handles NaN and infinity values, and avoids overflow.

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **other** – Input complex number

**Returns** *Complex* area hyperbolic cosine of the input complex number

```
template<typename T>
Complex<T> asinh(const Complex<T> &other)
```

Compute the complex arc hyperbolic sine of a complex number.

This function computes the complex arc hyperbolic sine of the input complex number,  $z = \text{asinh}(z)$

The algorithm handles NaN and infinity values, and avoids overflow.

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **other** – Input complex number

**Returns** *Complex* arc hyperbolic sine of the input complex number

```
template<typename T>
Complex<T> asin(const Complex<T> &other)
```

Compute the complex arc sine of a complex number.

This function computes the complex arc sine of the input complex number,  $z = \text{asin}(z)$

It calculates the complex arc sine by using the complex hyperbolic sine function.

**See also:**

asinh

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **other** – Input complex number

**Returns** *Complex* arc sine of the input complex number

```
template<typename T>
```

```
Complex<T> atanh(const Complex<T> &other)
```

Compute the complex arc hyperbolic tangent of a complex number.

This function computes the complex arc hyperbolic tangent of the input complex number,  $z = \operatorname{atanh}(z)$

This function performs error checking and supports NaNs and Infs.

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **other** – Input complex number

**Returns** *Complex* arc hyperbolic tangent of the input complex number

```
template<typename T>
```

```
Complex<T> atan(const Complex<T> &other)
```

Compute the complex arc tangent of a complex number.

This function computes the complex arc tangent of the input complex number,  $z = \operatorname{atan}(z)$

The algorithm handles NaN and infinity values, and avoids overflow.

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **other** – Input complex number

**Returns** *Complex* arc tangent of the input complex number

```
template<typename T>
```

```
Complex<T> cosh(const Complex<T> &other)
```

Compute the complex hyperbolic cosine of a complex number.

This function computes the complex hyperbolic cosine of the input complex number,  $z = \operatorname{cosh}(z)$

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **other** – Input complex number

**Returns** *Complex* hyperbolic cosine of the input complex number

```
template<typename T>
```

```
Complex<T> polarPositiveNanInfZeroRho(const T &rho, const T &theta)
```

```
template<typename T>
```

```
Complex<T> exp(const Complex<T> &other)
```

Compute the complex exponential of a complex number.

This function computes the complex exponential of the input complex number,  $z = e^z$

The algorithm handles NaN and infinity values.

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **other** – Input complex number

**Returns** *Complex* exponential of the input complex number

```
template<typename T>
```



Complex<T> **exp2**(const Complex<T> &other)

Compute the complex exponential base 2 of a complex number.

**See also:**

*exp*

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **other** – Input complex number

**Returns** *Complex* exponential base 2 of the input complex number

template<typename T>

Complex<T> **exp10**(const Complex<T> &other)

Compute the complex exponential base 10 of a complex number.

**See also:**

*exp*

**Template Parameters** **T** – Scalar type of the complex number

**Parameters** **other** – Input complex number

**Returns** *Complex* exponential base 10 of the input complex number

template<typename T>

T **\_fabs**(const Complex<T> &other, int64\_t \*exp)

template<typename T>

T **\_logAbs**(const Complex<T> &other) noexcept

template<>

mpfr **\_logAbs**(const Complex<mpfr> &other) noexcept

template<>

float **\_logAbs**(const Complex<float> &other) noexcept

template<typename T>

Complex<T> **log**(const Complex<T> &other)

Calculates the natural logarithm of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns** Natural logarithm of the complex number

template<typename T, typename B>

Complex<T> **log**(const Complex<T> &other, const Complex<T> &base)

Calculates the logarithm of a complex number with a complex base.

$$\log_{\text{base}}(z) = \log(z) / \log(\text{base})$$

**See also:**

*log*

**Template Parameters**

- **T** – Scalar type
- **B** – Base type

**Parameters**

- **other** – *Complex* number
- **base** – Base of the logarithm

**Returns** Logarithm of the complex number with the given base

```
template<typename T, typename B>
```

```
Complex<T> log(const Complex<T> &other, const B &base)
```

Calculates the logarithm of a complex number with a real base.

$$\log_{\text{base}}(z) = \log(z) / \log(\text{base})$$

**See also:**

*log*

**Template Parameters**

- **T** – Scalar type of the complex number
- **B** – Scalar type of the base

**Parameters**

- **other** – *Complex* number
- **base** – Base of the logarithm (real)

**Returns** Logarithm of the complex number with the given base

```
template<typename T>
```

```
Complex<T> _pow(const T &left, const T &right)
```

```
template<typename T, typename V, typename std::enable_if_t<typetraits::TypeInfo<V>::type ==  
detail::LibRapidType::Scalar, int> = 0>
```

```
Complex<T> pow(const Complex<T> &left, const V &right)
```

Calculate  $\text{left}^{\text{right}}$  for a complex-valued left-hand side.

**Template Parameters**

- **T** – Value type for the left-hand side
- **V** – Value type for the right-hand side

**Parameters**

- **left** – *Complex* base
- **right** – Real exponent

**Returns**  $\text{left}^{\text{right}}$

```
template<typename T, typename V, typename std::enable_if_t<typetraits::TypeInfo<V>::type ==  
detail::LibRapidType::Scalar, int> = 0>
```

Complex<T> **pow**(const V &left, const Complex<T> &right)

Calculate  $\text{left}^{\text{right}}$  for a complex-valued right-hand side.

**Template Parameters**

- **T** – Value type for the left-hand side
- **V** – Value type for the right-hand side

**Parameters**

- **left** – Real base
- **right** – *Complex* exponent

**Returns**  $\text{left}^{\text{right}}$

template<typename T>

Complex<T> **pow**(const Complex<T> &left, const Complex<T> &right)

Calculate  $\text{left}^{\text{right}}$  for complex numbers.

**Template Parameters** **T** – *Complex* number component type

**Parameters**

- **left** – *Complex* base
- **right** – *Complex* exponent

**Returns**  $\text{left}^{\text{right}}$

template<typename T>

Complex<T> **sinh**(const Complex<T> &other)

Calculate the hyperbolic sine of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\sinh(z)$

template<typename T>

Complex<T> **tanh**(const Complex<T> &other)

Calculate the hyperbolic tangent of a complex number.

This function supports propagation of NaNs and Infs.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\tanh(z)$

template<typename T>

T **arg**(const Complex<T> &other)

Return the phase angle of a complex value as a real.

This function calls  $\text{atan2}(\text{imag}(z), \text{real}(z))$ .

**See also:**

`atan2`

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\arg(z)$

template<typename **T**>

Complex<**T**> **proj**(const Complex<**T**> &other)

Project a complex number onto the Riemann sphere.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\text{proj}(z)$

template<typename **T**>

Complex<**T**> **cos**(const Complex<**T**> &other)

Calculate the cosine of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\cos(z)$

template<typename **T**>

Complex<**T**> **csc**(const Complex<**T**> &other)

Calculate the cosecant of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\text{csc}(z)$

template<typename **T**>

Complex<**T**> **sec**(const Complex<**T**> &other)

Calculate the secant of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\text{sec}(z)$

template<typename **T**>

Complex<**T**> **cot**(const Complex<**T**> &other)

Calculate the cotangent of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\cot(z)$

template<typename **T**>

Complex<**T**> **acsc**(const Complex<**T**> &other)

Calculate the arc cosecant of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\text{arccsc}(z)$

template<typename **T**>

Complex<T> **asec**(const Complex<T> &other)

Calculate the arc secant of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\operatorname{arcsec}(z)$

template<typename T>

Complex<T> **acot**(const Complex<T> &other)

Calculate the arc cotangent of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\operatorname{arccot}(z)$

template<typename T>

Complex<T> **log2**(const Complex<T> &other)

Calculate the logarithm base 2 of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\log_2(z)$

template<typename T>

Complex<T> **log10**(const Complex<T> &other)

Calculate the logarithm base 10 of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\log_{10}(z)$

template<typename T>

T **norm**(const Complex<T> &other)

Calculate the magnitude squared of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $|z|^2$

template<typename T>

Complex<T> **polar**(const T &rho, const T &theta)

Return a complex number from polar coordinates.

Given a radius, **rho**, and an angle, **theta**, this function returns the complex number  $\rho e^{i\theta}$ .

The function returns NaN, infinity or zero based on the input values of **rho**.

**Template Parameters** **T** – Scalar type of the complex number

**Parameters**

- **rho** – Radius of the polar coordinate system
- **theta** – Angle of the polar coordinate system

**Returns** *Complex* number in polar form.

```
template<typename T>  
Complex<T> sin(const Complex<T> &other)
```

Compute the sine of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\sin(z)$

```
template<typename T>  
Complex<T> tan(const Complex<T> &other)
```

Compute the tangent of a complex number.

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $\tan(z)$

```
template<typename T>  
Complex<T> floor(const Complex<T> &other)
```

Round the real and imaginary parts of a complex number towards  $-\infty$ .

