

---

# **Umpire Documentation**

*Release 2.0.0*

**David Beckingsale**

**Jan 08, 2020**



# BASICS

<b>1</b>	<b>Getting Started</b>	<b>3</b>
1.1	Installation . . . . .	3
1.2	Basic Usage . . . . .	4
<b>2</b>	<b>Umpire Tutorial</b>	<b>5</b>
2.1	Allocators . . . . .	5
2.2	Resources . . . . .	6
2.3	Operations . . . . .	7
2.4	Dynamic Pools . . . . .	13
2.5	Introspection . . . . .	16
2.6	Typed Allocators . . . . .	17
2.7	Replay . . . . .	18
2.8	C API: Allocators . . . . .	19
2.9	C API: Resources . . . . .	20
2.10	C API: Pools . . . . .	20
2.11	FORTRAN API: Allocators . . . . .	21
<b>3</b>	<b>Advanced Configuration</b>	<b>23</b>
<b>4</b>	<b>Umpire Cookbook</b>	<b>25</b>
4.1	Growing and Shrinking a Pool . . . . .	25
4.2	Disable Introspection . . . . .	27
4.3	Apply Memory Advice to a Pool . . . . .	28
4.4	Apply Memory Advice with a Specific Device ID . . . . .	29
4.5	Moving Host Data to Managed Memory . . . . .	30
4.6	Improving DynamicPool Performance with a Coalesce Heuristic . . . . .	31
4.7	Move Allocations Between NUMA Nodes . . . . .	33
4.8	Determining the Largest Block of Available Memory in Pool . . . . .	35
4.9	Coalescing Pool Memory . . . . .	36
4.10	Building a Pinned Memory Pool in FORTRAN . . . . .	37
4.11	Visualizing Allocators . . . . .	38
4.12	Mixed Pool Creation and Algorithm Basics . . . . .	39
4.13	Thread Safe Allocator . . . . .	40
<b>5</b>	<b>Features</b>	<b>43</b>
5.1	Allocators . . . . .	43
5.2	Strategies . . . . .	43
5.3	Operations . . . . .	44
5.4	Logging and Replay of Umpire Events . . . . .	46
5.5	File I/O . . . . .	49

<b>6</b>	<b>API</b>	<b>51</b>
6.1	Class Hierarchy . . . . .	51
6.2	File Hierarchy . . . . .	51
6.3	Full API . . . . .	51
<b>7</b>	<b>Contribution Guide</b>	<b>245</b>
7.1	Forking Umpire . . . . .	245
<b>8</b>	<b>Developer Guide</b>	<b>247</b>
	<b>Index</b>	<b>249</b>

Umpire is a resource management library that allows the discovery, provision, and management of memory on next-generation hardware architectures with NUMA memory hierarchies.

- Take a look at our Getting Started guide for all you need to get up and running with Umpire.
- If you are looking for developer documentation on a particular function, check out the code documentation.
- Want to contribute? Take a look at our developer and contribution guides.

Any questions? File an issue on GitHub, or email [umpire-dev@llnl.gov](mailto:umpire-dev@llnl.gov)



## GETTING STARTED

This page provides information on how to quickly get up and running with Umpire.

### 1.1 Installation

Umpire is hosted on GitHub [here](#). To clone the repo into your local working space, type:

```
$ git clone --recursive https://github.com/LLNL/Umpire.git
```

The `--recursive` argument is required to ensure that the *BLT* submodule is also checked out. *BLT* is the build system we use for Umpire.

#### 1.1.1 Building Umpire

Umpire uses CMake and BLT to handle builds. Make sure that you have a modern compiler loaded and the configuration is as simple as:

```
$ mkdir build && cd build  
$ cmake -DCUDA_TOOLKIT_ROOT_DIR=/path/to/cuda ../
```

By default, Umpire will attempt to build with CUDA. CMake will provide output about which compiler is being used, and what version of CUDA was detected. Once CMake has completed, Umpire can be built with Make:

```
$ make
```

For more advanced configuration, see *Advanced Configuration*.

#### 1.1.2 Installing Umpire

To install Umpire, just run:

```
$ make install
```

Umpire install files to the `lib`, `include` and `bin` directories of the `CMAKE_INSTALL_PREFIX`. Additionally, Umpire installs a CMake configuration file that can help you use Umpire in other projects. By setting `umpire_DIR` to point to the root of your Umpire installation, you can call `find_package(umpire)` inside your CMake project and Umpire will be automatically detected and available for use.

## 1.2 Basic Usage

Let's take a quick tour through Umpire's most important features. A complete listing you can compile is included at the bottom of the page. First, let's grab an Allocator and allocate some memory. This is the interface through which you will want to access data:

```
auto& rm = umpire::ResourceManager::getInstance();
umpire::Allocator allocator = rm.getAllocator("HOST");

float* my_data = static_cast<float*>(allocator.allocate(100*sizeof(float)));
```

This code grabs the default allocator for the host memory, and uses it to allocate an array of 100 floats. We can ask for different Allocators to allocate memory in different places. Let's ask for a device allocator:

```
umpire::Allocator device_allocator = rm.getAllocator("DEVICE");

float* my_data_device = static_cast<float*>(device_allocator.
↳allocate(100*sizeof(float)));
```

This code gets the default device allocator, and uses it to allocate an array of 100 floats. Remember, since this is a device pointer, there is no guarantee you will be able to access it on the host. Luckily, Umpire's ResourceManager can copy one pointer to another transparently. Let's copy the data from our first pointer to the DEVICE-allocated pointer.

```
rm.copy(my_data, my_data_device);
```

To free any memory allocated, you can use the deallocate function of the Allocator, or the ResourceManager. Asking the ResourceManager to deallocate memory is slower, but useful if you don't know how or where an allocation was made:

```
allocator.deallocate(my_data); // deallocate using Allocator
rm.deallocate(my_data_device); // deallocate using ResourceManager
```



## UMPIRE TUTORIAL

This section is a tutorial introduction to Umpire. We start with the most basic memory allocation, and move through topics like allocating on different resources, using allocation strategies to change how memory is allocated, using operations to move and modify data, and how to use Umpire introspection capability to find out information about Allocators and allocations.

These examples are all built as part of Umpire, and you can find the files in the [examples](#) directory at the root of the Umpire repository. Feel free to play around and modify these examples to experiment with all of Umpire's functionality.

The following tutorial examples assume a working knowledge of C++ and a general understanding of how memory is laid out in modern heterogeneous computers. The main thing to remember is that in many systems, memory on other execution devices (like GPUs) might not be directly accessible from the CPU. If you try and access this memory your program will error! Luckily, Umpire makes it easy to move data around, and check where it is, as you will see in the following sections.

### 2.1 Allocators

The fundamental concept for accessing memory through Umpire is the `umpire::Allocator`. An `umpire::Allocator` is a C++ object that can be used to allocate and deallocate memory, as well as query a pointer to get some extra information about it.

All `umpire::Allocator`s are created and managed by Umpire's `umpire::ResourceManager`. To get an Allocator, you need to ask for one:

```
auto& rm = umpire::ResourceManager::getInstance();

umpire::Allocator allocator = rm.getAllocator("HOST");
```

Once you have an `umpire::Allocator` you can use it to allocate and deallocate memory:

```
double* data = static_cast<double*>(
    allocator.allocate(SIZE*sizeof(double)));

std::cout << "Allocated " << (SIZE*sizeof(double)) << " bytes using the "
    << allocator.getName() << " allocator...";

allocator.deallocate(data);
```

In the next section, we will see how to allocate memory using different resources.

```
////////////////////////////////////  
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire  
// project contributors. See the COPYRIGHT file for details.  
//  
// SPDX-License-Identifier: (MIT)  
////////////////////////////////////  
#include "umpire/Allocator.hpp"  
#include "umpire/ResourceManager.hpp"  
  
int main(int, char**) {  
  
    constexpr std::size_t SIZE = 1024;  
  
    auto& rm = umpire::ResourceManager::getInstance();  
  
    umpire::Allocator allocator = rm.getAllocator("HOST");  
  
    double* data = static_cast<double*>( allocator.allocate(SIZE*sizeof(double)));  
  
    std::cout << "Allocated " << (SIZE*sizeof(double)) << " bytes using the "  
        << allocator.getName() << " allocator...";  
  
    allocator.deallocate(data);  
  
    std::cout << " deallocated." << std::endl;  
  
    return 0;  
}
```

## 2.2 Resources

Each computer system will have a number of distinct places in which the system will allow you to allocate memory. In Umpire’s world, these are *memory resources*. A memory resource can correspond to a hardware resource, but can also be used to identify memory with a particular characteristic, like “pinned” memory in a GPU system.

When you configure Umpire, it will create `umpire::resource::MemoryResource`s according to what is available on the system you are building for. For each resource, Umpire will create a default `umpire::Allocator` that you can use. In the previous example, we were actually using an `umpire::Allocator` created for the memory resource corresponding to the CPU memory

The easiest way to identify resources is by name. The “HOST” resource is always available. In a modern NVIDIA GPU system, we also have resources that represent global GPU memory (“DEVICE”), unified memory that can be accessed by the CPU or GPU (“UM”) and host memory that can be accessed by the GPU (“PINNED”);

Umpire will create an `umpire::Allocator` for each of these resources, and you can get them using the same `umpire::ResourceManager::getAllocator()` call you saw in the previous example:

```
umpire::Allocator allocator = rm.getAllocator(resource);
```

Note that every allocator supports the same calls, no matter which resource it is for, this means we can run the same code for all the resources available in the system.

In the next example, we will learn how to move data between resources using operations.

```

////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

void allocate_and_deallocate(const std::string& resource)
{
    constexpr std::size_t SIZE = 1024;

    auto& rm = umpire::ResourceManager::getInstance();

    umpire::Allocator allocator = rm.getAllocator(resource);

    double* data = static_cast<double*>(
        allocator.allocate(SIZE*sizeof(double)));

    std::cout << "Allocated " << (SIZE*sizeof(double)) << " bytes using the "
        << allocator.getName() << " allocator...";

    allocator.deallocate(data);

    std::cout << " deallocated." << std::endl;
}

int main(int, char**) {
    allocate_and_deallocate("HOST");

#ifdef UMPIRE_ENABLE_CUDA
    allocate_and_deallocate("DEVICE");
    allocate_and_deallocate("UM");
    allocate_and_deallocate("PINNED");
#endif
#ifdef UMPIRE_ENABLE_HIP
    allocate_and_deallocate("DEVICE");
    allocate_and_deallocate("PINNED");
#endif

    return 0;
}

```

## 2.3 Operations

Moving and modifying data in a heterogenous memory system can be annoying. You have to keep track of the source and destination, and often use vendor-specific APIs to perform the modifications. In Umpire, all data modification and movement is wrapped up in a concept we call *operations*. Full documentation for all of these is available [here](#). The full code listing for each example is include at the bottom of the page.

### 2.3.1 Copy

Let's start by looking at how we copy data around. The `umpire::ResourceManager` provides an interface to copy that handles figuring out where the source and destination pointers were allocated, and selects the correct implementation to copy the data:

```
rm.copy(dest_data, source_data);
```

This example allocates the destination data using any valid Allocator.

### 2.3.2 Move

If you want to move data to a new Allocator and deallocate the old copy, Umpire provides a `umpire::ResourceManager::move()` operation.

```
double* dest_data = static_cast<double*>(
    rm.move(source_data, dest_allocator));
```

The move operation combines an allocation, a copy, and a deallocate into one function call, allowing you to move data without having to have the destination data allocated. As always, this operation will work with any valid destination Allocator.

### 2.3.3 Memset

Setting a whole block of memory to a value (like 0) is a common operation, that most people know as a memset. Umpire provides a `umpire::ResourceManager::memset()` implementation that can be applied to any allocation, regardless of where it came from:

```
rm.memset(data, 0);
```

### 2.3.4 Reallocate

Reallocating CPU memory is easy, there is a function designed specifically to do it: `realloc`. When the original allocation was made in a different memory however, you can be out of luck. Umpire provides a `umpire::ResourceManager::reallocate()` operation:

```
data = static_cast<double*>(rm.reallocate(data, REALLOCATED_SIZE));
```

This method returns a pointer to the reallocated data. Like all operations, this can be used regardless of the Allocator used for the source data.

### 2.3.5 Listings

Copy Example Listing

```
////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

void copy_data(double* source_data, std::size_t size, const std::string& destination)
{
    auto& rm = umpire::ResourceManager::getInstance();
    auto dest_allocator = rm.getAllocator(destination);

    double* dest_data = static_cast<double*>(
        dest_allocator.allocate(size*sizeof(double)));

    rm.copy(dest_data, source_data);

    std::cout << "Copied source data (" << source_data << ") to destination "
        << destination << " (" << dest_data << ")" << std::endl;

    dest_allocator.deallocate(dest_data);
}

int main(int, char**) {
    constexpr std::size_t SIZE = 1024;

    auto& rm = umpire::ResourceManager::getInstance();
    auto allocator = rm.getAllocator("HOST");

    double* data = static_cast<double*>(
        allocator.allocate(SIZE*sizeof(double)));

    std::cout << "Allocated " << (SIZE*sizeof(double)) << " bytes using the "
        << allocator.getName() << " allocator." << std::endl;

    std::cout << "Filling with 0.0...";

    for (std::size_t i = 0; i < SIZE; i++) {
        data[i] = 0.0;
    }

    std::cout << "done." << std::endl;

    copy_data(data, SIZE, "HOST");
    #if defined(UMPIRE_ENABLE_CUDA)
    copy_data(data, SIZE, "DEVICE");
    copy_data(data, SIZE, "UM");
    copy_data(data, SIZE, "PINNED");
    #endif
    #if defined(UMPIRE_ENABLE_HIP)
    copy_data(data, SIZE, "DEVICE");
    copy_data(data, SIZE, "PINNED");
    #endif

    allocator.deallocate(data);

    return 0;
}

```

Move Example Listing

```

////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

double* move_data(double* source_data, const std::string& destination)
{
    auto& rm = umpire::ResourceManager::getInstance();
    auto dest_allocator = rm.getAllocator(destination);

    std::cout << "Moved source data (" << source_data << ") to destination ";

    double* dest_data = static_cast<double*>(
        rm.move(source_data, dest_allocator));

    std::cout << destination << " (" << dest_data << ")" << std::endl;

    return dest_data;
}

int main(int, char**) {
    constexpr std::size_t SIZE = 1024;

    auto& rm = umpire::ResourceManager::getInstance();

    auto allocator = rm.getAllocator("HOST");

    double* data = static_cast<double*>(
        allocator.allocate(SIZE*sizeof(double)));

    std::cout << "Allocated " << (SIZE*sizeof(double)) << " bytes using the "
        << allocator.getName() << " allocator." << std::endl;

    std::cout << "Filling with 0.0...";

    for (std::size_t i = 0; i < SIZE; i++) {
        data[i] = 0.0;
    }

    std::cout << "done." << std::endl;

    data = move_data(data, "HOST");
    #if defined(UMPIRE_ENABLE_CUDA)
    data = move_data(data, "DEVICE");
    data = move_data(data, "UM");
    data = move_data(data, "PINNED");
    #endif
    #if defined(UMPIRE_ENABLE_HIP)
    data = move_data(data, "DEVICE");
    data = move_data(data, "PINNED");
    #endif

    rm.deallocate(data);
}

```

(continues on next page)

(continued from previous page)

```

return 0;
}

```

### Memset Example Listing

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

int main(int, char**) {
    constexpr std::size_t SIZE = 1024;

    auto& rm = umpire::ResourceManager::getInstance();

    const std::string destinations[] = {
        "HOST"
#ifdef UMPIRE_ENABLE_CUDA
        , "DEVICE"
        , "UM"
        , "PINNED"
#endif
#ifdef UMPIRE_ENABLE_HIP
        , "DEVICE"
        , "PINNED"
#endif
    };

    for (auto& destination : destinations) {
        auto allocator = rm.getAllocator(destination);
        double* data = static_cast<double*>(
            allocator.allocate(SIZE*sizeof(double)));

        std::cout << "Allocated " << (SIZE*sizeof(double)) << " bytes using the "
            << allocator.getName() << " allocator." << std::endl;

        rm.memset(data, 0);

        std::cout << "Set data from " << destination << " (" << data << ") to 0." <<
        ↪std::endl;

        allocator.deallocate(data);
    }

    return 0;
}

```

### Reallocate Example Listing

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.

```

(continues on next page)

```

//
// SPDX-License-Identifier: (MIT)
//
//
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

int main(int, char**) {
    constexpr std::size_t SIZE = 1024;
    constexpr std::size_t REALLOCATED_SIZE = 256;

    auto& rm = umpire::ResourceManager::getInstance();

    const std::string destinations[] = {
        "HOST"
#ifdef UMPIRE_ENABLE_CUDA
        , "DEVICE"
        , "UM"
        , "PINNED"
#endif
#ifdef UMPIRE_ENABLE_HIP
        , "DEVICE"
        , "PINNED"
#endif
    };

    for (auto& destination : destinations) {
        auto allocator = rm.getAllocator(destination);
        double* data = static_cast<double*>(
            allocator.allocate(SIZE*sizeof(double)));

        std::cout << "Allocated " << (SIZE*sizeof(double)) << " bytes using the "
            << allocator.getName() << " allocator." << std::endl;

        std::cout << "Reallocating data (" << data << ") to size "
            << REALLOCATED_SIZE << "...";

        data = static_cast<double*>(rm.reallocate(data, REALLOCATED_SIZE));

        std::cout << "done. Reallocated data (" << data << ")" << std::endl;

        allocator.deallocate(data);
    }

    return 0;
}

```



## 2.4 Dynamic Pools

Frequently allocating and deallocating memory can be quite costly, especially when you are making large allocations or allocating on different memory resources. To mitigate this, Umpire provides allocation strategies that can be used to customize how data is obtained from the system.

In this example, we will look at the `umpire::strategy::DynamicPool` strategy. This is a simple pooling algorithm that can fulfill requests for allocations of any size. To create a new Allocator using the `umpire::strategy::DynamicPool` strategy:

```
auto allocator = rm.getAllocator(resource);

auto pooled_allocator =
    rm.makeAllocator<umpire::strategy::DynamicPool>(resource + "_pool",
                                                    allocator);
```

We have to provide a new name for the Allocator, as well as the underlying Allocator we wish to use to grab memory.

Once you have an Allocator, you can allocate and deallocate memory as before, without needing to worry about the underlying algorithm used for the allocations:

```
double* data = static_cast<double*>(
    pooled_allocator.allocate(SIZE*sizeof(double)));

std::cout << "Allocated " << (SIZE*sizeof(double)) << " bytes using the "
    << pooled_allocator.getName() << " allocator...";

pooled_allocator.deallocate(data);
```

Don't forget, these strategies can be created on top of any valid Allocator:

```
allocate_and_deallocate_pool("HOST");

#ifdef UMPIRE_ENABLE_CUDA
    allocate_and_deallocate_pool("DEVICE");
    allocate_and_deallocate_pool("UM");
    allocate_and_deallocate_pool("PINNED");
#endif
#ifdef UMPIRE_ENABLE_HIP
    allocate_and_deallocate_pool("DEVICE");
    allocate_and_deallocate_pool("PINNED");
#endif
```

Most Umpire users will make allocations that use the GPU via the `umpire::strategy::DynamicPool`, to help mitigate the cost of allocating memory on these devices.

You can tune the way that `umpire::strategy::DynamicPool` allocates memory using two parameters: the initial size, and the minimum size. The initial size controls how large the first underlying allocation made will be, regardless of the requested size. The minimum size controls the minimum size of any future underlying allocations. These two parameters can be passed when constructing a pool:

```
auto allocator = rm.getAllocator(resource);

auto pooled_allocator =
    rm.makeAllocator<umpire::strategy::DynamicPool>(resource + "_pool",
                                                    allocator,
```

(continues on next page)

(continued from previous page)

```

initial_size, /* default = 512Mb*/
min_block_size /* default = 1Mb */
↪);

```

Depending on where you are allocating data, you might want to use different sizes. It's easy to construct multiple pools with different configurations:

```

allocate_and_deallocate_pool("HOST", 65536, 512);
#ifdef UMPIRE_ENABLE_CUDA
allocate_and_deallocate_pool("DEVICE", (1024*1024*1024), (1024*1024));
allocate_and_deallocate_pool("UM", (1024*64), 1024);
allocate_and_deallocate_pool("PINNED", (1024*16), 1024);
#endif
#ifdef UMPIRE_ENABLE_HIP
allocate_and_deallocate_pool("DEVICE", (1024*1024*1024), (1024*1024));
allocate_and_deallocate_pool("PINNED", (1024*16), 1024);
#endif

```

There are lots of different strategies that you can use, we will look at some of them in this tutorial. A complete list of strategies can be found [here](#).

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

#include "umpire/strategy/DynamicPool.hpp"

void allocate_and_deallocate_pool(const std::string& resource)
{
    constexpr std::size_t SIZE = 1024;

    auto& rm = umpire::ResourceManager::getInstance();

    auto allocator = rm.getAllocator(resource);

    auto pooled_allocator =
        rm.makeAllocator<umpire::strategy::DynamicPool>(resource + "_pool",
                                                       allocator);

    double* data = static_cast<double*>(
        pooled_allocator.allocate(SIZE*sizeof(double)));

    std::cout << "Allocated " << (SIZE*sizeof(double)) << " bytes using the "
              << pooled_allocator.getName() << " allocator...";

    pooled_allocator.deallocate(data);

    std::cout << " deallocated." << std::endl;
}

int main(int, char**) {

```

(continues on next page)

(continued from previous page)

```

    allocate_and_deallocate_pool("HOST");

#ifdef UMPIRE_ENABLE_CUDA
    allocate_and_deallocate_pool("DEVICE");
    allocate_and_deallocate_pool("UM");
    allocate_and_deallocate_pool("PINNED");
#endif
#ifdef UMPIRE_ENABLE_HIP
    allocate_and_deallocate_pool("DEVICE");
    allocate_and_deallocate_pool("PINNED");
#endif

    return 0;
}

```

```

////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

#include "umpire/strategy/DynamicPool.hpp"

void allocate_and_deallocate_pool(
    const std::string& resource,
    std::size_t initial_size,
    std::size_t min_block_size)
{
    constexpr std::size_t SIZE = 1024;

    auto& rm = umpire::ResourceManager::getInstance();

    auto allocator = rm.getAllocator(resource);

    auto pooled_allocator =
        rm.makeAllocator<umpire::strategy::DynamicPool>(resource + "_pool",
                                                       allocator,
                                                       initial_size, /* default = 512Mb*/
                                                       min_block_size /* default = 1Mb */
    );

    double* data = static_cast<double*>(
        pooled_allocator.allocate(SIZE*sizeof(double)));

    std::cout << "Allocated " << (SIZE*sizeof(double)) << " bytes using the "
              << pooled_allocator.getName() << " allocator...";

    pooled_allocator.deallocate(data);

    std::cout << " deallocated." << std::endl;
}

int main(int, char**) {

```

(continues on next page)

(continued from previous page)

```

    allocate_and_deallocate_pool("HOST", 65536, 512);
#ifdef UMPIRE_ENABLE_CUDA
    allocate_and_deallocate_pool("DEVICE", (1024*1024*1024), (1024*1024));
    allocate_and_deallocate_pool("UM", (1024*64), 1024);
    allocate_and_deallocate_pool("PINNED", (1024*16), 1024);
#endif
#ifdef UMPIRE_ENABLE_HIP
    allocate_and_deallocate_pool("DEVICE", (1024*1024*1024), (1024*1024));
    allocate_and_deallocate_pool("PINNED", (1024*16), 1024);
#endif

    return 0;
}

```

## 2.5 Introspection

When writing code to run on computers with a complex memory hierarchy, one of the most difficult things can be keeping track of where each pointer has been allocated. Umpire's introspection capability keeps track of this information, as well as other useful bits and pieces you might want to know.

The `umpire::ResourceManager` can be used to find the allocator associated with an address:

```
auto found_allocator = rm.getAllocator(data);
```

Once you have this, it's easy to query things like the name of the Allocator:

```
<< found_allocator.getName()
```

You can also find out the associated `umpire::Platform`, which can help you decide where to operate on this data:

```
<< static_cast<int>(found_allocator.getPlatform()) << std::endl;
```

You can also find out how big the allocation is, in case you forgot:

```
<< found_allocator.getSize(data) << std::endl;
```

Remember that these functions will work on any allocation made using an Allocator or `umpire::TypedAllocator`.

```

////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

int main(int, char**) {
    constexpr std::size_t SIZE = 1024;

    auto& rm = umpire::ResourceManager::getInstance();

    const std::string destinations[] = {

```

(continues on next page)

(continued from previous page)

```

    "HOST"
#if defined(UMPIRE_ENABLE_CUDA)
    , "DEVICE"
    , "UM"
    , "PINNED"
#endif
#if defined(UMPIRE_ENABLE_HIP)
    , "DEVICE"
    , "PINNED"
#endif
};

for (auto& destination : destinations) {
    auto allocator = rm.getAllocator(destination);
    double* data = static_cast<double*>(
        allocator.allocate(SIZE*sizeof(double)));

    std::cout << "Allocated " << (SIZE*sizeof(double)) << " bytes using the "
        << allocator.getName() << " allocator." << std::endl;

    auto found_allocator = rm.getAllocator(data);

    std::cout << "According to the ResourceManager, the Allocator used is "
        << found_allocator.getName()
        << ", which has the Platform "
        << static_cast<int>(found_allocator.getPlatform()) << std::endl;

    std::cout << "The size of the allocation is << "
        << found_allocator.getSize(data) << std::endl;

    allocator.deallocate(data);
}

return 0;
}

```

## 2.6 Typed Allocators

Sometimes, you might want to construct an allocator that allocates objects of a specific type. Umpire provides a `umpire::TypedAllocator` for this purpose. It can also be used with STL objects like `std::vector`.

A `umpire::TypedAllocator` is constructed from any existing `Allocator`, and provides the same interface as the normal `umpire::Allocator`. However, when you call `allocate`, this argument is the number of objects you want to allocate, not the total number of bytes:

```

umpire::TypedAllocator<double> double_allocator{alloc};

double* my_doubles = double_allocator.allocate(1024);

double_allocator.deallocate(my_doubles, 1024);

```

To use this allocator with an STL object like a vector, you need to pass the type as a template parameter for the vector, and also pass the allocator to the vector when you construct it:

```
std::vector< double, umpire::TypedAllocator<double> >
    my_vector(double_allocator);

my_vector.resize(100);
```

One thing to remember is that whatever allocator you use with an STL object, it must be compatible with the inner workings of that object. For example, if you try and use a “DEVICE”-based allocator it will fail, since the vector will try and construct each element. The CPU cannot access DEVICE memory in most systems, thus causing a segfault. Be careful!

```
////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
#include "umpire/ResourceManager.hpp"
#include "umpire/Allocator.hpp"

#include "umpire/TypedAllocator.hpp"

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();
    auto alloc = rm.getAllocator("HOST");

    umpire::TypedAllocator<double> double_allocator{alloc};

    double* my_doubles = double_allocator.allocate(1024);

    double_allocator.deallocate(my_doubles, 1024);

    std::vector< double, umpire::TypedAllocator<double> >
        my_vector(double_allocator);

    my_vector.resize(100);

    return 0;
}
```

## 2.7 Replay

Umpire provides a lightweight replay capability that can be used to investigate performance of particular allocation patterns and reproduce bugs.

## 2.7.1 Input Example

A log can be captured and stored as a JSON file, then used as input to the `replay` application (available under the `bin` directory). The `replay` program will read the replay log, and recreate the events that occurred as part of the run that generated the log.

The file `tut_replay.cpp` makes a `umpire::strategy::DynamicPool`:

```
auto pool = rm.makeAllocator<umpire::strategy::DynamicPool>(
    "pool",
    allocator);
```

This allocator is used to perform some randomly sized allocations, and later free them:

```
std::generate(allocations.begin(), allocations.end(),
             [&] () { return pool.allocate(random_number()); });
```

```
for (auto& ptr : allocations) pool.deallocate(ptr);
```

## 2.7.2 Running the Example

Running this program:

```
UMPIRE_REPLAY="On" ./bin/examples/tutorial/tut_replay > tut_replay_log.json
```

will write Umpire replay events to the file `tut_replay_log.json`. You can see that this file contains JSON formatted lines.

## 2.7.3 Replaying the session

Loading this file with the `replay` program will replay this sequence of `umpire::Allocator` creation, allocations, and deallocations:

```
./bin/replay -i ../tutorial/examples/tut_replay_log.json
```

We also have a tutorial for the C interface to Umpire. Complete example listings are available, and will be compiled if you have configured Umpire with `-DENABLE_C=On`.

The C tutorial assumes an understanding of C, and it would be useful to have some knowledge of C++ to understand how the C API maps to the native C++ classes that Umpire provides.

## 2.8 C API: Allocators

The fundamental concept for accessing memory through Umpire is an `umpire::Allocator`. In C, this means using the type `umpire_allocator`. There are corresponding functions that take an `umpire_allocator` and let you allocate and deallocate memory.

As with the native C++ interface, all allocators are accessed via the `umpire::ResourceManager`. In the C API, there is a corresponding `umpire_resourcemanager` type. To get an `umpire_allocator`:

```
umpire_resourcemanager rm;
umpire_resourcemanager_get_instance(&rm);

umpire_allocator allocator;
umpire_resourcemanager_get_allocator_by_name(&rm, "HOST", &allocator);
```

Once you have an `umpire_allocator`, you can use it to allocate and deallocate memory:

```
double* data = (double*) umpire_allocator_allocate(&allocator, SIZE*sizeof(double));

printf("Allocated %lu bytes using the %s allocator...", (SIZE*sizeof(double)),
↳umpire_allocator_get_name(&allocator));

umpire_allocator_deallocate(&allocator, data);
```

In the next section, we will see how to allocate memory in different places.

## 2.9 C API: Resources

Each computer system will have a number of distinct places in which the system will allow you to allocate memory. In Umpire’s world, these are *memory resources*. A memory resource can correspond to a hardware resource, but can also be used to identify memory with a particular characteristic, like “pinned” memory in a GPU system.

When you configure Umpire, it will create `umpire::resource::MemoryResource`s according to what is available on the system you are building for. For each resource, Umpire will create a default `umpire_allocator` that you can use. In the previous example, we were actually using an `umpire_allocator` created for the memory resource corresponding to the CPU memory.

The easiest way to identify resources is by name. The “HOST” resource is always available. In a system configured with NVIDIA GPUs, we also have resources that represent global GPU memory (“DEVICE”), unified memory that can be accessed by the CPU or GPU (“UM”) and host memory that can be accessed by the GPU (“PINNED”);

Umpire will create an `umpire_allocator` for each of these resources, and you can get them using the same `umpire_resourcemanager_get_allocator_by_name` call you saw in the previous example:

Note that every allocator supports the same calls, no matter which resource it is for, this means we can run the same code for all the resources available in the system:

As you can see, we can call this function with any valid resource name:

In the next example, we will learn how to move data between resources using operations.

## 2.10 C API: Pools

Frequently allocating and deallocating memory can be quite costly, especially when you are making large allocations or allocating on different memory resources. To mitigate this, Umpire provides allocation strategies that can be used to customize how data is obtained from the system.

In this example, we will look at creating a pool that can fulfill requests for allocations of any size. To create a new `umpire_allocator` using the pooling algorithm:

The two arguments are the size of the initial block that is allocated, and the minimum size of any future blocks. We have to provide a new name for the allocator, as well as the underlying `umpire_allocator` we wish to use to grab memory.



Once you have the allocator, you can allocate and deallocate memory as before, without needing to worry about the underlying algorithm used for the allocations:

This pool can be created with any valid underlying `umpire_allocator`.

Finally, we have a tutorial for Umpire's FORTRAN API. These examples will be compiled when configuring with `-DENABLE_FORTRAN=On`. The FORTRAN tutorial assumes an understanding of FORTRAN. Familiarity with the FORTRAN's ISO C bindings can be useful for understanding why the interface looks the way it does.

## 2.11 FORTRAN API: Allocators

The fundamental concept for accessing memory through Umpire is an `umpire:Allocator`. In FORTRAN, this means using the type `UmpireAllocator`. This type provides an `allocate_pointer` function to allocate raw memory, and a generic `allocate` procedure that takes an array pointer and an array of dimensions and will allocate the correct amount of memory.

As with the native C++ interface, all allocators are accessed via the `umpire::ResourceManager`. In the FORTRAN API, there is a corresponding `UmpireResourceManager` type. To get an `UmpireAllocator`:

```
rm = rm%get_instance()
allocator = rm%get_allocator_by_id(0)
```

In this example we fetch the allocator by id, using 0 means you will always get a host allocator. Once you have an `UmpireAllocator`, you can use it to allocate and deallocate memory:

```
call allocator%allocate(array, [ 10 ])

write(10,*) "Allocated array of size ", 10

call allocator%deallocate(array)
```

In this case, we allocate a one-dimensional array using the generic `allocate` function.



## ADVANCED CONFIGURATION

In addition to the normal options provided by CMake, Umpire uses some additional configuration arguments to control optional features and behavior. Each argument is a boolean option, and can be turned on or off:

```
-DENABLE_CUDA=Off
```

Here is a summary of the configuration options, their default value, and meaning:

Variable	Default	Meaning
ENABLE_CUDA	Off	Enable CUDA support
ENABLE_HIP	Off	Enable HIP support
ENABLE_HCC	Off	Enable HCC support
ENABLE_NUMA	Off	Enable NUMA support
ENABLE_STATISTICS	Off	Enable collection of memory statistics
ENABLE_TESTING	On	Build test executables
ENABLE_BENCHMARKS	On	Build benchmark programs
ENABLE_LOGGING	On	Enable Logging within Umpire
ENABLE_SLIC	Off	Enable SLIC logging
ENABLE_TOOLS	Off	Enable tools like replay
ENABLE_DOCS	Off	Build documentation (requires Sphinx and/or Doxygen)
ENABLE_C	Off	Build the C API
ENABLE_FORTRAN	Off	Build the Fortran API

These arguments are explained in more detail below:

- **ENABLE\_CUDA** This option enables support for NVIDIA GPUs using the CUDA programming model. If Umpire is built without CUDA, HCC, or HIP support, then only the `HOST` allocator is available for use.
- **ENABLE\_HIP** This option enables support for AMD GPUs using the ROCm stack and HIP programming model. If Umpire is built without CUDA, HCC, or HIP support, then only the `HOST` allocator is available for use.
- **ENABLE\_HCC** This option enables support for AMD GPUs using the ROCm stack and HCC programming model. If Umpire is built without CUDA, HCC, or HIP support, then only the `HOST` allocator is available for use.
- **ENABLE\_NUMA** This option enables support for NUMA. The `umpire::strategy::NumaPolicy` is available when built with this option, which may be used to locate the allocation to a specific node.
- **ENABLE\_STATISTICS** This option enables collection of memory statistics. If Umpire is built with this option, the Conduit library will also be built.
- **ENABLE\_TESTING** This option controls whether or not test executables will be built.
- **ENABLE\_BENCHMARKS** This option will build the benchmark programs used to test performance.

- `ENABLE_LOGGING` This option enables usage of Logging services for Umpire
- `ENABLE_SLIC` This option enables usage of logging services provided by SLIC.
- `ENABLE_TOOLS` Enable development tools for Umpire (replay, etc.)
- `ENABLE_DOCS` Build user documentation (with Sphinx) and code documentation (with Doxygen)
- `ENABLE_C` Build the C API, this allows accessing Umpire Allocators and the ResourceManager through a C interface.
- `ENABLE_FORTRAN` Build the Fortran API.

## UMPIRE COOKBOOK

This section provides a set of recipes that show you how to accomplish specific tasks using Umpire. The main focus is things that can be done by composing different parts of Umpire to achieve a particular use case.

Examples include being able to grow and shrink a pool, constructing Allocators that have introspection disabled for improved performance, and applying CUDA “memory advise” to all the allocations in a particular pool.

### 4.1 Growing and Shrinking a Pool

When sharing a pool between different parts of your application, or even between co-ordinating libraries in the same application, you might want to grow and shrink a pool on demand. By limiting the size of a pool using device memory, you leave more space on the GPU for “unified memory” to move data there.

The basic idea is to create a pool that allocates a block of your minimum size, and then allocate a single word from this pool to ensure the initial block is never freed:

```
auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>(
    "GPU_POOL",
    allocator,
    4ul * 1024ul * 1024ul * 1024ul + 1);
void* hold = pooled_allocator.allocate(64);
UMPIRE_USE_VAR(hold);
```

To increase the pool size you can preallocate a large chunk and then immediately free it. The pool will retain this memory for use by later allocations:

```
void* grow = pooled_allocator.allocate( 8ul * 1024ul * 1024ul * 1024ul );
pooled_allocator.deallocate(grow);
```

Assuming that there are no allocations left in the larger “chunk” of the pool, you can shrink the pool back down to the initial size by calling `umpire::Allocator::release()`:

```
pooled_allocator.release();
```

The complete example is included below:

```
////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
#include "umpire/strategy/DynamicPool.hpp"
```

(continues on next page)

```

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

#include "umpire/util/Macros.hpp"

#include <iostream>

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();

    auto allocator = rm.getAllocator("DEVICE");

    /*
     * Create a 4 Gb pool and reserve one word (to maintain alignment)
     */
    auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>(
        "GPU_POOL",
        allocator,
        4ul * 1024ul * 1024ul * 1024ul + 1);
    void* hold = pooled_allocator.allocate(64);
    UMPIRE_USE_VAR(hold);

    std::cout << "Pool has allocated " << pooled_allocator.getActualSize()
              << " bytes of memory. " << pooled_allocator.getCurrentSize()
              << " bytes are used" << std::endl;

    /*
     * Grow pool to ~12 by grabbing a 8Gb chunk
     */
    void* grow = pooled_allocator.allocate( 8ul * 1024ul * 1024ul * 1024ul );
    pooled_allocator.deallocate(grow);

    std::cout << "Pool has allocated " << pooled_allocator.getActualSize()
              << " bytes of memory. " << pooled_allocator.getCurrentSize()
              << " bytes are used" << std::endl;

    /*
     * Shrink pool back to ~4Gb
     */
    pooled_allocator.release();
    std::cout << "Pool has allocated " << pooled_allocator.getActualSize()
              << " bytes of memory. " << pooled_allocator.getCurrentSize()
              << " bytes are used" << std::endl;

    return 0;
}

```

## 4.2 Disable Introspection

If you know that you won't be using any of Umpire's introspection capabilities for allocations that come from a particular `umpire::Allocator`, you can turn off the introspection and avoid the overhead of tracking the associated metadata.

**Warning:** Disabling introspection means that allocations from this Allocator cannot be used for operations, or size and location queries.

In this recipe, we look at disabling introspection for a pool. To turn off introspection, you pass a boolean as the second template parameter to the `umpire::ResourceManager::makeAllocator()` method:

```
auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool, false>(
    "NO_INTROSPECTION_POOL",
    allocator);
```

Remember that disabling introspection will stop tracking the size of allocations made from the pool, so the `umpire::Allocator::getCurrentSize()` method will return 0:

```
<< " bytes of memory. " << pooled_allocator.getCurrentSize()
```

The complete example is included below:

```
////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
#include "umpire/strategy/DynamicPool.hpp"

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

#include "umpire/util/Macros.hpp"

#include <iostream>

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();

    auto allocator = rm.getAllocator("HOST");

    /*
     * Create a pool with introspection disabled (can improve performance)
     */
    auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool, false>(
        "NO_INTROSPECTION_POOL",
        allocator);

    void* data = pooled_allocator.allocate(1024);

    std::cout << "Pool has allocated " << pooled_allocator.getActualSize()
              << " bytes of memory. " << pooled_allocator.getCurrentSize()
              << " bytes are used" << std::endl;
```

(continues on next page)

(continued from previous page)

```
pooled_allocator.deallocate(data);

return 0;
}
```

### 4.3 Apply Memory Advice to a Pool

When using unified memory on systems with CUDA GPUs, various types of memory advice can be applied to modify how the CUDA runtime moves this memory around between the CPU and GPU. One type of advice that can be applied is “preferred location”, and you can specify where you want the preferred location of the memory to be. This can be useful for ensuring that the memory is kept on the GPU.

By creating a pool on top of an `umpire::strategy::AllocationAdvisor`, you can amortize the cost of applying memory advice:

```
auto preferred_location_allocator =
    rm.makeAllocator<umpire::strategy::AllocationAdvisor>(
        "preferred_location_host", allocator, "PREFERRED_LOCATION");

/*
 * Create a pool using the preferred_location_allocator. This makes all
 * allocations in the pool have the same preferred location, the GPU.
 */
auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>(
    "GPU_POOL",
    preferred_location_allocator);
```

The complete example is included below:

```
////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
#include "umpire/strategy/DynamicPool.hpp"
#include "umpire/strategy/AllocationAdvisor.hpp"

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

#include "umpire/util/Macros.hpp"

#include <iostream>

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();

    auto allocator = rm.getAllocator("UM");

    /*
     * Create an allocator that applied "PREFERRED_LOCATION" advice to set the
     * GPU as the preferred location.
     */
}
```

(continues on next page)



(continued from previous page)

```

*/
auto preferred_location_allocator =
    rm.makeAllocator<umpire::strategy::AllocationAdvisor>(
        "preferred_location_host", allocator, "PREFERRED_LOCATION");

/*
 * Create a pool using the preferred_location_allocator. This makes all
 * allocations in the pool have the same preferred location, the GPU.
 */
auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>(
    "GPU_POOL",
    preferred_location_allocator);

UMPIRE_USE_VAR(pooled_allocator);

return 0;
}

```

## 4.4 Apply Memory Advice with a Specific Device ID

When using unified memory on systems with CUDA GPUs, various types of memory advice can be applied to modify how the CUDA runtime moves this memory around between the CPU and GPU. When applying memory advice, a device ID can be used to specify which device the advice relates to. One type of advice that can be applied is “preferred location”, and you can specify where you want the preferred location of the memory to be. This can be useful for ensuring that the memory is kept on the GPU.

By passing a specific device id when constructing an `umpire::strategy::AllocationAdvisor`, you can ensure that the advice will be applied with respect to that device

```

auto preferred_location_allocator =
    rm.makeAllocator<umpire::strategy::AllocationAdvisor>(
        "preferred_location_device_2", allocator, "PREFERRED_LOCATION", device_id);

```

The complete example is included below:

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include "umpire/strategy/AllocationAdvisor.hpp"

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

#include "umpire/util/Exception.hpp"

#include <iostream>

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();

    auto allocator = rm.getAllocator("UM");

```

(continues on next page)

(continued from previous page)

```

/*
 * Create an allocator that applied "PREFERRED_LOCATION" advice to set a
 * specific GPU device as the preferred location.
 *
 * In this case, device #2.
 */
const int device_id = 2;

try {
    auto preferred_location_allocator =
        rm.makeAllocator<umpire::strategy::AllocationAdvisor>(
            "preferred_location_device_2", allocator, "PREFERRED_LOCATION", device_id);

    void* data = preferred_location_allocator.allocate(1024);

    preferred_location_allocator.deallocate(data);
} catch (umpire::util::Exception& e) {
    std::cout << "Couldn't create Allocator with device_id = " << device_id
              << std::endl;

    std::cout << e.message() << std::endl;
}

return 0;
}

```

## 4.5 Moving Host Data to Managed Memory

When using a system with NVIDIA GPUs, you may realize that some host data should be moved to unified memory in order to make it accessible by the GPU. You can do this with the `umpire::ResourceManager::move()` operation:

```
double* um_data = static_cast<double*>(rm.move(host_data, um_allocator));
```

The move operation will copy the data from host memory to unified memory, allocated using the provided `um_allocator`. The original allocation in host memory will be deallocated. The complete example is included below:

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

int main(int, char**) {
    constexpr std::size_t SIZE = 1024;

    auto& rm = umpire::ResourceManager::getInstance();
    auto allocator = rm.getAllocator("HOST");

```

(continues on next page)

(continued from previous page)

```

/*
 * Allocate host data
 */
double* host_data = static_cast<double*>(
    allocator.allocate(SIZE*sizeof(double));

/*
 * Move data to unified memory
 */
auto um_allocator = rm.getAllocator("UM");
double* um_data = static_cast<double*>(rm.move(host_data, um_allocator));

/*
 * Deallocate um_data, host_data is already deallocated by move operation.
 */
rm.deallocate(um_data);

return 0;
}

```

## 4.6 Improving DynamicPool Performance with a Coalesce Heuristic

As needed, the `umpire::strategy::DynamicPool` will continue to allocate blocks to satisfy allocation requests that cannot be satisfied by blocks currently in the pool it is managing. Under certain application-specific memory allocation patterns, fragmentation within the blocks or allocations that are for sizes greater than the size of the largest available block can cause the pool to grow too large. For example, a problematic allocation pattern is when an application makes several allocations of incrementing size where each allocation is larger than the previous block size allocated.

The `umpire::strategy::DynamicPool::coalesce()` method may be used to cause the `umpire::strategy::DynamicPool` to coalesce the releasable blocks into a single larger block. This is accomplished by: tallying the size of all blocks without allocations against them, releasing those blocks back to the memory resource, and creating a new block of the previously tallied size.

Applications may offer a heuristic function to the `umpire::strategy::DynamicPool` during instantiation that will return true whenever a pool reaches a specific threshold of releasable bytes (represented by completely free blocks) to the total size of the pool. The `DynamicPool` will call this heuristic function just before it returns from its `umpire::strategy::DynamicPool::deallocate()` method and when the function returns true, the `DynamicPool` will call the `umpire::strategy::DynamicPool::coalesce()` method.

The default heuristic of 100 will cause the `DynamicPool` to automatically coalesce when all of the bytes in the pool are releasable and there is more than one block in the pool.

A heuristic of 0 will cause the `DynamicPool` to never automatically coalesce.

Creation of the heuristic function is accomplished by:

```
//
```

The heuristic function is then provided as a parameter when the object is instantiated:

```

auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>(
    "HOST_POOL"
    , allocator
    , 1024ul

```

(continues on next page)

(continued from previous page)

```

        , 1024ul
        , 16

```

The complete example is included below:

```

////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
#include "umpire/strategy/DynamicPool.hpp"
#include "umpire/strategy/DynamicPoolHeuristic.hpp"

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

#include "umpire/util/Macros.hpp"

#include <iostream>

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();

    auto allocator = rm.getAllocator("HOST");

    //
    // Create a heuristic function that will return true to the DynamicPool
    // object when the threshold of releasable size to total size is 75%.
    //
    auto heuristic_function = umpire::strategy::heuristic_percent_releasable(75);

    //
    // Create a pool with an initial block size of 1 Kb and 1 Kb block size for
    // all subsequent allocations and with our previously created heuristic
    // function.
    //
    auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>(
        "HOST_POOL"
        , allocator
        , 1024ul
        , 1024ul
        , 16
        , heuristic_function);

    //
    // Obtain a pointer to our specifi DynamicPool instance in order to see the
    // DynamicPool-specific statistics
    //
    auto dynamic_pool = umpire::util::unwrap_allocator<umpire::strategy::DynamicPool>
    ↪(pooled_allocator);

    void* a[4];
    for (int i = 0; i < 4; ++i)
        a[i] = pooled_allocator.allocate(1024);

    for (int i = 0; i < 4; ++i) {

```

(continues on next page)

(continued from previous page)

```

pooled_allocator.deallocate(a[i]);
std::cout
  << "Pool has " << pooled_allocator.getActualSize() << " bytes of memory. "
  << pooled_allocator.getCurrentSize() << " bytes are used. "
  << dynamic_pool->getBlocksInPool() << " blocks are in the pool. "
  << dynamic_pool->getReleasableSize() << " bytes are releasable. "
  << std::endl;
}

return 0;
}

```

## 4.7 Move Allocations Between NUMA Nodes

When using NUMA (cache coherent or non uniform memory access) systems, there are different latencies to parts of the memory. From an application perspective, the memory looks the same, yet especially for high-performance computing it is advantageous to have finer control. `malloc()` attempts to allocate memory close to your node, but it can make no guarantees. Therefore, Linux provides both a process-level interface for setting NUMA policies with the system utility `numactl`, and a fine-grained interface with `libnuma`. These interfaces work on ranges of memory in multiples of the page size, which is the length or unit of address space loaded into a processor cache at once.

A page range may be bound to a NUMA node using the `umpire::strategy::NumaPolicy`. It can therefore also be moved between NUMA nodes using the `umpire::ResourceManager::move()` with a different allocator. The power of using such an abstraction is that the NUMA node can be associated with a device, in which case the memory is moved to, for example, GPU memory.

In this recipe we create an allocation bound to a NUMA node, and move it to another NUMA node.

The complete example is included below:

```

////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

#include "umpire/strategy/NumaPolicy.hpp"

#include "umpire/util/numa.hpp"
#include "umpire/util/Macros.hpp"

#include <iostream>

#ifdef UMPIRE_ENABLE_CUDA
#include <cuda_runtime_api.h>
#endif

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();

    const std::size_t alloc_size = 5 * umpire::get_page_size();

```

(continues on next page)

(continued from previous page)

```

// Get a list of the host NUMA nodes (e.g. one per socket)
auto host_nodes = umpire::numa::get_host_nodes();

if (host_nodes.size() < 1) {
    UMPIRE_ERROR("No NUMA nodes detected");
}

// Create an allocator on the first NUMA node
auto host_src_alloc = rm.makeAllocator<umpire::strategy::NumaPolicy>(
    "host_numa_src_alloc", rm.getAllocator("HOST"), host_nodes[0]);

// Create an allocation on that node
void* src_ptr = host_src_alloc.allocate(alloc_size);

if (host_nodes.size() > 1) {
    // Create an allocator on another host NUMA node.
    auto host_dst_alloc = rm.makeAllocator<umpire::strategy::NumaPolicy>(
        "host_numa_dst_alloc", rm.getAllocator("HOST"), host_nodes[1]);

    // Move the memory
    void* dst_ptr = rm.move(src_ptr, host_dst_alloc);

    // The pointer shouldn't change even though the memory location changes
    if (dst_ptr != src_ptr) {
        UMPIRE_ERROR("Pointers should match");
    }

    // Touch it
    rm.memset(dst_ptr, 0);

    // Verify NUMA node
    if (umpire::numa::get_location(dst_ptr) != host_nodes[1]) {
        UMPIRE_ERROR("Move was unsuccessful");
    }
}

#ifdef UMPIRE_ENABLE_DEVICE
// Get a list of the device nodes
auto device_nodes = umpire::numa::get_device_nodes();

if (device_nodes.size() > 0) {
    // Create an allocator on the first device NUMA node. Note that
    // this still requires using the "HOST" allocator. The allocations
    // are moved after the address space is reserved.
    auto device_alloc = rm.makeAllocator<umpire::strategy::NumaPolicy>(
        "device_numa_src_alloc", rm.getAllocator("HOST"), device_nodes[0]);

    // Move the memory
    void* dst_ptr = rm.move(src_ptr, device_alloc);

    // The pointer shouldn't change even though the memory location changes
    if (dst_ptr != src_ptr) {
        UMPIRE_ERROR("Pointers should match");
    }

    // Touch it -- this currently uses the host memset operation (thus, copying the
    ↪memory back)

```

(continues on next page)

(continued from previous page)

```

rm.memset(dst_ptr, 0);

// Verify NUMA node
if (umpire::numa::get_location(dst_ptr) != device_nodes[0]) {
    UMPIRE_ERROR("Move was unsuccessful");
}
}
#endif

// Clean up by deallocating from the original allocator, since the
// allocation record is still associated with that allocator
host_src_alloc.deallocate(src_ptr);

return 0;
}

```

## 4.8 Determining the Largest Block of Available Memory in Pool

The `umpire::strategy::DynamicPool` provides a `umpire::strategy::DynamicPool::getLargestAvailableBlock` that may be used to determine the size of the largest block currently available for allocation within the pool. To call this function, you must get the pointer to the `umpire::strategy::AllocationStrategy` from the `umpire::Allocator`:

```

auto& rm = umpire::ResourceManager::getInstance();

auto pool = rm.makeAllocator<umpire::strategy::DynamicPool>(
    "pool", rm.getAllocator("HOST"));

auto dynamic_pool =
    umpire::util::unwrap_allocator<umpire::strategy::DynamicPool>(pool);

```

Once you have the pointer to the appropriate strategy, you can call the function:

```

std::cout
    << "Largest available block in pool is "
    << dynamic_pool->getLargestAvailableBlock() << " bytes in size"
    << std::endl;

```

The complete example is included below:

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include "umpire/strategy/DynamicPool.hpp"

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

#include "umpire/util/Exception.hpp"
#include "umpire/util/wrap_allocator.hpp"

```

(continues on next page)

(continued from previous page)

```

#include <iostream>

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();

    auto pool = rm.makeAllocator<umpire::strategy::DynamicPool>(
        "pool", rm.getAllocator("HOST"));

    auto dynamic_pool =
        umpire::util::unwrap_allocator<umpire::strategy::DynamicPool>(pool);

    if ( dynamic_pool == nullptr ) {
        UMPIRE_ERROR(pool.getName() << " is not a DynamicPool");
    }

    auto ptr = pool.allocate(1024);

    std::cout
        << "Largest available block in pool is "
        << dynamic_pool->getLargestAvailableBlock() << " bytes in size"
        << std::endl;

    pool.deallocate(ptr);

    return 0;
}

```

## 4.9 Coalescing Pool Memory

The `umpire::strategy::DynamicPool` provides a `umpire::strategy::DynamicPool::coalesce()` that can be used to release unused memory and allocate a single large block that will be able to satisfy allocations up to the previously observed high-watermark. To call this function, you must get the pointer to the `umpire::strategy::AllocationStrategy` from the `umpire::Allocator`:

```

auto dynamic_pool =
    umpire::util::unwrap_allocator<umpire::strategy::DynamicPool>(pool);

```

Once you have the pointer to the appropriate strategy, you can call the function:

```
dynamic_pool->coalesce();
```

The complete example is included below:

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include "umpire/strategy/DynamicPool.hpp"
#include "umpire/strategy/AllocationTracker.hpp"

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

```

(continues on next page)



(continued from previous page)

```

#include "umpire/util/Exception.hpp"
#include "umpire/util/wrap_allocator.hpp"

#include <iostream>

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();

    auto pool = rm.makeAllocator<umpire::strategy::DynamicPool>(
        "pool", rm.getAllocator("HOST"));

    auto dynamic_pool =
        umpire::util::unwrap_allocator<umpire::strategy::DynamicPool>(pool);

    if (dynamic_pool) {
        dynamic_pool->coalesce();
    } else {
        UMPIRE_ERROR(pool.getName() << " is not a DynamicPool, cannot coalesce!");
    }

    return 0;
}

```

## 4.10 Building a Pinned Memory Pool in FORTRAN

In this recipe, we show you how to build a pool in pinned memory using Umpire’s FORTRAN API. These kinds of pools can be useful for allocating buffers to be used in communication routines in various scientific applications.

Building the pool takes two steps: 1) getting a base “PINNED” allocator, and 2) creating the pool:

```

base_allocator = rm%get_allocator_by_name("PINNED")

pinned_pool = rm%make_allocator_pool("PINNED_POOL", &
                                     base_allocator, &
                                     512_8*1024_8, &
                                     1024_8)

```

The complete example is included below:

```

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
! project contributors. See the COPYRIGHT file for details.
!
! SPDX-License-Identifier: (MIT)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

program umpire_f_pinned_pool
    use umpire_mod
    implicit none
    logical ok

    integer(C_INT), pointer, dimension(:) :: array(:)
    type(UmpireAllocator) base_allocator

```

(continues on next page)

(continued from previous page)

```

type(UmpireAllocator) pinned_pool
type(UmpireResourceManager) rm

rm = rm%get_instance()
base_allocator = rm%get_allocator_by_name("PINNED")

pinned_pool = rm%make_allocator_pool("PINNED_POOL", &
                                     base_allocator, &
                                     512_8*1024_8, &
                                     1024_8)

call pinned_pool%allocate(array, [10])
end program umpire_f_pinned_pool

```

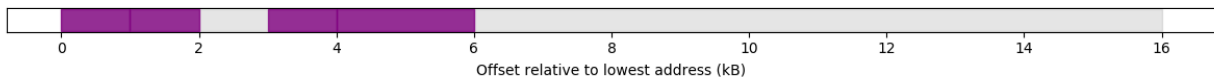
## 4.11 Visualizing Allocators

The python script *plot\_allocations.py* is included with Umpire to plot allocations. This script uses series of three arguments: an output file with allocation records, a color, and an alpha (transparency) value *0.0-1.0*. Although these could be used to plot records from a single allocator, 3 arguments, it can also be used to overlay multiple allocators, by passing 3n arguments after the script name. In this cookbook we use this feature to visualize a pooled allocator.

The cookbook generates two files, *allocator.log* and *pooled\_allocator.log*, that contain the allocation records from the underlying allocator and the pool. These can then be plotted using a command similar to the following:

```
tools/plot_allocations allocator.log gray 0.2 pooled_allocator.log purple 0.8
```

That script uses Python and Matplotlib to generate the following image



The complete example is included below:

```

////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////
#include "umpire/Umpire.hpp"
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/DynamicPool.hpp"

#include <fstream>

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();

    auto allocator = rm.getAllocator("HOST");
    auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>("HOST_POOL",
                                                                              allocator,
                                                                              1024 * 16);

```

(continues on next page)

(continued from previous page)

```

void* a[4];
for (int i = 0; i < 4; ++i) a[i] = pooled_allocator.allocate(1024);

// Create fragmentation
pooled_allocator.deallocate(a[2]);
a[2] = pooled_allocator.allocate(1024 * 2);

// Output the records from the underlying host allocator
{
    std::ofstream out("allocator.log");
    umpire::print_allocator_records(allocator, out);
    out.close();
}

// Output the records from the pooled allocator
{
    std::ofstream out("pooled_allocator.log");
    umpire::print_allocator_records(pooled_allocator, out);
    out.close();
}

for (int i = 0; i < 4; ++i) pooled_allocator.deallocate(a[i]);

// Visualize this using the python script. Example usage:
// tools/analysis/plot_allocations allocator.log gray 0.2 pooled_allocator.log_
→purple 0.8

return 0;
}

```

## 4.12 Mixed Pool Creation and Algorithm Basics

This recipe shows how to create a default mixed pool, and one that might be tailored to a specific application's needs. Mixed pools allocate in an array of `umpire::strategy::FixedPool` for small allocations, because these have simpler bookkeeping and are very fast, and a `umpire::strategy::DynamicPool` for larger allocations.

The class `umpire::strategy::MixedPool` uses a generic choice of `umpire::strategy::FixedPool` of size 256 bytes to 4MB in increments of powers of 2, while `umpire::strategy::MixedPoolImpl` has template arguments that select the first, power of 2 increment, and last fixed pool size.

The complete example is included below:

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include "umpire/strategy/MixedPool.hpp"

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

#include <iostream>

```

(continues on next page)

```

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();
    auto allocator = rm.getAllocator("HOST");

    /*
     * Create a default mixed pool.
     */
    auto default_mixed_allocator = rm.makeAllocator<umpire::strategy::MixedPool>(
        "default_mixed_pool", allocator);

    UMPIRE_USE_VAR(default_mixed_allocator);

    /*
     * Create a mixed pool using fixed pool bins of size 2^8 = 256 Bytes
     * to 2^14 = 16 kB in increments of 5x, where each individual fixed
     * pool is kept under 4MB in size to begin.
     */
    auto custom_mixed_allocator = rm.makeAllocator<umpire::strategy::MixedPool>(
        "custom_mixed_pool", allocator, 256, 16*1024, 4*1024*1024, 5);

    /*
     * Although this calls for only 4*4=16 bytes, this allocation will
     * come from the smallest fixed pool, thus ptr will actually be the
     * first address in a range of 256 bytes.
     */
    void *ptr1 = custom_mixed_allocator.allocate(4 * sizeof(int));

    /*
     * This is too beyond the range of the fixed pools, and therefore is
     * allocated from a dynamic pool. The range of address space
     * reserved will be exactly what was requested by the allocate()
     * method.
     */
    void *ptr2 = custom_mixed_allocator.allocate(1 << 18);

    /*
     * Clean up
     */
    custom_mixed_allocator.deallocate(ptr1);
    custom_mixed_allocator.deallocate(ptr2);

    return 0;
}

```

## 4.13 Thread Safe Allocator

If you want thread-safe access to allocations that come from a particular `umpire::Allocator`, you can create an instance of a `umpire::strategy::ThreadSafeAllocator` object that will synchronize access to it.

In this recipe, we look at creating a `umpire::strategy::ThreadSafeAllocator` for an `umpire::strategy::DynamicPool` object:

```

auto& rm = umpire::ResourceManager::getInstance();

auto pool = rm.makeAllocator<umpire::strategy::DynamicPool>(
    "pool", rm.getAllocator("HOST"));

auto thread_safe_pool =
    rm.makeAllocator<umpire::strategy::ThreadSafeAllocator>
    ("thread_safe_pool", pool);

```

The complete example is included below:

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-19, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include "umpire/strategy/DynamicPool.hpp"
#include "umpire/strategy/ThreadSafeAllocator.hpp"

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

int main(int, char**) {
    auto& rm = umpire::ResourceManager::getInstance();

    auto pool = rm.makeAllocator<umpire::strategy::DynamicPool>(
        "pool", rm.getAllocator("HOST"));

    auto thread_safe_pool =
        rm.makeAllocator<umpire::strategy::ThreadSafeAllocator>
        ("thread_safe_pool", pool);

    auto allocation = thread_safe_pool.allocate(256);
    thread_safe_pool.deallocate(allocation);

    return 0;
}

```



## 5.1 Allocators

Allocators are the fundamental object used to allocate and deallocate memory using Umpire.

### **class Allocator**

Provides a unified interface to allocate and free data.

An *Allocator* encapsulates all the details of how and where allocations will be made, and can also be used to introspect the memory resource. *Allocator* objects do not return typed allocations, so the pointer returned from the allocate method must be cast to the relevant type.

See *TypedAllocator*

template<typename T>

### **class TypedAllocator**

*Allocator* for objects of type T.

This class is an adaptor that allows using an *Allocator* to allocate objects of type T. You can use this class as an allocator for STL containers like `std::vector`.

## 5.2 Strategies

Strategies are used in Umpire to allow custom algorithms to be applied when allocating memory. These strategies can do anything, from providing different pooling methods to speed up allocations to applying different operations to every allocation. Strategies can be composed to combine their functionality, allowing flexible and reusable implementations of different components.

### **class AllocationStrategy**

*AllocationStrategy* provides a unified interface to all classes that can be used to allocate and free data.

Subclassed by *umpire::resource::MemoryResource*, *umpire::strategy::AllocationAdvisor*, *umpire::strategy::AllocationPrefetcher*, *umpire::strategy::AllocationTracker*, *umpire::strategy::DynamicPoolList*, *umpire::strategy::DynamicPoolMap*, *umpire::strategy::FixedPool*, *umpire::strategy::MixedPool*, *umpire::strategy::MonotonicAllocationStrategy*, *umpire::strategy::NamedAllocationStrategy*, *umpire::strategy::NumaPolicy*, *umpire::strategy::SizeLimiter*, *umpire::strategy::SlotPool*, *umpire::strategy::ThreadSafeAllocator*, *umpire::strategy::ZeroByteHandler*

## 5.2.1 Provided Strategies

**class AllocationAdvisor** : public umpire::strategy::AllocationStrategy  
Applies the given MemoryOperation to every allocation.

This AllocationStrategy is designed to be used with the following operations:

- *op::CudaAdviseAccessedByOperation*
- *op::CudaAdvisePreferredLocationOperation*
- *op::CudaAdviseReadMostlyOperation*

Using this AllocationStrategy when combined with a pool like DynamicPool is a good way to mitigate the overhead of applying the memory advice.

**Warning:** doxygenclass: Cannot find class “umpire::strategy::DynamicPool” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

**class FixedPool** : public umpire::strategy::AllocationStrategy  
Pool for fixed size allocations.

This AllocationStrategy provides an efficient pool for fixed size allocations, and used to quickly allocate and deallocate objects.

**class MonotonicAllocationStrategy** : public umpire::strategy::AllocationStrategy

**class SlotPool** : public umpire::strategy::AllocationStrategy

**class ThreadSafeAllocator** : public umpire::strategy::AllocationStrategy  
Make an Allocator thread safe.

Using this AllocationStrategy will make the provided allocator thread-safe by synchronizing access to the allocators interface.

## 5.3 Operations

Operations provide an abstract interface to modifying and moving data between Umpire :class: `umpire::Allocator`s.

### 5.3.1 Provided Operations

namespace op

**class CudaCopyFromOperation** : public umpire::op::MemoryOperation  
*#include <umpire/op/CudaCopyFromOperation.hpp>* Copy operation to move data from a NVIDIA GPU to CPU memory.

**class CudaCopyOperation** : public umpire::op::MemoryOperation  
*#include <umpire/op/CudaCopyOperation.hpp>* Copy operation to move data between two GPU addresses.

**class CudaCopyToOperation** : public umpire::op::MemoryOperation  
*#include <umpire/op/CudaCopyToOperation.hpp>* Copy operation to move data from CPU to NVIDIA GPU memory.