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $(\lfloor \text{real}(z) \rfloor, \lfloor \text{imag}(z) \rfloor)$

```
template<typename T>  
Complex<T> ceil(const Complex<T> &other)
```

Round the real and imaginary parts of a complex number towards  $+\infty$ .

**Template Parameters** **T** – Scalar type

**Parameters** **other** – *Complex* number

**Returns**  $(\lceil \text{real}(z) \rceil, \lceil \text{imag}(z) \rceil)$

```
template<typename T>  
auto random(const Complex<T> &min, const Complex<T> &max, uint64_t seed = -1) -> Complex<T>
```

Generate a random complex number between two given complex numbers.

This function generates a random complex number in the range  $[\text{min}, \text{max}]$ , where min and max are given as input. The function uses a default seed if none is provided.

**Template Parameters** **T** – Scalar type of the complex number

**Parameters**

- **min** – Minimum complex number
- **max** – Maximum complex number
- **seed** – Seed for the random number generator

**Returns** Random complex number between min and max

```
template<typename T = double>
```

```
class Complex
```

*#include <complex.hpp>* A class representing a complex number of the form  $a + bi$ , where  $a$  and  $b$  are real numbers.

This class represents a complex number of the form  $a + bi$ , where  $a$  and  $b$  are real numbers. The class is templated, allowing the user to specify the type of the real and imaginary components. The default type is `double`.

**Template Parameters** **T** – The type of the real and imaginary components

## Public Types

using **Scalar** = typename *typetraits*::TypeInfo<T>::Scalar

## Public Functions

inline **Complex**()

Default constructor.

Create a new complex number. Both the real and imaginary components are set to zero

template<typename **R**>

inline explicit **Complex**(const **R** &realVal)

Construct a complex number from a real number.

Create a complex number, setting only the real component. The imaginary component is initialized to zero

**Template Parameters** **R** – The type of the real component

**Parameters** **realVal** – The real component

template<typename **R**, typename **I**>

inline **Complex**(const **R** &realVal, const **I** &imagVal)

Construct a complex number from real and imaginary components.

Create a new complex number where both the real and imaginary parts are set from the passed parameters

**Template Parameters**

- **R** – The type of the real component

- **I** – The type of the imaginary component

**Parameters**

- **realVal** – The real component

- **imagVal** – The imaginary component

inline **Complex**(const Complex<T> &other)

*Complex* number copy constructor.

**Parameters** **other** – The complex number to copy

inline **Complex**(Complex<T> &&other) noexcept

*Complex* number move constructor.

**Parameters** **other** – The complex number to move

template<typename **Other**>

inline **Complex**(const Complex<Other> &other)

Construct a complex number from another complex number with a different type.

**Template Parameters** **Other** – Type of the components of the other complex number

**Parameters** **other** – The complex number to copy

inline explicit **Complex**(const std::complex<T> &other)

Construct a complex number from a std::complex.

**Parameters** **other** – The std::complex value to copy

inline auto **operator**=(const Complex<T> &other) -> Complex<T>&

*Complex* number assignment operator.

**Parameters** **other** – The value to assign

**Returns** \*this

template<typename P>

inline void **store**(P \*ptr) const

template<typename P>

inline void **load**(const P \*ptr)

inline void **real**(const T &val)

Assign to the real component.

Set the real component of this complex number to val

**Parameters** **val** – The value to assign

inline void **imag**(const T &val)

Assign to the imaginary component.

Set the imaginary component of this complex number to val

**Parameters** **val** – The value to assign

inline auto **real**() const -> const T&

Access the real component.

Returns a const reference to the real component of this complex number

**Returns** Real component

inline auto **imag**() const -> const T&

Access the imaginary component.

Returns a const reference to the imaginary component of this complex number

**Returns** Imaginary component

inline auto **real**() -> T&

Access the real component.

Returns a reference to the real component of this complex number. Since this is a reference type, it can be assigned to

**Returns** Real component

inline auto **imag**() -> T&

Access the imaginary component.

Returns a reference to the imaginary component of this complex number. Since this is a reference type, it can be assigned to

**Returns** imaginary component

inline auto **operator**=(const T &other) -> Complex&

*Complex* number assignment operator.

Set the real component of this complex number to other, and the imaginary component to 0

**Parameters** **other** –

**Returns** \*this

template<typename **Other**>



inline auto **operator**=(const Complex<Other> &other) -> Complex&

*Complex* number assignment operator.

Assign another complex number to this one, copying the real and imaginary components

**Template Parameters** **Other** – The type of the other complex number

**Parameters** **other** – *Complex* number to assign

**Returns** \*this

inline auto **operator**+=(const T &other) -> Complex&

Inplace addition.

Add a scalar value to the real component of this imaginary number

**Parameters** **other** – Scalar value to add

**Returns** \*this

inline auto **operator**-= (const T &other) -> Complex&

Inplace subtraction.

Subtract a scalar value from the real component of this imaginary number

**Parameters** **other** – Scalar value to subtract

**Returns** \*this

inline auto **operator**\*=(const T &other) -> Complex&

Inplace multiplication.

Multiply both the real and imaginary components of this complex number by a scalar

**Parameters** **other** – Scalar value to multiply by

**Returns** \*this

inline auto **operator**/=(const T &other) -> Complex&

Inplace division.

Divide both the real and imaginary components of this complex number by a scalar

**Parameters** **other** – Scalar value to divide by

**Returns** \*this

inline auto **operator**+=(const Complex &other) -> Complex&

Inplace addition.

Add a complex number to this one

**Parameters** **other** – *Complex* number to add

**Returns** \*this

inline auto **operator**-= (const Complex &other) -> Complex&

Inplace subtraction.

Subtract a complex number from this one

**Parameters** **other** – *Complex* number to subtract

**Returns** \*this

inline auto **operator**\*=(const Complex &other) -> Complex&

Inplace multiplication.

Multiply this complex number by another one

**Parameters** **other** – *Complex* number to multiply by

**Returns** \*this

inline auto **operator**/=(const Complex &other) -> Complex&

Inplace division.

Divide this complex number by another one

**Parameters** **other** – *Complex* number to divide by

**Returns** \*this

```
template<typename To>
```

```
inline explicit operator To() const
```

Cast to scalar types.

Cast this complex number to a scalar type. This will extract only the real component.

**Template Parameters** **To** – Type to cast to

**Returns** Scalar

```
template<typename To>
```

```
inline explicit operator Complex<To>() const
```

Cast to a complex number with a different scalar type.

Cast the real and imaginary components of this complex number to a different type and return the result as a new complex number

**Template Parameters** **To** – Scalar type to cast to

**Returns** *Complex* number

```
inline auto str(const std::string &format = "{}") const -> std::string
```

*Complex* number to string.

Create a std::string representation of a complex number, formatting each component with the format string

**Parameters** **format** – Format string

**Returns** std::string

## Public Static Functions

```
static inline constexpr auto size() -> size_t
```

## Protected Functions

```
template<typename Other>
```

```
inline void _add(const Complex<Other> &other)
```

Add a complex number to this one.

**Template Parameters** **Other** – Scalar type of the other complex number

**Parameters** **other** – Other complex number

```
template<typename Other>
```

```
inline void _sub(const Complex<Other> &other)
```

Subtract a complex number from this one.

**Template Parameters** **Other** – Scalar type of the other complex number

**Parameters** **other** – Other complex number

```
template<typename Other>
```

```
inline void _mul(const Complex<Other> &other)
```

Multiply this complex number by another one.

**Template Parameters** **Other** – Scalar type of the other complex number

**Parameters** **other** – Other complex number

```
template<typename Other>
```

```
inline void _div(const Complex<Other> &other)
```

Divide this complex number by another one.

**Template Parameters** **Other** – Scalar type of the other complex number

**Parameters** **other** – Other complex number

## Private Members

```
T m_val[2]
```

## Private Static Attributes

```
static constexpr size_t RE = 0
```

```
static constexpr size_t IM = 1
```

```
namespace detail
```

```
namespace algorithm
```

## Functions

```
template<typename T>
```

```
auto normMinusOne(const T x, const T y) noexcept -> T
```

Calculates  $x^2 + y^2 - 1$  for  $|x| \geq |y|$  and  $0.5 \leq |x| < 2^{12}$ .

**Template Parameters** **T** – Template type

**Parameters**

- **x** – First value
- **y** – Second value

**Returns**  $x * x + y * y - 1$

```
template<bool safe = true, typename T>
```

```
auto logP1(const T x) -> T
```

Calculates  $\log(1 + x)$ .

May be inaccurate for small inputs

**Template Parameters**

- **safe** – If true, will check for NaNs and overflow
- **T** – Template type

**Parameters** **x** – Input value

**Returns**  $\log(1 + x)$

```
template<bool safe = true, typename T>
```

```
auto logHypot(const T x, const T y) noexcept -> T
```

Calculates  $\log(\sqrt{x^2 + y^2})$ .

**Template Parameters**

- **safe** – If true, will check for NaNs and overflow
- **T** – Template type

**Parameters**

- **x** – Horizontal component

- **y** – Vertical component

**Returns**  $\log(\sqrt{x^2 + y^2})$

```
template<typename T>
```

```
auto expMul(T *pleft, T right, short exponent) -> short
```

Compute  $e^{\text{pleft}} \times \text{right} \times 2^{\text{exponent}}$ .

**Template Parameters** **T** – Template type

**Parameters**

- **pleft** – Pointer to the value to be exponentiated
- **right** – Multiplier for the exponentiated value
- **exponent** – Exponent for the power of 2 multiplication

**Returns** 1 if the result is NaN or Inf, -1 otherwise

## Variables

```
template<typename T>
```

```
static T HypotLegHuge = HypotLegHugeHelper<T>::val
```

```
template<typename T>
```

```
static T HypotLegTiny = HypotLegTinyHelper<T>::val
```

```
template<typename T>
```

```
struct HypotLegHugeHelper
```

```
    #include <complex.hpp>
```

## Public Static Attributes

```
static T val = (std::is_integral_v<T>)?
```

```
    (::librapid::sqrt(typetraits::TypeInfo<T>::max()) / T(2)): (T(0.5) * ::librapid::sqrt(typetraits::TypeInfo<T>::max()))
```

```
template<>
```

```
struct HypotLegHugeHelper<double>
```

```
    #include <complex.hpp>
```

## Public Static Attributes

```
static constexpr double val = 6.703903964971298e+153
```

```
template<>
```

```
struct HypotLegHugeHelper<float>
```

```
    #include <complex.hpp>
```

### Public Static Attributes

```
static constexpr double val = 9.2233715e+18f
```

```
template<typename T>
```

```
struct HypotLegTinyHelper
```

```
    #include <complex.hpp>
```

### Public Static Attributes

```
static T val = ::librapid::sqrt(T(2) * typetraits::TypeInfo<T>::min() /  
typetraits::TypeInfo<T>::epsilon())
```

```
template<>
```

```
struct HypotLegTinyHelper<double>
```

```
    #include <complex.hpp>
```

### Public Static Attributes

```
static constexpr double val = 1.4156865331029228e-146
```

```
template<>
```

```
struct HypotLegTinyHelper<float>
```

```
    #include <complex.hpp>
```

### Public Static Attributes

```
static constexpr double val = 4.440892e-16f
```

```
namespace multiprec
```

### Functions

```
template<typename T>
```

```
constexpr auto addX2(const T &x, const T &y) noexcept -> Fmp<T>
```

Summarizes two 1x precision values combined into a 2x precision result.

This function is exact when:

- I. The result doesn't overflow
- II. Either underflow is gradual, or no internal underflow occurs
- III. Intermediate precision is either the same as T, or greater than twice the precision of T
- IV. Parameters and local variables do not retain extra intermediate precision
- V. Rounding mode is rounding to nearest.

Violation of condition 3 or 5 could lead to relative error on the order of  $\epsilon^2$ .

Violation of other conditions could lead to worse results

**Template Parameters** **T** – Template type  
**Parameters**

- **x** – First value
- **y** – Second value

**Returns** Sum of x and y

```
template<typename T>
```

```
constexpr auto addSmallX2(const T x, const T y) noexcept -> Fmp<T>
```

Combines two 1x precision values into a 2x precision result with the requirement of specific exponent relationship.

Requires:  $\text{exponent}(x) + \text{countr\_zero}(\text{significand}(x)) \geq \text{exponent}(y)$  or  $x == 0$

The result is exact when:

- I. The requirement above is satisfied
- II. No internal overflow occurs
- III. Either underflow is gradual, or no internal underflow occurs
- IV. Intermediate precision is either the same as T, or greater than twice the precision of T
- V. Parameters and local variables do not retain extra intermediate precision
- VI. Rounding mode is rounding to nearest

Violation of condition 3 or 5 could lead to relative error on the order of  $\epsilon^2$ .

Violation of other conditions could lead to worse results

**Template Parameters** **T** – Template type

**Parameters**

- **x** – First value
- **y** – Second value

**Returns** Sum of x and y

```
template<typename T>
```

```
constexpr auto addSmallX2(const T &x, const Fmp<T> &y) noexcept -> Fmp<T>
```

Combines a 1x precision value with a 2x precision value.

Requires:  $\text{exponent}(x) + \text{countr\_zero}(\text{significand}(x)) \geq \text{exponent}(y.\text{val0})$  or  $x == 0$

**Template Parameters** **T** – Template type

**Parameters**

- **x** – First value
- **y** – Second value

**Returns** Sum of x and y

```
template<typename T>
```

```
constexpr auto addX1(const Fmp<T> &x, const Fmp<T> &y) noexcept -> T
```

Combines two 2x precision values into a 1x precision result.

**Template Parameters** **T** – Template type

**Parameters**

- **x** – First value
- **y** – Second value

**Returns** Sum of x and y

```
constexpr auto highHalf(const double x) noexcept -> double
```

Rounds a 2x precision value to 26 significant bits.

**Parameters** **x** – Value to round

**Returns** Rounded value

```
constexpr double sqrError(const double x, const double prod0) noexcept
```

Fallback method for *sqrError(const double, const double)* when SIMD is not available.

```
template<typename T>
```

```

auto sqrError(const T x, const T prod0) noexcept -> T
    Type-agnostic version of sqrError(const double, const double)
    Template Parameters T – Template type
    Parameters
        • x – Input value
        • prod0 – Faithfully rounded product of  $x^2$ 

auto sqrX2(const double x) noexcept -> Fmp<double>
    Calculates the square of a 1x precision value and returns a 2x precision result.

    The result is exact when no internal overflow or underflow occurs.
    Parameters x – Input value
    Returns 2x precision square of x

template<typename T>
auto sqrX2(const T x) noexcept -> Fmp<T>
    Type-agnostic version of sqrX2(const double)
    Template Parameters T – Template type
    Parameters x – Input value
    Returns 2x precision square of x

template<typename Scalar>
struct Fmp
    #include <complex.hpp>

```

### Public Members

Scalar **val0**

Scalar **val1**

namespace **typetraits**

```

template<typename T>
struct TypeInfo<Complex<T>>
    #include <complex.hpp>

```

### Public Types

using **Scalar** = Complex<T>

using **Packet** = typename std::conditional\_t<(TypeInfo<T>::packetWidth > 1), Complex<typename TypeInfo<T>::Packet>, std::false\_type>

### Public Functions

```
inline LIMIT_IMPL(min)

inline LIMIT_IMPL(max)

inline LIMIT_IMPL(epsilon)

inline LIMIT_IMPL(roundError)

inline LIMIT_IMPL(denormMin)

inline LIMIT_IMPL(infinity)

inline LIMIT_IMPL(quietNaN)

inline LIMIT_IMPL(signalingNaN)
```

### Public Static Attributes

```
static constexpr detail::LibRapidType type = detail::LibRapidType::Scalar

static constexpr int64_t packetWidth = TypeInfo<typename TypeInfo<T>::Scalar>::packetWidth

static constexpr char name[] = "Complex"

static constexpr bool supportsArithmetic = true

static constexpr bool supportsLogical = true

static constexpr bool supportsBinary = false

static constexpr bool allowVectorisation = false

static constexpr cudaDataType_t CudaType = cudaDataType_t::CUDA_C_64F

static constexpr bool canAlign = TypeInfo<T>::canAlign

static constexpr bool canMemcpy = TypeInfo<T>::canMemcpy
```



### 1.3.4.2 Complex Number Examples

*To do*

### 1.3.4.3 Complex Number Implementation Details

*To do*

## 1.3.5 Mathematics

### 1.3.6 Multi-Precision Arithmetic

LibRapid has support for [MPIR](#) and [MPFR](#), which support arbitrary-precision integers, floating points and rationals.

We provide a simple wrapper around these libraries, enabling all mathematical operations to be performed on these data types – you don’t even need to use a different function name!

#### 1.3.6.1 Multi-Precision Listing

**Warning:** doxygenclass: Cannot find class “librapid::mpz” in doxygen xml output for project “librapid” from directory: ../xml

---

**Warning:** doxygenclass: Cannot find class “librapid::mpq” in doxygen xml output for project “librapid” from directory: ../xml

---

**Warning:** doxygenclass: Cannot find class “librapid::mpf” in doxygen xml output for project “librapid” from directory: ../xml

---

**Warning:** doxygenclass: Cannot find class “librapid::mpfr” in doxygen xml output for project “librapid” from directory: ../xml

---

## 1.4 Tutorials

## 1.5 Performance and Benchmarks

LibRapid is high-performance library and is fast by default, but there are still ways to make your code even faster.

### 1.5.1 Lazy Evaluation

Operations performed on Arrays are evaluated only when needed, meaning functions can be chained together and evaluated in one go. In many cases, the compiler can optimise these chained calls into a single loop, resulting in much faster code.