```

class CudaMemsetOperation : public umpire::op::MemoryOperation
    #include <umpire/op/CudaMemsetOperation.hpp> Memset on NVIDIA device memory.

class GenericReallocateOperation : public umpire::op::MemoryOperation
    #include <umpire/op/GenericReallocateOperation.hpp> Generic reallocate operation to work on any current_ptr location.

class HipCopyFromOperation : public umpire::op::MemoryOperation
    #include <umpire/op/HipCopyFromOperation.hpp> Copy operation to move data from a AMD GPU to CPU memory.

class HipCopyOperation : public umpire::op::MemoryOperation
    #include <umpire/op/HipCopyOperation.hpp> Copy operation to move data between two GPU addresses.

class HipCopyToOperation : public umpire::op::MemoryOperation
    #include <umpire/op/HipCopyToOperation.hpp> Copy operation to move data from CPU to AMD GPU memory.

class HipMemsetOperation : public umpire::op::MemoryOperation
    #include <umpire/op/HipMemsetOperation.hpp> Memset on AMD device memory.

class HostCopyOperation : public umpire::op::MemoryOperation
    #include <umpire/op/HostCopyOperation.hpp> Copy memory between two allocations in CPU memory.

class HostMemsetOperation : public umpire::op::MemoryOperation
    #include <umpire/op/HostMemsetOperation.hpp> Memset an allocation in CPU memory.

class HostReallocateOperation : public umpire::op::MemoryOperation
    #include <umpire/op/HostReallocateOperation.hpp> Reallocate data in CPU memory.

class MemoryOperation
    #include <umpire/op/MemoryOperation.hpp> Base class of an operation on memory.

```

Neither the transform or apply methods are pure virtual, so inheriting classes only need overload the appropriate method. However, both methods will throw an error if called.

Subclassed by *umpire::op::CudaAdviseAccessedByOperation*, *umpire::op::CudaAdvisePreferredLocationOperation*, *umpire::op::CudaAdviseReadMostlyOperation*, *umpire::op::CudaAdviseUnsetAccessedByOperation*, *umpire::op::CudaAdviseUnsetPreferredLocationOperation*, *umpire::op::CudaAdviseUnsetReadMostlyOperation*, *umpire::op::CudaCopyFromOperation*, *umpire::op::CudaCopyOperation*, *umpire::op::CudaCopyToOperation*, *umpire::op::CudaMemPrefetchOperation*, *umpire::op::CudaMemsetOperation*, *umpire::op::GenericReallocateOperation*, *umpire::op::HipCopyFromOperation*, *umpire::op::HipCopyOperation*, *umpire::op::HipCopyToOperation*, *umpire::op::HipMemsetOperation*, *umpire::op::HostCopyOperation*, *umpire::op::HostMemsetOperation*, *umpire::op::HostReallocateOperation*, *umpire::op::NumaMoveOperation*, *umpire::op::RocmCopyOperation*, *umpire::op::RocmMemsetOperation*

```

class MemoryOperationRegistry
    #include <umpire/op/MemoryOperationRegistry.hpp> The MemoryOperationRegistry serves as a registry for MemoryOperation objects. It is a singleton class, typically accessed through the ResourceManager.

```

The *MemoryOperationRegistry* class provides lookup mechanisms allowing searching for the appropriate *MemoryOperation* to be applied to allocations made with particular *AllocationStrategy* objects.

MemoryOperations provided by Umpire are registered with the *MemoryOperationRegistry* when it is constructed. Additional MemoryOperations can be registered later using the registerOperation method.

The following operations are pre-registered for all *AllocationStrategy* pairs:

- "COPY"

- "MEMSET"
- "REALLOCATE"

See *MemoryOperation*

See *AllocationStrategy*

```
class NumaMoveOperation : public umpire::op::MemoryOperation
    #include <umpire/op/NumaMoveOperation.hpp> Relocate a pointer to a different NUMA node.
```

```
class RocmCopyOperation : public umpire::op::MemoryOperation
    #include <umpire/op/RocmCopyOperation.hpp> Copy operation to move data between two GPU ad-
    dresses.
```

```
class RocmMemsetOperation : public umpire::op::MemoryOperation
    #include <umpire/op/RocmMemsetOperation.hpp> Memset an allocation in CPU memory.
```

## 5.4 Logging and Replay of Umpire Events

### 5.4.1 Logging

When debugging memory operation problems, it is sometimes helpful to enable Umpire's logging facility. The logging functionality is enabled for default builds unless `-DENABLE_LOGGING='Off'` has been specified in which case it is disabled.

If Umpire logging is enabled, it may be controlled by setting the `UMPIRE_LOG_LEVEL` environment variable to Error, Warning, Info, or Debug. The Debug value is the most verbose.

When `UMPIRE_LOG_LEVEL` has been set, events will be logged to the standard output.

### 5.4.2 Replay

Umpire provides a lightweight replay capability that can be used to investigate performance of particular allocation patterns and reproduce bugs. By running an executable that uses Umpire with the environment variable `UMPIRE_REPLAY` set to On, Umpire will emit information for the following Umpire events:

- **version** `umpire::get_major_version()`, `umpire::get_minor_version()`, and `umpire::get_patch_version()`
- **makeMemoryResource** `umpire::resource::MemoryResourceRegistry::makeMemoryResource()`
- **makeAllocator** `umpire::ResourceManager::makeAllocator()`
- **allocate** `umpire::Allocator::allocate()`
- **deallocate** `umpire::Allocator::deallocate()`

### 5.4.3 Running with Replay

To enable Umpire replay, one may execute as follows:

```
UMPIRE_REPLAY="On" ./my_umpire_using_program > replay_log.json
```

will write Umpire replay events to the file `replay_log.json` that will contain the following kinds of information:

### 5.4.4 Interpreting Results - Version Event

The first event captured is the **version** event which shows the version information as follows:

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052211435757, "event": "version",
  ↪ "payload": { "major":0, "minor":3, "patch":3 } }
```

Each line contains the following set of common elements:

**kind** Always set to `replay`

**uid** This is the MPI rank of the process generating the event for mpi programs or the PID for non-mpi.

**timestamp** Set to the time when the event occurred.

**event** Set to one of: `version`, `makeMemoryResource`, `makeAllocator`, `allocate`, or `deallocate`

**payload** Optional and varies upon event type

**result** Optional and varies upon event type

As can be seen, the *major*, *minor*, and *patch* version numbers are captured within the *payload* for this event.

### 5.4.5 makeMemoryResource Event

Next you will see events for the creation of the default memory resources provided by Umpire with the **makeMemoryResource** event:

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052211477678, "event":
  ↪ "makeMemoryResource", "payload": { "name": "HOST" }, "result": "0x101626b0" }
{ "kind":"replay", "uid":27494, "timestamp":1558388052471684134, "event":
  ↪ "makeMemoryResource", "payload": { "name": "DEVICE" }, "result": "0x101d79a0" }
{ "kind":"replay", "uid":27494, "timestamp":1558388052471698804, "event":
  ↪ "makeMemoryResource", "payload": { "name": "PINNED" }, "result": "0x101d7a50" }
{ "kind":"replay", "uid":27494, "timestamp":1558388052472972935, "event":
  ↪ "makeMemoryResource", "payload": { "name": "UM" }, "result": "0x101d7b00" }
{ "kind":"replay", "uid":27494, "timestamp":1558388052595814979, "event":
  ↪ "makeMemoryResource", "payload": { "name": "DEVICE_CONST" }, "result": "0x101d7bb0"
  ↪ }
```

The *payload* shows that a memory resource was created for *HOST*, *DEVICE*, *PINNED*, *UM*, and *DEVICE\_CONST* respectively. The *result* is a reference to the object that was created within Umpire for that resource.

## 5.4.6 makeAllocator Event

The **makeAllocator** event occurs whenever a new allocator instance is being created. Each call to *makeAllocator* will generate a pair of JSON lines. The first line will show the intent of the call and the second line will show both the intent and the result. This is because the *makeAllocator* call can fail and keeping both the intent and result allows us to reproduce this failure later.

```
umpire::Allocator:
```

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052595864262, "event":  
↪"makeAllocator", "payload": { "type":"umpire::strategy::DynamicPool", "with_  
↪introspection":true, "allocator_name":"pool", "args": [ "HOST" ] } }  
{ "kind":"replay", "uid":27494, "timestamp":1558388052595903505, "event":  
↪"makeAllocator", "payload": { "type":"umpire::strategy::DynamicPool", "with_  
↪introspection":true, "allocator_name":"pool", "args": [ "HOST" ] }, "result": {  
↪"allocator_ref":"0x108a8730" } }
```

The *payload* shows how the allocator was constructed. The *result* shows the reference to the allocated object.

## 5.4.7 allocate Event

Like the **makeAllocator** event, the **allocate** event is captured as an intention/result pair so that an error may be replayed in the event that there is an allocation failure.

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052595911583, "event": "allocate",  
↪"payload": { "allocator_ref": "0x108a8730", "size": 0 } }  
{ "kind":"replay", "uid":27494, "timestamp":1558388052595934822, "event": "allocate",  
↪"payload": { "allocator_ref": "0x108a8730", "size": 0 }, "result": { "memory_ptr":  
↪"0x200040000010" } }
```

The *payload* shows the object reference of the allocator and the size of the allocation request. The *result* shows the pointer to the memory allocated.

## 5.4.8 deallocate Event

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052596358577, "event": "deallocate  
↪", "payload": { "allocator_ref": "0x108a8730", "memory_ptr": "0x200040000010" } }
```

The *payload* shows the reference to the allocator object and the pointer to the allocated memory that is to be freed.

## 5.4.9 Replaying the session

Loading this file with the *replay* program will replay this sequence of *umpire::Allocator* creation, allocations, and deallocations:

```
./bin/replay -i replay_log.json
```

## 5.5 File I/O

Umpire provides support for writing files containing log and replay data, rather than directing this output to stdout. When logging or replay are enabled, the following environment variables can be used to determine where the output is written:

`UMPIRE_OUTPUT_DIR` . Directory to write log and replay files  
`UMPIRE_OUTPUT_BASENAME` `umpire`  
    Basename of logging and relpay files

The values of these variables are used to construct unique filenames for output. The extension `.log` is used for logging output, and `.replay` for replay output. The filenames additionally contain three integers, one corresponding to the rank of the process, one corresponding to the process ID, and one that is used to make multiple files with the same basename and rank unique. This ensures that multiple runs with the same IO configuration do not overwrite files.

The format of the filenames is:

```
<UMPIRE_OUTPUT_BASENAME>.<RANK>.<PID>.<UID>.<log|replay>
```

If Umpire is compiled without MPI support, then rank will always be 0.



## 6.1 Class Hierarchy

## 6.2 File Hierarchy

## 6.3 Full API

### 6.3.1 Namespaces

#### Namespace `genumpiresplicer`

##### Contents

- *Functions*
- *Variables*

##### Functions

- *Function `genumpiresplicer::gen_bounds`*
- *Function `genumpiresplicer::gen_fortran`*
- *Function `genumpiresplicer::gen_methods`*

##### Variables

- *Variable `genumpiresplicer::maxdims`*
- *Variable `genumpiresplicer::types`*

## Namespace `iso_c_binding`

## Namespace `std`

STL namespace.

## Namespace `umpire`

### Contents

- *Namespaces*
- *Classes*
- *Enums*
- *Functions*
- *Variables*

## Namespaces

- *Namespace `umpire::alloc`*
- *Namespace `umpire::numa`*
- *Namespace `umpire::op`*
- *Namespace `umpire::resource`*
- *Namespace `umpire::strategy`*
- *Namespace `umpire::util`*

## Classes

- *Struct `MemoryResourceTraits`*
- *Class `Allocator`*
- *Class `DeviceAllocator`*
- *Class `Replay`*
- *Class `ResourceManager`*
- *Template Class `TypedAllocator`*



## Enums

- *Enum Platform*

## Functions

- *Function `umpire::cpu_vendor_type`*
- *Function `umpire::error`*
- *Function `umpire::finalize`*
- *Function `umpire::free`*
- *Function `umpire::get_allocator_records`*
- *Function `umpire::get_major_version`*
- *Function `umpire::get_minor_version`*
- *Function `umpire::get_page_size`*
- *Function `umpire::get_patch_version`*
- *Function `umpire::get_rc_version`*
- *Function `umpire::initialize`*
- *Function `umpire::log`*
- *Function `umpire::malloc`*
- *Function `umpire::operator<<(std::ostream&, const Allocator&)`*
- *Function `umpire::operator<<(std::ostream&, umpire::Allocator&)`*
- *Function `umpire::operator<<(std::ostream&, umpire::strategy::DynamicPoolMap::CoalesceHeuristic&)`*
- *Function `umpire::operator<<(std::ostream&, umpire::strategy::DynamicPoolList::CoalesceHeuristic&)`*
- *Function `umpire::print_allocator_records`*
- *Function `umpire::replay`*

## Variables

- *Variable `umpire::env_name`*

## Namespace `umpire::alloc`

### Contents

- *Classes*

## Classes

- *Struct AmAllocAllocator*
- *Struct AmPinnedAllocator*
- *Struct CudaMallocAllocator*
- *Struct CudaMallocManagedAllocator*
- *Struct CudaPinnedAllocator*
- *Struct HipMallocAllocator*
- *Struct HipPinnedAllocator*
- *Struct MallocAllocator*
- *Struct PosixMemalignAllocator*

## Namespace `umpire::numa`

### Contents

- *Functions*

## Functions

- *Function `umpire::numa::get_allocatable_nodes`*
- *Function `umpire::numa::get_device_nodes`*
- *Function `umpire::numa::get_host_nodes`*
- *Function `umpire::numa::get_location`*
- *Function `umpire::numa::move_to_node`*
- *Function `umpire::numa::preferred_node`*

## Namespace `umpire::op`

### Contents

- *Classes*

## Classes

- *Struct pair\_hash*
- *Class CudaAdviseAccessedByOperation*
- *Class CudaAdvisePreferredLocationOperation*
- *Class CudaAdviseReadMostlyOperation*
- *Class CudaAdviseUnsetAccessedByOperation*
- *Class CudaAdviseUnsetPreferredLocationOperation*
- *Class CudaAdviseUnsetReadMostlyOperation*
- *Class CudaCopyFromOperation*
- *Class CudaCopyOperation*
- *Class CudaCopyToOperation*
- *Class CudaMemPrefetchOperation*
- *Class CudaMemsetOperation*
- *Class GenericReallocateOperation*
- *Class HipCopyFromOperation*
- *Class HipCopyOperation*
- *Class HipCopyToOperation*
- *Class HipMemsetOperation*
- *Class HostCopyOperation*
- *Class HostMemsetOperation*
- *Class HostReallocateOperation*
- *Class MemoryOperation*
- *Class MemoryOperationRegistry*
- *Class NumaMoveOperation*
- *Class RocmCopyOperation*
- *Class RocmMemsetOperation*

## Namespace `umpire::resource`

### Contents

- *Classes*
- *Enums*

## Classes

- *Struct* `MemoryResourceTypeHash`
- *Class* `CudaConstantMemoryResource`
- *Class* `CudaConstantMemoryResourceFactory`
- *Class* `CudaDeviceResourceFactory`
- *Class* `CudaPinnedMemoryResourceFactory`
- *Class* `CudaUnifiedMemoryResourceFactory`
- *Template Class* `DefaultMemoryResource`
- *Class* `HipConstantMemoryResource`
- *Class* `HipConstantMemoryResourceFactory`
- *Class* `HipDeviceResourceFactory`
- *Class* `HipPinnedMemoryResourceFactory`
- *Class* `HostResourceFactory`
- *Class* `MemoryResource`
- *Class* `MemoryResourceFactory`
- *Class* `MemoryResourceRegistry`
- *Class* `NullMemoryResource`
- *Class* `NullMemoryResourceFactory`
- *Class* `RocmDeviceResourceFactory`
- *Class* `RocmPinnedMemoryResourceFactory`

## Enums

- *Enum* `MemoryResourceType`

## Namespace `umpire::strategy`

### Contents

- *Namespaces*
- *Classes*
- *Functions*
- *Typedefs*
- *Variables*

## Namespaces

- *Namespace* `umpire::strategy::mixins`

## Classes

- *Struct* `FixedPool::Pool`
- *Class* `AllocationAdvisor`
- *Class* `AllocationPrefetcher`
- *Class* `AllocationStrategy`
- *Class* `AllocationTracker`
- *Class* `DynamicPoolList`
- *Class* `DynamicPoolMap`
- *Class* `FixedPool`
- *Class* `MixedPool`
- *Class* `MonotonicAllocationStrategy`
- *Class* `NamedAllocationStrategy`
- *Class* `NumaPolicy`
- *Class* `SizeLimiter`
- *Class* `SlotPool`
- *Class* `ThreadSafeAllocator`
- *Class* `ZeroByteHandler`

## Functions

- *Function* `umpire::strategy::find_first_set`
- *Function* `umpire::strategy::heuristic_percent_releasable`
- *Function* `umpire::strategy::heuristic_percent_releasable_list`
- *Function* `umpire::strategy::operator<<`

## Typedefs

- *Typedef* `umpire::strategy::DynamicPool`

## Variables

- Variable *umpire::strategy::bits\_per\_int*
- Variable *umpire::strategy::heuristic\_percent\_releasable*
- Variable *umpire::strategy::heuristic\_percent\_releasable\_list*

## Namespace *umpire::strategy::mixins*

### Contents

- *Classes*

## Classes

- *Class Inspector*

## Namespace *umpire::util*

### Contents

- *Namespaces*
- *Classes*
- *Functions*
- *Variables*

## Namespaces

- *Namespace *umpire::util::@171**
- *Namespace *umpire::util::detail**
- *Namespace *umpire::util::message**

## Classes

- *Template Struct *RecordList::Block**
- *Struct *AllocationRecord**
- *Struct *FixedMallocPool::Pool**
- *Struct *iterator\_begin**
- *Struct *iterator\_end**
- *Class *AllocationMap**

- *Class AllocationMap::ConstIterator*
- *Class AllocationMap::RecordList*
- *Class RecordList::ConstIterator*
- *Class Exception*
- *Class FixedMallocPool*
- *Class Logger*
- *Template Class MemoryMap*
- *Template Class MemoryMap::Iterator\_*
- *Class MPI*
- *Class OutputBuffer*
- *Class Statistic*
- *Class StatisticsDatabase*

## Functions

- *Function umpire::util::case\_insensitive\_match*
- *Function umpire::util::directory\_exists*
- *Template Function umpire::util::do\_wrap(std::unique\_ptr<Base>&&)*
- *Template Function umpire::util::do\_wrap(std::unique\_ptr<Base>&&)*
- *Function umpire::util::file\_exists*
- *Function umpire::util::flush\_files*
- *Function umpire::util::initialize\_io*
- *Template Function umpire::util::make\_unique*
- *Function umpire::util::make\_unique\_filename*
- *Function umpire::util::relative\_fragmentation*
- *Template Function umpire::util::unwrap\_allocation\_strategy*
- *Template Function umpire::util::unwrap\_allocator*
- *Template Function umpire::util::wrap\_allocator*

## Variables

- *Variable umpire::util::defaultLevel*
- *Variable umpire::util::env\_name*
- *Variable umpire::util::MessageLevelName*

## Namespace `umpire::util::@171`

### Namespace `umpire::util::detail`

#### Contents

- *Functions*

#### Functions

- *Template Function `umpire::util::detail::add_entry(conduit::Node&, T, U)`*
- *Template Function `umpire::util::detail::add_entry(conduit::Node&, T, U, Args...)`*
- *Function `umpire::util::detail::add_entry(conduit::Node&)`*
- *Template Function `umpire::util::detail::record_statistic`*

### Namespace `umpire::util::message`

#### Contents

- *Enums*

#### Enums

- *Enum `Level`*

### Namespace `umpire_mod`

#### Contents

- *Functions*

#### Functions

- *Function `umpire_mod::allocator_allocate`*
- *Function `umpire_mod::allocator_allocate_double_array_1d`*
- *Function `umpire_mod::allocator_allocate_double_array_2d`*
- *Function `umpire_mod::allocator_allocate_double_array_3d`*
- *Function `umpire_mod::allocator_allocate_double_array_4d`*
- *Function `umpire_mod::allocator_allocate_float_array_1d`*
- *Function `umpire_mod::allocator_allocate_float_array_2d`*



- *Function `umpire_mod::allocator_allocate_float_array_3d`*
- *Function `umpire_mod::allocator_allocate_float_array_4d`*
- *Function `umpire_mod::allocator_allocate_int_array_1d`*
- *Function `umpire_mod::allocator_allocate_int_array_2d`*
- *Function `umpire_mod::allocator_allocate_int_array_3d`*
- *Function `umpire_mod::allocator_allocate_int_array_4d`*
- *Function `umpire_mod::allocator_allocate_long_array_1d`*
- *Function `umpire_mod::allocator_allocate_long_array_2d`*
- *Function `umpire_mod::allocator_allocate_long_array_3d`*
- *Function `umpire_mod::allocator_allocate_long_array_4d`*
- *Function `umpire_mod::allocator_associated`*
- *Function `umpire_mod::allocator_deallocate`*
- *Function `umpire_mod::allocator_deallocate_double_array_1d`*
- *Function `umpire_mod::allocator_deallocate_double_array_2d`*
- *Function `umpire_mod::allocator_deallocate_double_array_3d`*
- *Function `umpire_mod::allocator_deallocate_double_array_4d`*
- *Function `umpire_mod::allocator_deallocate_float_array_1d`*
- *Function `umpire_mod::allocator_deallocate_float_array_2d`*
- *Function `umpire_mod::allocator_deallocate_float_array_3d`*
- *Function `umpire_mod::allocator_deallocate_float_array_4d`*
- *Function `umpire_mod::allocator_deallocate_int_array_1d`*
- *Function `umpire_mod::allocator_deallocate_int_array_2d`*
- *Function `umpire_mod::allocator_deallocate_int_array_3d`*
- *Function `umpire_mod::allocator_deallocate_int_array_4d`*
- *Function `umpire_mod::allocator_deallocate_long_array_1d`*
- *Function `umpire_mod::allocator_deallocate_long_array_2d`*
- *Function `umpire_mod::allocator_deallocate_long_array_3d`*
- *Function `umpire_mod::allocator_deallocate_long_array_4d`*
- *Function `umpire_mod::allocator_delete`*
- *Function `umpire_mod::allocator_eq`*
- *Function `umpire_mod::allocator_get_actual_size`*
- *Function `umpire_mod::allocator_get_current_size`*
- *Function `umpire_mod::allocator_get_high_watermark`*
- *Function `umpire_mod::allocator_get_id`*
- *Function `umpire_mod::allocator_get_instance`*
- *Function `umpire_mod::allocator_get_name`*

- *Function `umpire_mod::allocator_get_size`*
- *Function `umpire_mod::allocator_ne`*
- *Function `umpire_mod::allocator_release`*
- *Function `umpire_mod::allocator_set_instance`*
- *Function `umpire_mod::resourcemanager_associated`*
- *Function `umpire_mod::resourcemanager_copy_all`*
- *Function `umpire_mod::resourcemanager_copy_with_size`*
- *Function `umpire_mod::resourcemanager_deallocate`*
- *Function `umpire_mod::resourcemanager_eq`*
- *Function `umpire_mod::resourcemanager_get_allocator_by_id`*
- *Function `umpire_mod::resourcemanager_get_allocator_by_name`*
- *Function `umpire_mod::resourcemanager_get_allocator_for_ptr`*
- *Function `umpire_mod::resourcemanager_get_instance`*
- *Function `umpire_mod::resourcemanager_get_size`*
- *Function `umpire_mod::resourcemanager_has_allocator`*
- *Function `umpire_mod::resourcemanager_is_allocator`*
- *Function `umpire_mod::resourcemanager_make_allocator_advisor`*
- *Function `umpire_mod::resourcemanager_make_allocator_fixed_pool`*
- *Function `umpire_mod::resourcemanager_make_allocator_list_pool`*
- *Function `umpire_mod::resourcemanager_make_allocator_named`*
- *Function `umpire_mod::resourcemanager_make_allocator_pool`*
- *Function `umpire_mod::resourcemanager_make_allocator_prefetcher`*
- *Function `umpire_mod::resourcemanager_make_allocator_thread_safe`*
- *Function `umpire_mod::resourcemanager_memset_all`*
- *Function `umpire_mod::resourcemanager_memset_with_size`*
- *Function `umpire_mod::resourcemanager_move`*
- *Function `umpire_mod::resourcemanager_ne`*
- *Function `umpire_mod::resourcemanager_reallocate_default`*
- *Function `umpire_mod::resourcemanager_reallocate_with_allocator`*
- *Function `umpire_mod::resourcemanager_register_allocator`*

## Namespace `umpire_strategy_mod`

### Contents

- *Functions*

### Functions

- *Function `umpire_strategy_mod::allocationadvisor_associated`*
- *Function `umpire_strategy_mod::allocationadvisor_eq`*
- *Function `umpire_strategy_mod::allocationadvisor_get_instance`*
- *Function `umpire_strategy_mod::allocationadvisor_ne`*
- *Function `umpire_strategy_mod::allocationadvisor_set_instance`*
- *Function `umpire_strategy_mod::dynamicpool_associated`*
- *Function `umpire_strategy_mod::dynamicpool_eq`*
- *Function `umpire_strategy_mod::dynamicpool_get_instance`*
- *Function `umpire_strategy_mod::dynamicpool_ne`*
- *Function `umpire_strategy_mod::dynamicpool_set_instance`*
- *Function `umpire_strategy_mod::namedallocationstrategy_associated`*
- *Function `umpire_strategy_mod::namedallocationstrategy_eq`*
- *Function `umpire_strategy_mod::namedallocationstrategy_get_instance`*
- *Function `umpire_strategy_mod::namedallocationstrategy_ne`*
- *Function `umpire_strategy_mod::namedallocationstrategy_set_instance`*

## 6.3.2 Classes and Structs

### Struct `DynamicSizePool::Block`

- Defined in file `umpire_strategy_DynamicSizePool.hpp`

### Nested Relationships

This struct is a nested type of *Template Class `DynamicSizePool`*.

## Struct Documentation

### struct Block

#### Public Members

template<>  
char \***data**

template<>  
std::size\_t **size**

template<>  
std::size\_t **blockSize**

template<>  
Block \***next**

### Struct FixedSizePool::Pool

- Defined in file\_umpire\_strategy\_FixedSizePool.hpp

## Nested Relationships

This struct is a nested type of *Template Class FixedSizePool*.

## Struct Documentation

### struct Pool

#### Public Members

template<>  
unsigned char \***data**

template<>  
unsigned int \***avail**

template<>  
unsigned int **numAvail**

template<>  
**struct** Pool \***next**

### Struct `s_umpire_allocator`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

#### Struct Documentation

```
struct s_umpire_allocator
```

##### Public Members

```
void *addr
```

```
int idtor
```

### Struct `s_umpire_resourcemanager`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

#### Struct Documentation

```
struct s_umpire_resourcemanager
```

##### Public Members

```
void *addr
```

```
int idtor
```

### Struct `s_umpire_SHROUD_array`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

#### Struct Documentation

```
struct s_umpire_SHROUD_array
```

##### Public Members

```
umpire_SHROUD_capsule_data cxx
```

```
const void *cvoidp
```

```
const char *ccharp
```

```
union s_umpire_SHROUD_array::[anonymous] addr
```

```
size_t len
```

```
size_t size
```

### Struct `s_umpire_SHROUD_capsule_data`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

#### Struct Documentation

```
struct s_umpire_SHROUD_capsule_data
```

##### Public Members

```
void *addr
```

```
int idtor
```

### Struct `s_umpire_strategy_allocationadvisor`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

#### Struct Documentation

```
struct s_umpire_strategy_allocationadvisor
```

##### Public Members

```
void *addr
```

```
int idtor
```

### Struct `s_umpire_strategy_allocationprefetcher`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

#### Struct Documentation

```
struct s_umpire_strategy_allocationprefetcher
```

##### Public Members

```
void *addr
```

```
int idtor
```

### Struct `s_umpire_strategy_dynamicpool`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

#### Struct Documentation

```
struct s_umpire_strategy_dynamicpool
```

##### Public Members

```
void *addr
```

```
int idtor
```

### Struct `s_umpire_strategy_dynamicpoollist`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

#### Struct Documentation

```
struct s_umpire_strategy_dynamicpoollist
```

##### Public Members

```
void *addr
```

```
int idtor
```

### Struct `s_umpire_strategy_fixedpool`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

#### Struct Documentation

```
struct s_umpire_strategy_fixedpool
```

##### Public Members

```
void *addr
```

```
int idtor
```

### Struct `s_umpire_strategy_namedallocationstrategy`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

#### Struct Documentation

```
struct s_umpire_strategy_namedallocationstrategy
```

##### Public Members

```
void *addr
```

```
int idtor
```

### Struct `s_umpire_strategy_threadsafeallocator`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

#### Struct Documentation

```
struct s_umpire_strategy_threadsafeallocator
```

##### Public Members

```
void *addr
```

```
int idtor
```

### Struct `StdAllocator`

- Defined in file `umpire_strategy_StdAllocator.hpp`

#### Struct Documentation

```
struct StdAllocator
```

##### Public Static Functions

```
static void *allocate (std::size_t size)
```

```
static void deallocate (void *ptr)
```



## Struct AmAllocAllocator

- Defined in file\_umpire\_alloc\_AmAllocAllocator.hpp

## Struct Documentation

### struct AmAllocAllocator

Uses hcAlloc and hcAlloc to allocate and deallocate memory on AMD GPUs that support ROCm.

#### Public Functions

void \***allocate** (std::size\_t *bytes*)  
Allocate bytes of memory using hcAlloc.

**Return** Pointer to start of the allocation.

#### Parameters

- *bytes*: Number of bytes to allocate.

#### Exceptions

- *umpire::util::Exception*: if memory cannot be allocated.

void **deallocate** (void \**ptr*)  
Deallocate memory using hcFree.

#### Parameters

- *ptr*: Address to deallocate.

#### Exceptions

- *umpire::util::Exception*: if memory cannot be free'd.

## Struct AmPinnedAllocator

- Defined in file\_umpire\_alloc\_AmPinnedAllocator.hpp

## Struct Documentation

### struct AmPinnedAllocator

Uses hcAlloc and hcAlloc to allocate and deallocate memory on AMD GPUs that support ROCm.

## Public Functions

void **\*allocate** (std::size\_t *bytes*)  
Allocate bytes of pinned memory using `am_alloc`.

**Return** Pointer to start of the allocation.

### Parameters

- `bytes`: Number of bytes to allocate.

### Exceptions

- `umpire::util::Exception`: if memory cannot be allocated.

void **deallocate** (void *\*ptr*)  
Deallocate memory using `am_free`.

### Parameters

- `ptr`: Address to deallocate.

### Exceptions

- `umpire::util::Exception`: if memory cannot be free'd.

## Struct CudaMallocAllocator

- Defined in file `umpire_alloc_CudaMallocAllocator.hpp`

## Struct Documentation

### struct CudaMallocAllocator

Uses `cudaMalloc` and `cudaFree` to allocate and deallocate memory on NVIDIA GPUs.

## Public Functions

void **\*allocate** (std::size\_t *size*)  
Allocate bytes of memory using `cudaMalloc`.

**Return** Pointer to start of the allocation.

### Parameters

- `bytes`: Number of bytes to allocate.

### Exceptions

- `umpire::util::Exception`: if memory cannot be allocated.

void **deallocate** (void *\*ptr*)  
Deallocate memory using `cudaFree`.

### Parameters

- `ptr`: Address to deallocate.

**Exceptions**

- `umpire::util::Exception`: if memory cannot be free'd.

**Struct CudaMallocManagedAllocator**

- Defined in file\_umpire\_alloc\_CudaMallocManagedAllocator.hpp

**Struct Documentation****struct CudaMallocManagedAllocator**

Uses `cudaMallocManaged` and `cudaFree` to allocate and deallocate unified memory on NVIDIA GPUs.

**Public Functions**

void **\*allocate** (std::size\_t *bytes*)  
Allocate bytes of memory using `cudaMallocManaged`.

**Return** Pointer to start of the allocation.

**Parameters**

- *bytes*: Number of bytes to allocate.

**Exceptions**

- `umpire::util::Exception`: if memory cannot be allocated.

void **deallocate** (void *\*ptr*)  
Deallocate memory using `cudaFree`.

**Parameters**

- *ptr*: Address to deallocate.

**Exceptions**

- `umpire::util::Exception`: if memory be free'd.

**Struct CudaPinnedAllocator**

- Defined in file\_umpire\_alloc\_CudaPinnedAllocator.hpp

**Struct Documentation****struct CudaPinnedAllocator**

## Public Functions

void **\*allocate** (std::size\_t *bytes*)

void **deallocate** (void *\*ptr*)

## Struct HipMallocAllocator

- Defined in file\_umpire\_alloc\_HipMallocAllocator.hpp

## Struct Documentation

### struct HipMallocAllocator

Uses hipMalloc and hipFree to allocate and deallocate memory on AMD GPUs.

## Public Functions

void **\*allocate** (std::size\_t *size*)

Allocate bytes of memory using hipMalloc.

**Return** Pointer to start of the allocation.

### Parameters

- *bytes*: Number of bytes to allocate.

### Exceptions

- *umpire::util::Exception*: if memory cannot be allocated.

void **deallocate** (void *\*ptr*)

Deallocate memory using hipFree.

### Parameters

- *ptr*: Address to deallocate.

### Exceptions

- *umpire::util::Exception*: if memory cannot be free'd.

## Struct HipPinnedAllocator

- Defined in file\_umpire\_alloc\_HipPinnedAllocator.hpp

## Struct Documentation

### struct HipPinnedAllocator

#### Public Functions

void \***allocate** (std::size\_t *bytes*)

void **deallocate** (void \**ptr*)

### Struct MallocAllocator

- Defined in file\_umpire\_alloc\_MallocAllocator.hpp

## Struct Documentation

### struct MallocAllocator

Uses malloc and free to allocate and deallocate CPU memory.

#### Public Functions

void \***allocate** (std::size\_t *bytes*)

Allocate bytes of memory using malloc.