Look at the example below:

```
lrc::Array<float> A, B, C, D;
A = lrc::fromData({{1, 2}, {3, 4}});
B = lrc::fromData({{5, 6}, {7, 8}});
C = lrc::fromData({{9, 10}, {11, 12}});
D = A + B * C;
```

Without lazy-evaluation, the operation  $A+B*C$  must be performed in multiple stages:

```
auto tmp1 = B * C;    // First operation and temporary object
auto tmp2 = A + tmp1; // Second operation and ANOTHER temporary object
D = tmp2;             // Unnecessary copy
```

This is clearly suboptimal.

With lazy-evaluation, however, the compiler can generate a loop similar to the pseudocode below:

```
FOR index IN A.size DO
    D[i] = A[i] + B[i] * C[i]
ENDFOR
```

This has no unnecessary copies, no temporary variables, no additional memory allocation, etc. and is substantially quicker.

#### 1.5.1.1 Making Use of LibRapid's Lazy Evaluation

To make use of LibRapid's lazy evaluation, try to avoid creating temporary objects and always assign results directly to an existing array object, instead of creating a new one. This means no heap allocations are performed, which is a very costly operation.

**Warning:** Be very careful not to reference invalid memory. This is, unfortunately, an unavoidable side effect of returning lazy-objects. See [Caution](#) for more information.

Note that, sometimes, it is faster to evaluate intermediate results than to use the combined operation. To do this, you can call `eval()` on the result of any operation to generate an Array object directly from it.

### 1.5.2 Linear Algebra

Linear algebra methods in LibRapid also return temporary objects, meaning they are not evaluated fully until they are needed. One implication of this is that expressions involving *more than one operation* will be evaluated *very slowly*.

**Danger:** Be careful when calling `eval` on the result of a linear algebra operation. Sometimes, LibRapid will be able to combine multiple operations into a single function call, which can lead to much better performance. Check the documentation for that specific function to see what further optimisations it supports.

### 1.5.2.1 Solution

To get around this issue, it'll often be quicker to simply evaluate (`myExpression.eval()`) the result of any linear algebra operations inside the larger expression.

```
auto slowExpression = a + b * c.dot(d);
auto fastExpression = a + b * c.dot(d).eval();
```

### 1.5.2.2 Explanation

Since `c.dot(d)` is a lazy object, the lazy evaluator will calculate each element of the resulting array independently as and when it is required by the rest of the expression. This means it is not possible to make use of the extremely fast BLAS and LAPACK functions.

By forcing the result to be evaluated independently of the rest of the expression, LibRapid can call `gemm`, for example, making the program significantly faster.

## 1.6 Caution

**Warning:** LibRapid developers had to make certain decisions regarding the underlying data layout used by the library. We made these decisions with the best interests of the library in mind, and while they may improve performance or usability, they may also incur adverse side effects.

While the developers of LibRapid may not be aware of all the side effects of their design choices, we have done our best to identify and justify those we know of.

### 1.6.1 Array Referencing Issues

LibRapid uses lazy evaluation to reduce the number of intermediate variables and copies required for any given operation, significantly improving performance. A side effect of this is that combined operations store references to Array objects.

As a result, if any of the referenced Array instances go out of scope before the lazy object is evaluated, an invalid memory location will be accessed, incurring a segmentation fault.

The easiest fix for this is to make sure you evaluate temporary results in time, though this is easier said than done. LibRapid aims to identify when a lazy object is using an invalid value and notify the user, but this will not work in all cases.

The code below will cause a segmentation fault since `testArray` will go out of scope upon returning from the function while the returned object contains two references to the array.

```
1  /* References invalid memory
2  vvvv */
3  auto doesThisBreak() {
4      lrc::Array<float> testArray(lrc::Shape({3, 3}));
5      testArray << 1, 2, 3, 4, 5, 6, 7, 8, 9;
6      return testArray + testArray;
7  }
```

```
1  /*    Changed
2  -----vvv----- */
3  lrc::Array<float> doesThisBreak() {
4      lrc::Array<float> testArray(lrc::Shape({3, 3}));
5      testArray << 1, 2, 3, 4, 5, 6, 7, 8, 9;
6      return testArray + testArray;
7  }
```

## WHY USE LIBRAPID?

LibRapid aims to provide a cohesive ecosystem of functions that interoperate with each other, allowing for faster development and faster code execution.

For example, LibRapid implements a wide range of mathematical functions which can operate on primitive types, multi-precision types, vectors, and arrays. Due to the way these functions are implemented, a single function call can be used to operate on all of these types, reducing code duplication.

### 2.1 A Small Example

To prove the point made above, let's take a look at a simple example. Here, we have a function that maps a value from one range to another:

```
1 // Standard "double" implementation
2 double map(double val, double start1, double stop1, double start2, double stop2) {
3     return start2 + (stop2 - start2) * ((val - start1) / (stop1 - start1));
4 }
5
6 // map(0.5, 0, 1, 0, 10) = 5
7 // map(10, 0, 100, 0, 1) = 0.1
8 // map(5, 0, 10, 0, 100) = 50
```

This function will accept integers, floats and doubles, but nothing else can be used, limiting its functionality.

Of course, this could be templated to accept other types, but if you passed a `std::vector<double>` to this function, for example, you'd have to create an edge case to support it. **This is where LibRapid comes in.**

Look at the function below:

```
1 // An extremely versatile mapping function (used within LibRapid!)
2 template<typename V, typename B1, typename E1, typename B2, typename E2>
3 V map(V val, B1 start1, E1 stop1, B2 start2, E2 stop2) {
4     return start2 + (stop2 - start2) * ((val - start1) / (stop1 - start1));
5 }
```

This may look excessively complicated with that many template parameters, but you don't actually need all of those! This just gives the greatest flexibility. This function can be called with **almost any LibRapid type!**.

```
1 map(0.5, 0, 1, 0, 100); // . . . . . | 50
2 map(lrc::Vec2d(0.2, 0.8), 0, 1, 0, 100); // . . . . . | (20, 80)
3 map(0.5, 0, 1, 0, lrc::Vec2d(100, 200)); // . . . . . | (50, 100)
4 map(lrc::Vec2d(-1, -2), 1, 0, lrc::Vec2d(100, 300)); // . | (75, 250)
```

(continues on next page)

(continued from previous page)

```

5
6 // -----
7
8 using namespace lrc::literals; // To use "_f" suffix
9                               // (also requires multiprecision to be enabled)
10 // "0.5"_f in this case creates a multiprecision float :)
11 map("0.5"_f, "0"_f, "1"_f, "0"_f, "100"_f); // . . . . . | 50.0000000000000000
12
13 // -----
14
15 auto val      = lrc::fromData<float>({{1, 2}, {3, 4}});
16 auto start1   = lrc::fromData<float>({{0, 0}, {0, 0}});
17 auto end1     = lrc::fromData<float>({{10, 10}, {10, 10}});
18 auto start2   = lrc::fromData<float>({{0, 0}, {0, 0}});
19 auto end2     = lrc::fromData<float>({{100, 100}, {100, 100}});
20
21 fmt::print("{}\n", lrc::map(val, start1, end1, start2, end2));
22 // [[10 20]
23 //  [30 40]]

```

Note: LibRapid's built-in map function has even more functionality! See the [Map Function](#) details.

This is just one example of how LibRapid's functions can be used to make your code more concise and more efficient, and hopefully it's clear to see how powerful this could be when working with more complex functions and types.

## CURRENT DEVELOPMENT STAGE

At the current point in time, LibRapid C++ is under rapid development by me ([Pencilcaseman](#)).

I am currently doing my A-Levels and do not have time to work on the library as much as I would like, so if you or someone you know might be willing to support the development of the library, feel free to create a pull request or chat to us on [Discord](#). Any help is greatly appreciated!





## ROADMAP

The [Roadmap](#) is a rough outline of what I want to get implemented in the library and by what point, but **please don't count on features being implemented quickly** – I can't promise I'll have the time to implement everything as soon as I'd like... (I'll try my best though!)

If you have any feature requests or suggestions, feel free to create an issue describing it. I'll try to get it working as soon as possible. If you really need something implemented quickly, a small donation would be appreciated, and would allow me to bump it to the top of my list of features.



## **LICENCING**

LibRapid is produced under the MIT License, so you are free to use the library how you like for personal and commercial purposes, though this is subject to some conditions, which can be found in full here: [LibRapid License](#)



## A

ARRAY\_FROM\_DATA\_DEF (*C macro*), 14

## C

CUDA\_REF\_OPERATOR (*C macro*), 91

CUDA\_REF\_OPERATOR\_NO\_ASSIGN (*C macro*), 91

## H

HIGHER\_DIMENSIONAL\_FROM\_DATA (*C macro*), 26

## I

IS\_ARRAY\_OP (*C macro*), 30

IS\_ARRAY\_OP\_ARRAY (*C macro*), 30

IS\_ARRAY\_OP\_WITH\_SCALAR (*C macro*), 31

## L

librapid (*C++ type*), 13, 15, 26, 27, 47, 81, 82, 91, 101

librapid::\_fabs (*C++ function*), 109

librapid::\_logAbs (*C++ function*), 109

librapid::\_pow (*C++ function*), 110

librapid::abs (*C++ function*), 106

librapid::acos (*C++ function*), 107

librapid::acosh (*C++ function*), 107

librapid::acot (*C++ function*), 113

librapid::acsc (*C++ function*), 112

librapid::arg (*C++ function*), 111

librapid::array (*C++ type*), 15, 27, 51

librapid::array::ArrayContainer (*C++ class*), 15

librapid::array::ArrayContainer::ArrayContainer  
(*C++ function*), 16, 17

librapid::array::ArrayContainer::assign  
(*C++ function*), 17, 20

librapid::array::ArrayContainer::Backend  
(*C++ type*), 16

librapid::array::ArrayContainer::begin (*C++  
function*), 19, 20

librapid::array::ArrayContainer::copy (*C++  
function*), 18

librapid::array::ArrayContainer::DirectRefSubscriptType  
(*C++ type*), 16

librapid::array::ArrayContainer::DirectSubscriptType  
(*C++ type*), 16

librapid::array::ArrayContainer::end (*C++  
function*), 20

librapid::array::ArrayContainer::fromData  
(*C++ function*), 21, 22

librapid::array::ArrayContainer::get (*C++  
function*), 18

librapid::array::ArrayContainer::Iterator  
(*C++ type*), 16

librapid::array::ArrayContainer::m\_shape  
(*C++ member*), 22

librapid::array::ArrayContainer::m\_storage  
(*C++ member*), 22

librapid::array::ArrayContainer::ndim (*C++  
function*), 18

librapid::array::ArrayContainer::operator()  
(*C++ function*), 18, 20

librapid::array::ArrayContainer::operator\*=  
(*C++ function*), 19, 20

librapid::array::ArrayContainer::operator+=  
(*C++ function*), 19, 20

librapid::array::ArrayContainer::operator/=  
(*C++ function*), 19, 20

librapid::array::ArrayContainer::operator=  
(*C++ function*), 17, 18, 20

librapid::array::ArrayContainer::operator%=  
(*C++ function*), 19, 20

librapid::array::ArrayContainer::operator&=  
(*C++ function*), 19, 21

librapid::array::ArrayContainer::operator-=  
(*C++ function*), 19, 20

librapid::array::ArrayContainer::operator^=  
(*C++ function*), 19, 21

librapid::array::ArrayContainer::operator|=  
(*C++ function*), 19, 21

librapid::array::ArrayContainer::operator>>=  
(*C++ function*), 19, 21

librapid::array::ArrayContainer::operator<<  
(*C++ function*), 18, 20

librapid::array::ArrayContainer::operator<<=  
(*C++ function*), 19, 21

librapid::array::ArrayContainer::operator[]  
(*C++ function*), 18

librapid::array::ArrayContainer::packet  
(C++ function), 19

librapid::array::ArrayContainer::Packet  
(C++ type), 16

librapid::array::ArrayContainer::scalar  
(C++ function), 19

librapid::array::ArrayContainer::Scalar  
(C++ type), 16

librapid::array::ArrayContainer::shape (C++  
function), 18

librapid::array::ArrayContainer::ShapeType  
(C++ type), 16

librapid::array::ArrayContainer::SizeType  
(C++ type), 16

librapid::array::ArrayContainer::storage  
(C++ function), 18

librapid::array::ArrayContainer::StorageType  
(C++ type), 16

librapid::array::ArrayContainer::str (C++  
function), 20

librapid::array::ArrayContainer::StrideType  
(C++ type), 16

librapid::array::ArrayContainer::write (C++  
function), 19

librapid::array::ArrayContainer::writePacket  
(C++ function), 19

librapid::array::ArrayView (C++ class), 27

librapid::array::ArrayView::ArrayType (C++  
type), 27

librapid::array::ArrayView::ArrayView (C++  
function), 27, 28

librapid::array::ArrayView::Backend (C++  
type), 27

librapid::array::ArrayView::BaseType (C++  
type), 27

librapid::array::ArrayView::begin (C++ func-  
tion), 29

librapid::array::ArrayView::ConstReference  
(C++ type), 27

librapid::array::ArrayView::end (C++ function),  
29

librapid::array::ArrayView::eval (C++ func-  
tion), 29

librapid::array::ArrayView::get (C++ function),  
28

librapid::array::ArrayView::Iterator (C++  
type), 27

librapid::array::ArrayView::m\_offset (C++  
member), 29

librapid::array::ArrayView::m\_ref (C++ mem-  
ber), 29

librapid::array::ArrayView::m\_shape (C++  
member), 29

librapid::array::ArrayView::m\_stride (C++  
member), 29

librapid::array::ArrayView::ndim (C++ func-  
tion), 29

librapid::array::ArrayView::offset (C++ func-  
tion), 29

librapid::array::ArrayView::operator CAST  
(C++ function), 28

librapid::array::ArrayView::operator= (C++  
function), 28, 29

librapid::array::ArrayView::operator[] (C++  
function), 28

librapid::array::ArrayView::Reference (C++  
type), 27

librapid::array::ArrayView::scalar (C++ func-  
tion), 29

librapid::array::ArrayView::Scalar (C++ type),  
27

librapid::array::ArrayView::setOffset (C++  
function), 29

librapid::array::ArrayView::setShape (C++  
function), 29

librapid::array::ArrayView::setStride (C++  
function), 29

librapid::array::ArrayView::shape (C++ func-  
tion), 28

librapid::array::ArrayView::ShapeType (C++  
type), 27

librapid::array::ArrayView::str (C++ function),  
29

librapid::array::ArrayView::stride (C++ func-  
tion), 28

librapid::array::ArrayView::StrideType (C++  
type), 27

librapid::asec (C++ function), 112

librapid::asin (C++ function), 107

librapid::asinh (C++ function), 107

librapid::atan (C++ function), 108

librapid::atanh (C++ function), 108

librapid::ceil (C++ function), 114

librapid::Complex (C++ class), 114

librapid::Complex::\_add (C++ function), 118

librapid::Complex::\_div (C++ function), 118

librapid::Complex::\_mul (C++ function), 118

librapid::Complex::\_sub (C++ function), 118

librapid::Complex::Complex (C++ function), 115

librapid::Complex::IM (C++ member), 119

librapid::Complex::imag (C++ function), 116

librapid::Complex::load (C++ function), 116

librapid::Complex::m\_val (C++ member), 119

librapid::Complex::operator Complex<To>  
(C++ function), 118

librapid::Complex::operator To (C++ function),  
118

librapid::Complex::operator\*= (C++ function),

---

```

117
librapid::Complex::operator+= (C++ function),
117
librapid::Complex::operator/= (C++ function),
117
librapid::Complex::operator=(C++ function), 116
librapid::Complex::operator-= (C++ function),
117
librapid::Complex::RE (C++ member), 119
librapid::Complex::real (C++ function), 116
librapid::Complex::Scalar (C++ type), 115
librapid::Complex::size (C++ function), 118
librapid::Complex::store (C++ function), 116
librapid::Complex::str (C++ function), 118
librapid::conj (C++ function), 107
librapid::cos (C++ function), 112
librapid::cosh (C++ function), 108
librapid::cot (C++ function), 112
librapid::csc (C++ function), 112
librapid::CudaStorage (C++ class), 91
librapid::CudaStorage::~~CudaStorage (C++
function), 93
librapid::CudaStorage::begin (C++ function), 93
librapid::CudaStorage::ConstPointer (C++
type), 92
librapid::CudaStorage::ConstReference (C++
type), 92
librapid::CudaStorage::copy (C++ function), 93
librapid::CudaStorage::CudaStorage (C++ func-
tion), 92
librapid::CudaStorage::data (C++ function), 93
librapid::CudaStorage::defaultShape (C++
function), 94
librapid::CudaStorage::DifferenceType (C++
type), 92
librapid::CudaStorage::end (C++ function), 93
librapid::CudaStorage::fromData (C++ function),
93, 94
librapid::CudaStorage::initData (C++ function),
94
librapid::CudaStorage::m_begin (C++ member),
94
librapid::CudaStorage::m_ownsData (C++ mem-
ber), 94
librapid::CudaStorage::m_size (C++ member), 94
librapid::CudaStorage::operator= (C++ func-
tion), 92, 93
librapid::CudaStorage::operator[] (C++ func-
tion), 93
librapid::CudaStorage::Pointer (C++ type), 92
librapid::CudaStorage::Reference (C++ type), 92
librapid::CudaStorage::resize (C++ function), 93
librapid::CudaStorage::resizeImpl (C++ func-
tion), 94
librapid::CudaStorage::Scalar (C++ type), 92
librapid::CudaStorage::set (C++ function), 93
librapid::CudaStorage::size (C++ function), 93
librapid::CudaStorage::SizeType (C++ type), 92
librapid::detail (C++ type), 22, 55, 89, 94, 119
librapid::detail::Abs (C++ struct), 62
librapid::detail::Abs::operator() (C++ func-
tion), 62
librapid::detail::Abs::packet (C++ function), 62
librapid::detail::Acos (C++ struct), 59
librapid::detail::Acos::operator() (C++ func-
tion), 59
librapid::detail::Acos::packet (C++ function),
59
librapid::detail::algorithm (C++ type), 119
librapid::detail::algorithm::expMul (C++
function), 120
librapid::detail::algorithm::HypotLegHuge
(C++ member), 120
librapid::detail::algorithm::HypotLegHugeHelper
(C++ struct), 120
librapid::detail::algorithm::HypotLegHugeHelper<double>
(C++ struct), 120
librapid::detail::algorithm::HypotLegHugeHelper<double>::v
(C++ member), 120
librapid::detail::algorithm::HypotLegHugeHelper<float>
(C++ struct), 120
librapid::detail::algorithm::HypotLegHugeHelper<float>::va
(C++ member), 121
librapid::detail::algorithm::HypotLegTiny
(C++ member), 120
librapid::detail::algorithm::HypotLegTinyHelper
(C++ struct), 121
librapid::detail::algorithm::HypotLegTinyHelper::val
(C++ member), 121
librapid::detail::algorithm::HypotLegTinyHelper<double>
(C++ struct), 121
librapid::detail::algorithm::HypotLegTinyHelper<double>::v
(C++ member), 121
librapid::detail::algorithm::HypotLegTinyHelper<float>
(C++ struct), 121
librapid::detail::algorithm::HypotLegTinyHelper<float>::va
(C++ member), 121
librapid::detail::algorithm::logHypot (C++
function), 119
librapid::detail::algorithm::logP1 (C++ func-
tion), 119
librapid::detail::algorithm::normMinusOne
(C++ function), 119
librapid::detail::Asin (C++ struct), 59
librapid::detail::Asin::operator() (C++ func-
tion), 59
librapid::detail::Asin::packet (C++ function),
59