**Return** Pointer to start of the allocation.

#### Parameters

- *bytes*: Number of bytes to allocate.

#### Exceptions

- *umpire::util::Exception*: if memory cannot be allocated.

void **deallocate** (void \**ptr*)

Deallocate memory using free.

#### Parameters

- *ptr*: Address to deallocate.

#### Exceptions

- *umpire::util::Exception*: if memory cannot be free'd.

## Struct PosixMemalignAllocator

- Defined in file\_umpire\_alloc\_PosixMemalignAllocator.hpp

## Struct Documentation

### struct PosixMemalignAllocator

Uses `posix_memalign()` and `free()` to allocate page-aligned memory.

#### Public Functions

void \***allocate** (std::size\_t *bytes*)  
Allocate bytes of memory using `posix_memalign`.

**Return** Pointer to start of the allocation.

#### Parameters

- *bytes*: Number of bytes to allocate. Does not have to be a multiple of the system page size.

#### Exceptions

- `umpire::util::Exception`: if memory cannot be allocated.

void **deallocate** (void \**ptr*)  
Deallocate memory using `free`.

#### Parameters

- *ptr*: Address to deallocate.

#### Exceptions

- `umpire::util::Exception`: if memory cannot be free'd.

## Struct MemoryResourceTraits

- Defined in file\_umpire\_util\_MemoryResourceTraits.hpp

## Struct Documentation

### struct MemoryResourceTraits

#### Public Types

enum **optimized\_for**

*Values:*

**any**

**latency**

**bandwidth**

**access**

```
enum vendor_type
```

*Values:*

```
UNKNOWN
```

```
AMD
```

```
IBM
```

```
INTEL
```

```
NVIDIA
```

```
enum memory_type
```

*Values:*

```
UNKNOWN
```

```
DDR
```

```
GDDR
```

```
HBM
```

```
NVME
```

### Public Members

```
bool unified = false
```

```
std::size_t size = 0
```

```
vendor_type vendor = vendor_type::UNKNOWN
```

```
memory_type kind = memory_type::UNKNOWN
```

```
optimized_for used_for = optimized_for::any
```

### Struct pair\_hash

- Defined in file\_umpire\_op\_MemoryOperationRegistry.hpp

### Struct Documentation

```
struct pair_hash
```

### Public Functions

```
std::size_t operator () (const std::pair<Platform, Platform> &p) const
```

## Struct MemoryResourceTypeHash

- Defined in file\_umpire\_resource\_MemoryResourceTypes.hpp

## Struct Documentation

**struct** MemoryResourceTypeHash

### Public Functions

```
template<typename T>  
std::size_t operator () (T t) const
```

## Struct FixedPool::Pool

- Defined in file\_umpire\_strategy\_FixedPool.hpp

## Nested Relationships

This struct is a nested type of *Class FixedPool*.

## Struct Documentation

**struct** Pool

### Public Functions

```
Pool (AllocationStrategy *allocation_strategy, const std::size_t object_bytes, const std::size_t ob-  
jects_per_pool, const std::size_t avail_bytes)
```

### Public Members

```
AllocationStrategy *strategy
```

```
char *data
```

```
int *avail
```

```
std::size_t num_avail
```



### Template Struct RecordList::Block

- Defined in file\_umpire\_util\_AllocationMap.hpp

### Nested Relationships

This struct is a nested type of *Class AllocationMap::RecordList*.

### Struct Documentation

```
template<typename T>  
struct Block
```

#### Public Members

```
T rec  
Block *prev
```

### Struct AllocationRecord

- Defined in file\_umpire\_util\_AllocationRecord.hpp

### Struct Documentation

```
struct AllocationRecord
```

#### Public Members

```
void *ptr  
std::size_t size  
strategy::AllocationStrategy *strategy
```

### Struct FixedMallocPool::Pool

- Defined in file\_umpire\_util\_FixedMallocPool.hpp

### Nested Relationships

This struct is a nested type of *Class FixedMallocPool*.

## Struct Documentation

**struct Pool**

### Public Functions

**Pool** (**const** std::size\_t *object\_bytes*, **const** std::size\_t *objects\_per\_pool*)

### Public Members

unsigned char \***data**

unsigned char \***next**

unsigned int **num\_initialized**

std::size\_t **num\_free**

## Struct iterator\_begin

- Defined in file\_umpire\_util\_MemoryMap.hpp

## Struct Documentation

**struct iterator\_begin**

## Struct iterator\_end

- Defined in file\_umpire\_util\_MemoryMap.hpp

## Struct Documentation

**struct iterator\_end**

## Template Class DynamicSizePool

- Defined in file\_umpire\_strategy\_DynamicSizePool.hpp

## Nested Relationships

### Nested Types

- *Struct DynamicSizePool::Block*

## Class Documentation

```
template<class IA = StdAllocator>
class DynamicSizePool
```

### Public Functions

```
DynamicSizePool (umpire::strategy::AllocationStrategy *strat, const std::size_t _minInitialBytes =
(16 * 1024), const std::size_t _minBytes = 256)
```

```
~DynamicSizePool ()
```

```
void *allocate (std::size_t size)
```

```
void deallocate (void *ptr)
```

```
std::size_t getCurrentSize () const
```

```
std::size_t getActualSize () const
```

```
std::size_t getHighWatermark () const
```

```
std::size_t getBlocksInPool () const
```

```
std::size_t getLargestAvailableBlock () const
```

```
std::size_t getReleasableSize () const
```

```
std::size_t getFreeBlocks () const
```

```
std::size_t getInUseBlocks () const
```

```
void coalesce ()
```

```
void release ()
```

### Protected Types

```
typedef FixedSizePool<struct Block, IA, IA, (1 << 6)> BlockPool
```

### Protected Functions

```
void findUsableBlock (struct Block *&best, struct Block *&prev, std::size_t size)
```

```
std::size_t alignmentAdjust (const std::size_t size)
```

```
void allocateBlock (struct Block *&curr, struct Block *&prev, const std::size_t size)
```

```
void splitBlock (struct Block *&curr, struct Block *&prev, const std::size_t size)
```

```
void releaseBlock (struct Block *curr, struct Block *prev)
```

```
std::size_t freeReleasedBlocks ()
```

```
void coalesceFreeBlocks (std::size_t size)
```

```
void freeAllBlocks ()
```

## Protected Attributes

*BlockPool* **blockPool**

**struct** *Block* \***usedBlocks**

**struct** *Block* \***freeBlocks**

std::size\_t **totalBlocks**

std::size\_t **totalBytes**

std::size\_t **allocBytes**

std::size\_t **minInitialBytes**

std::size\_t **minBytes**

std::size\_t **highWatermark**

umpire::strategy::*AllocationStrategy* \***allocator**

**struct** **Block**

## Public Members

template<>  
char \***data**

template<>  
std::size\_t **size**

template<>  
std::size\_t **blockSize**

template<>  
Block \***next**

## Template Class FixedSizePool

- Defined in file\_umpire\_strategy\_FixedSizePool.hpp

## Nested Relationships

### Nested Types

- *Struct FixedSizePool::Pool*

## Class Documentation

```
template<class T, class MA, class IA = StdAllocator, int NP = (1 << 6)>
class FixedSizePool
```

### Public Functions

```
FixedSizePool ()
```

```
~FixedSizePool ()
```

```
T *allocate ()
```

```
void deallocate (T *ptr)
```

```
std::size_t getCurrentSize () const
    Return allocated size to user.
```

```
std::size_t getActualSize () const
    Return total size with internal overhead.
```

```
std::size_t numPools () const
    Return the number of pools.
```

```
std::size_t poolSize () const
    Return the pool size.
```

### Public Static Functions

```
static FixedSizePool &getInstance ()
```

### Protected Functions

```
void newPool (struct Pool **pnew)
```

```
T *allocInPool (struct Pool *p)
```

### Protected Attributes

```
struct Pool *pool
```

```
const std::size_t numPerPool
```

```
const std::size_t totalPoolSize
```

```
std::size_t numBlocks
```

```
struct Pool
```

## Public Members

```
template<>
unsigned char *data

template<>
unsigned int *avail

template<>
unsigned int numAvail

template<>
struct Pool *next
```

## Class Allocator

- Defined in file\_umpire\_Allocator.hpp

## Class Documentation

### class Allocator

Provides a unified interface to allocate and free data.

An *Allocator* encapsulates all the details of how and where allocations will be made, and can also be used to introspect the memory resource. *Allocator* objects do not return typed allocations, so the pointer returned from the allocate method must be cast to the relevant type.

See *TypedAllocator*

## Public Functions

void \***allocate** (std::size\_t *bytes*)

Allocate bytes of memory.

The memory will be allocated as determined by the AllocationStrategy used by this *Allocator*. Note that this method does not guarantee new memory pages being requested from the underlying memory system, as the associated AllocationStrategy could have already allocated sufficient memory, or re-use existing allocations that were not returned to the system.

**Return** Pointer to start of the allocation.

### Parameters

- *bytes*: Number of bytes to allocate (>= 0)

void **deallocate** (void \**ptr*)

Free the memory at *ptr*.

This method will throw an `umpire::Exception` if *ptr* was not allocated using this *Allocator*. If you need to deallocate memory allocated by an unknown object, use the *ResourceManager::deallocate* method.

### Parameters

- *ptr*: Pointer to free (!nullptr)

void **release** ()

Release any and all unused memory held by this *Allocator*.

std::size\_t **getSize** (void \**ptr*) **const**

Return number of bytes allocated for allocation.

**Return** number of bytes allocated for ptr

**Parameters**

- *ptr*: Pointer to allocation in question

std::size\_t **getHighWatermark** () **const**

Return the memory high watermark for this *Allocator*.

This is the largest amount of memory allocated by this *Allocator*. Note that this may be larger than the largest value returned by `getCurrentSize`.

**Return** Memory high watermark.

std::size\_t **getCurrentSize** () **const**

Return the current size of this *Allocator*.

This is sum of the sizes of all the tracked allocations. Note that this doesn't ever have to be equal to `getHighWatermark`.

**Return** current size of *Allocator*.

std::size\_t **getActualSize** () **const**

Return the actual size of this *Allocator*.

For non-pool allocators, this will be the same as `getCurrentSize()`.

For pools, this is the total amount of memory allocated for blocks managed by the pool.

**Return** actual size of *Allocator*.

**const** std::string &**getName** () **const**

Get the name of this *Allocator*.

Allocators are uniquely named, and the name of the *Allocator* can be used to retrieve the same *Allocator* from the *ResourceManager* at a later time.

**See** *ResourceManager::getAllocator*

**Return** name of *Allocator*.

int **getId** () **const**

Get the integer ID of this *Allocator*.

Allocators are uniquely identified, and the ID of the *Allocator* can be used to retrieve the same *Allocator* from the *ResourceManager* at a later time.

**See** *ResourceManager::getAllocator*

**Return** integer id of *Allocator*.

`strategy::AllocationStrategy *getAllocationStrategy ()`  
Get the AllocationStrategy object used by this *Allocator*.

**Return** Pointer to the AllocationStrategy.

*Platform* `getPlatform ()`  
Get the Platform object appropriate for this *Allocator*.

**Return** Platform for this *Allocator*.

`Allocator ()`

## Friends

```
friend umpire::Allocator::AllocatorTest
std::ostream &operator<< (std::ostream &os, const Allocator &allocator)
```

## Class DeviceAllocator

- Defined in file\_umpire\_DeviceAllocator.hpp

## Class Documentation

`class DeviceAllocator`  
Lightweight allocator for use in GPU code.

## Public Functions

`__host__ DeviceAllocator (Allocator allocator, size_t size)`  
Construct a new *DeviceAllocator* that will use allocator to allocate data.

### Parameters

- `allocator`: *Allocator* to use for allocating memory.

`__host__ ~DeviceAllocator ()`

```
__host__ __device__ umpire::DeviceAllocator::DeviceAllocator(const DeviceAllocator &
__device__ void * umpire::DeviceAllocator::allocate(size_t size)
```



## Class `CudaAdviseAccessedByOperation`

- Defined in file `umpire_op_CudaAdviseAccessedByOperation.hpp`

## Inheritance Relationships

### Base Type

- `public umpire::op::MemoryOperation` (*Class `MemoryOperation`*)

## Class Documentation

```
class CudaAdviseAccessedByOperation : public umpire::op::MemoryOperation
```

### Public Functions

```
void apply (void *src_ptr, util::AllocationRecord *src_allocation, int val, std::size_t length)
    Apply val to the first length bytes of src_ptr.
```

Uses `cudaMemAdvise` to set data as accessed by the appropriate device.

#### Parameters

- `src_ptr`: Pointer to source memory location.
- `src_allocation`: `AllocationRecord` of source.
- `val`: Value to apply.
- `length`: Number of bytes to modify.

#### Exceptions

- `util::Exception`:

```
void transform (void *src_ptr, void **dst_ptr, util::AllocationRecord *src_allocation,
                util::AllocationRecord *dst_allocation, std::size_t length)
    Transform length bytes of memory from src_ptr to dst_ptr.
```

#### Parameters

- `src_ptr`: Pointer to source memory location.
- `dst_ptr`: Pointer to destination memory location.
- `src_allocation`: `AllocationRecord` of source.
- `dst_allocation`: `AllocationRecord` of destination.
- `length`: Number of bytes to transform.

#### Exceptions

- `util::Exception`:

## Class CudaAdvisePreferredLocationOperation

- Defined in file\_umpire\_op\_CudaAdvisePreferredLocationOperation.hpp

## Inheritance Relationships

### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

## Class Documentation

```
class CudaAdvisePreferredLocationOperation : public umpire::op::MemoryOperation
```

### Public Functions

```
void apply (void *src_ptr, util::AllocationRecord *src_allocation, int val, std::size_t length)  
    Apply val to the first length bytes of src_ptr.
```

Uses cudaMemAdvise to set preferred location of data.

#### Parameters

- src\_ptr: Pointer to source memory location.
- src\_allocation: AllocationRecord of source.
- val: Value to apply.
- length: Number of bytes to modify.

#### Exceptions

- *util::Exception:*

```
void transform (void *src_ptr, void **dst_ptr, util::AllocationRecord *src_allocation,  
               util::AllocationRecord *dst_allocation, std::size_t length)  
    Transform length bytes of memory from src_ptr to dst_ptr.
```

#### Parameters

- src\_ptr: Pointer to source memory location.
- dst\_ptr: Pointer to destination memory location.
- src\_allocation: AllocationRecord of source.
- dst\_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

#### Exceptions

- *util::Exception:*

## Class CudaAdviseReadMostlyOperation

- Defined in file\_umpire\_op\_CudaAdviseReadMostlyOperation.hpp

## Inheritance Relationships

### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

## Class Documentation

```
class CudaAdviseReadMostlyOperation : public umpire::op::MemoryOperation
```

### Public Functions

```
void apply (void *src_ptr, util::AllocationRecord *src_allocation, int val, std::size_t length)
```

Apply val to the first length bytes of src\_ptr.

Uses cudaMemAdvise to set data as “read mostly” on the appropriate device.

#### Parameters

- src\_ptr: Pointer to source memory location.
- src\_allocation: AllocationRecord of source.
- val: Value to apply.
- length: Number of bytes to modify.

#### Exceptions

- *util::Exception:*

```
void transform (void *src_ptr, void **dst_ptr, util::AllocationRecord *src_allocation,  
               util::AllocationRecord *dst_allocation, std::size_t length)
```

Transform length bytes of memory from src\_ptr to dst\_ptr.

#### Parameters

- src\_ptr: Pointer to source memory location.
- dst\_ptr: Pointer to destination memory location.
- src\_allocation: AllocationRecord of source.
- dst\_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

#### Exceptions

- *util::Exception:*

## Class CudaAdviseUnsetAccessedByOperation

- Defined in file\_umpire\_op\_CudaAdviseUnsetAccessedByOperation.hpp

## Inheritance Relationships

### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

## Class Documentation

```
class CudaAdviseUnsetAccessedByOperation : public umpire::op::MemoryOperation
```

### Public Functions

```
void apply (void *src_ptr, util::AllocationRecord *src_allocation, int val, std::size_t length)  
    Apply val to the first length bytes of src_ptr.
```

Uses cudaMemAdvise to set data as accessed by the appropriate device.

#### Parameters

- src\_ptr: Pointer to source memory location.
- src\_allocation: AllocationRecord of source.
- val: Value to apply.
- length: Number of bytes to modify.

#### Exceptions

- *util::Exception:*

```
void transform (void *src_ptr, void **dst_ptr, util::AllocationRecord *src_allocation,  
               util::AllocationRecord *dst_allocation, std::size_t length)  
    Transform length bytes of memory from src_ptr to dst_ptr.
```

#### Parameters

- src\_ptr: Pointer to source memory location.
- dst\_ptr: Pointer to destination memory location.
- src\_allocation: AllocationRecord of source.
- dst\_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

#### Exceptions

- *util::Exception:*

## Class CudaAdviseUnsetPreferredLocationOperation

- Defined in file\_umpire\_op\_CudaAdviseUnsetPreferredLocationOperation.hpp

### Inheritance Relationships

#### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

### Class Documentation

```
class CudaAdviseUnsetPreferredLocationOperation : public umpire::op::MemoryOperation
```

#### Public Functions

```
void apply (void *src_ptr, util::AllocationRecord *src_allocation, int val, std::size_t length)
    Apply val to the first length bytes of src_ptr.
```

Uses cudaMemAdvise to set preferred location of data.

#### Parameters

- src\_ptr: Pointer to source memory location.
- src\_allocation: AllocationRecord of source.
- val: Value to apply.
- length: Number of bytes to modify.

#### Exceptions

- *util::Exception:*

```
void transform (void *src_ptr, void **dst_ptr, util::AllocationRecord *src_allocation,
               util::AllocationRecord *dst_allocation, std::size_t length)
    Transform length bytes of memory from src_ptr to dst_ptr.
```

#### Parameters

- src\_ptr: Pointer to source memory location.
- dst\_ptr: Pointer to destination memory location.
- src\_allocation: AllocationRecord of source.
- dst\_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

#### Exceptions

- *util::Exception:*

## Class CudaAdviseUnsetReadMostlyOperation

- Defined in file\_umpire\_op\_CudaAdviseUnsetReadMostlyOperation.hpp

## Inheritance Relationships

### Base Type

- public `umpire::op::MemoryOperation` (*Class MemoryOperation*)

## Class Documentation

```
class CudaAdviseUnsetReadMostlyOperation : public umpire::op::MemoryOperation
```

### Public Functions

```
void apply (void *src_ptr, util::AllocationRecord *src_allocation, int val, std::size_t length)  
    Apply val to the first length bytes of src_ptr.
```

Uses `cudaMemAdvise` to set data as “read mostly” on the appropriate device.

#### Parameters

- `src_ptr`: Pointer to source memory location.
- `src_allocation`: `AllocationRecord` of source.
- `val`: Value to apply.
- `length`: Number of bytes to modify.

#### Exceptions

- `util::Exception`:

```
void transform (void *src_ptr, void **dst_ptr, util::AllocationRecord *src_allocation,  
               util::AllocationRecord *dst_allocation, std::size_t length)  
    Transform length bytes of memory from src_ptr to dst_ptr.
```

#### Parameters

- `src_ptr`: Pointer to source memory location.
- `dst_ptr`: Pointer to destination memory location.
- `src_allocation`: `AllocationRecord` of source.
- `dst_allocation`: `AllocationRecord` of destination.
- `length`: Number of bytes to transform.

#### Exceptions

- `util::Exception`:

## Class CudaCopyFromOperation

- Defined in file\_umpire\_op\_CudaCopyFromOperation.hpp

## Inheritance Relationships

### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

## Class Documentation

**class CudaCopyFromOperation** : public umpire::op::MemoryOperation  
Copy operation to move data from a NVIDIA GPU to CPU memory.

### Public Functions

void **transform**(void \*src\_ptr, void \*\*dst\_ptr, util::AllocationRecord \*src\_allocation, util::AllocationRecord \*dst\_allocation, std::size\_t length)  
Transform length bytes of memory from src\_ptr to dst\_ptr.

Uses cudaMemcpy to move data when src\_ptr is on a NVIDIA GPU and dst\_ptr is on the CPU.

#### Parameters

- src\_ptr: Pointer to source memory location.
- dst\_ptr: Pointer to destination memory location.
- src\_allocation: AllocationRecord of source.
- dst\_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

#### Exceptions

- util::Exception:

void **apply**(void \*src\_ptr, util::AllocationRecord \*src\_allocation, int val, std::size\_t length)  
Apply val to the first length bytes of src\_ptr.

#### Parameters

- src\_ptr: Pointer to source memory location.
- src\_allocation: AllocationRecord of source.
- val: Value to apply.
- length: Number of bytes to modify.

#### Exceptions

- util::Exception:

## Class `CudaCopyOperation`

- Defined in file `umpire_op_CudaCopyOperation.hpp`

## Inheritance Relationships

### Base Type

- `public umpire::op::MemoryOperation` (*Class `MemoryOperation`*)

## Class Documentation

**class `CudaCopyOperation`** : **public** `umpire::op::MemoryOperation`

Copy operation to move data between two GPU addresses.

### Public Functions

void **transform**(void \**src\_ptr*, void \*\**dst\_ptr*, `umpire::util::AllocationRecord` \**src\_allocation*, `umpire::util::AllocationRecord` \**dst\_allocation*, `std::size_t` *length*)

Transform *length* bytes of memory from *src\_ptr* to *dst\_ptr*.

Uses `cudaMemcpy` to move data when both *src\_ptr* and *dst\_ptr* are on NVIDIA GPUs.

#### Parameters

- *src\_ptr*: Pointer to source memory location.
- *dst\_ptr*: Pointer to destination memory location.
- *src\_allocation*: `AllocationRecord` of source.
- *dst\_allocation*: `AllocationRecord` of destination.
- *length*: Number of bytes to transform.

#### Exceptions

- `util::Exception`:

void **apply**(void \**src\_ptr*, `util::AllocationRecord` \**src\_allocation*, int *val*, `std::size_t` *length*)

Apply *val* to the first *length* bytes of *src\_ptr*.

#### Parameters

- *src\_ptr*: Pointer to source memory location.
- *src\_allocation*: `AllocationRecord` of source.
- *val*: Value to apply.
- *length*: Number of bytes to modify.

#### Exceptions

- `util::Exception`:



## Class `CudaCopyToOperation`

- Defined in file `umpire_op_CudaCopyToOperation.hpp`

## Inheritance Relationships

### Base Type

- `public umpire::op::MemoryOperation` (*Class `MemoryOperation`*)

## Class Documentation

**class `CudaCopyToOperation`** : **public** `umpire::op::MemoryOperation`

Copy operation to move data from CPU to NVIDIA GPU memory.

### Public Functions

void **transform**(void \**src\_ptr*, void \*\**dst\_ptr*, `umpire::util::AllocationRecord` \**src\_allocation*, `umpire::util::AllocationRecord` \**dst\_allocation*, `std::size_t` *length*)

Transform *length* bytes of memory from *src\_ptr* to *dst\_ptr*.

Uses `cudaMemcpy` to move data when *src\_ptr* is on the CPU and *dst\_ptr* is on an NVIDIA GPU.

#### Parameters

- *src\_ptr*: Pointer to source memory location.
- *dst\_ptr*: Pointer to destination memory location.
- *src\_allocation*: `AllocationRecord` of source.
- *dst\_allocation*: `AllocationRecord` of destination.
- *length*: Number of bytes to transform.

#### Exceptions

- `util::Exception`:

void **apply**(void \**src\_ptr*, `util::AllocationRecord` \**src\_allocation*, int *val*, `std::size_t` *length*)

Apply *val* to the first *length* bytes of *src\_ptr*.

#### Parameters

- *src\_ptr*: Pointer to source memory location.
- *src\_allocation*: `AllocationRecord` of source.
- *val*: Value to apply.
- *length*: Number of bytes to modify.

#### Exceptions

- `util::Exception`:

## Class CudaMemPrefetchOperation

- Defined in file\_umpire\_op\_CudaMemPrefetchOperation.hpp

## Inheritance Relationships

### Base Type

- public `umpire::op::MemoryOperation` (*Class MemoryOperation*)

## Class Documentation

```
class CudaMemPrefetchOperation : public umpire::op::MemoryOperation
```

### Public Functions

```
void apply (void *src_ptr, umpire::util::AllocationRecord *src_allocation, int val, std::size_t length)  
    Apply val to the first length bytes of src_ptr.
```

#### Parameters

- `src_ptr`: Pointer to source memory location.
- `src_allocation`: AllocationRecord of source.
- `val`: Value to apply.
- `length`: Number of bytes to modify.

#### Exceptions

- *util::Exception*:

```
void transform (void *src_ptr, void **dst_ptr, util::AllocationRecord *src_allocation,  
               util::AllocationRecord *dst_allocation, std::size_t length)  
    Transform length bytes of memory from src_ptr to dst_ptr.
```

#### Parameters

- `src_ptr`: Pointer to source memory location.
- `dst_ptr`: Pointer to destination memory location.
- `src_allocation`: AllocationRecord of source.
- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

#### Exceptions

- *util::Exception*:

## Class CudaMemsetOperation

- Defined in file\_umpire\_op\_CudaMemsetOperation.hpp

## Inheritance Relationships

### Base Type

- `public umpire::op::MemoryOperation` (*Class MemoryOperation*)

## Class Documentation

**class CudaMemsetOperation** : **public** `umpire::op::MemoryOperation`  
 Memset on NVIDIA device memory.

### Public Functions

void **apply** (void \**src\_ptr*, `util::AllocationRecord` \**ptr*, int *value*, `std::size_t` *length*)  
 Apply val to the first length bytes of src\_ptr.

Uses cudaMemset to set first length bytes of src\_ptr to value.

#### Parameters

- *src\_ptr*: Pointer to source memory location.
- *src\_allocation*: AllocationRecord of source.
- *val*: Value to apply.
- *length*: Number of bytes to modify.

#### Exceptions

- `util::Exception`:

void **transform** (void \**src\_ptr*, void \*\**dst\_ptr*, `util::AllocationRecord` \**src\_allocation*,  
`util::AllocationRecord` \**dst\_allocation*, `std::size_t` *length*)  
 Transform length bytes of memory from src\_ptr to dst\_ptr.

#### Parameters

- *src\_ptr*: Pointer to source memory location.
- *dst\_ptr*: Pointer to destination memory location.
- *src\_allocation*: AllocationRecord of source.
- *dst\_allocation*: AllocationRecord of destination.
- *length*: Number of bytes to transform.

#### Exceptions

- `util::Exception`:

## Class GenericReallocateOperation

- Defined in file\_umpire\_op\_GenericReallocateOperation.hpp

## Inheritance Relationships

### Base Type

- public `umpire::op::MemoryOperation` (*Class MemoryOperation*)

## Class Documentation

**class** `GenericReallocateOperation` : public `umpire::op::MemoryOperation`  
Generic reallocate operation to work on any `current_ptr` location.

### Public Functions

void **transform**(void \**current\_ptr*, void \*\**new\_ptr*, `util::AllocationRecord` \**current\_allocation*,  
`util::AllocationRecord` \**new\_allocation*, `std::size_t` *new\_size*)  
Transform length bytes of memory from `src_ptr` to `dst_ptr`.

This operation relies on `ResourceManager::copy`, `AllocationStrategy::allocate` and `AllocationStrategy::deallocate` to implement a reallocate operation that can work for any `current_ptr` location.

#### Parameters

- `src_ptr`: Pointer to source memory location.
- `dst_ptr`: Pointer to destination memory location.
- `src_allocation`: `AllocationRecord` of source.
- `dst_allocation`: `AllocationRecord` of destination.
- `length`: Number of bytes to transform.

#### Exceptions

- `util::Exception`:

void **apply**(void \**src\_ptr*, `util::AllocationRecord` \**src\_allocation*, int *val*, `std::size_t` *length*)  
Apply `val` to the first `length` bytes of `src_ptr`.

#### Parameters

- `src_ptr`: Pointer to source memory location.
- `src_allocation`: `AllocationRecord` of source.
- `val`: Value to apply.
- `length`: Number of bytes to modify.

#### Exceptions

- `util::Exception`:

## Class HipCopyFromOperation

- Defined in file\_umpire\_op\_HipCopyFromOperation.hpp

## Inheritance Relationships

### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

## Class Documentation

**class HipCopyFromOperation**: public umpire::op::MemoryOperation

Copy operation to move data from a AMD GPU to CPU memory.

### Public Functions

void **transform**(void \*src\_ptr, void \*\*dst\_ptr, util::AllocationRecord \*src\_allocation, util::AllocationRecord \*dst\_allocation, std::size\_t length)

Transform length bytes of memory from src\_ptr to dst\_ptr.

Uses hipMemcpy to move data when src\_ptr is on a AMD GPU and dst\_ptr is on the CPU.

#### Parameters

- src\_ptr: Pointer to source memory location.
- dst\_ptr: Pointer to destination memory location.
- src\_allocation: AllocationRecord of source.
- dst\_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

#### Exceptions

- util::Exception:

void **apply**(void \*src\_ptr, util::AllocationRecord \*src\_allocation, int val, std::size\_t length)

Apply val to the first length bytes of src\_ptr.

#### Parameters

- src\_ptr: Pointer to source memory location.
- src\_allocation: AllocationRecord of source.
- val: Value to apply.
- length: Number of bytes to modify.

#### Exceptions

- util::Exception:

## Class HipCopyOperation

- Defined in file\_umpire\_op\_HipCopyOperation.hpp

## Inheritance Relationships

### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

## Class Documentation

**class HipCopyOperation** : public umpire::op::MemoryOperation  
Copy operation to move data between two GPU addresses.

### Public Functions

void **transform**(void \*src\_ptr, void \*\*dst\_ptr, umpire::util::AllocationRecord \*src\_allocation, umpire::util::AllocationRecord \*dst\_allocation, std::size\_t length)  
Transform length bytes of memory from src\_ptr to dst\_ptr.

Uses hipMemcpy to move data when both src\_ptr and dst\_ptr are on AMD GPUs.

#### Parameters

- src\_ptr: Pointer to source memory location.
- dst\_ptr: Pointer to destination memory location.
- src\_allocation: AllocationRecord of source.
- dst\_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

#### Exceptions

- *util::Exception*:

void **apply**(void \*src\_ptr, umpire::util::AllocationRecord \*src\_allocation, int val, std::size\_t length)  
Apply val to the first length bytes of src\_ptr.

#### Parameters

- src\_ptr: Pointer to source memory location.
- src\_allocation: AllocationRecord of source.
- val: Value to apply.
- length: Number of bytes to modify.

#### Exceptions

- *util::Exception*:

## Class HipCopyToOperation

- Defined in file\_umpire\_op\_HipCopyToOperation.hpp

## Inheritance Relationships

### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

## Class Documentation

**class HipCopyToOperation** : public umpire::op::MemoryOperation

Copy operation to move data from CPU to AMD GPU memory.

### Public Functions

void **transform**(void \*src\_ptr, void \*\*dst\_ptr, umpire::util::AllocationRecord \*src\_allocation, umpire::util::AllocationRecord \*dst\_allocation, std::size\_t length)

Transform length bytes of memory from src\_ptr to dst\_ptr.

Uses hipMemcpy to move data when src\_ptr is on the CPU and dst\_ptr is on an AMD GPU.

#### Parameters

- src\_ptr: Pointer to source memory location.
- dst\_ptr: Pointer to destination memory location.
- src\_allocation: AllocationRecord of source.
- dst\_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

#### Exceptions

- *util::Exception*:

void **apply**(void \*src\_ptr, umpire::util::AllocationRecord \*src\_allocation, int val, std::size\_t length)

Apply val to the first length bytes of src\_ptr.

#### Parameters

- src\_ptr: Pointer to source memory location.
- src\_allocation: AllocationRecord of source.
- val: Value to apply.
- length: Number of bytes to modify.

#### Exceptions

- *util::Exception*:

## Class HipMemsetOperation

- Defined in file\_umpire\_op\_HipMemsetOperation.hpp

## Inheritance Relationships

### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

## Class Documentation

**class HipMemsetOperation** : public umpire::op::MemoryOperation  
Memset on AMD device memory.

### Public Functions

void **apply** (void \*src\_ptr, util::AllocationRecord \*ptr, int value, std::size\_t length)  
Apply val to the first length bytes of src\_ptr.  
Uses hipMemset to set first length bytes of src\_ptr to value.

#### Parameters

- src\_ptr: Pointer to source memory location.
- src\_allocation: AllocationRecord of source.
- val: Value to apply.
- length: Number of bytes to modify.

#### Exceptions

- util::Exception:

void **transform** (void \*src\_ptr, void \*\*dst\_ptr, util::AllocationRecord \*src\_allocation,  
util::AllocationRecord \*dst\_allocation, std::size\_t length)  
Transform length bytes of memory from src\_ptr to dst\_ptr.

#### Parameters

- src\_ptr: Pointer to source memory location.
- dst\_ptr: Pointer to destination memory location.
- src\_allocation: AllocationRecord of source.
- dst\_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

#### Exceptions

- util::Exception:



## Class HostCopyOperation

- Defined in file\_umpire\_op\_HostCopyOperation.hpp

## Inheritance Relationships

### Base Type

- public `umpire::op::MemoryOperation` (*Class MemoryOperation*)

## Class Documentation

**class HostCopyOperation** : public `umpire::op::MemoryOperation`

Copy memory between two allocations in CPU memory.