```

```

librapid::detail::Atan(C++ struct), 59
librapid::detail::Atan::operator() (C++ function), 60
librapid::detail::Atan::packet (C++ function), 60
librapid::detail::Cbirt(C++ struct), 62
librapid::detail::Cbirt::operator() (C++ function), 62
librapid::detail::Cbirt::packet (C++ function), 62
librapid::detail::Ceil(C++ struct), 62
librapid::detail::Ceil::operator() (C++ function), 63
librapid::detail::Ceil::packet (C++ function), 63
librapid::detail::ContainsArrayType (C++ struct), 24
librapid::detail::ContainsArrayType::evaluator (C++ function), 24
librapid::detail::ContainsArrayType::val (C++ member), 25
librapid::detail::Cos(C++ struct), 58
librapid::detail::Cos::operator() (C++ function), 59
librapid::detail::Cos::packet (C++ function), 59
librapid::detail::Cosh(C++ struct), 60
librapid::detail::Cosh::operator() (C++ function), 60
librapid::detail::Cosh::packet (C++ function), 60
librapid::detail::CudaRef(C++ class), 99
librapid::detail::CudaRef::CudaRef(C++ function), 99
librapid::detail::CudaRef::get(C++ function), 99
librapid::detail::CudaRef::m_offset (C++ member), 99
librapid::detail::CudaRef::m_ptr (C++ member), 99
librapid::detail::CudaRef::operator CAST (C++ function), 99
librapid::detail::CudaRef::operator= (C++ function), 99
librapid::detail::CudaRef::PtrType(C++ type), 99
librapid::detail::CudaRef::str(C++ function), 99
librapid::detail::cudaSharedPtrAllocate (C++ function), 95
librapid::detail::Divide(C++ struct), 56
librapid::detail::Divide::operator() (C++ function), 56
librapid::detail::Divide::packet (C++ function), 56
librapid::detail::ElementWiseEqual (C++ struct), 57
librapid::detail::ElementWiseEqual::operator() (C++ function), 58
librapid::detail::ElementWiseEqual::packet (C++ function), 58
librapid::detail::ElementWiseNotEqual (C++ struct), 58
librapid::detail::ElementWiseNotEqual::operator() (C++ function), 58
librapid::detail::ElementWiseNotEqual::packet (C++ function), 58
librapid::detail::Exp(C++ struct), 60
librapid::detail::Exp::operator() (C++ function), 61
librapid::detail::Exp::packet (C++ function), 61
librapid::detail::Floor(C++ struct), 62
librapid::detail::Floor::operator() (C++ function), 62
librapid::detail::Floor::packet (C++ function), 62
librapid::detail::GreaterThan(C++ struct), 57
librapid::detail::GreaterThan::operator() (C++ function), 57
librapid::detail::GreaterThan::packet (C++ function), 57
librapid::detail::GreaterThanEqual (C++ struct), 57
librapid::detail::GreaterThanEqual::operator() (C++ function), 57
librapid::detail::GreaterThanEqual::packet (C++ function), 57
librapid::detail::isArrayOp(C++ function), 55
librapid::detail::isArrayOpArray (C++ function), 55
librapid::detail::isArrayOpWithScalar (C++ function), 55
librapid::detail::IsArrayType(C++ struct), 23
librapid::detail::IsArrayType::val (C++ member), 24
librapid::detail::IsArrayType<array::ArrayView<T>> (C++ struct), 15
librapid::detail::IsArrayType<array::ArrayView<T>>::val (C++ member), 15
librapid::detail::IsArrayType<ArrayRef<T>> (C++ struct), 24
librapid::detail::IsArrayType<ArrayRef<T>>::val (C++ member), 24
librapid::detail::IsArrayType<FunctionRef<T...>> (C++ struct), 24
librapid::detail::IsArrayType<FunctionRef<T...>>::val (C++ member), 24
librapid::detail::LessThan(C++ struct), 56
librapid::detail::LessThan::operator() (C++

```



`function), 57`  
`librapid::detail::LessThan::packet (C++ function), 57`  
`librapid::detail::LessThanEqual (C++ struct), 57`  
`librapid::detail::LessThanEqual::operator() (C++ function), 57`  
`librapid::detail::LessThanEqual::packet (C++ function), 57`  
`librapid::detail::Log (C++ struct), 61`  
`librapid::detail::Log10 (C++ struct), 61`  
`librapid::detail::Log10::operator() (C++ function), 61`  
`librapid::detail::Log10::packet (C++ function), 61`  
`librapid::detail::Log2 (C++ struct), 61`  
`librapid::detail::Log2::operator() (C++ function), 61`  
`librapid::detail::Log2::packet (C++ function), 61`  
`librapid::detail::Log::operator() (C++ function), 61`  
`librapid::detail::Log::packet (C++ function), 61`  
`librapid::detail::makeFunction (C++ function), 55`  
`librapid::detail::Minus (C++ struct), 56`  
`librapid::detail::Minus::operator() (C++ function), 56`  
`librapid::detail::Minus::packet (C++ function), 56`  
`librapid::detail::Multiply (C++ struct), 56`  
`librapid::detail::Multiply::operator() (C++ function), 56`  
`librapid::detail::Multiply::packet (C++ function), 56`  
`librapid::detail::multiprec (C++ type), 121`  
`librapid::detail::multiprec::addSmallX2 (C++ function), 122`  
`librapid::detail::multiprec::addX1 (C++ function), 122`  
`librapid::detail::multiprec::addX2 (C++ function), 121`  
`librapid::detail::multiprec::Fmp (C++ struct), 123`  
`librapid::detail::multiprec::Fmp::val0 (C++ member), 123`  
`librapid::detail::multiprec::Fmp::val1 (C++ member), 123`  
`librapid::detail::multiprec::highHalf (C++ function), 122`  
`librapid::detail::multiprec::sqrError (C++ function), 122`  
`librapid::detail::multiprec::sqrX2 (C++ function), 123`  
`librapid::detail::Neg (C++ struct), 58`  
`librapid::detail::Neg::operator() (C++ function), 58`  
`librapid::detail::Neg::packet (C++ function), 58`  
`librapid::detail::operator!= (C++ function), 98`  
`librapid::detail::operator* (C++ function), 95, 96`  
`librapid::detail::operator*= (C++ function), 96`  
`librapid::detail::operator+ (C++ function), 95`  
`librapid::detail::operator+= (C++ function), 95`  
`librapid::detail::operator/ (C++ function), 96`  
`librapid::detail::operator/= (C++ function), 96`  
`librapid::detail::operator== (C++ function), 98`  
`librapid::detail::operator% (C++ function), 96`  
`librapid::detail::operator%= (C++ function), 96`  
`librapid::detail::operator& (C++ function), 97`  
`librapid::detail::operator&= (C++ function), 97`  
`librapid::detail::operator- (C++ function), 95`  
`librapid::detail::operator-= (C++ function), 95`  
`librapid::detail::operator^ (C++ function), 96`  
`librapid::detail::operator^= (C++ function), 96`  
`librapid::detail::operator| (C++ function), 97`  
`librapid::detail::operator|= (C++ function), 97`  
`librapid::detail::operator> (C++ function), 98`  
`librapid::detail::operator>= (C++ function), 98`  
`librapid::detail::operator>> (C++ function), 97`  
`librapid::detail::operator>>= (C++ function), 97, 98`  
`librapid::detail::operator< (C++ function), 98`  
`librapid::detail::operator<= (C++ function), 98`  
`librapid::detail::operator<< (C++ function), 97`  
`librapid::detail::operator<<= (C++ function), 97`  
`librapid::detail::PhonyNameDueToError::val (C++ member), 24`  
`librapid::detail::Plus (C++ struct), 55`  
`librapid::detail::Plus::operator() (C++ function), 56`  
`librapid::detail::Plus::packet (C++ function), 56`  
`librapid::detail::safeAllocate (C++ function), 89`  
`librapid::detail::safeDeallocate (C++ function), 89`  
`librapid::detail::safePointerCopy (C++ function), 90`  
`librapid::detail::safePointerCopyCuda (C++ function), 99`  
`librapid::detail::Sin (C++ struct), 58`  
`librapid::detail::Sin::operator() (C++ function), 58`  
`librapid::detail::Sin::packet (C++ function), 58`  
`librapid::detail::Sinh (C++ struct), 60`  
`librapid::detail::Sinh::operator() (C++ function), 60`