### Public Functions

void **transform**(void \**src\_ptr*, void \*\**dst\_ptr*, `umpire::util::AllocationRecord` \**src\_allocation*, `umpire::util::AllocationRecord` \**dst\_allocation*, `std::size_t` *length*)  
 Transform *length* bytes of memory from *src\_ptr* to *dst\_ptr*.

#### Parameters

- *src\_ptr*: Pointer to source memory location.
- *dst\_ptr*: Pointer to destination memory location.
- *src\_allocation*: `AllocationRecord` of source.
- *dst\_allocation*: `AllocationRecord` of destination.
- *length*: Number of bytes to transform.

#### Exceptions

- `util::Exception`:

void **apply**(void \**src\_ptr*, `util::AllocationRecord` \**src\_allocation*, int *val*, `std::size_t` *length*)  
 Apply *val* to the first *length* bytes of *src\_ptr*.

#### Parameters

- *src\_ptr*: Pointer to source memory location.
- *src\_allocation*: `AllocationRecord` of source.
- *val*: Value to apply.
- *length*: Number of bytes to modify.

#### Exceptions

- `util::Exception`:

## Class HostMemsetOperation

- Defined in file\_umpire\_op\_HostMemsetOperation.hpp

## Inheritance Relationships

### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

## Class Documentation

**class HostMemsetOperation** : public umpire::op::MemoryOperation  
Memset an allocation in CPU memory.

### Public Functions

void **apply** (void \*src\_ptr, util::AllocationRecord \*allocation, int value, std::size\_t length)  
Apply val to the first length bytes of src\_ptr.  
Uses std::memset to set the first length bytes of src\_ptr to value.

#### Parameters

- src\_ptr: Pointer to source memory location.
- src\_allocation: AllocationRecord of source.
- val: Value to apply.
- length: Number of bytes to modify.

#### Exceptions

- util::Exception:

void **transform** (void \*src\_ptr, void \*\*dst\_ptr, util::AllocationRecord \*src\_allocation,  
util::AllocationRecord \*dst\_allocation, std::size\_t length)  
Transform length bytes of memory from src\_ptr to dst\_ptr.

#### Parameters

- src\_ptr: Pointer to source memory location.
- dst\_ptr: Pointer to destination memory location.
- src\_allocation: AllocationRecord of source.
- dst\_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

#### Exceptions

- util::Exception:

## Class HostReallocateOperation

- Defined in file\_umpire\_op\_HostReallocateOperation.hpp

## Inheritance Relationships

### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

## Class Documentation

**class HostReallocateOperation** : public umpire::op::MemoryOperation  
 Reallocate data in CPU memory.

### Public Functions

void **transform**(void \*current\_ptr, void \*\*new\_ptr, util::AllocationRecord \*current\_allocation,  
 util::AllocationRecord \*new\_allocation, std::size\_t new\_size)  
 Transform length bytes of memory from src\_ptr to dst\_ptr.

Uses POSIX realloc to reallocate memory in the CPU memory.

#### Parameters

- src\_ptr: Pointer to source memory location.
- dst\_ptr: Pointer to destination memory location.
- src\_allocation: AllocationRecord of source.
- dst\_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

#### Exceptions

- util::Exception:

void **apply**(void \*src\_ptr, util::AllocationRecord \*src\_allocation, int val, std::size\_t length)  
 Apply val to the first length bytes of src\_ptr.

#### Parameters

- src\_ptr: Pointer to source memory location.
- src\_allocation: AllocationRecord of source.
- val: Value to apply.
- length: Number of bytes to modify.

#### Exceptions

- util::Exception:

## Class MemoryOperation

- Defined in file\_umpire\_op\_MemoryOperation.hpp

## Inheritance Relationships

## Derived Types

- `public umpire::op::CudaAdviseAccessedByOperation` (*Class CudaAdviseAccessedByOperation*)
- `public umpire::op::CudaAdvisePreferredLocationOperation` (*Class CudaAdvisePreferredLocationOperation*)
- `public umpire::op::CudaAdviseReadMostlyOperation` (*Class CudaAdviseReadMostlyOperation*)
- `public umpire::op::CudaAdviseUnsetAccessedByOperation` (*Class CudaAdviseUnsetAccessedByOperation*)
- `public umpire::op::CudaAdviseUnsetPreferredLocationOperation` (*Class CudaAdviseUnsetPreferredLocationOperation*)
- `public umpire::op::CudaAdviseUnsetReadMostlyOperation` (*Class CudaAdviseUnsetReadMostlyOperation*)
- `public umpire::op::CudaCopyFromOperation` (*Class CudaCopyFromOperation*)
- `public umpire::op::CudaCopyOperation` (*Class CudaCopyOperation*)
- `public umpire::op::CudaCopyToOperation` (*Class CudaCopyToOperation*)
- `public umpire::op::CudaMemPrefetchOperation` (*Class CudaMemPrefetchOperation*)
- `public umpire::op::CudaMemsetOperation` (*Class CudaMemsetOperation*)
- `public umpire::op::GenericReallocateOperation` (*Class GenericReallocateOperation*)
- `public umpire::op::HipCopyFromOperation` (*Class HipCopyFromOperation*)
- `public umpire::op::HipCopyOperation` (*Class HipCopyOperation*)
- `public umpire::op::HipCopyToOperation` (*Class HipCopyToOperation*)
- `public umpire::op::HipMemsetOperation` (*Class HipMemsetOperation*)
- `public umpire::op::HostCopyOperation` (*Class HostCopyOperation*)
- `public umpire::op::HostMemsetOperation` (*Class HostMemsetOperation*)
- `public umpire::op::HostReallocateOperation` (*Class HostReallocateOperation*)
- `public umpire::op::NumaMoveOperation` (*Class NumaMoveOperation*)
- `public umpire::op::RocmCopyOperation` (*Class RocmCopyOperation*)
- `public umpire::op::RocmMemsetOperation` (*Class RocmMemsetOperation*)

## Class Documentation

### class `MemoryOperation`

Base class of an operation on memory.

Neither the transform or apply methods are pure virtual, so inheriting classes only need overload the appropriate method. However, both methods will throw an error if called.

Subclassed by `umpire::op::CudaAdviseAccessedByOperation`, `umpire::op::CudaAdvisePreferredLocationOperation`, `umpire::op::CudaAdviseReadMostlyOperation`, `umpire::op::CudaAdviseUnsetAccessedByOperation`, `umpire::op::CudaAdviseUnsetPreferredLocationOperation`, `umpire::op::CudaAdviseUnsetReadMostlyOperation`, `umpire::op::CudaCopyFromOperation`, `umpire::op::CudaCopyOperation`, `umpire::op::CudaCopyToOperation`, `umpire::op::CudaMemPrefetchOperation`, `umpire::op::CudaMemsetOperation`, `umpire::op::GenericReallocateOperation`, `umpire::op::HipCopyFromOperation`, `umpire::op::HipCopyOperation`, `umpire::op::HipCopyToOperation`, `umpire::op::HipMemsetOperation`, `umpire::op::HostCopyOperation`, `umpire::op::HostMemsetOperation`, `umpire::op::HostReallocateOperation`, `umpire::op::NumaMoveOperation`, `umpire::op::RocmCopyOperation`, `umpire::op::RocmMemsetOperation`

### Public Functions

**virtual** `~MemoryOperation()`

void **transform**(void \**src\_ptr*, void \*\**dst\_ptr*, util::AllocationRecord \**src\_allocation*, util::AllocationRecord \**dst\_allocation*, std::size\_t *length*)

Transform length bytes of memory from *src\_ptr* to *dst\_ptr*.

#### Parameters

- *src\_ptr*: Pointer to source memory location.
- *dst\_ptr*: Pointer to destination memory location.
- *src\_allocation*: AllocationRecord of source.
- *dst\_allocation*: AllocationRecord of destination.
- *length*: Number of bytes to transform.

#### Exceptions

- `util::Exception`:

void **apply**(void \**src\_ptr*, util::AllocationRecord \**src\_allocation*, int *val*, std::size\_t *length*)

Apply *val* to the first *length* bytes of *src\_ptr*.

#### Parameters

- *src\_ptr*: Pointer to source memory location.
- *src\_allocation*: AllocationRecord of source.
- *val*: Value to apply.
- *length*: Number of bytes to modify.

#### Exceptions

- `util::Exception`:

## Class MemoryOperationRegistry

- Defined in file\_umpire\_op\_MemoryOperationRegistry.hpp

## Class Documentation

### class MemoryOperationRegistry

The *MemoryOperationRegistry* serves as a registry for *MemoryOperation* objects. It is a singleton class, typically accessed through the *ResourceManager*.

The *MemoryOperationRegistry* class provides lookup mechanisms allowing searching for the appropriate *MemoryOperation* to be applied to allocations made with particular *AllocationStrategy* objects.

MemoryOperations provided by Umpire are registered with the *MemoryOperationRegistry* when it is constructed. Additional MemoryOperations can be registered later using the registerOperation method.

The following operations are pre-registered for all *AllocationStrategy* pairs:

- "COPY"
- "MEMSET"
- "REALLOCATE"

See *MemoryOperation*

See *AllocationStrategy*

## Public Functions

```
std::shared_ptr<umpire::op::MemoryOperation> find(const std::string &name, strategy::AllocationStrategy *source_allocator, strategy::AllocationStrategy *dst_allocator)
```

Function to find a *MemoryOperation* object.

Finds the *MemoryOperation* object that matches the given name and *AllocationStrategy* objects. If the requested *MemoryOperation* is not found, this method will throw an Exception.

### Parameters

- name: Name of operation.
- src\_allocator: *AllocationStrategy* of the source allocation.
- dst\_allocator: *AllocationStrategy* of the destination allocation.

### Exceptions

- *umpire::util::Exception*: if the requested *MemoryOperation* is not found.

```
void registerOperation(const std::string &name, std::pair<Platform, Platform> platforms, std::shared_ptr<MemoryOperation> &&operation)
```

Add a new *MemoryOperation* to the registry.

This object will register the provided *MemoryOperation*, making it available for later retrieval using *MemoryOperation::find*

### Parameters

- name: Name of the operation.

- `platforms`: pair of `Platforms` for the source and destination.
- `operation`: pointer to the `MemoryOperation`.

```

MemoryOperationRegistry (const MemoryOperationRegistry&)
MemoryOperationRegistry &operator= (const MemoryOperationRegistry&)
~MemoryOperationRegistry ()

```

### Public Static Functions

```

MemoryOperationRegistry &getInstance ()
    Get the MemoryOperationRegistry singleton instance.

```

### Protected Functions

```

MemoryOperationRegistry ()

```

## Class NumaMoveOperation

- Defined in file `umpire_op_NumaMoveOperation.hpp`

### Inheritance Relationships

#### Base Type

- `public` `umpire::op::MemoryOperation` (*Class MemoryOperation*)

### Class Documentation

```

class NumaMoveOperation : public umpire::op::MemoryOperation
    Relocate a pointer to a different NUMA node.

```

#### Public Functions

```

void transform (void *src_ptr, void **dst_ptr, umpire::util::AllocationRecord *src_allocation, umpire::util::AllocationRecord *dst_allocation, std::size_t length)
    Transfrom length bytes of memory from src_ptr to dst_ptr.

```

#### Parameters

- `src_ptr`: Pointer to source memory location.
- `dst_ptr`: Pointer to destinatio memory location.
- `src_allocation`: `AllocationRecord` of source.
- `dst_allocation`: `AllocationRecord` of destination.
- `length`: Number of bytes to transform.

### Exceptions

- `util::Exception`:

void **apply** (void \**src\_ptr*, util::AllocationRecord \**src\_allocation*, int *val*, std::size\_t *length*)  
Apply *val* to the first *length* bytes of *src\_ptr*.

### Parameters

- *src\_ptr*: Pointer to source memory location.
- *src\_allocation*: AllocationRecord of source.
- *val*: Value to apply.
- *length*: Number of bytes to modify.

### Exceptions

- `util::Exception`:

## Class RocmCopyOperation

- Defined in file\_umpire\_op\_RocmCopyOperation.hpp

## Inheritance Relationships

### Base Type

- public umpire::op::MemoryOperation (*Class MemoryOperation*)

## Class Documentation

**class RocmCopyOperation** : public umpire::op::MemoryOperation  
Copy operation to move data between two GPU addresses.

### Public Functions

void **transform** (void \**src\_ptr*, void \*\**dst\_ptr*, umpire::util::AllocationRecord \**src\_allocation*, umpire::util::AllocationRecord \**dst\_allocation*, std::size\_t *length*)  
Transform *length* bytes of memory from *src\_ptr* to *dst\_ptr*.  
Uses `hc::accelerator_view::copy` to move data when either *src\_ptr* and *dst\_ptr* are on AMD GPUs.

### Parameters

- *src\_ptr*: Pointer to source memory location.
- *dst\_ptr*: Pointer to destination memory location.
- *src\_allocation*: AllocationRecord of source.
- *dst\_allocation*: AllocationRecord of destination.
- *length*: Number of bytes to transform.

### Exceptions



- `util::Exception`:

void **apply** (void \**src\_ptr*, `util::AllocationRecord` \**src\_allocation*, int *val*, `std::size_t` *length*)  
Apply val to the first length bytes of src\_ptr.

#### Parameters

- `src_ptr`: Pointer to source memory location.
- `src_allocation`: AllocationRecord of source.
- `val`: Value to apply.
- `length`: Number of bytes to modify.

#### Exceptions

- `util::Exception`:

### Class RocmMemsetOperation

- Defined in file\_umpire\_op\_RocmMemsetOperation.hpp

### Inheritance Relationships

#### Base Type

- public `umpire::op::MemoryOperation` (*Class MemoryOperation*)

### Class Documentation

**class RocmMemsetOperation** : public `umpire::op::MemoryOperation`  
Memset an allocation in CPU memory.

#### Public Functions

void **apply** (void \**src\_ptr*, `util::AllocationRecord` \**allocation*, int *value*, `std::size_t` *length*)  
Apply val to the first length bytes of src\_ptr.

Uses `std::memset` to set the first length bytes of src\_ptr to value.

#### Parameters

- `src_ptr`: Pointer to source memory location.
- `src_allocation`: AllocationRecord of source.
- `val`: Value to apply.
- `length`: Number of bytes to modify.

#### Exceptions

- `util::Exception`:

```
void transform (void *src_ptr, void **dst_ptr, util::AllocationRecord *src_allocation,  
               util::AllocationRecord *dst_allocation, std::size_t length)  
  Transform length bytes of memory from src_ptr to dst_ptr.
```

#### Parameters

- *src\_ptr*: Pointer to source memory location.
- *dst\_ptr*: Pointer to destination memory location.
- *src\_allocation*: *AllocationRecord* of source.
- *dst\_allocation*: *AllocationRecord* of destination.
- *length*: Number of bytes to transform.

#### Exceptions

- *util::Exception*:

### Class Replay

- Defined in file\_umpire\_Replay.hpp

### Class Documentation

```
class Replay
```

#### Public Functions

```
void logMessage (const std::string &message)
```

```
bool replayLoggingEnabled ()
```

```
uint64_t replayUid ()
```

#### Public Static Functions

```
Replay *getReplayLogger ()
```

```
static std::string printReplayAllocator (void)
```

```
template<typename T, typename ...Args>
```

```
static std::string printReplayAllocator (T &&firstArg, Args&&... args)
```

### Class CudaConstantMemoryResource

- Defined in file\_umpire\_resource\_CudaConstantMemoryResource.hpp

## Inheritance Relationships

### Base Type

- `public umpire::resource::MemoryResource` (*Class MemoryResource*)

### Class Documentation

**class** `CudaConstantMemoryResource` : **public** `umpire::resource::MemoryResource`

#### Public Functions

**CudaConstantMemoryResource** (**const** `std::string &name`, `int id`, *MemoryResourceTraits traits*)

`void *allocate` (`std::size_t bytes`)

Allocate bytes of memory.

This function is pure virtual and must be implemented by the inheriting class.

**Return** Pointer to start of allocation.

#### Parameters

- `bytes`: Number of bytes to allocate.

`void deallocate` (`void *ptr`)

Free the memory at `ptr`.

This function is pure virtual and must be implemented by the inheriting class.

#### Parameters

- `ptr`: Pointer to free.

`std::size_t getCurrentSize` () **const**

Return the current size of this *MemoryResource*.

This is sum of the sizes of all the tracked allocations. Note that this doesn't ever have to be equal to `getHighWatermark`.

**Return** current total size of active allocations in this *MemoryResource*.

`std::size_t getHighWatermark` () **const**

Return the memory high watermark for this *MemoryResource*.

This is the largest amount of memory allocated by this *Allocator*. Note that this may be larger than the largest value returned by `getCurrentSize`.

**Return** Memory high watermark.

*Platform* `getPlatform` ()

Get the Platform associated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

**Return** Platform associated with this *MemoryResource*.

*MemoryResourceTraits* **getTraits** ()

void **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

std::size\_t **getActualSize** () **const**

Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

**const** std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

*MemoryResourceTraits* **m\_traits**

std::string **m\_name**

int **m\_id**

### Class CudaConstantMemoryResourceFactory

- Defined in file\_umpire\_resource\_CudaConstantMemoryResourceFactory.hpp

### Inheritance Relationships

#### Base Type

- public umpire::resource::MemoryResourceFactory (*Class MemoryResourceFactory*)

## Class Documentation

**class CudaConstantMemoryResourceFactory** : public umpire::resource::MemoryResourceFactory  
Factory class for constructing *MemoryResource* objects that use GPU memory.

## Class CudaDeviceResourceFactory

- Defined in file\_umpire\_resource\_CudaDeviceResourceFactory.hpp

## Inheritance Relationships

### Base Type

- public umpire::resource::MemoryResourceFactory (Class *MemoryResourceFactory*)

## Class Documentation

**class CudaDeviceResourceFactory** : public umpire::resource::MemoryResourceFactory  
Factory class for constructing *MemoryResource* objects that use GPU memory.

## Class CudaPinnedMemoryResourceFactory

- Defined in file\_umpire\_resource\_CudaPinnedMemoryResourceFactory.hpp

## Inheritance Relationships

### Base Type

- public umpire::resource::MemoryResourceFactory (Class *MemoryResourceFactory*)

## Class Documentation

**class CudaPinnedMemoryResourceFactory** : public umpire::resource::MemoryResourceFactory

## Class CudaUnifiedMemoryResourceFactory

- Defined in file\_umpire\_resource\_CudaUnifiedMemoryResourceFactory.hpp

## Inheritance Relationships

### Base Type

- `public umpire::resource::MemoryResourceFactory` (*Class MemoryResourceFactory*)

### Class Documentation

**class CudaUnifiedMemoryResourceFactory** : **public** `umpire::resource::MemoryResourceFactory`  
Factory class to construct a *MemoryResource* that uses NVIDIA “unified” memory, accessible from both the CPU and NVIDIA GPUs.

### Template Class DefaultMemoryResource

- Defined in file `umpire_resource_DefaultMemoryResource.hpp`

## Inheritance Relationships

### Base Type

- `public umpire::resource::MemoryResource` (*Class MemoryResource*)

### Class Documentation

```
template<typename _allocator>
class DefaultMemoryResource : public umpire::resource::MemoryResource
    Concrete MemoryResource object that uses the template _allocator to allocate and deallocate memory.
```

#### Public Functions

**DefaultMemoryResource** (*Platform platform*, **const** `std::string &name`, `int id`, *MemoryResource-Traits traits*)

`void *allocate` (`std::size_t bytes`)  
Allocate bytes of memory.

This function is pure virtual and must be implemented by the inheriting class.

**Return** Pointer to start of allocation.

#### Parameters

- `bytes`: Number of bytes to allocate.

`void deallocate` (`void *ptr`)  
Free the memory at `ptr`.

This function is pure virtual and must be implemented by the inheriting class.

#### Parameters

- `ptr`: Pointer to free.

`std::size_t` **getCurrentSize** () **const**

Return the current size of this *MemoryResource*.

This is sum of the sizes of all the tracked allocations. Note that this doesn't ever have to be equal to `getHighWatermark`.

**Return** current total size of active allocations in this *MemoryResource*.

`std::size_t` **getHighWatermark** () **const**

Return the memory high watermark for this *MemoryResource*.

This is the largest amount of memory allocated by this *Allocator*. Note that this may be larger than the largest value returned by `getCurrentSize`.

**Return** Memory high watermark.

*Platform* **getPlatform** ()

Get the Platform associated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

**Return** Platform associated with this *MemoryResource*.

*MemoryResourceTraits* **getTraits** ()

`void` **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

`std::size_t` **getActualSize** () **const**

Get the current amount of memory allocated by this allocator.

Note that this can be larger than `getCurrentSize()`, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

**const** `std::string` &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

`int` **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

`_allocator` `m_allocator`

*Platform* `m_platform`

*MemoryResourceTraits* `m_traits`

`std::string` `m_name`

`int` `m_id`

### Class HipConstantMemoryResource

- Defined in file `umpire_resource_HipConstantMemoryResource.hpp`

### Inheritance Relationships

#### Base Type

- `public` `umpire::resource::MemoryResource` (*Class MemoryResource*)

### Class Documentation

```
class HipConstantMemoryResource : public umpire::resource::MemoryResource
```

#### Public Functions

```
HipConstantMemoryResource (const std::string &name, int id, MemoryResourceTraits traits)
```

```
void *allocate (std::size_t bytes)
```

Allocate bytes of memory.

This function is pure virtual and must be implemented by the inheriting class.

**Return** Pointer to start of allocation.

#### Parameters

- `bytes`: Number of bytes to allocate.

```
void deallocate (void *ptr)
```

Free the memory at `ptr`.

This function is pure virtual and must be implemented by the inheriting class.

#### Parameters

- `ptr`: Pointer to free.

```
std::size_t getCurrentSize () const
```

Return the current size of this *MemoryResource*.

This is sum of the sizes of all the tracked allocations. Note that this doesn't ever have to be equal to `getHighWatermark`.



**Return** current total size of active allocations in this *MemoryResource*.

`std::size_t getHighWatermark () const`

Return the memory high watermark for this *MemoryResource*.

This is the largest amount of memory allocated by this *Allocator*. Note that this may be larger than the largest value returned by `getCurrentSize`.

**Return** Memory high watermark.

*Platform* `getPlatform ()`

Get the Platform associated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

**Return** Platform associated with this *MemoryResource*.

*MemoryResourceTraits* `getTraits ()`

`void release ()`

Release any and all unused memory held by this *AllocationStrategy*.

`std::size_t getActualSize () const`

Get the current amount of memory allocated by this allocator.

Note that this can be larger than `getCurrentSize()`, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

`const std::string &getName ()`

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

`int getId ()`

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

## Protected Attributes

*MemoryResourceTraits* `m_traits`

`std::string m_name`

`int m_id`

## Class HipConstantMemoryResourceFactory

- Defined in file\_umpire\_resource\_HipConstantMemoryResourceFactory.hpp

### Inheritance Relationships

#### Base Type

- `public umpire::resource::MemoryResourceFactory` (Class *MemoryResourceFactory*)

### Class Documentation

**class HipConstantMemoryResourceFactory** : **public** umpire::resource::MemoryResourceFactory  
Factory class for constructing *MemoryResource* objects that use GPU memory.

## Class HipDeviceResourceFactory

- Defined in file\_umpire\_resource\_HipDeviceResourceFactory.hpp

### Inheritance Relationships

#### Base Type

- `public umpire::resource::MemoryResourceFactory` (Class *MemoryResourceFactory*)

### Class Documentation

**class HipDeviceResourceFactory** : **public** umpire::resource::MemoryResourceFactory  
Factory class for constructing *MemoryResource* objects that use GPU memory.

## Class HipPinnedMemoryResourceFactory

- Defined in file\_umpire\_resource\_HipPinnedMemoryResourceFactory.hpp

### Inheritance Relationships

#### Base Type

- `public umpire::resource::MemoryResourceFactory` (Class *MemoryResourceFactory*)

## Class Documentation

**class HipPinnedMemoryResourceFactory** : public `umpire::resource::MemoryResourceFactory`

## Class HostResourceFactory

- Defined in `file_umpire_resource_HostResourceFactory.hpp`

## Inheritance Relationships

### Base Type

- `public umpire::resource::MemoryResourceFactory` (*Class MemoryResourceFactory*)

## Class Documentation

**class HostResourceFactory** : public `umpire::resource::MemoryResourceFactory`  
 Factory class to construct a *MemoryResource* that uses CPU memory.

## Class MemoryResource

- Defined in `file_umpire_resource_MemoryResource.hpp`

## Inheritance Relationships

### Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

## Derived Types

- `public umpire::resource::CudaConstantMemoryResource` (*Class CudaConstantMemoryResource*)
- `public umpire::resource::DefaultMemoryResource< _allocator >` (*Template Class DefaultMemoryResource*)
- `public umpire::resource::HipConstantMemoryResource` (*Class HipConstantMemoryResource*)
- `public umpire::resource::NullMemoryResource` (*Class NullMemoryResource*)

## Class Documentation

**class MemoryResource** : public `umpire::strategy::AllocationStrategy`

Base class to represent the available hardware resources for memory allocation in the system.

Objects of this inherit from `strategy::AllocationStrategy`, allowing them to be used directly.

Subclassed by `umpire::resource::CudaConstantMemoryResource`, `umpire::resource::DefaultMemoryResource<_allocator>`, `umpire::resource::HipConstantMemoryResource`, `umpire::resource::NullMemoryResource`

### Public Functions

**MemoryResource** (`const std::string &name`, `int id`, `MemoryResourceTraits traits`)

Construct a *MemoryResource* with the given name and id.

#### Parameters

- `name`: Name of the *MemoryResource*.
- `id`: ID of the *MemoryResource* (must be unique).

**virtual ~MemoryResource** ()

**virtual void \*allocate** (`std::size_t bytes`) = 0

Allocate bytes of memory.

This function is pure virtual and must be implemented by the inheriting class.

**Return** Pointer to start of allocation.

#### Parameters

- `bytes`: Number of bytes to allocate.

**virtual void deallocate** (`void *ptr`) = 0

Free the memory at `ptr`.

This function is pure virtual and must be implemented by the inheriting class.

#### Parameters

- `ptr`: Pointer to free.

**virtual std::size\_t getCurrentSize** () **const** = 0

Return the current size of this *MemoryResource*.

This is sum of the sizes of all the tracked allocations. Note that this doesn't ever have to be equal to `getHighWatermark`.

**Return** current total size of active allocations in this *MemoryResource*.

**virtual std::size\_t getHighWatermark** () **const** = 0

Return the memory high watermark for this *MemoryResource*.

This is the largest amount of memory allocated by this *Allocator*. Note that this may be larger than the largest value returned by `getCurrentSize`.

**Return** Memory high watermark.

**virtual** *Platform* **getPlatform** () = 0

Get the Platform associated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

**Return** Platform associated with this *MemoryResource*.

*MemoryResourceTraits* **getTraits** ()

void **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

std::size\_t **getActualSize** () **const**

Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

**const** std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

*MemoryResourceTraits* **m\_traits**

std::string **m\_name**

int **m\_id**

### Class MemoryResourceFactory

- Defined in file\_umpire\_resource\_MemoryResourceFactory.hpp

### Inheritance Relationships

### Derived Types

- public `umpire::resource::CudaConstantMemoryResourceFactory` (Class *CudaConstantMemoryResourceFactory*)
- public `umpire::resource::CudaDeviceResourceFactory` (Class *CudaDeviceResourceFactory*)
- public `umpire::resource::CudaPinnedMemoryResourceFactory` (Class *CudaPinnedMemoryResourceFactory*)

- `public umpire::resource::CudaUnifiedMemoryResourceFactory` (Class *CudaUnifiedMemoryResourceFactory*)
- `public umpire::resource::HipConstantMemoryResourceFactory` (Class *HipConstantMemoryResourceFactory*)
- `public umpire::resource::HipDeviceResourceFactory` (Class *HipDeviceResourceFactory*)
- `public umpire::resource::HipPinnedMemoryResourceFactory` (Class *HipPinnedMemoryResourceFactory*)
- `public umpire::resource::HostResourceFactory` (Class *HostResourceFactory*)
- `public umpire::resource::NullMemoryResourceFactory` (Class *NullMemoryResourceFactory*)
- `public umpire::resource::RocmDeviceResourceFactory` (Class *RocmDeviceResourceFactory*)
- `public umpire::resource::RocmPinnedMemoryResourceFactory` (Class *RocmPinnedMemoryResourceFactory*)

## Class Documentation

### class `MemoryResourceFactory`

Abstract factory class for constructing *MemoryResource* objects.

Concrete implementations of this class are used by the *MemoryResourceRegistry* to construct *MemoryResource* objects.

See *MemoryResourceRegistry*

Subclassed by *umpire::resource::CudaConstantMemoryResourceFactory*, *umpire::resource::CudaDeviceResourceFactory*, *umpire::resource::CudaPinnedMemoryResourceFactory*, *umpire::resource::CudaUnifiedMemoryResourceFactory*, *umpire::resource::HipConstantMemoryResourceFactory*, *umpire::resource::HipDeviceResourceFactory*, *umpire::resource::HipPinnedMemoryResourceFactory*, *umpire::resource::HostResourceFactory*, *umpire::resource::NullMemoryResourceFactory*, *umpire::resource::RocmDeviceResourceFactory*, *umpire::resource::RocmPinnedMemoryResourceFactory*

### Public Functions

`virtual ~MemoryResourceFactory ()`

`virtual bool isValidMemoryResourceFor (const std::string &name) = 0`

`virtual std::unique_ptr<resource::MemoryResource> create (const std::string &name, int id) = 0`  
Construct a *MemoryResource* with the given name and id.

#### Parameters

- `name`: Name of the *MemoryResource*.
- `id`: ID of the *MemoryResource*.

## Class MemoryResourceRegistry

- Defined in file\_umpire\_resource\_MemoryResourceRegistry.hpp

### Class Documentation

```
class MemoryResourceRegistry
```

#### Public Functions

```
std::unique_ptr<resource::MemoryResource> makeMemoryResource (const std::string &name, int
                                                                id)
```

```
void registerMemoryResource (std::unique_ptr<MemoryResourceFactory> &&factory)
```

```
MemoryResourceRegistry (const MemoryResourceRegistry&)
```

```
MemoryResourceRegistry &operator= (const MemoryResourceRegistry&)
```

```
~MemoryResourceRegistry ()
```

#### Public Static Functions

```
MemoryResourceRegistry &getInstance ()
```

## Class NullMemoryResource

- Defined in file\_umpire\_resource\_NullMemoryResource.hpp

### Inheritance Relationships

#### Base Type

- public umpire::resource::MemoryResource (*Class MemoryResource*)

### Class Documentation

```
class NullMemoryResource : public umpire::resource::MemoryResource
```

#### Public Functions

```
NullMemoryResource (Platform platform, const std::string &name, int id, MemoryResourceTraits
                    traits)
```

```
void *allocate (std::size_t bytes)
    Allocate bytes of memory.
```

This function is pure virtual and must be implemented by the inheriting class.

**Return** Pointer to start of allocation.

**Parameters**

- `bytes`: Number of bytes to allocate.

void **deallocate** (void \**ptr*)

Free the memory at *ptr*.

This function is pure virtual and must be implemented by the inheriting class.

**Parameters**

- `ptr`: Pointer to free.

std::size\_t **getCurrentSize** () **const**

Return the current size of this *MemoryResource*.

This is sum of the sizes of all the tracked allocations. Note that this doesn't ever have to be equal to `getHighWatermark`.

**Return** current total size of active allocations in this *MemoryResource*.

std::size\_t **getHighWatermark** () **const**

Return the memory high watermark for this *MemoryResource*.

This is the largest amount of memory allocated by this *Allocator*. Note that this may be larger than the largest value returned by `getCurrentSize`.

**Return** Memory high watermark.

*Platform* **getPlatform** ()

Get the Platform associated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

**Return** Platform associated with this *MemoryResource*.

*MemoryResourceTraits* **getTraits** ()

void **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

std::size\_t **getActualSize** () **const**

Get the current amount of memory allocated by this allocator.

Note that this can be larger than `getCurrentSize()`, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

**const** std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.



```
int getId ()  
    Get the id of this AllocationStrategy.  
  
    Return The id of this AllocationStrategy.
```

### Protected Attributes

```
Platform m_platform  
MemoryResourceTraits m_traits  
std::string m_name  
int m_id
```

### Class NullMemoryResourceFactory

- Defined in file\_umpire\_resource\_NullMemoryResourceFactory.hpp

### Inheritance Relationships

#### Base Type

- public umpire::resource::MemoryResourceFactory (*Class MemoryResourceFactory*)

### Class Documentation

```
class NullMemoryResourceFactory : public umpire::resource::MemoryResourceFactory  
    Factory class for constructing MemoryResource objects that use GPU memory.
```

### Class RocmDeviceResourceFactory

- Defined in file\_umpire\_resource\_RocmDeviceResourceFactory.hpp

### Inheritance Relationships

#### Base Type

- public umpire::resource::MemoryResourceFactory (*Class MemoryResourceFactory*)

## Class Documentation

**class RocmDeviceResourceFactory** : public umpire::resource::MemoryResourceFactory  
Factory class for constructing *MemoryResource* objects that use GPU memory.