---

```

librapid::detail::Sinh::packet (C++ function), 60
librapid::detail::Sqrt (C++ struct), 61
librapid::detail::Sqrt::operator() (C++ function), 62
librapid::detail::Sqrt::packet (C++ function), 62
librapid::detail::SubscriptType (C++ struct), 22
librapid::detail::SubscriptType::Direct (C++ type), 22
librapid::detail::SubscriptType::Ref (C++ type), 22
librapid::detail::SubscriptType::Scalar (C++ type), 22
librapid::detail::SubscriptType<CudaStorage<T>> (C++ struct), 23
librapid::detail::SubscriptType<CudaStorage<T>>::Direct (C++ type), 23
librapid::detail::SubscriptType<CudaStorage<T>>::Ref (C++ type), 23
librapid::detail::SubscriptType<CudaStorage<T>>::Scalar (C++ type), 23
librapid::detail::SubscriptType<FixedStorage<T, Dims...>> (C++ struct), 23
librapid::detail::SubscriptType<FixedStorage<T, Dims...>>::Direct (C++ type), 23
librapid::detail::SubscriptType<FixedStorage<T, Dims...>>::Ref (C++ type), 23
librapid::detail::SubscriptType<FixedStorage<T, Dims...>>::Scalar (C++ type), 23
librapid::detail::SubscriptType<Storage<T>> (C++ struct), 22
librapid::detail::SubscriptType<Storage<T>>::Direct (C++ type), 23
librapid::detail::SubscriptType<Storage<T>>::Ref (C++ type), 23
librapid::detail::SubscriptType<Storage<T>>::Scalar (C++ type), 23
librapid::detail::Tan (C++ struct), 59
librapid::detail::Tan::operator() (C++ function), 59
librapid::detail::Tan::packet (C++ function), 59
librapid::detail::Tanh (C++ struct), 60
librapid::detail::Tanh::operator() (C++ function), 60
librapid::detail::Tanh::packet (C++ function), 60
librapid::exp (C++ function), 108
librapid::exp10 (C++ function), 109
librapid::exp2 (C++ function), 108
librapid::FixedStorage (C++ class), 86
librapid::FixedStorage::~FixedStorage (C++ function), 88
librapid::FixedStorage::begin (C++ function), 88
librapid::FixedStorage::cbegin (C++ function), 88
librapid::FixedStorage::cend (C++ function), 88
librapid::FixedStorage::ConstIterator (C++ type), 87
librapid::FixedStorage::ConstPointer (C++ type), 87
librapid::FixedStorage::ConstReference (C++ type), 87
librapid::FixedStorage::ConstReverseIterator (C++ type), 87
librapid::FixedStorage::copy (C++ function), 88
librapid::FixedStorage::crbegin (C++ function), 89
librapid::FixedStorage::crend (C++ function), 89
librapid::FixedStorage::data (C++ function), 88
librapid::FixedStorage::defaultShape (C++ function), 89
librapid::FixedStorage::DifferenceType (C++ type), 87
librapid::FixedStorage::end (C++ function), 88
librapid::FixedStorage::FixedStorage (C++ function), 87
librapid::FixedStorage::Iterator (C++ type), 87
librapid::FixedStorage::m_data (C++ member), 89
librapid::FixedStorage::operator= (C++ function), 88
librapid::FixedStorage::operator[] (C++ function), 88
librapid::FixedStorage::Pointer (C++ type), 87
librapid::FixedStorage::rbegin (C++ function), 88, 89
librapid::FixedStorage::Reference (C++ type), 87
librapid::FixedStorage::rend (C++ function), 89
librapid::FixedStorage::resize (C++ function), 88
librapid::FixedStorage::ReverseIterator (C++ type), 87
librapid::FixedStorage::Scalar (C++ type), 87
librapid::FixedStorage::size (C++ function), 88
librapid::FixedStorage::Size (C++ member), 89
librapid::FixedStorage::SizeType (C++ type), 87
librapid::floor (C++ function), 114
librapid::imag (C++ function), 106
librapid::linalg (C++ type), 13
librapid::linalg::cublasGemmComputeType (C++ function), 14
librapid::linalg::CuBLASGemmComputeType (C++ struct), 14
librapid::linalg::CuBLASGemmComputeType::computeType (C++ member), 14

```

---

```

librapid::linalg::CuBLASgemmComputeType::scaleType (C++ member), 14
librapid::linalg::gemm (C++ function), 13, 14
librapid::log (C++ function), 109, 110
librapid::log10 (C++ function), 113
librapid::log2 (C++ function), 113
librapid::norm (C++ function), 113
librapid::operator!= (C++ function), 105, 106
librapid::operator* (C++ function), 103
librapid::operator+ (C++ function), 101
librapid::operator/ (C++ function), 103, 104
librapid::operator== (C++ function), 104, 105
librapid::operator- (C++ function), 101, 102
librapid::polar (C++ function), 113
librapid::polarPositiveNanInfZeroRho (C++ function), 108
librapid::pow (C++ function), 110, 111
librapid::proj (C++ function), 112
librapid::random (C++ function), 114
librapid::real (C++ function), 106
librapid::sec (C++ function), 112
librapid::sin (C++ function), 113
librapid::sinh (C++ function), 111
librapid::sqrt (C++ function), 106
librapid::Storage (C++ class), 82
librapid::Storage::~~Storage (C++ function), 84
librapid::Storage::begin (C++ function), 85
librapid::Storage::cbegin (C++ function), 85
librapid::Storage::cend (C++ function), 85
librapid::Storage::ConstIterator (C++ type), 83
librapid::Storage::ConstPointer (C++ type), 83
librapid::Storage::ConstRawPointer (C++ type), 83
librapid::Storage::ConstReference (C++ type), 83
librapid::Storage::ConstReverseIterator (C++ type), 83
librapid::Storage::copy (C++ function), 84
librapid::Storage::crbegin (C++ function), 85
librapid::Storage::crend (C++ function), 85
librapid::Storage::data (C++ function), 85
librapid::Storage::defaultShape (C++ function), 85, 86
librapid::Storage::DifferenceType (C++ type), 83
librapid::Storage::end (C++ function), 85
librapid::Storage::fromData (C++ function), 85, 86
librapid::Storage::initData (C++ function), 86
librapid::Storage::Iterator (C++ type), 83
librapid::Storage::m_begin (C++ member), 86
librapid::Storage::m_ownsData (C++ member), 86
librapid::Storage::m_size (C++ member), 86
librapid::Storage::operator= (C++ function), 84
librapid::Storage::operator[] (C++ function), 85
librapid::Storage::Pointer (C++ type), 83
librapid::Storage::RawPointer (C++ type), 83
librapid::Storage::rbegin (C++ function), 85
librapid::Storage::Reference (C++ type), 83
librapid::Storage::rend (C++ function), 85
librapid::Storage::resize (C++ function), 84, 85
librapid::Storage::resizeImpl (C++ function), 86
librapid::Storage::ReverseIterator (C++ type), 83
librapid::Storage::Scalar (C++ type), 83
librapid::Storage::set (C++ function), 84
librapid::Storage::size (C++ function), 85
librapid::Storage::SizeType (C++ type), 83
librapid::Storage::Storage (C++ function), 83, 84
librapid::Storage::toHostStorage (C++ function), 84
librapid::Storage::toHostStorageUnsafe (C++ function), 84
librapid::Stride (C++ class), 81
librapid::Stride::operator= (C++ function), 82
librapid::Stride::Stride (C++ function), 82
librapid::tan (C++ function), 114
librapid::tanh (C++ function), 111
librapid::typetraits (C++ type), 25, 30, 63, 82, 90, 99, 123
librapid::typetraits::DescriptorExtractor (C++ struct), 63
librapid::typetraits::DescriptorExtractor::Type (C++ type), 64
librapid::typetraits::DescriptorExtractor<:librapid::detail::Functor, Args...> (C++ struct), 31
librapid::typetraits::DescriptorExtractor<:librapid::detail::Functor, Args...>::Type (C++ type), 31
librapid::typetraits::DescriptorExtractor<array::ArrayContStorageType> (C++ struct), 31
librapid::typetraits::DescriptorExtractor<array::ArrayContStorageType>::Type (C++ type), 31
librapid::typetraits::DescriptorExtractor<array::ArrayView (C++ struct), 31
librapid::typetraits::DescriptorExtractor<array::ArrayView (C++ type), 31
librapid::typetraits::DescriptorMerger (C++ struct), 63
librapid::typetraits::DescriptorMerger::Type (C++ type), 63
librapid::typetraits::DescriptorMerger<Descriptor1, Descriptor1> (C++ struct), 63
librapid::typetraits::DescriptorMerger<Descriptor1, Descriptor1>::Type (C++ type), 63
librapid::typetraits::DescriptorType (C++ struct), 64
librapid::typetraits::DescriptorType::FirstDescriptor (C++ type), 65