## Class RocmPinnedMemoryResourceFactory

- Defined in file\_umpire\_resource\_RocmPinnedMemoryResourceFactory.hpp

## Inheritance Relationships

### Base Type

- public umpire::resource::MemoryResourceFactory (*Class MemoryResourceFactory*)

## Class Documentation

**class RocmPinnedMemoryResourceFactory** : public umpire::resource::MemoryResourceFactory  
Factory class for constructing *MemoryResource* objects that use GPU memory.

## Class ResourceManager

- Defined in file\_umpire\_ResourceManager.hpp

## Class Documentation

**class ResourceManager**

### Public Functions

void **initialize** ()

Initialize the *ResourceManager*.

This will create all registered *MemoryResource* objects

std::vector<std::string> **getAllocatorNames** () const

Get the names of all available *Allocator* objects.

std::vector<int> **getAllocatorIds** () const

Get the ids of all available *Allocator* objects.

*Allocator* **getAllocator** (const std::string &name)

Get the *Allocator* with the given name.

*Allocator* **getAllocator** (const char \*name)

*Allocator* **getAllocator** (resource::MemoryResourceType resource\_type)

Get the default *Allocator* for the given resource\_type.

*Allocator* **getAllocator** (int *id*)  
Get the *Allocator* with the given ID.

*Allocator* **getDefaultAllocator** ()  
Get the default *Allocator*.

The default *Allocator* is used whenever an *Allocator* is required and one is not provided, or cannot be inferred.

**Return** The default *Allocator*.

void **setDefaultAllocator** (*Allocator* *allocator*)  
Set the default *Allocator*.

The default *Allocator* is used whenever an *Allocator* is required and one is not provided, or cannot be inferred.

#### Parameters

- *allocator*: The *Allocator* to use as the default.

template<typename **Strategy**, bool **introspection** = true, typename ...**Args**>  
*Allocator* **makeAllocator** (const std::string &*name*, *Args*&&... *args*)  
Construct a new *Allocator*.

void **registerAllocator** (const std::string &*name*, *Allocator* *allocator*)  
Register an *Allocator* with the *ResourceManager*.

After registration, the *Allocator* can be retrieved by calling `getAllocator(name)`.

The same *Allocator* can be registered under multiple names.

#### Parameters

- *name*: Name to register *Allocator* with.
- *allocator*: *Allocator* to register.

*Allocator* **getAllocator** (void \**ptr*)  
Get the *Allocator* used to allocate *ptr*.

**Return** *Allocator* for the given *ptr*.

#### Parameters

- *ptr*: Pointer to find the *Allocator* for.

bool **isAllocator** (const std::string &*name*)

bool **hasAllocator** (void \**ptr*)  
Does the given pointer have an associated *Allocator*.

**Return** True if the pointer has an associated *Allocator*.

void **registerAllocation** (void \**ptr*, util::AllocationRecord *record*)  
register an allocation with the manager.

util::*AllocationRecord* **deregisterAllocation** (void \*ptr)  
de-register the address ptr with the manager.

**Return** the allocation record removed from the manager.

const util::*AllocationRecord* \***findAllocationRecord** (void \*ptr) const  
Find the allocation record associated with an address ptr.

**Return** the record if found, or throws an exception if not found.

bool **isAllocatorRegistered** (const std::string &name)  
Check whether the named *Allocator* exists.

void **copy** (void \*dst\_ptr, void \*src\_ptr, std::size\_t size = 0)  
Copy size bytes of data from src\_ptr to dst\_ptr.

Both the src\_ptr and dst\_ptr addresses must be allocated by Umpire. They can be offset from any Umpire-managed base address.

The dst\_ptr must be large enough to accommodate size bytes of data.

#### Parameters

- dst\_ptr: Destination pointer.
- src\_ptr: Source pointer.
- size: Size in bytes.

void **memset** (void \*ptr, int val, std::size\_t length = 0)  
Set the first length bytes of ptr to the value val.

#### Parameters

- ptr: Pointer to data.
- val: Value to set.
- length: Number of bytes to set to val.

void \***reallocate** (void \*current\_ptr, std::size\_t new\_size)  
Reallocate current\_ptr to new\_size.

If current\_ptr is nullptr, then the default allocator will be used to allocate data. The default allocator may be set with a call to *setDefaultAllocator(Allocator allocator)*.

#### Parameters

- current\_ptr: Source pointer to reallocate.
- new\_size: New size of pointer.

NOTE 1: This is not thread safe NOTE 2: If the allocator for which current\_ptr is intended is different from the default allocator, then all subsequent reallocate calls will result in allocations from the default allocator which may not be the intended behavior.

If new\_size is 0, then the current\_ptr will be deallocated if it is not a nullptr, and a zero-byte allocation will be returned.

**Return** Reallocated pointer.

void **\*reallocate** (void *\*current\_ptr*, std::size\_t *new\_size*, *Allocator allocator*)  
 Reallocate *current\_ptr* to *new\_size*.

If *current\_ptr* is null, then *allocator* will be used to allocate the data.

#### Parameters

- *current\_ptr*: Source pointer to reallocate.
- *new\_size*: New size of pointer.
- *allocator*: *Allocator* to use if *current\_ptr* is null.

If *new\_size* is 0, then the *current\_ptr* will be deallocated if it is not a nullptr, and a zero-byte allocation will be returned.

**Return** Reallocated pointer.

void **\*move** (void *\*src\_ptr*, *Allocator allocator*)  
 Move *src\_ptr* to memory from *allocator*.

**Return** Pointer to new location of data.

#### Parameters

- *src\_ptr*: Pointer to move.
- *allocator*: *Allocator* to use to allocate new memory for moved data.

void **deallocate** (void *\*ptr*)  
 Deallocate any pointer allocated by an Umpire-managed resource.

#### Parameters

- *ptr*: Pointer to deallocate.

std::size\_t **getSize** (void *\*ptr*) **const**  
 Get the size in bytes of the allocation for the given pointer.

**Return** Size of allocation in bytes.

#### Parameters

- *ptr*: Pointer to find size of.

std::shared\_ptr<op::MemoryOperation> **getOperation** (**const** std::string &*operation\_name*, *Allocator src\_allocator*, *Allocator dst\_allocator*)

**~ResourceManager** ()

**ResourceManager** (**const** *ResourceManager*&)

*ResourceManager* &**operator=** (**const** *ResourceManager*&)

## Public Static Functions

*ResourceManager* &**getInstance** ()

## Class AllocationAdvisor

- Defined in file\_umpire\_strategy\_AllocationAdvisor.hpp

## Inheritance Relationships

### Base Type

- public `umpire::strategy::AllocationStrategy` (Class *AllocationStrategy*)

## Class Documentation

**class AllocationAdvisor** : public `umpire::strategy::AllocationStrategy`  
Applies the given MemoryOperation to every allocation.

This *AllocationStrategy* is designed to be used with the following operations:

- *op::CudaAdviseAccessedByOperation*
- *op::CudaAdvisePreferredLocationOperation*
- *op::CudaAdviseReadMostlyOperation*

Using this *AllocationStrategy* when combined with a pool like `DynamicPool` is a good way to mitigate the overhead of applying the memory advice.

## Public Functions

**AllocationAdvisor** (**const** `std::string &name`, `int id`, *Allocator* `allocator`, **const** `std::string &advice_operation`, `int device_id = 0`)

**AllocationAdvisor** (**const** `std::string &name`, `int id`, *Allocator* `allocator`, **const** `std::string &advice_operation`, *Allocator* `accessing_allocator`, `int device_id = 0`)

`void *allocate` (`std::size_t bytes`)  
Allocate bytes of memory.

**Return** Pointer to start of allocated bytes.

### Parameters

- `bytes`: Number of bytes to allocate.

`void deallocate` (`void *ptr`)  
Free the memory at `ptr`.

### Parameters

- `ptr`: Pointer to free.

`std::size_t getCurrentSize () const`

Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

`std::size_t getHighWatermark () const`

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

*Platform* `getPlatform ()`

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

`void release ()`

Release any and all unused memory held by this *AllocationStrategy*.

`std::size_t getActualSize () const`

Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

`const std::string &getName ()`

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

`int getId ()`

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

`std::string m_name`

`int m_id`

## Class AllocationPrefetcher

- Defined in file\_umpire\_strategy\_AllocationPrefetcher.hpp

## Inheritance Relationships

### Base Type

- public `umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

## Class Documentation

**class AllocationPrefetcher** : public `umpire::strategy::AllocationStrategy`

Apply the appropriate “PREFETCH” operation to every allocation.

### Public Functions

**AllocationPrefetcher** (`const` `std::string` &*name*, `int` *id*, *Allocator* *allocator*, `int` *device\_id* = 0)

`void` \***allocate** (`std::size_t` *bytes*)  
Allocate bytes of memory.

**Return** Pointer to start of allocated bytes.

#### Parameters

- *bytes*: Number of bytes to allocate.

`void` **deallocate** (`void` \**ptr*)  
Free the memory at *ptr*.

#### Parameters

- *ptr*: Pointer to free.

`std::size_t` **getCurrentSize** () `const`  
Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

`std::size_t` **getHighWatermark** () `const`  
Get the high watermark of the total allocated size.  
This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

*Platform* **getPlatform** ()  
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.



**Return** The platform associated with this *AllocationStrategy*.

void **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

std::size\_t **getActualSize** () const

Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

const std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

std::string **m\_name**

int **m\_id**

### Class AllocationStrategy

- Defined in file\_umpire\_strategy\_AllocationStrategy.hpp

### Inheritance Relationships

### Derived Types

- public umpire::resource::MemoryResource (*Class MemoryResource*)
- public umpire::strategy::AllocationAdvisor (*Class AllocationAdvisor*)
- public umpire::strategy::AllocationPrefetcher (*Class AllocationPrefetcher*)
- public umpire::strategy::AllocationTracker (*Class AllocationTracker*)
- public umpire::strategy::DynamicPoolList (*Class DynamicPoolList*)
- public umpire::strategy::DynamicPoolMap (*Class DynamicPoolMap*)
- public umpire::strategy::FixedPool (*Class FixedPool*)
- public umpire::strategy::MixedPool (*Class MixedPool*)
- public umpire::strategy::MonotonicAllocationStrategy (*Class MonotonicAllocationStrategy*)

- `public umpire::strategy::NamedAllocationStrategy` (*Class NamedAllocationStrategy*)
- `public umpire::strategy::NumaPolicy` (*Class NumaPolicy*)
- `public umpire::strategy::SizeLimiter` (*Class SizeLimiter*)
- `public umpire::strategy::SlotPool` (*Class SlotPool*)
- `public umpire::strategy::ThreadSafeAllocator` (*Class ThreadSafeAllocator*)
- `public umpire::strategy::ZeroByteHandler` (*Class ZeroByteHandler*)

## Class Documentation

### class AllocationStrategy

*AllocationStrategy* provides a unified interface to all classes that can be used to allocate and free data.

Subclassed by `umpire::resource::MemoryResource`, `umpire::strategy::AllocationAdvisor`, `umpire::strategy::AllocationPrefetcher`, `umpire::strategy::AllocationTracker`, `umpire::strategy::DynamicPoolList`, `umpire::strategy::DynamicPoolMap`, `umpire::strategy::FixedPool`, `umpire::strategy::MixedPool`, `umpire::strategy::MonotonicAllocationStrategy`, `umpire::strategy::NamedAllocationStrategy`, `umpire::strategy::NumaPolicy`, `umpire::strategy::SizeLimiter`, `umpire::strategy::SlotPool`, `umpire::strategy::ThreadSafeAllocator`, `umpire::strategy::ZeroByteHandler`

### Public Functions

**AllocationStrategy** (`const std::string &name`, `int id`)

Construct a new *AllocationStrategy* object.

All *AllocationStrategy* objects must will have a unique name and id. This uniqueness is enforced by the *ResourceManager*.

#### Parameters

- `name`: The name of this *AllocationStrategy* object.
- `id`: The id of this *AllocationStrategy* object.

**virtual ~AllocationStrategy** ()

**virtual void \*allocate** (`std::size_t bytes`) = 0

Allocate bytes of memory.

**Return** Pointer to start of allocated bytes.

#### Parameters

- `bytes`: Number of bytes to allocate.

**virtual void deallocate** (`void *ptr`) = 0

Free the memory at `ptr`.

#### Parameters

- `ptr`: Pointer to free.

**void release** ()

Release any and all unused memory held by this *AllocationStrategy*.

**virtual** std::size\_t **getCurrentSize** () **const** = 0

Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

**virtual** std::size\_t **getHighWatermark** () **const** = 0

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

std::size\_t **getActualSize** () **const**

Get the current amount of memory allocated by this allocator.

Note that this can be larger than `getCurrentSize()`, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

**virtual** Platform **getPlatform** () = 0

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

**const** std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

std::string **m\_name**

int **m\_id**

## Friends

`std::ostream &operator<<` (`std::ostream &os`, `const AllocationStrategy &strategy`)

## Class AllocationTracker

- Defined in file `umpire_strategy_AllocationTracker.hpp`

## Inheritance Relationships

### Base Types

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)
- `private umpire::strategy::mixins::Inspector` (*Class Inspector*)

## Class Documentation

```
class AllocationTracker : public umpire::strategy::AllocationStrategy, private umpire::strategy::mixins::Inspector
```

### Public Functions

```
AllocationTracker (std::unique_ptr<AllocationStrategy> &&allocator)
```

```
void *allocate (std::size_t bytes)  
    Allocate bytes of memory.
```

**Return** Pointer to start of allocated bytes.

#### Parameters

- `bytes`: Number of bytes to allocate.

```
void deallocate (void *ptr)  
    Free the memory at ptr.
```

#### Parameters

- `ptr`: Pointer to free.

```
void release ()  
    Release any and all unused memory held by this AllocationStrategy.
```

```
std::size_t getCurrentSize () const  
    Get current (total) size of the allocated memory.
```

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

std::size\_t **getHighWatermark** () **const**

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

std::size\_t **getActualSize** () **const**

Get the current amount of memory allocated by this allocator.

Note that this can be larger than `getCurrentSize()`, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

*Platform* **getPlatform** ()

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

strategy::AllocationStrategy \***getAllocationStrategy** ()

**const** std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

## Protected Attributes

std::string **m\_name**

int **m\_id**

## Private Functions

void **registerAllocation** (void \**ptr*, std::size\_t *size*, strategy::AllocationStrategy \**strategy*)

util::AllocationRecord **deregisterAllocation** (void \**ptr*, strategy::AllocationStrategy \**strategy*)

## Private Members

`std::size_t m_current_size`  
`std::size_t m_high_watermark`

## Class DynamicPoolList

- Defined in file `umpire_strategy_DynamicPoolList.hpp`

## Inheritance Relationships

### Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

## Class Documentation

**class** `DynamicPoolList` : **public** `umpire::strategy::AllocationStrategy`  
Simple dynamic pool for allocations.

This *AllocationStrategy* uses Simpool to provide pooling for allocations of any size. The behavior of the pool can be controlled by two parameters: the initial allocation size, and the minimum allocation size.

The initial size controls how large the first piece of memory allocated is, and the minimum size controls the lower bound on all future chunk allocations.

## Public Types

**using** `CoalesceHeuristic` = `std::function<bool (const strategy::DynamicPoolList&) >`  
Callback Heuristic to trigger coalesce of free blocks in pool.

The registered heuristic callback function will be called immediately after a `deallocation()` has completed from the pool.

## Public Functions

`DynamicPoolList` (**const** `std::string &name`, `int id`, *Allocator* `allocator`, **const** `std::size_t min_initial_alloc_size = (512 * 1024 * 1024)`, **const** `std::size_t min_alloc_size = (1 * 1024 * 1024)`, *CoalesceHeuristic* `coalesce_heuristic = heuristic_percent_releasable_list(100)`)

Construct a new *DynamicPoolList*.

### Parameters

- `name`: Name of this instance of the *DynamicPoolList*.
- `id`: Id of this instance of the *DynamicPoolList*.
- `min_initial_alloc_size`: The minimum size of the first allocation the pool will make.
- `min_alloc_size`: The minimum size of all future allocations.
- `coalesce_heuristic`: Heuristic callback function.

void **\*allocate** (size\_t *bytes*)

void **deallocate** (void *\*ptr*)  
Free the memory at ptr.

#### Parameters

- *ptr*: Pointer to free.

void **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

std::size\_t **getCurrentSize** () **const**

Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

std::size\_t **getActualSize** () **const**

Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

std::size\_t **getHighWatermark** () **const**

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of *getCurrentSize*.

**Return** High watermark allocation size.

*Platform* **getPlatform** ()

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

std::size\_t **getReleasableSize** () **const**

Get the number of bytes that may be released back to resource.

A memory pool has a set of blocks that have no allocations against them. If the size of the set is greater than one, then the pool will have a number of bytes that may be released back to the resource or coalesced into a larger block.

**Return** The total number of bytes that are releasable

std::size\_t **getBlocksInPool** () **const**

Get the number of memory blocks that the pool has.

**Return** The total number of blocks that are allocated by the pool

`std::size_t getLargestAvailableBlock () const`

Get the largest allocatable number of bytes from pool before the pool will grow.

return The largest number of bytes that may be allocated without causing pool growth

void `coalesce ()`

`virtual void *allocate (std::size_t bytes) = 0`

Allocate bytes of memory.

**Return** Pointer to start of allocated bytes.

**Parameters**

- `bytes`: Number of bytes to allocate.

`const std::string &getName ()`

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

`int getId ()`

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

`std::string m_name`

`int m_id`

### Class DynamicPoolMap

- Defined in file\_umpire\_strategy\_DynamicPoolMap.hpp

### Inheritance Relationships

#### Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

### Class Documentation

`class DynamicPoolMap : public umpire::strategy::AllocationStrategy`

Simple dynamic pool for allocations.

This *AllocationStrategy* uses Simpool to provide pooling for allocations of any size. The behavior of the pool can be controlled by two parameters: the initial allocation size, and the minimum allocation size.

The initial size controls how large the first piece of memory allocated is, and the minimum size controls the lower bound on all future chunk allocations.



## Public Types

**using Pointer** = void \*

**using CoalesceHeuristic** = std::function<bool (const strategy::DynamicPoolMap&) >  
 Callback heuristic to trigger coalesce of free blocks in pool.

The registered heuristic callback function will be called immediately after a deallocation() has completed from the pool.

## Public Functions

**DynamicPoolMap** (const std::string &name, int id, Allocator allocator, const std::size\_t initial\_alloc\_size = (512 \* 1024 \* 1024), const std::size\_t min\_alloc\_size = (1 \* 1024 \* 1024), const std::size\_t align\_bytes = 16, CoalesceHeuristic coalesce\_heuristic = heuristic\_percent\_releasable(100))

Construct a new *DynamicPoolMap*.

### Parameters

- name: Name of this instance of the *DynamicPoolMap*
- id: Unique identifier for this instance
- initial\_alloc\_bytes: Size the pool initially allocates
- min\_alloc\_bytes: The minimum size of all future allocations
- coalesce\_heuristic: Heuristic callback function
- align\_bytes: Number of bytes with which to align allocation sizes

**~DynamicPoolMap** ()

Destructs the *DynamicPoolMap*.

**DynamicPoolMap** (const *DynamicPoolMap*&)

void \***allocate** (std::size\_t bytes)

Allocate bytes of memory.

**Return** Pointer to start of allocated bytes.

### Parameters

- bytes: Number of bytes to allocate.

void **deallocate** (void \*ptr)

Free the memory at ptr.

### Parameters

- ptr: Pointer to free.

void **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

`std::size_t getCurrentSize () const`

Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

`std::size_t getActualSize () const`

Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

`std::size_t getHighWatermark () const`

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of *getCurrentSize*.

**Return** High watermark allocation size.

*Platform* `getPlatform ()`

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

`std::size_t getReleasableSize () const`

Returns the number of bytes of unallocated data held by this pool that could be immediately released back to the resource.

A memory pool has a set of blocks that are not leased out to the application as allocations. Allocations from the resource begin as a single chunk, but these could be split, and only the first chunk can be deallocated back to the resource immediately.

**Return** The total number of bytes that are immediately releasable.

`std::size_t getFreeBlocks () const`

Return the number of free memory blocks that the pools holds.

`std::size_t getInUseBlocks () const`

Return the number of used memory blocks that the pools holds.

`std::size_t getBlocksInPool () const`

Return the number of memory blocks both leased to application and internal free memory that the pool holds.

`std::size_t getLargestAvailableBlock ()`

Get the largest allocatable number of bytes from pool before the pool will grow.

return The largest number of bytes that may be allocated without causing pool growth

`void coalesce ()`

Merge as many free records as possible, release all possible free blocks, then reallocate a chunk to keep the actual size the same.

**const** std::string &**getName** ()  
Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()  
Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

std::string **m\_name**

int **m\_id**

### Class FixedPool

- Defined in file\_umpire\_strategy\_FixedPool.hpp

### Nested Relationships

#### Nested Types

- *Struct FixedPool::Pool*

### Inheritance Relationships

#### Base Type

- public umpire::strategy::AllocationStrategy (*Class AllocationStrategy*)

### Class Documentation

**class FixedPool** : public umpire::strategy::AllocationStrategy

Pool for fixed size allocations.

This *AllocationStrategy* provides an efficient pool for fixed size allocations, and used to quickly allocate and deallocate objects.

## Public Functions

**FixedPool** (**const** std::string &name, int id, *Allocator* allocator, **const** std::size\_t object\_bytes, **const** std::size\_t objects\_per\_pool = 64 \* sizeof(int) \* 8)  
Constructs a *FixedPool*.

### Parameters

- name: The allocator name for reference later in *ResourceManager*
- id: The allocator id for reference later in *ResourceManager*
- allocator: Used for data allocation. It uses std::malloc for internal tracking of these allocations.
- object\_bytes: The fixed size (in bytes) for each allocation
- objects\_per\_pool: Number of objects in each sub-pool internally. Performance likely improves if this is large, at the cost of memory usage. This does not have to be a multiple of sizeof(int)\*8, but it will also likely improve performance if so.

**~FixedPool** ()

**FixedPool** (**const** *FixedPool*&)

void \***allocate** (std::size\_t bytes = 0)  
Allocate bytes of memory.

**Return** Pointer to start of allocated bytes.

### Parameters

- bytes: Number of bytes to allocate.

void **deallocate** (void \*ptr)  
Free the memory at ptr.

### Parameters

- ptr: Pointer to free.

std::size\_t **getCurrentSize** () **const**  
Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

std::size\_t **getHighWatermark** () **const**  
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

std::size\_t **getActualSize** () **const**  
Get the current amount of memory allocated by this allocator.

Note that this can be larger than `getCurrentSize()`, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

*Platform* **getPlatform** ()

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

bool **pointerIsFromPool** (void \*ptr) **const**

std::size\_t **numPools** () **const**

void **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

**const** std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

std::string **m\_name**

int **m\_id**

### Class MixedPool

- Defined in file\_umpire\_strategy\_MixedPool.hpp

### Inheritance Relationships

#### Base Type

- public umpire::strategy::AllocationStrategy (*Class AllocationStrategy*)

## Class Documentation

**class MixedPool** : public `umpire::strategy::AllocationStrategy`

A faster pool that pulls from a series of pools.

Pool implementation using a series of `FixedPools` for small sizes, and a `DynamicPool` for sizes larger than  $(1 \llcorner \llcorner \text{LastFixed})$  bytes.

### Public Functions

**MixedPool**(**const** `std::string` &name, `int` id, *Allocator* allocator, `std::size_t` *smallest\_fixed\_obj\_size* =  $(1 \llcorner \llcorner 8)$ , `std::size_t` *largest\_fixed\_obj\_size* =  $(1 \llcorner \llcorner 17)$ , `std::size_t` *max\_initial\_fixed\_pool\_size* =  $1024 * 1024 * 2$ , `std::size_t` *fixed\_size\_multiplier* = 16, **const** `std::size_t` *dynamic\_initial\_alloc\_size* =  $(512 * 1024 * 1024)$ , **const** `std::size_t` *dynamic\_min\_alloc\_size* =  $(1 * 1024 * 1024)$ , **const** `std::size_t` *dynamic\_align\_bytes* = 16, *DynamicPoolMap::CoalesceHeuristic* *dynamic\_coalesce\_heuristic* = *heuristic\_percent\_releasable*(100))

Creates a *MixedPool* of one or more fixed pools and a dynamic pool for large allocations.

#### Parameters

- name: Name of the pool
- id: Unique identifier for lookup later in *ResourceManager*
- allocator: Underlying allocator
- smallest\_fixed\_obj\_size: Smallest fixed pool object size in bytes
- largest\_fixed\_obj\_size: Largest fixed pool object size in bytes
- max\_initial\_fixed\_pool\_size: Largest initial size of any fixed pool
- fixed\_size\_multiplier: Fixed pool object size increase factor
- dynamic\_initial\_alloc\_size: Size the dynamic pool initially allocates
- dynamic\_min\_alloc\_bytes: Minimum size of all future allocations in the dynamic pool
- dynamic\_align\_bytes: Size with which to align allocations (for the dynamic pool)
- coalesce\_heuristic: Heuristic callback function (for the dynamic pool)

`void *allocate` (`std::size_t` bytes)

Allocate bytes of memory.

**Return** Pointer to start of allocated bytes.

#### Parameters

- bytes: Number of bytes to allocate.

`void deallocate` (`void *ptr`)

Free the memory at ptr.

#### Parameters

- ptr: Pointer to free.

void **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

std::size\_t **getCurrentSize** () const

Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

std::size\_t **getActualSize** () const

Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

std::size\_t **getHighWatermark** () const

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of *getCurrentSize*.

**Return** High watermark allocation size.

*Platform* **getPlatform** ()

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

const std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

std::string **m\_name**

int **m\_id**

## Class Inspector

- Defined in file\_umpire\_strategy\_mixins\_Inspector.hpp

## Inheritance Relationships

### Derived Type

- `private` `umpire::strategy::AllocationTracker` (*Class AllocationTracker*)

## Class Documentation

### class Inspector

Subclassed by `umpire::strategy::AllocationTracker`

### Public Functions

`Inspector()`

`void registerAllocation` (`void *ptr`, `std::size_t size`, `strategy::AllocationStrategy *strategy`)

`util::AllocationRecord deregisterAllocation` (`void *ptr`, `strategy::AllocationStrategy *strategy`)

### Protected Attributes

`std::size_t m_current_size`

`std::size_t m_high_watermark`

## Class MonotonicAllocationStrategy

- Defined in file\_umpire\_strategy\_MonotonicAllocationStrategy.hpp

## Inheritance Relationships

### Base Type

- `public` `umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)



## Class Documentation

**class MonotonicAllocationStrategy** : public `umpire::strategy::AllocationStrategy`

### Public Functions

**MonotonicAllocationStrategy** (`const` `std::string` &*name*, `int` *id*, *Allocator* *allocator*, `std::size_t` *capacity*)

**~MonotonicAllocationStrategy** ()

`void` \***allocate** (`std::size_t` *bytes*)

Allocate bytes of memory.

**Return** Pointer to start of allocated bytes.

#### Parameters

- *bytes*: Number of bytes to allocate.

`void` **deallocate** (`void` \**ptr*)

Free the memory at *ptr*.

#### Parameters

- *ptr*: Pointer to free.

`std::size_t` **getCurrentSize** () `const`

Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

`std::size_t` **getHighWatermark** () `const`

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

*Platform* **getPlatform** ()

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

`void` **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

`std::size_t` **getActualSize** () `const`

Get the current amount of memory allocated by this allocator.

Note that this can be larger than `getCurrentSize()`, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

`const std::string &getName ()`  
Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

`int getId ()`  
Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

`std::string m_name`

`int m_id`

### Class NamedAllocationStrategy

- Defined in file\_umpire\_strategy\_NamedAllocationStrategy.hpp

### Inheritance Relationships

#### Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

### Class Documentation

```
class NamedAllocationStrategy : public umpire::strategy::AllocationStrategy
```

#### Public Functions

```
NamedAllocationStrategy (const std::string &name, int id, Allocator allocator)
```

```
void *allocate (std::size_t bytes)  
Allocate bytes of memory.
```

**Return** Pointer to start of allocated bytes.

#### Parameters

- `bytes`: Number of bytes to allocate.

```
void deallocate (void *ptr)  
Free the memory at ptr.
```

#### Parameters

- `ptr`: Pointer to free.

std::size\_t **getCurrentSize** () const

Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

std::size\_t **getHighWatermark** () const

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

Platform **getPlatform** ()

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

void **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

std::size\_t **getActualSize** () const

Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

const std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

strategy::AllocationStrategy \*m\_allocator

std::string m\_name

int m\_id

## Class NumaPolicy

- Defined in file\_umpire\_strategy\_NumaPolicy.hpp

## Inheritance Relationships

### Base Type

- public umpire::strategy::AllocationStrategy (*Class AllocationStrategy*)

## Class Documentation

**class NumaPolicy** : public umpire::strategy::AllocationStrategy

Use NUMA interface to locate memory to a specific NUMA node.

This *AllocationStrategy* provides a method of ensuring memory sits on a specific NUMA node. This can be used either for optimization, or for moving memory between the host and devices.

### Public Functions

**NumaPolicy** (**const** std::string &name, int id, *Allocator allocator*, int numa\_node)

void \***allocate** (std::size\_t bytes)

Allocate bytes of memory.

**Return** Pointer to start of allocated bytes.

#### Parameters

- bytes: Number of bytes to allocate.

void **deallocate** (void \*ptr)

Free the memory at ptr.

#### Parameters

- ptr: Pointer to free.

std::size\_t **getCurrentSize** () **const**

Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

std::size\_t **getHighWatermark** () **const**

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

*Platform* **getPlatform ()**

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

**int getNode () const**

**void release ()**

Release any and all unused memory held by this *AllocationStrategy*.

**std::size\_t getActualSize () const**

Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

**const std::string &getName ()**

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

**int getId ()**

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

**std::string m\_name**

**int m\_id**

### Class SizeLimiter

- Defined in file\_umpire\_strategy\_SizeLimiter.hpp

### Inheritance Relationships

#### Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

## Class Documentation

**class** `SizeLimiter` : public `umpire::strategy::AllocationStrategy`

An allocator with a limited total size.

Using this *AllocationStrategy* with another can be a good way to limit the total size of allocations made on a particular resource or from a particular context.

### Public Functions

**SizeLimiter** (`const` `std::string` &*name*, `int` *id*, *Allocator* *allocator*, `std::size_t` *size\_limit*)

`void` **allocate** (`std::size_t` *bytes*)

Allocate *bytes* of memory.

**Return** Pointer to start of allocated bytes.

#### Parameters

- *bytes*: Number of bytes to allocate.

`void` **deallocate** (`void` \**ptr*)

Free the memory at *ptr*.

#### Parameters

- *ptr*: Pointer to free.

`std::size_t` **getCurrentSize** () `const`

Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

`std::size_t` **getHighWatermark** () `const`

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

*Platform* **getPlatform** ()

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

`void` **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

`std::size_t` **getActualSize** () **const**

Get the current amount of memory allocated by this allocator.

Note that this can be larger than `getCurrentSize()`, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

**const** `std::string` &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

`int` **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

`std::string` **m\_name**

`int` **m\_id**

### Class SlotPool

- Defined in file\_umpire\_strategy\_SlotPool.hpp

### Inheritance Relationships

#### Base Type

- `public` `umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

### Class Documentation

```
class SlotPool : public umpire::strategy::AllocationStrategy
```

#### Public Functions

```
SlotPool (const std::string &name, int id, Allocator allocator, std::size_t slots)
```

```
~SlotPool ()
```

```
void *allocate (std::size_t bytes)
```

Allocate bytes of memory.

**Return** Pointer to start of allocated bytes.

#### Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate** (void \**ptr*)  
Free the memory at *ptr*.

**Parameters**

- `ptr`: Pointer to free.

std::size\_t **getCurrentSize** () **const**

Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

std::size\_t **getHighWatermark** () **const**

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

Platform **getPlatform** ()

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

void **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

std::size\_t **getActualSize** () **const**

Get the current amount of memory allocated by this allocator.

Note that this can be larger than `getCurrentSize()`, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

const std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.



## Protected Attributes

```
std::string m_name
int m_id
```

## Class ThreadSafeAllocator

- Defined in file\_umpire\_strategy\_ThreadSafeAllocator.hpp

## Inheritance Relationships

### Base Type

- public umpire::strategy::AllocationStrategy (*Class AllocationStrategy*)

## Class Documentation

```
class ThreadSafeAllocator : public umpire::strategy::AllocationStrategy
```

Make an *Allocator* thread safe.

Using this *AllocationStrategy* will make the provided allocator thread-safe by synchronizing access to the allocators interface.

### Public Functions

```
ThreadSafeAllocator (const std::string &name, int id, Allocator allocator)
```

```
void *allocate (std::size_t bytes)
    Allocate bytes of memory.
```

**Return** Pointer to start of allocated bytes.

#### Parameters

- `bytes`: Number of bytes to allocate.

```
void deallocate (void *ptr)
    Free the memory at ptr.
```

#### Parameters

- `ptr`: Pointer to free.

```
std::size_t getCurrentSize () const
    Get current (total) size of the allocated memory.
```

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

std::size\_t **getHighWatermark** () **const**

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

*Platform* **getPlatform** ()

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

void **release** ()

Release any and all unused memory held by this *AllocationStrategy*.

std::size\_t **getActualSize** () **const**

Get the current amount of memory allocated by this allocator.

Note that this can be larger than `getCurrentSize()`, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

**const** std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

## Protected Attributes

strategy::AllocationStrategy \***m\_allocator**

std::mutex **m\_mutex**

std::string **m\_name**

int **m\_id**

## Class ZeroByteHandler

- Defined in file\_umpire\_strategy\_ZeroByteHandler.hpp

## Inheritance Relationships

### Base Type

- public `umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

### Class Documentation

**class** `ZeroByteHandler` : public `umpire::strategy::AllocationStrategy`

#### Public Functions

**ZeroByteHandler** (`std::unique_ptr<AllocationStrategy> &&allocator`)

void **\*allocate** (`std::size_t bytes`)  
Allocate bytes of memory.

**Return** Pointer to start of allocated bytes.

#### Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate** (`void *ptr`)  
Free the memory at `ptr`.

#### Parameters

- `ptr`: Pointer to free.

void **release** ()  
Release any and all unused memory held by this *AllocationStrategy*.

`std::size_t` **getCurrentSize** () **const**  
Get current (total) size of the allocated memory.

This is the total size of all allocation currently ‘live’ that have been made by this *AllocationStrategy* object.

**Return** Current total size of allocations.

`std::size_t` **getHighWatermark** () **const**  
Get the high watermark of the total allocated size.  
This is equivalent to the highest observed value of `getCurrentSize`.

**Return** High watermark allocation size.

`std::size_t` **getActualSize** () **const**  
Get the current amount of memory allocated by this allocator.

Note that this can be larger than `getCurrentSize()`, particularly if the *AllocationStrategy* implements some kind of pooling.

**Return** The total size of all the memory this object has allocated.

*Platform* **getPlatform** ()

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

**Return** The platform associated with this *AllocationStrategy*.

*strategy::AllocationStrategy* \***getAllocationStrategy** ()

**const** std::string &**getName** ()

Get the name of this *AllocationStrategy*.

**Return** The name of this *AllocationStrategy*.

int **getId** ()

Get the id of this *AllocationStrategy*.

**Return** The id of this *AllocationStrategy*.

### Protected Attributes

std::string **m\_name**

int **m\_id**

### Template Class TypedAllocator

- Defined in file\_umpire\_TypedAllocator.hpp

### Class Documentation

template<typename T>

**class TypedAllocator**

*Allocator* for objects of type T.

This class is an adaptor that allows using an *Allocator* to allocate objects of type T. You can use this class as an allocator for STL containers like std::vector.

### Public Types

**typedef** T **value\_type**

## Public Functions

**TypedAllocator** (*Allocator allocator*)

Construct a new *TypedAllocator* that will use *allocator* to allocate data.

### Parameters

- *allocator*: *Allocator* to use for allocating memory.

template<typename **U**>

**TypedAllocator** (**const** *TypedAllocator*<**U**> &*other*)

**T \*allocate** (std::size\_t *size*)

void **deallocate** (**T \*ptr**, std::size\_t *size*)

Deallocate *ptr*, the passed size is ignored.

### Parameters

- *ptr*: Pointer to deallocate
- *size*: Size of allocation (ignored).

## Friends

**friend** **umpire::TypedAllocator::TypedAllocator**

## Class AllocationMap

- Defined in file\_umpire\_util\_AllocationMap.hpp

## Nested Relationships

### Nested Types

- *Class AllocationMap::ConstIterator*
- *Class AllocationMap::RecordList*
- *Template Struct RecordList::Block*
- *Class RecordList::ConstIterator*

## Class Documentation

**class** **AllocationMap**

## Public Types

```
using Map = MemoryMap<RecordList>
```

## Public Functions

```
AllocationMap ()
```

```
AllocationMap (const AllocationMap&)
```

```
void insert (void *ptr, AllocationRecord record)
```

```
const AllocationRecord *find (void *ptr) const
```

```
AllocationRecord *find (void *ptr)
```

```
const AllocationRecord *findRecord (void *ptr) const
```

```
AllocationRecord *findRecord (void *ptr)
```

```
AllocationRecord remove (void *ptr)
```

```
bool contains (void *ptr) const
```

```
void clear ()
```

```
std::size_t size () const
```

```
void print (const std::function<bool> const AllocationRecord&  
> &&predicate, std::ostream &os = std::cout) const
```

```
void printAll (std::ostream &os = std::cout) const
```

```
AllocationMap::ConstIterator begin () const
```

```
AllocationMap::ConstIterator end () const
```

```
class ConstIterator : public std::iterator<std::forward_iterator_tag, AllocationRecord>
```

## Public Functions

```
ConstIterator (const AllocationMap *map, iterator_begin)
```

```
ConstIterator (const AllocationMap *map, iterator_end)
```

```
ConstIterator (const ConstIterator&)
```

```
const AllocationRecord &operator* ()
```

```
const AllocationRecord *operator--> ()
```

```
AllocationMap::ConstIterator &operator++ ()
```

```
AllocationMap::ConstIterator operator++ (int)
```

```
bool operator== (const ConstIterator &other) const
```

```
bool operator!= (const ConstIterator &other) const
```

## Class AllocationMap::ConstIterator

- Defined in file\_umpire\_util\_AllocationMap.hpp

### Nested Relationships

This class is a nested type of *Class AllocationMap*.

### Inheritance Relationships

#### Base Type

- `public std::iterator< std::forward_iterator_tag, AllocationRecord >`

### Class Documentation

```
class ConstIterator : public std::iterator<std::forward_iterator_tag, AllocationRecord>
```

#### Public Functions

```
ConstIterator (const AllocationMap *map, iterator_begin)
```

```
ConstIterator (const AllocationMap *map, iterator_end)
```

```
ConstIterator (const ConstIterator&)
```

```
const AllocationRecord &operator* ()
```

```
const AllocationRecord *operator-> ()
```

```
AllocationMap::ConstIterator &operator++ ()
```

```
AllocationMap::ConstIterator operator++ (int)
```

```
bool operator== (const ConstIterator &other) const
```

```
bool operator!= (const ConstIterator &other) const
```

## Class AllocationMap::RecordList

- Defined in file\_umpire\_util\_AllocationMap.hpp

## Nested Relationships

This class is a nested type of *Class AllocationMap*.

## Nested Types

- *Template Struct RecordList::Block*
- *Class RecordList::ConstIterator*

## Class Documentation

**class RecordList**

### Public Types

```
using RecordBlock = Block<AllocationRecord>
```

### Public Functions

```
RecordList (AllocationMap &map, AllocationRecord record)
```

```
~RecordList ()
```

```
void push_back (const AllocationRecord &rec)
```

```
AllocationRecord pop_back ()
```

```
AllocationMap::RecordList::ConstIterator begin () const
```

```
AllocationMap::RecordList::ConstIterator end () const
```

```
std::size_t size () const
```

```
bool empty () const
```

```
AllocationRecord *back ()
```

```
const AllocationRecord *back () const
```

```
template<typename T>
```

```
struct Block
```

### Public Members

```
T rec
```

```
Block *prev
```

```
class ConstIterator : public std::iterator<std::forward_iterator_tag, AllocationRecord>
```



## Public Functions

```

ConstIterator ()
ConstIterator (const RecordList *list, iterator_begin)
ConstIterator (const RecordList *list, iterator_end)
ConstIterator (const ConstIterator&)
const AllocationRecord &operator* ()
const AllocationRecord *operator-> ()
AllocationMap::RecordList::ConstIterator &operator++ ()
AllocationMap::RecordList::ConstIterator operator++ (int)
bool operator== (const ConstIterator &other) const
bool operator!= (const ConstIterator &other) const

```

## Class RecordList::ConstIterator

- Defined in file\_umpire\_util\_AllocationMap.hpp

## Nested Relationships

This class is a nested type of *Class AllocationMap::RecordList*.

## Inheritance Relationships

### Base Type

- `public std::iterator< std::forward_iterator_tag, AllocationRecord >`

## Class Documentation

```
class ConstIterator : public std::iterator<std::forward_iterator_tag, AllocationRecord>
```

### Public Functions

```

ConstIterator ()
ConstIterator (const RecordList *list, iterator_begin)
ConstIterator (const RecordList *list, iterator_end)
ConstIterator (const ConstIterator&)
const AllocationRecord &operator* ()

```

```
const AllocationRecord *operator-> ()  
AllocationMap::RecordList::ConstIterator &operator++ ()  
AllocationMap::RecordList::ConstIterator operator++ (int)  
bool operator== (const ConstIterator &other) const  
bool operator!= (const ConstIterator &other) const
```

## Class Exception

- Defined in file\_umpire\_util\_Exception.hpp

## Inheritance Relationships

### Base Type

- public std::exception

## Class Documentation

```
class Exception : public std::exception
```

### Public Functions

```
Exception (const std::string &msg, const std::string &file, int line)  
virtual ~Exception ()  
std::string message () const  
const char *what () const
```

## Class FixedMallocPool

- Defined in file\_umpire\_util\_FixedMallocPool.hpp

## Nested Relationships

### Nested Types

- Struct FixedMallocPool::Pool

## Class Documentation

### class FixedMallocPool

Pool for fixed size allocations using *malloc()*

Another version of this class exists in `umpire::strategy`, but this version does not rely on *Allocator* and all the memory tracking statistics, so it is useful for building objects in `umpire::util`.

### Public Functions

```
FixedMallocPool (const std::size_t object_bytes, const std::size_t objects_per_pool = 1024 *
                1024)
```

```
~FixedMallocPool ()
```

```
void *allocate (std::size_t bytes = 0)
```

```
void deallocate (void *ptr)
```

```
std::size_t numPools () const
```

## Class Logger

- Defined in file `umpire_util_Logger.hpp`

## Class Documentation

### class Logger

### Public Functions

```
void setLoggingMsgLevel (message::Level level)
```

```
void logMessage (message::Level level, const std::string &message, const std::string &fileName,
                int line)
```

```
bool logLevelEnabled (message::Level level)
```

```
~Logger ()
```

```
Logger (const Logger&)
```

```
Logger &operator= (const Logger&)
```

## Public Static Functions

**static** void **initialize** ()

**static** void **finalize** ()

*Logger* \***getActiveLogger** ()

## Template Class MemoryMap

- Defined in file `_umpire_util_MemoryMap.hpp`

## Nested Relationships

### Nested Types

- *Template Class MemoryMap::Iterator\_*

## Class Documentation

template<typename **V**>

**class** **MemoryMap**

A fast replacement for `std::map<void*, Value>` for a generic `Value`.

This uses *FixedMallocPool* and Judy arrays and provides forward const and non-const iterators.

### Public Types

template<>

**using** **Key** = void \*

template<>

**using** **Value** = **V**

template<>

**using** **KeyValuePair** = `std::pair<Key, Value *>`

template<>

**using** **Iterator** = `Iterator_<false>`

template<>

**using** **ConstIterator** = `Iterator_<true>`

## Public Functions

**MemoryMap** ()

**~MemoryMap** ()

**MemoryMap** (const *MemoryMap*&)

std::pair<typename *MemoryMap*<V>::Iterator, bool> **insert** (Key *ptr*, const Value &*val*)  
 Insert Value at *ptr* in the map if *ptr* does not exist. Uses copy constructor on Value once.

**Return** Pair of iterator position into map and boolean value whether entry was added. The iterator will be set to *end()* if no insertion was made.

template<typename **P**>

std::pair<Iterator, bool> **insert** (*P* &&*pair*)

Insert a key-value pair if *pair.first* does not exist as a key. Must have first and second fields. Calls the first version.

**Return** See alternative version.

template<typename ...**Args**>

std::pair<Iterator, bool> **insert** (Key *ptr*, *Args*&&... *args*)

Emplaces a new value at *ptr* in the map, forwarding *args* to the placement new constructor.

**Return** See alternative version.

*MemoryMap*<V>::Iterator **findOrBefore** (Key *ptr*)

Find a value at *ptr*.

**Return** iterator into map at *ptr* or preceeding position.

*MemoryMap*<V>::ConstIterator **findOrBefore** (Key *ptr*) const

*MemoryMap*<V>::Iterator **find** (Key *ptr*)

Find a value at *ptr*.

**Return** iterator into map at *ptr* or *end()* if not found.

*MemoryMap*<V>::ConstIterator **find** (Key *ptr*) const

*MemoryMap*<V>::ConstIterator **begin** () const

Iterator to first value or *end()* if empty.

*MemoryMap*<V>::Iterator **begin** ()

*MemoryMap*<V>::ConstIterator **end** () const

Iterator to one-past-last value.

*MemoryMap*<V>::Iterator **end** ()

void **erase** (Key *ptr*)

Remove an entry from the map.

void **erase** (Iterator *iter*)

void **erase** (ConstIterator *oter*)

void **removeLast** ()

Remove/deallocate the last found entry.

WARNING: Use this with caution, only directly after using a method above. *erase(Key)* is safer, but requires an additional lookup.

void **clear** ()

Clear all entris from the map.

std::size\_t **size** () **const**

Return number of entries in the map.

template<typename ...**Args**>

std::pair<**typename** MemoryMap<V>::Iterator, bool> **doInsert** (Key *ptr*, *Args*&&... *args*)

template<typename **P**>

std::pair<**typename** MemoryMap<V>::Iterator, bool> **insert** (*P* &&*pair*)

template<typename ...**Args**>

std::pair<**typename** MemoryMap<V>::Iterator, bool> **insert** (Key *ptr*, *Args*&&... *args*)

## Friends

**friend** **umpire::util::MemoryMap::Iterator\_**

template<bool **Const** = false>

**class** **Iterator\_** : **public** std::iterator<std::forward\_iterator\_tag, Value>

## Public Types

template<>

template<>

**using** **Map** = **typename** std::conditional<Const, **const** MemoryMap<Value>, MemoryMap<Value>>::type

template<>

template<>

**using** **ValuePtr** = **typename** std::conditional<Const, **const** Value \*, Value \*>::type

template<>

template<>

**using** **Content** = std::pair<Key, ValuePtr>

template<>

template<>

**using** **Reference** = **typename** std::conditional<Const, **const** Content&, Content&>::type

template<>

template<>

**using** **Pointer** = **typename** std::conditional<Const, **const** Content \*, Content \*>::type

## Public Functions

```

template<>
Iterator_(Map *map, Key ptr)

template<>
Iterator_(Map *map, iterator_begin)

template<>
Iterator_(Map *map, iterator_end)

template<bool OtherConst>
Iterator_(const Iterator_<OtherConst> &other)

template<>
MemoryMap<V>::template Iterator_<Const>::Reference operator* ()

template<>
MemoryMap<V>::template Iterator_<Const>::Pointer operator-> ()

template<>
MemoryMap<V>::template Iterator_<Const> &operator++ ()

template<>
MemoryMap<V>::template Iterator_<Const> operator++ (int)

template<bool OtherConst>
bool operator== (const Iterator_<OtherConst> &other) const

template<bool OtherConst>
bool operator!= (const Iterator_<OtherConst> &other) const

template<bool OtherConst>
bool operator== (const MemoryMap<V>::Iterator_<OtherConst> &other) const

template<bool OtherConst>
bool operator!= (const MemoryMap<V>::Iterator_<OtherConst> &other) const

```

## Template Class MemoryMap::Iterator\_

- Defined in file\_umpire\_util\_MemoryMap.hpp

## Nested Relationships

This class is a nested type of *Template Class MemoryMap*.

## Inheritance Relationships

### Base Type

- `public std::iterator< std::forward_iterator_tag, Value >`

### Class Documentation

```
template<bool Const = false>  
class Iterator_ : public std::iterator<std::forward_iterator_tag, Value>
```

#### Public Types

```
template<>  
template<>  
using Map = typename std::conditional<Const, const MemoryMap<Value>, MemoryMap<Value>>::type  
  
template<>  
template<>  
using ValuePtr = typename std::conditional<Const, const Value *, Value *>::type  
  
template<>  
template<>  
using Content = std::pair<Key, ValuePtr>  
  
template<>  
template<>  
using Reference = typename std::conditional<Const, const Content&, Content&>::type  
  
template<>  
template<>  
using Pointer = typename std::conditional<Const, const Content *, Content *>::type
```

#### Public Functions

```
template<>  
Iterator_(Map *map, Key ptr)  
  
template<>  
Iterator_(Map *map, iterator_begin)  
  
template<>  
Iterator_(Map *map, iterator_end)  
  
template<bool OtherConst>  
Iterator_(const Iterator_<OtherConst> &other)  
  
template<>  
MemoryMap<V>::template Iterator_<Const>::Reference operator* ()  
  
template<>  
MemoryMap<V>::template Iterator_<Const>::Pointer operator-- ()  
  
template<>
```



```

MemoryMap<V>::template Iterator_<Const> &operator++ ()

template<>
MemoryMap<V>::template Iterator_<Const> operator++ (int)

template<bool OtherConst>
bool operator== (const Iterator_<OtherConst> &other) const

template<bool OtherConst>
bool operator!= (const Iterator_<OtherConst> &other) const

template<bool OtherConst>
bool operator== (const MemoryMap<V>::Iterator_<OtherConst> &other) const

template<bool OtherConst>
bool operator!= (const MemoryMap<V>::Iterator_<OtherConst> &other) const

```

## Class MPI

- Defined in file\_umpire\_util\_MPI.hpp

## Class Documentation

### class MPI

#### Public Static Functions

```

void initialize ()
void finalize ()
int getRank ()
int getSize ()
void sync ()
void logMpiInfo ()
bool isInitialized ()

```

## Class OutputBuffer

- Defined in file\_umpire\_util\_OutputBuffer.hpp

## Inheritance Relationships

### Base Type

- `public streambuf`

### Class Documentation

```
class OutputBuffer : public streambuf
```

#### Public Functions

```
OutputBuffer ()
```

```
~OutputBuffer ()
```

```
void setConsoleStream (std::ostream *stream)
```

```
void setFileStream (std::ostream *stream)
```

```
int overflow (int ch)
```

```
int sync ()
```

### Class Statistic

- Defined in file\_umpire\_util\_Statistic.hpp

### Class Documentation

```
class Statistic
```

#### Public Functions

```
~Statistic ()
```

```
void recordStatistic (conduit::Node &&n)
```

```
void printData (std::ostream &stream)
```

#### Protected Functions

```
Statistic (const std::string &name)
```

## Class StatisticsDatabase

- Defined in file\_umpire\_util\_StatisticsDatabase.hpp

## Class Documentation

**class** StatisticsDatabase

### Public Functions

std::shared\_ptr<Statistic> **getStatistic** (const std::string &name)

void **printStatistics** (std::ostream &stream)

### Public Static Functions

StatisticsDatabase \***getDatabase** ()

## 6.3.3 Enums

### Enum Platform

- Defined in file\_umpire\_util\_Platform.hpp

### Enum Documentation

**enum** umpire::Platform

*Values:*

**none**

**cpu**

**cuda**

**rocm**

**hip**

### Enum MemoryResourceType

- Defined in file\_umpire\_resource\_MemoryResourceTypes.hpp

## Enum Documentation

**enum** `umpire::resource::MemoryResourceType`

*Values:*

**Host**  
**Device**  
**Unified**  
**Pinned**  
**Constant**

## Enum Level

- Defined in `file_umpire_util_Logger.hpp`

## Enum Documentation

**enum** `umpire::util::message::Level`

*Values:*

**Error**  
**Warning**  
**Info**  
**Debug**  
**Num\_Levels**

## 6.3.4 Functions

### Function `find_first_set`

- Defined in `file_umpire_strategy_FixedSizePool.hpp`

## Function Documentation

`int` `umpire::strategy::find_first_set` (`int` *i*)

### Function `genumpiresplicer::gen_bounds`

- Defined in `file_umpire_interface_c_fortran_genumpiresplicer.py`

## Function Documentation

`genumpiresplicer.gen_bounds()`

## Function `genumpiresplicer::gen_fortran`

- Defined in file\_umpire\_interface\_c\_fortran\_genumpiresplicer.py

## Function Documentation

`genumpiresplicer.gen_fortran()`

## Function `genumpiresplicer::gen_methods`

- Defined in file\_umpire\_interface\_c\_fortran\_genumpiresplicer.py

## Function Documentation

`genumpiresplicer.gen_methods()`

## Function `round_up`

- Defined in file\_umpire\_strategy\_DynamicPoolMap.cpp

## Function Documentation

**Warning:** doxygenfunction: Cannot find function “round\_up” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

## Function `shroud_c_loc`

- Defined in file\_umpire\_interface\_c\_fortran\_shroudr.cpp

## Function Documentation

**Warning:** doxygenfunction: Cannot find function “shroud\_c\_loc” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

## Function `shroud_c_loc_`

- Defined in file\_umpire\_interface\_c\_fortran\_shroudr.cpp

## Function Documentation

**Warning:** doxygenfunction: Cannot find function “shroud\_c\_loc\_” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

### Function `shroud_FccCopy(char *, int, const char *)`

- Defined in file\_umpire\_interface\_c\_fortran\_shroudr.cpp

## Function Documentation

void `shroud_FccCopy` (char \**a*, int *la*, const char \**s*)

### Function `shroud_FccCopy(char *, int, const char *)`

- Defined in file\_umpire\_interface\_c\_fortran\_shroudr.hpp

## Function Documentation

void `shroud_FccCopy` (char \**a*, int *la*, const char \**s*)

### Function `umpire::cpu_vendor_type`

- Defined in file\_umpire\_util\_detect\_vendor.cpp

## Function Documentation

*MemoryResourceTraits::vendor\_type* `umpire::cpu_vendor_type` ()

### Function `umpire::error`

- Defined in file\_umpire\_util\_io.cpp

## Function Documentation

std::ostream &`umpire::error` ()

### Function `umpire::finalize`

- Defined in `file_umpire_Umpire.hpp`

### Function Documentation

```
void umpire::finalize ()
```

### Function `umpire::free`

- Defined in `file_umpire_Umpire.hpp`

### Function Documentation

```
void umpire::free (void *ptr)
```

Free any memory allocated with Umpire.

This method is a convenience wrapper around calls to the *ResourceManager*, it can be used to free allocations from any *MemorySpace*. \*

#### Parameters

- `ptr`: Address to free.

### Function `umpire::get_allocator_records`

- Defined in `file_umpire_Umpire.cpp`

### Function Documentation

```
std::vector<util::AllocationRecord> umpire::get_allocator_records (Allocator allocator)
```

Returns vector of *AllocationRecords* created by the allocator.

#### Parameters

- `allocator`: source *Allocator*.

### Function `umpire::get_major_version`

- Defined in `file_umpire_Umpire.hpp`

### Function Documentation

`int umpire::get_major_version()`

### Function `umpire::get_minor_version`

- Defined in `file_umpire_Umpire.hpp`

### Function Documentation

`int umpire::get_minor_version()`

### Function `umpire::get_page_size`

- Defined in `file_umpire_util_numa.cpp`

### Function Documentation

`long umpire::get_page_size()`

### Function `umpire::get_patch_version`

- Defined in `file_umpire_Umpire.hpp`

### Function Documentation

`int umpire::get_patch_version()`

### Function `umpire::get_rc_version`

- Defined in `file_umpire_Umpire.hpp`

### Function Documentation

`std::string umpire::get_rc_version()`

### Function `umpire::initialize`

- Defined in `file_umpire_Umpire.hpp`



## Function Documentation

`void umpire::initialize()`

## Function `umpire::log`

- Defined in `file_umpire_util_io.cpp`

## Function Documentation

`std::ostream &umpire::log()`

## Function `umpire::malloc`

- Defined in `file_umpire_Umpire.hpp`

## Function Documentation

`void *umpire::malloc (std::size_t size)`

Allocate memory in the default space, with the default allocator.

This method is a convenience wrapper around calls to the *ResourceManager* to allocate memory in the default *MemorySpace*.

### Parameters

- `size`: Number of bytes to allocate.

## Function `umpire::numa::get_allocatable_nodes`

- Defined in `file_umpire_util_numa.cpp`

## Function Documentation

`std::vector<int> umpire::numa::get_allocatable_nodes()`

## Function `umpire::numa::get_device_nodes`

- Defined in `file_umpire_util_numa.cpp`

## Function Documentation

`std::vector<int> umpire::numa::get_device_nodes()`

## Function `umpire::numa::get_host_nodes`

- Defined in `file_umpire_util_numa.cpp`

## Function Documentation

`std::vector<int> umpire::numa::get_host_nodes()`

## Function `umpire::numa::get_location`

- Defined in `file_umpire_util_numa.cpp`

## Function Documentation

`int umpire::numa::get_location(void *ptr)`

## Function `umpire::numa::move_to_node`

- Defined in `file_umpire_util_numa.cpp`

## Function Documentation

`void umpire::numa::move_to_node(void *ptr, std::size_t bytes, int node)`

## Function `umpire::numa::preferred_node`

- Defined in `file_umpire_util_numa.cpp`

## Function Documentation

`int umpire::numa::preferred_node()`

## Function `umpire::operator<<(std::ostream&, const Allocator&)`

- Defined in `file_umpire_Allocator.cpp`

## Function Documentation

`std::ostream &umpire::operator<<` (`std::ostream &os`, `const Allocator &allocator`)

### Function `umpire::operator<<`(`std::ostream&`, `umpire::Allocator&`)

- Defined in `file_umpire_Replay.cpp`

## Function Documentation

`std::ostream &umpire::operator<<` (`std::ostream &out`, `umpire::Allocator &alloc`)

### Function `umpire::operator<<`(`std::ostream&`, `umpire::strategy::DynamicPoolMap::CoalesceHeuristic&`)

- Defined in `file_umpire_Replay.cpp`

## Function Documentation

`std::ostream &umpire::operator<<` (`std::ostream &out`, `umpire::strategy::DynamicPoolMap::CoalesceHeuristic&`)

### Function `umpire::operator<<`(`std::ostream&`, `umpire::strategy::DynamicPoolList::CoalesceHeuristic&`)

- Defined in `file_umpire_Replay.cpp`

## Function Documentation

`std::ostream &umpire::operator<<` (`std::ostream &out`, `umpire::strategy::DynamicPoolList::CoalesceHeuristic&`)

### Function `umpire::print_allocator_records`

- Defined in `file_umpire_Umpire.cpp`

## Function Documentation

`void umpire::print_allocator_records` (`Allocator allocator`, `std::ostream &os = std::cout`)

Print the allocations from a specific allocator in a human-readable format.

#### Parameters

- `allocator`: source *Allocator*.
- `os`: output stream

### Function `umpire::replay`

- Defined in `file_umpire_util_io.cpp`

### Function Documentation

```
std::ostream &umpire::replay()
```

### Function `umpire::strategy::find_first_set`

- Defined in `file_umpire_strategy_FixedPool.cpp`

### Function Documentation

```
int umpire::strategy::find_first_set(int i)
```

### Function `umpire::strategy::heuristic_percent_releasable`

- Defined in `file_umpire_strategy_DynamicPoolHeuristic.cpp`

### Function Documentation

```
std::function<bool (const strategy::DynamicPoolMap&)> umpire::strategy::heuristic_percent_releasable  
int percentage
```

### Function `umpire::strategy::heuristic_percent_releasable_list`

- Defined in `file_umpire_strategy_DynamicPoolHeuristic.cpp`

### Function Documentation

```
std::function<bool (const strategy::DynamicPoolList&)> umpire::strategy::heuristic_percent_releasable_list  
int percentage
```

### Function `umpire::strategy::operator<<`

- Defined in `file_umpire_strategy_AllocationStrategy.cpp`

## Function Documentation

`std::ostream &umpire::strategy::operator<<` (`std::ostream &os`, `const AllocationStrategy &strategy`)

## Function `umpire::util::case_insensitive_match`

- Defined in `file_umpire_util_Logger.cpp`

## Function Documentation

`static int umpire::util::case_insensitive_match` (`const std::string s1`, `const std::string s2`)

## Function `umpire::util::detail::add_entry(conduit::Node&)`

- Defined in `file_umpire_util_statistic_helper.hpp`

## Function Documentation

`conduit::Node umpire::util::detail::add_entry` (`conduit::Node &n`)

## Template Function `umpire::util::detail::add_entry(conduit::Node&, T, U)`

- Defined in `file_umpire_util_statistic_helper.hpp`

## Function Documentation

`template<typename T, typename U>`  
`conduit::Node umpire::util::detail::add_entry` (`conduit::Node &n`, `T k`, `U v`)

## Template Function `umpire::util::detail::add_entry(conduit::Node&, T, U, Args...)`

- Defined in `file_umpire_util_statistic_helper.hpp`

## Function Documentation

`template<typename T, typename U, typename ...Args>`  
`conduit::Node umpire::util::detail::add_entry` (`conduit::Node &n`, `T k`, `U v`, `Args... args`)

### Template Function `umpire::util::detail::record_statistic`

- Defined in `file_umpire_util_statistic_helper.hpp`

### Function Documentation

```
template<typename ...Args>  
void umpire::util::detail::record_statistic (const std::string &name, Args&&... args)
```

### Function `umpire::util::directory_exists`

- Defined in `file_umpire_util_io.cpp`

### Function Documentation

```
static bool umpire::util::directory_exists (const std::string &file)
```

### Template Function `umpire::util::do_wrap(std::unique_ptr<Base>&&)`

- Defined in `file_umpire_util_wrap_allocator.hpp`

### Function Documentation

```
template<typename Base>  
std::unique_ptr<Base> umpire::util::do_wrap (std::unique_ptr<Base> &&allocator)
```

### Template Function `umpire::util::do_wrap(std::unique_ptr<Base>&&)`

- Defined in `file_umpire_util_wrap_allocator.hpp`

### Function Documentation

```
template<typename Base>  
std::unique_ptr<Base> umpire::util::do_wrap (std::unique_ptr<Base> &&allocator)
```

### Function `umpire::util::file_exists`

- Defined in `file_umpire_util_io.cpp`

## Function Documentation

**static** bool `umpire::util::file_exists` (**const** std::string &*file*)

### Function `umpire::util::flush_files`

- Defined in `file_umpire_util_io.cpp`

## Function Documentation

void `umpire::util::flush_files` ()

Synchronize all stream buffers to their respective output sequences. This function is usually called by exception generating code like `UMPIRE_ERROR`.

### Function `umpire::util::initialize_io`

- Defined in `file_umpire_util_io.cpp`

## Function Documentation

void `umpire::util::initialize_io` (**const** bool *enable\_log*, **const** bool *enable\_replay*)

Initialize the streams. This method is called when `ResourceManger` is initialized. Do not call this manually.

### Template Function `umpire::util::make_unique`

- Defined in `file_umpire_util_make_unique.hpp`

## Function Documentation

template<typename **T**, typename ...**Args**>

**constexpr** std::unique\_ptr<*T*> `umpire::util::make_unique` (*Args*&&... *args*)

### Function `umpire::util::make_unique_filename`

- Defined in `file_umpire_util_io.cpp`

## Function Documentation

**static** std::string `umpire::util::make_unique_filename` (**const** std::string &*base\_dir*, **const** std::string &*name*, **const** int *pid*, **const** std::string &*extension*)

### Function `umpire::util::relative_fragmentation`

- Defined in `file_umpire_util_allocation_statistics.cpp`

### Function Documentation

`float umpire::util::relative_fragmentation (std::vector<util::AllocationRecord> &recs)`

Compute the relative fragmentation of a set of allocation records.