```

librapid::typetraits::DescriptorType::FirstType (C++ member), 65–71  
 (C++ type), 65 librapid::typetraits::PhonyNameDueToError::kernelNameScalar  
 librapid::typetraits::DescriptorType::RestDescriptor (C++ member), 65–71  
 (C++ type), 65 librapid::typetraits::PhonyNameDueToError::name  
 librapid::typetraits::DescriptorType::Type (C++ member), 65–81  
 (C++ type), 65 librapid::typetraits::PhonyNameDueToError::Packet  
 librapid::typetraits::DescriptorType\_t (C++ (C++ type), 25 librapid::typetraits::PhonyNameDueToError::packetWidth  
 type), 63 librapid::typetraits::PhonyNameDueToError::Scalar  
 librapid::typetraits::impl (C++ type), 81 (C++ member), 25  
 librapid::typetraits::impl::descriptorExtractor librapid::typetraits::PhonyNameDueToError::Scalar  
 (C++ function), 81 (C++ type), 25, 30  
 librapid::typetraits::IsArrayContainer (C++ librapid::typetraits::PhonyNameDueToError::supportsArithmetic  
 struct), 26 (C++ member), 25  
 librapid::typetraits::IsArrayContainer<array::ArrayContainerTypeShape<SizeType, dims>, StorageScalar>> (C++ struct), 15 (C++ member), 25  
 librapid::typetraits::IsCudaStorage (C++ librapid::typetraits::PhonyNameDueToError::supportsLogical  
 struct), 100 (C++ member), 25  
 librapid::typetraits::IsCudaStorage<CudaStorage<Scalar>> librapid::typetraits::PhonyNameDueToError::type  
 (C++ struct), 100 (C++ member), 25, 30  
 librapid::typetraits::IsFixedStorage (C++ librapid::typetraits::PhonyNameDueToError::Type  
 struct), 91 (C++ type), 64  
 librapid::typetraits::IsFixedStorage<FixedStorage<Size...>> (C++ struct), 91 librapid::typetraits::TypeInfo<::librapid::detail::Abs>  
 (C++ struct), 46  
 librapid::typetraits::IsStorage (C++ struct), librapid::typetraits::TypeInfo<::librapid::detail::Abs>:f  
 91 (C++ member), 46  
 librapid::typetraits::IsStorage<Storage<Scalar>> librapid::typetraits::TypeInfo<::librapid::detail::Abs>:g  
 (C++ struct), 91 (C++ function), 46  
 librapid::typetraits::LIBRAPID\_DEFINE\_AS\_TYPE librapid::typetraits::TypeInfo<::librapid::detail::Abs>:g  
 (C++ function), 30, 90, 99 (C++ function), 46  
 librapid::typetraits::PhonyNameDueToError::all librapid::typetraits::TypeInfo<::librapid::detail::Abs>:k  
 (C++ member), 25, 30 (C++ member), 46  
 librapid::typetraits::PhonyNameDueToError::Back librapid::typetraits::TypeInfo<::librapid::detail::Abs>:r  
 (C++ type), 25, 30 (C++ member), 46  
 librapid::typetraits::PhonyNameDueToError::can librapid::typetraits::TypeInfo<::librapid::detail::Acos>  
 (C++ member), 25 (C++ struct), 40  
 librapid::typetraits::PhonyNameDueToError::can librapid::typetraits::TypeInfo<::librapid::detail::Acos>:s  
 (C++ member), 26 (C++ member), 41  
 librapid::typetraits::PhonyNameDueToError::cuda librapid::typetraits::TypeInfo<::librapid::detail::Acos>:s  
 (C++ member), 25 (C++ function), 41  
 librapid::typetraits::PhonyNameDueToError::Cuda librapid::typetraits::TypeInfo<::librapid::detail::Acos>:s  
 (C++ member), 25 (C++ function), 41  
 librapid::typetraits::PhonyNameDueToError::fill librapid::typetraits::TypeInfo<::librapid::detail::Acos>:s  
 (C++ member), 65–81 (C++ member), 41  
 librapid::typetraits::PhonyNameDueToError::get librapid::typetraits::TypeInfo<::librapid::detail::Acos>:s  
 (C++ function), 65–81 (C++ member), 41  
 librapid::typetraits::PhonyNameDueToError::get librapid::typetraits::TypeInfo<::librapid::detail::Asin>  
 (C++ function), 65–71 (C++ struct), 40  
 librapid::typetraits::PhonyNameDueToError::get librapid::typetraits::TypeInfo<::librapid::detail::Asin>:s  
 (C++ function), 65–81 (C++ member), 40  
 librapid::typetraits::PhonyNameDueToError::get librapid::typetraits::TypeInfo<::librapid::detail::Asin>:s  
 (C++ function), 65–71 (C++ function), 40  
 librapid::typetraits::PhonyNameDueToError::kernel librapid::typetraits::TypeInfo<::librapid::detail::Asin>:s  
 (C++ member), 65–81 (C++ function), 40  
 librapid::typetraits::PhonyNameDueToError::kernel librapid::typetraits::TypeInfo<::librapid::detail::Asin>:s  
 (C++ member), 65–81 (C++ function), 40

<b>Index</b>	<b>145</b>
--------------	------------





<b>Index</b>	<b>147</b>
--------------	------------





```

librapid::typetraits::TypeInfo<array::ArrayView<T>> Dims...>::Scalar (C++ type), 91
    (C++ struct), 26
librapid::typetraits::TypeInfo<array::ArrayView<T>>::allLowerVectorisation
    (C++ member), 27
librapid::typetraits::TypeInfo<array::ArrayView<T>>::Backend, 90
    (C++ type), 26
librapid::typetraits::TypeInfo<array::ArrayView<T>>::Scalar, 90
    (C++ type), 26
librapid::typetraits::TypeInfo<array::ArrayView<T>>::Type, 90
    (C++ member), 27
librapid::typetraits::TypeInfo<Complex<T>>
    (C++ struct), 123
librapid::typetraits::TypeInfo<Complex<T>>::allLowerVectorisation
    (C++ member), 124
librapid::typetraits::TypeInfo<Complex<T>>::canMultiply
    (C++ member), 124
librapid::typetraits::TypeInfo<Complex<T>>::canMultiply
    (C++ member), 124
librapid::typetraits::TypeInfo<Complex<T>>::CudaType
    (C++ member), 124
librapid::typetraits::TypeInfo<Complex<T>>::LIMIT_TPL
    (C++ function), 124
librapid::typetraits::TypeInfo<Complex<T>>::name
    (C++ member), 124
librapid::typetraits::TypeInfo<Complex<T>>::Packet
    (C++ type), 123
librapid::typetraits::TypeInfo<Complex<T>>::packetWidth
    (C++ member), 124
librapid::typetraits::TypeInfo<Complex<T>>::Scalar
    (C++ type), 123
librapid::typetraits::TypeInfo<Complex<T>>::supportsArithmetic
    (C++ member), 124
librapid::typetraits::TypeInfo<Complex<T>>::supportsBinary
    (C++ member), 124
librapid::typetraits::TypeInfo<Complex<T>>::supportsLogical
    (C++ member), 124
librapid::typetraits::TypeInfo<Complex<T>>::type
    (C++ member), 124
librapid::typetraits::TypeInfo<CudaStorage<Scalar_>>
    (C++ struct), 99
librapid::typetraits::TypeInfo<CudaStorage<Scalar_>>::Backend
    (C++ type), 100
librapid::typetraits::TypeInfo<CudaStorage<Scalar_>>::isLibRapidType
    (C++ member), 100
librapid::typetraits::TypeInfo<CudaStorage<Scalar_>>::Scalar
    (C++ type), 100
librapid::typetraits::TypeInfo<FixedStorage<Scalar_,
    Dims...>> (C++ struct), 90
librapid::typetraits::TypeInfo<FixedStorage<Scalar_,
    Dims...>>::Backend (C++ type), 91
librapid::typetraits::TypeInfo<FixedStorage<Scalar_,
    Dims...>>::isLibRapidType (C++ mem-
    ber), 91
librapid::typetraits::TypeInfo<FixedStorage<Scalar_,

```