Fragmentation =  $1 - (\text{largest free block}) / (\text{total free space})$

### Template Function `umpire::util::unwrap_allocation_strategy`

- Defined in `file_umpire_util_wrap_allocator.hpp`

### Function Documentation

`template<typename Strategy>`

`Strategy *umpire::util::unwrap_allocation_strategy (strategy::AllocationStrategy *base_strategy)`

### Template Function `umpire::util::unwrap_allocator`

- Defined in `file_umpire_util_wrap_allocator.hpp`

### Function Documentation

`template<typename Strategy>`

`Strategy *umpire::util::unwrap_allocator (Allocator allocator)`

### Template Function `umpire::util::wrap_allocator`

- Defined in `file_umpire_util_wrap_allocator.hpp`

### Function Documentation

`template<typename ...Strategies>`

`std::unique_ptr<strategy::AllocationStrategy> umpire::util::wrap_allocator (std::unique_ptr<strategy::AllocationStrategy> &&allocator)`



**Function `umpire_allocator_allocate(umpire_allocator *, size_t)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.cpp

**Function Documentation**

void \***umpire\_allocator\_allocate** (*umpire\_allocator \*self*, size\_t bytes)

**Function `umpire_allocator_allocate(umpire_allocator *, size_t)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.h

**Function Documentation**

void \***umpire\_allocator\_allocate** (*umpire\_allocator \*self*, size\_t bytes)

**Function `umpire_allocator_deallocate(umpire_allocator *, void *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.cpp

**Function Documentation**

void **umpire\_allocator\_deallocate** (*umpire\_allocator \*self*, void \*ptr)

**Function `umpire_allocator_deallocate(umpire_allocator *, void *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.h

**Function Documentation**

void **umpire\_allocator\_deallocate** (*umpire\_allocator \*self*, void \*ptr)

**Function `umpire_allocator_delete(umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.cpp

**Function Documentation**

void **umpire\_allocator\_delete** (*umpire\_allocator \*self*)

### Function `umpire_allocator_delete(umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapAllocator.h`

### Function Documentation

void `umpire_allocator_delete` (*umpire\_allocator* \*self)

### Function `umpire_allocator_get_actual_size(umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapAllocator.cpp`

### Function Documentation

size\_t `umpire_allocator_get_actual_size` (*umpire\_allocator* \*self)

### Function `umpire_allocator_get_actual_size(umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapAllocator.h`

### Function Documentation

size\_t `umpire_allocator_get_actual_size` (*umpire\_allocator* \*self)

### Function `umpire_allocator_get_current_size(umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapAllocator.cpp`

### Function Documentation

size\_t `umpire_allocator_get_current_size` (*umpire\_allocator* \*self)

### Function `umpire_allocator_get_current_size(umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapAllocator.h`

### Function Documentation

size\_t `umpire_allocator_get_current_size` (*umpire\_allocator* \*self)

**Function `umpire_allocator_get_high_watermark(umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.cpp

**Function Documentation**

`size_t umpire_allocator_get_high_watermark (umpire_allocator *self)`

**Function `umpire_allocator_get_high_watermark(umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.h

**Function Documentation**

`size_t umpire_allocator_get_high_watermark (umpire_allocator *self)`

**Function `umpire_allocator_get_id(umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.cpp

**Function Documentation**

`size_t umpire_allocator_get_id (umpire_allocator *self)`

**Function `umpire_allocator_get_id(umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.h

**Function Documentation**

`size_t umpire_allocator_get_id (umpire_allocator *self)`

**Function `umpire_allocator_get_name(umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.cpp

**Function Documentation**

`const char *umpire_allocator_get_name (umpire_allocator *self)`

### Function `umpire_allocator_get_name(umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapAllocator.h`

### Function Documentation

```
const char *umpire_allocator_get_name (umpire_allocator *self)
```

### Function `umpire_allocator_get_name_bufferify(umpire_allocator *, umpire_SHROUD_array *)`

- Defined in file `umpire_interface_c_fortran_wrapAllocator.cpp`

### Function Documentation

```
void umpire_allocator_get_name_bufferify (umpire_allocator *self, umpire_SHROUD_array  
*DSHF_rv)
```

### Function `umpire_allocator_get_name_bufferify(umpire_allocator *, umpire_SHROUD_array *)`

- Defined in file `umpire_interface_c_fortran_wrapAllocator.h`

### Function Documentation

```
void umpire_allocator_get_name_bufferify (umpire_allocator *self, umpire_SHROUD_array  
*DSHF_rv)
```

### Function `umpire_allocator_get_size(umpire_allocator *, void *)`

- Defined in file `umpire_interface_c_fortran_wrapAllocator.cpp`

### Function Documentation

```
size_t umpire_allocator_get_size (umpire_allocator *self, void *ptr)
```

### Function `umpire_allocator_get_size(umpire_allocator *, void *)`

- Defined in file `umpire_interface_c_fortran_wrapAllocator.h`

### Function Documentation

`size_t umpire_allocator_get_size (umpire_allocator *self, void *ptr)`

### Function `umpire_allocator_release(umpire_allocator *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.cpp

### Function Documentation

`void umpire_allocator_release (umpire_allocator *self)`

### Function `umpire_allocator_release(umpire_allocator *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.h

### Function Documentation

`void umpire_allocator_release (umpire_allocator *self)`

### Function `umpire_mod::allocator_allocate`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

`type(c_ptr) function umpire_mod::allocator_allocate(obj obj, bytes bytes)`

### Function `umpire_mod::allocator_allocate_double_array_1d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

`subroutine umpire_mod::allocator_allocate_double_array_1d(this this, array array, dims dims)`

### Function `umpire_mod::allocator_allocate_double_array_2d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_double_array_2d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_double_array_3d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_double_array_3d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_double_array_4d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_double_array_4d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_float_array_1d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_float_array_1d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_float_array_2d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_float_array_2d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_float_array_3d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_float_array_3d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_float_array_4d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_float_array_4d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_int_array_1d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_int_array_1d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_int_array_2d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_int_array_2d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_int_array_3d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_int_array_3d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_int_array_4d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_int_array_4d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_long_array_1d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_long_array_1d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_long_array_2d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_long_array_2d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_long_array_3d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_long_array_3d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_allocate_long_array_4d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_allocate_long_array_4d(this this, array array, dims dims)
```

### Function `umpire_mod::allocator_associated`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f



## Function Documentation

logical function `umpire_mod::allocator_associated(obj obj)`

## Function `umpire_mod::allocator_deallocate`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

subroutine `umpire_mod::allocator_deallocate (obj obj, ptr ptr)`

## Function `umpire_mod::allocator_deallocate_double_array_1d`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

subroutine `umpire_mod::allocator_deallocate_double_array_1d(this this, array array)`

## Function `umpire_mod::allocator_deallocate_double_array_2d`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

subroutine `umpire_mod::allocator_deallocate_double_array_2d(this this, array array)`

## Function `umpire_mod::allocator_deallocate_double_array_3d`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

subroutine `umpire_mod::allocator_deallocate_double_array_3d(this this, array array)`

## Function `umpire_mod::allocator_deallocate_double_array_4d`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

```
subroutine umpire_mod::allocator_deallocate_double_array_4d(this this, array array)
```

## Function `umpire_mod::allocator_deallocate_float_array_1d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

## Function Documentation

```
subroutine umpire_mod::allocator_deallocate_float_array_1d(this this, array array)
```

## Function `umpire_mod::allocator_deallocate_float_array_2d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

## Function Documentation

```
subroutine umpire_mod::allocator_deallocate_float_array_2d(this this, array array)
```

## Function `umpire_mod::allocator_deallocate_float_array_3d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

## Function Documentation

```
subroutine umpire_mod::allocator_deallocate_float_array_3d(this this, array array)
```

## Function `umpire_mod::allocator_deallocate_float_array_4d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

## Function Documentation

```
subroutine umpire_mod::allocator_deallocate_float_array_4d(this this, array array)
```

## Function `umpire_mod::allocator_deallocate_int_array_1d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

subroutine `umpire_mod::allocator_deallocate_int_array_1d`(this this, array array)

### Function `umpire_mod::allocator_deallocate_int_array_2d`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

### Function Documentation

subroutine `umpire_mod::allocator_deallocate_int_array_2d`(this this, array array)

### Function `umpire_mod::allocator_deallocate_int_array_3d`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

### Function Documentation

subroutine `umpire_mod::allocator_deallocate_int_array_3d`(this this, array array)

### Function `umpire_mod::allocator_deallocate_int_array_4d`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

### Function Documentation

subroutine `umpire_mod::allocator_deallocate_int_array_4d`(this this, array array)

### Function `umpire_mod::allocator_deallocate_long_array_1d`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

### Function Documentation

subroutine `umpire_mod::allocator_deallocate_long_array_1d`(this this, array array)

### Function `umpire_mod::allocator_deallocate_long_array_2d`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

### Function Documentation

```
subroutine umpire_mod::allocator_deallocate_long_array_2d(this this, array array)
```

### Function `umpire_mod::allocator_deallocate_long_array_3d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_deallocate_long_array_3d(this this, array array)
```

### Function `umpire_mod::allocator_deallocate_long_array_4d`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_deallocate_long_array_4d(this this, array array)
```

### Function `umpire_mod::allocator_delete`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
subroutine umpire_mod::allocator_delete (obj obj)
```

### Function `umpire_mod::allocator_eq`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

```
logical function umpire_mod::allocator_eq(a a, b b)
```

### Function `umpire_mod::allocator_get_actual_size`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfumpire.f

### Function Documentation

`integer(c_size_t) function umpire_mod::allocator_get_actual_size(obj obj)`

### Function `umpire_mod::allocator_get_current_size`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

### Function Documentation

`integer(c_size_t) function umpire_mod::allocator_get_current_size(obj obj)`

### Function `umpire_mod::allocator_get_high_watermark`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

### Function Documentation

`integer(c_size_t) function umpire_mod::allocator_get_high_watermark(obj obj)`

### Function `umpire_mod::allocator_get_id`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

### Function Documentation

`integer(c_size_t) function umpire_mod::allocator_get_id(obj obj)`

### Function `umpire_mod::allocator_get_instance`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

### Function Documentation

`type(c_ptr) function umpire_mod::allocator_get_instance(obj obj)`

### Function `umpire_mod::allocator_get_name`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

`character(len=:)` `function`, `allocatable` `umpire_mod::allocator_get_name(obj obj)`

### Function `umpire_mod::allocator_get_size`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

`integer(c_size_t)` `function` `umpire_mod::allocator_get_size(obj obj, ptr ptr)`

### Function `umpire_mod::allocator_ne`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

`logical` `function` `umpire_mod::allocator_ne(a a, b b)`

### Function `umpire_mod::allocator_release`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

subroutine `umpire_mod::allocator_release(obj obj)`

### Function `umpire_mod::allocator_set_instance`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

subroutine `umpire_mod::allocator_set_instance(obj obj, cxxmem cxxmem)`

### Function `umpire_mod::resourcemanager_associated`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

`logical function umpire_mod::resource_manager_associated(obj obj)`

## Function `umpire_mod::resource_manager_copy_all`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

subroutine `umpire_mod::resource_manager_copy_all (obj obj, src_ptr src_ptr, dst_ptr dst_ptr)`

## Function `umpire_mod::resource_manager_copy_with_size`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

subroutine `umpire_mod::resource_manager_copy_with_size (obj obj, src_ptr src_ptr, dst_ptr dst_ptr, size size)`

## Function `umpire_mod::resource_manager_deallocate`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

subroutine `umpire_mod::resource_manager_deallocate (obj obj, ptr ptr)`

## Function `umpire_mod::resource_manager_eq`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

## Function Documentation

`logical function umpire_mod::resource_manager_eq(a a, b b)`

### Function `umpire_mod::resource_manager_get_allocator_by_id`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
type(umpireallocator) function umpire_mod::resource_manager_get_allocator_by_id(obj obj, id
```

### Function `umpire_mod::resource_manager_get_allocator_by_name`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
type(umpireallocator) function umpire_mod::resource_manager_get_allocator_by_name(obj obj, n
```

### Function `umpire_mod::resource_manager_get_allocator_for_ptr`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
type(umpireallocator) function umpire_mod::resource_manager_get_allocator_for_ptr(obj obj, p
```

### Function `umpire_mod::resource_manager_get_instance`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
type(umpireresource_manager) function umpire_mod::resource_manager_get_instance()
```

### Function `umpire_mod::resource_manager_get_size`

- Defined in file `umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
integer(c_size_t) function umpire_mod::resource_manager_get_size(obj obj, ptr ptr)
```



**Function `umpire_mod::resource_manager_has_allocator`**

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

**Function Documentation**

```
logical function umpire_mod::resource_manager_has_allocator(obj obj, ptr ptr)
```

**Function `umpire_mod::resource_manager_is_allocator`**

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

**Function Documentation**

```
logical function umpire_mod::resource_manager_is_allocator(obj obj, name name)
```

**Function `umpire_mod::resource_manager_make_allocator_advisor`**

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

**Function Documentation**

```
type(umpire_allocator) function umpire_mod::resource_manager_make_allocator_advisor(obj obj,
```

**Function `umpire_mod::resource_manager_make_allocator_fixed_pool`**

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

**Function Documentation**

```
type(umpire_allocator) function umpire_mod::resource_manager_make_allocator_fixed_pool(obj obj,
```

**Function `umpire_mod::resource_manager_make_allocator_list_pool`**

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

**Function Documentation**

```
type(umpire_allocator) function umpire_mod::resource_manager_make_allocator_list_pool(obj obj,
```

### Function `umpire_mod::resource_manager_make_allocator_named`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
type(umpire_allocator) function umpire_mod::resource_manager_make_allocator_named(obj obj, name name)
```

### Function `umpire_mod::resource_manager_make_allocator_pool`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
type(umpire_allocator) function umpire_mod::resource_manager_make_allocator_pool(obj obj, name name, pool_size pool_size)
```

### Function `umpire_mod::resource_manager_make_allocator_prefetcher`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
type(umpire_allocator) function umpire_mod::resource_manager_make_allocator_prefetcher(obj obj, name name, pool_size pool_size, prefetcher prefetcher)
```

### Function `umpire_mod::resource_manager_make_allocator_thread_safe`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
type(umpire_allocator) function umpire_mod::resource_manager_make_allocator_thread_safe(obj obj, name name, pool_size pool_size, prefetcher prefetcher, thread_safe thread_safe)
```

### Function `umpire_mod::resource_manager_memset_all`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
subroutine umpire_mod::resource_manager_memset_all(obj obj, ptr ptr, val val)
```

### Function `umpire_mod::resource_manager_memset_with_size`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
subroutine umpire_mod::resource_manager_memset_with_size (obj obj, ptr ptr, val val, length length)
```

### Function `umpire_mod::resource_manager_move`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
type(c_ptr) function umpire_mod::resource_manager_move(obj obj, src_ptr src_ptr, allocator allocator)
```

### Function `umpire_mod::resource_manager_ne`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
logical function umpire_mod::resource_manager_ne(a a, b b)
```

### Function `umpire_mod::resource_manager_reallocate_default`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
type(c_ptr) function umpire_mod::resource_manager_reallocate_default(obj obj, src_ptr src_ptr)
```

### Function `umpire_mod::resource_manager_reallocate_with_allocator`

- Defined in `file_umpire_interface_c_fortran_wrapfumpire.f`

#### Function Documentation

```
type(c_ptr) function umpire_mod::resource_manager_reallocate_with_allocator(obj obj, src_ptr src_ptr)
```

### Function `umpire_mod::resourcemanager_register_allocator`

- Defined in file `umpire_interface_c_fortran_wrapumpire.f`

### Function Documentation

subroutine `umpire_mod::resourcemanager_register_allocator` (*obj obj, name name, allocator allocator*)

### Function `umpire_resourcemanager_copy_all(umpire_resourcemanager *, void *, void *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

void `umpire_resourcemanager_copy_all` (*umpire\_resourcemanager \*self, void \*src\_ptr, void \*dst\_ptr*)

### Function `umpire_resourcemanager_copy_all(umpire_resourcemanager *, void *, void *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

### Function Documentation

void `umpire_resourcemanager_copy_all` (*umpire\_resourcemanager \*self, void \*src\_ptr, void \*dst\_ptr*)

### Function `umpire_resourcemanager_copy_with_size(umpire_resourcemanager *, void *, void *, size_t)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

void `umpire_resourcemanager_copy_with_size` (*umpire\_resourcemanager \*self, void \*src\_ptr, void \*dst\_ptr, size\_t size*)

### Function `umpire_resourcemanager_copy_with_size(umpire_resourcemanager *, void *, void *, size_t)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

## Function Documentation

void **umpire\_resourcemanager\_copy\_with\_size** (*umpire\_resourcemanager* \*self, void \*src\_ptr, void \*dst\_ptr, size\_t size)

## Function **umpire\_resourcemanager\_deallocate**(umpire\_resourcemanager \*, void \*)

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

## Function Documentation

void **umpire\_resourcemanager\_deallocate** (*umpire\_resourcemanager* \*self, void \*ptr)

## Function **umpire\_resourcemanager\_deallocate**(umpire\_resourcemanager \*, void \*)

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

## Function Documentation

void **umpire\_resourcemanager\_deallocate** (*umpire\_resourcemanager* \*self, void \*ptr)

## Function **umpire\_resourcemanager\_get\_allocator\_by\_id**(umpire\_resourcemanager \*, const int, umpire\_allocator \*)

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

## Function Documentation

*umpire\_allocator* \***umpire\_resourcemanager\_get\_allocator\_by\_id**(*umpire\_resourcemanager* \*self, **const** int id, *umpire\_allocator* \*SHC\_rv)

## Function **umpire\_resourcemanager\_get\_allocator\_by\_id**(umpire\_resourcemanager \*, const int, umpire\_allocator \*)

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

## Function Documentation

*umpire\_allocator* \***umpire\_resourcemanager\_get\_allocator\_by\_id**(*umpire\_resourcemanager* \*self, **const** int id, *umpire\_allocator* \*SHC\_rv)

**Function** `umpire_resource_manager_get_allocator_by_name(umpire_resource_manager *, const char *, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

```
umpire_allocator *umpire_resource_manager_get_allocator_by_name(umpire_resource_manager
                                                                *self,          const
                                                                char *name,      um-
                                                                pire_allocator
                                                                *SHC_rv)
```

**Function** `umpire_resource_manager_get_allocator_by_name(umpire_resource_manager *, const char *, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

### Function Documentation

```
umpire_allocator *umpire_resource_manager_get_allocator_by_name(umpire_resource_manager
                                                                *self,          const
                                                                char *name,      um-
                                                                pire_allocator
                                                                *SHC_rv)
```

**Function** `umpire_resource_manager_get_allocator_by_name_bufferify(umpire_resource_manager *, const char *, int, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

```
umpire_allocator *umpire_resource_manager_get_allocator_by_name_bufferify(umpire_resource_manager
                                                                *self,
                                                                const
                                                                char
                                                                *name,
                                                                int
                                                                Lname,
                                                                um-
                                                                pire_allocator
                                                                *SHC_rv)
```

**Function `umpire_resourcemanager_get_allocator_by_name_bufferify(umpire_resourcemanager *, const char *, int, umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_get_allocator_by_name_bufferify(umpire_resourcemanager
                                                                    *self,
                                                                    const
                                                                    char
                                                                    *name,
                                                                    int
                                                                    Lname,
                                                                    um-
                                                                    pire_allocator
                                                                    *SHC_rv)
```

**Function `umpire_resourcemanager_get_allocator_for_ptr(umpire_resourcemanager *, void *, umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_get_allocator_for_ptr(umpire_resourcemanager
                                                                    *self,    void    *ptr,
                                                                    umpire_allocator
                                                                    *SHC_rv)
```

**Function `umpire_resourcemanager_get_allocator_for_ptr(umpire_resourcemanager *, void *, umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_get_allocator_for_ptr(umpire_resourcemanager
                                                                    *self,    void    *ptr,
                                                                    umpire_allocator
                                                                    *SHC_rv)
```

### Function `umpire_resourcemanager_get_instance(umpire_resourcemanager *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

#### Function Documentation

*umpire\_resourcemanager* \***umpire\_resourcemanager\_get\_instance** (*umpire\_resourcemanager*  
\**SHC\_rv*)

### Function `umpire_resourcemanager_get_instance(umpire_resourcemanager *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

#### Function Documentation

*umpire\_resourcemanager* \***umpire\_resourcemanager\_get\_instance** (*umpire\_resourcemanager*  
\**SHC\_rv*)

### Function `umpire_resourcemanager_get_size(umpire_resourcemanager *, void *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

#### Function Documentation

size\_t **umpire\_resourcemanager\_get\_size** (*umpire\_resourcemanager* \**self*, void \**ptr*)

### Function `umpire_resourcemanager_get_size(umpire_resourcemanager *, void *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

#### Function Documentation

size\_t **umpire\_resourcemanager\_get\_size** (*umpire\_resourcemanager* \**self*, void \**ptr*)

### Function `umpire_resourcemanager_has_allocator(umpire_resourcemanager *, void *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp



### Function Documentation

bool `umpire_resourcemanager_has_allocator` (*umpire\_resourcemanager* \*self, void \*ptr)

Function `umpire_resourcemanager_has_allocator(umpire_resourcemanager *, void *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

### Function Documentation

bool `umpire_resourcemanager_has_allocator` (*umpire\_resourcemanager* \*self, void \*ptr)

Function `umpire_resourcemanager_is_allocator(umpire_resourcemanager *, const char *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

### Function Documentation

bool `umpire_resourcemanager_is_allocator` (*umpire\_resourcemanager* \*self, const char \*name)

Function `umpire_resourcemanager_is_allocator(umpire_resourcemanager *, const char *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

### Function Documentation

bool `umpire_resourcemanager_is_allocator` (*umpire\_resourcemanager* \*self, const char \*name)

Function `umpire_resourcemanager_is_allocator_bufferify(umpire_resourcemanager *, const char *, int)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

### Function Documentation

bool `umpire_resourcemanager_is_allocator_bufferify` (*umpire\_resourcemanager* \*self, const char \*name, int Lname)

Function `umpire_resourcemanager_is_allocator_bufferify(umpire_resourcemanager *, const char *, int)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

### Function Documentation

```
bool umpire_resourcemanager_is_allocator_bufferify (umpire_resourcemanager *self,
                                                    const char *name, int Lname)
```

Function `umpire_resourcemanager_make_allocator_advisor(umpire_resourcemanager *, const char *, umpire_allocator, const char *, int, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_advisor (umpire_resourcemanager
                                                                    *self,          const
                                                                    char *name,    um-
                                                                    pire_allocator alloca-
                                                                    tor, const char *ad-
                                                                    vice_op, int device_id,
                                                                    umpire_allocator
                                                                    *SHC_rv)
```

Function `umpire_resourcemanager_make_allocator_advisor(umpire_resourcemanager *, const char *, umpire_allocator, const char *, int, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_advisor (umpire_resourcemanager
                                                                    *self,          const
                                                                    char *name,    um-
                                                                    pire_allocator alloca-
                                                                    tor, const char *ad-
                                                                    vice_op, int device_id,
                                                                    umpire_allocator
                                                                    *SHC_rv)
```

Function `umpire_resourcemanager_make_allocator_bufferify_advisor(umpire_resourcemanager *, const char *, int, umpire_allocator, const char *, int, int, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_bufferify_advisor(umpire_resourcemanager
                                                                    *self,
                                                                    const
                                                                    char
                                                                    *name,
                                                                    int
                                                                    Lname,
                                                                    um-
                                                                    pire_allocator
                                                                    allo-
                                                                    cator,
                                                                    const
                                                                    char
                                                                    *ad-
                                                                    vice_op,
                                                                    int Lad-
                                                                    vice_op,
                                                                    int de-
                                                                    vice_id,
                                                                    um-
                                                                    pire_allocator
                                                                    *SHC_rv)
```

Function `umpire_resourcemanager_make_allocator_bufferify_advisor(umpire_resourcemanager *, const char *, int, umpire_allocator, const char *, int, int, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

## Function Documentation

*umpire\_allocator* \***umpire\_resourcemanager\_make\_allocator\_bufferify\_advisor**(*umpire\_resourcemanager* \*self, **const** char \*name, int Lname, *um-  
pire\_allocator* allocator, **const** char \*ad-  
vice\_op, int Lad-  
vice\_op, int de-  
vice\_id, *um-  
pire\_allocator* \*SHC\_rv)

**Function `umpire_resourcemanager_make_allocator_bufferify_fixed_pool(umpire_resourcemanager *, const char *, int, umpire_allocator, size_t, umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

## Function Documentation

*umpire\_allocator* \***umpire\_resourcemanager\_make\_allocator\_bufferify\_fixed\_pool**(*umpire\_resourcemanager* \*self, **const** char \*name, int Lname, *um-  
pire\_allocator* al-  
lo-  
ca-  
tor, size\_t ob-  
ject\_size, *um-  
pire\_allocator* \*SHC\_rv)

**Function `umpire_resourcemanager_make_allocator_bufferify_fixed_pool(umpire_resourcemanager *, const char *, int, umpire_allocator, size_t, umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_bufferify_fixed_pool(umpire_resourcemanager
    *self,
    const
    char
    *name,
    int
    Lname,
    um-
pire_allocator
    al-
    lo-
    ca-
    tor,
    size_t
    ob-
    ject_size,
    um-
pire_allocator
    *SHC_rv)
```

**Function `umpire_resourcemanager_make_allocator_bufferify_list_pool(umpire_resourcemanager *, const char *, int, umpire_allocator, size_t, size_t, umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

## Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_bufferify_list_pool(umpire_resourcemanager
    *self,
    const
    char
    *name,
    int
    Lname,
    um-
pire_allocator
    allo-
    ca-
    tor,
    size_t
    ini-
    tial_size,
    size_t
    block,
    um-
pire_allocator
    *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_bufferify_list_pool(umpire_resourcemanager *, const char *, int, umpire_allocator, size_t, size_t, umpire_allocator *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

## Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_bufferify_list_pool(umpire_resourcemanager
    *self,
    const
    char
    *name,
    int
    Lname,
    um-
pire_allocator
    allo-
    ca-
    tor,
    size_t
    ini-
    tial_size,
    size_t
    block,
    um-
pire_allocator
    *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_bufferify_named(umpire_resourcemanager *, const char *, int, umpire_allocator, umpire_allocator *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_bufferify_named(umpire_resourcemanager
                                                                    *self,
                                                                    const
                                                                    char
                                                                    *name, int
                                                                    Lname,
                                                                    um-
                                                                    pire_allocator
                                                                    allocator,
                                                                    um-
                                                                    pire_allocator
                                                                    *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_bufferify_named(umpire_resourcemanager *, const char *, int, umpire_allocator, umpire_allocator *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_bufferify_named(umpire_resourcemanager
                                                                    *self,
                                                                    const
                                                                    char
                                                                    *name, int
                                                                    Lname,
                                                                    um-
                                                                    pire_allocator
                                                                    allocator,
                                                                    um-
                                                                    pire_allocator
                                                                    *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_bufferify_pool(umpire_resourcemanager *, const char *, int, umpire_allocator, size_t, size_t, umpire_allocator *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

## Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_bufferify_pool(umpire_resourcemanager
    *self,
    const char
    *name, int
    Lname, um-
pire_allocator
    allocator,
    size_t ini-
    tial_size,
    size_t
    block, um-
pire_allocator
    *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_bufferify_pool(umpire_resourcemanager *, const char *, int, umpire_allocator, size_t, size_t, umpire_allocator *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

## Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_bufferify_pool(umpire_resourcemanager
    *self,
    const char
    *name, int
    Lname, um-
pire_allocator
    allocator,
    size_t ini-
    tial_size,
    size_t
    block, um-
pire_allocator
    *SHC_rv)
```



**Function `umpire_resourcemanager_make_allocator_bufferify_prefetcher(umpire_resourcemanager *, const char *, int, umpire_allocator, int, umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_bufferify_prefetcher(umpire_resourcemanager
    *self,
    const
    char
    *name,
    int
    Lname,
    um-
pire_allocator
    al-
    lo-
    ca-
    tor,
    int
    de-
    vice_id,
    um-
pire_allocator
    *SHC_rv)
```

**Function `umpire_resourcemanager_make_allocator_bufferify_prefetcher(umpire_resourcemanager *, const char *, int, umpire_allocator, int, umpire_allocator *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

## Function Documentation

*umpire\_allocator* \***umpire\_resourcemanager\_make\_allocator\_bufferify\_prefetcher**(*umpire\_resourcemanager* \*self, **const** char \*name, int Lname, *um-  
pire\_allocator* al-  
lo-  
ca-  
tor, int device\_id, *um-  
pire\_allocator* \*SHC\_rv)

**Function `umpire_resourcemanager_make_allocator_bufferify_thread_safe`**(*umpire\_resourcemanager* \*, **const** char \*, int, *umpire\_allocator*, *umpire\_allocator* \*)

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

## Function Documentation

*umpire\_allocator* \***umpire\_resourcemanager\_make\_allocator\_bufferify\_thread\_safe**(*umpire\_resourcemanager* \*self, **const** char \*name, int Lname, *um-  
pire\_allocator* al-  
lo-  
ca-  
tor, *um-  
pire\_allocator* \*SHC\_rv)

Function `umpire_resource_manager_make_allocator_bufferify_thread_safe(umpire_resource_manager *, const char *, int, umpire_allocator, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

### Function Documentation

```
umpire_allocator *umpire_resource_manager_make_allocator_bufferify_thread_safe(umpire_resource_manager
                                                                    *self,
                                                                    const
                                                                    char
                                                                    *name,
                                                                    int
                                                                    Lname,
                                                                    um-
                                                                    pire_allocator
                                                                    al-
                                                                    lo-
                                                                    ca-
                                                                    tor,
                                                                    um-
                                                                    pire_allocator
                                                                    *SHC_rv)
```

Function `umpire_resource_manager_make_allocator_fixed_pool(umpire_resource_manager *, const char *, umpire_allocator, size_t, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

```
umpire_allocator *umpire_resource_manager_make_allocator_fixed_pool(umpire_resource_manager
                                                                    *self, const char
                                                                    *name,      um-
                                                                    pire_allocator
                                                                    allocator, size_t
                                                                    object_size, um-
                                                                    pire_allocator
                                                                    *SHC_rv)
```

Function `umpire_resource_manager_make_allocator_fixed_pool(umpire_resource_manager *, const char *, umpire_allocator, size_t, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

## Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_fixed_pool(umpire_resourcemanager
    *self, const char
    *name, um-
pire_allocator
    allocator, size_t
    object_size, um-
pire_allocator
    *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_list_pool(umpire_resourcemanager *, const char *, umpire_allocator, size_t, size_t, umpire_allocator *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

## Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_list_pool(umpire_resourcemanager
    *self, const
    char *name, um-
pire_allocator
    allocator, size_t
    initial_size,
    size_t block,
    umpire_allocator
    *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_list_pool(umpire_resourcemanager *, const char *, umpire_allocator, size_t, size_t, umpire_allocator *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

## Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_list_pool(umpire_resourcemanager
    *self, const
    char *name, um-
pire_allocator
    allocator, size_t
    initial_size,
    size_t block,
    umpire_allocator
    *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_named(umpire_resourcemanager *, const char *, umpire_allocator, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_named(umpire_resourcemanager
                                                                *self, const char
                                                                *name, umpire_allocator
                                                                allocator, um-
                                                                pire_allocator *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_named(umpire_resourcemanager *, const char *, umpire_allocator, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_named(umpire_resourcemanager
                                                                *self, const char
                                                                *name, umpire_allocator
                                                                allocator, um-
                                                                pire_allocator *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_pool(umpire_resourcemanager *, const char *, umpire_allocator, size_t, size_t, umpire_allocator *)`

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_pool(umpire_resourcemanager
                                                                *self, const char *name,
                                                                umpire_allocator alloca-
                                                                tor, size_t initial_size,
                                                                size_t block, um-
                                                                pire_allocator *SHC_rv)
```

**Function `umpire_resource_manager_make_allocator_pool(umpire_resource_manager *, const char *, umpire_allocator, size_t, size_t, umpire_allocator *)`**

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

### Function Documentation

```
umpire_allocator *umpire_resource_manager_make_allocator_pool(umpire_resource_manager
                                                                *self, const char *name,
                                                                umpire_allocator allocator,
                                                                size_t initial_size,
                                                                size_t block, umpire_allocator *SHC_rv)
```

**Function `umpire_resource_manager_make_allocator_prefetcher(umpire_resource_manager *, const char *, umpire_allocator, int, umpire_allocator *)`**

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

```
umpire_allocator *umpire_resource_manager_make_allocator_prefetcher(umpire_resource_manager
                                                                *self, const char
                                                                *name, umpire_allocator
                                                                allocator, int
                                                                device_id, umpire_allocator
                                                                *SHC_rv)
```

**Function `umpire_resource_manager_make_allocator_prefetcher(umpire_resource_manager *, const char *, umpire_allocator, int, umpire_allocator *)`**

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

### Function Documentation

```
umpire_allocator *umpire_resource_manager_make_allocator_prefetcher(umpire_resource_manager
                                                                *self, const char
                                                                *name, umpire_allocator
                                                                allocator, int
                                                                device_id, umpire_allocator
                                                                *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_thread_safe(umpire_resourcemanager *, const char *, umpire_allocator, umpire_allocator *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_thread_safe(umpire_resourcemanager
                                                                    *self, const
                                                                    char *name, um-
                                                                    pire_allocator
                                                                    allocator, um-
                                                                    pire_allocator
                                                                    *SHC_rv)
```

**Function** `umpire_resourcemanager_make_allocator_thread_safe(umpire_resourcemanager *, const char *, umpire_allocator, umpire_allocator *)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

### Function Documentation

```
umpire_allocator *umpire_resourcemanager_make_allocator_thread_safe(umpire_resourcemanager
                                                                    *self, const
                                                                    char *name, um-
                                                                    pire_allocator
                                                                    allocator, um-
                                                                    pire_allocator
                                                                    *SHC_rv)
```

**Function** `umpire_resourcemanager_memset_all(umpire_resourcemanager *, void *, int)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

### Function Documentation

```
void umpire_resourcemanager_memset_all (umpire_resourcemanager *self, void *ptr, int val)
```

**Function** `umpire_resourcemanager_memset_all(umpire_resourcemanager *, void *, int)`

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

### Function Documentation

void **umpire\_resourcemanager\_memset\_all** (*umpire\_resourcemanager \*self*, void \**ptr*, int *val*)

**Function umpire\_resourcemanager\_memset\_with\_size**(*umpire\_resourcemanager \**, void \*, int, *size\_t*)

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

### Function Documentation

void **umpire\_resourcemanager\_memset\_with\_size** (*umpire\_resourcemanager \*self*, void \**ptr*, int *val*, *size\_t length*)

**Function umpire\_resourcemanager\_memset\_with\_size**(*umpire\_resourcemanager \**, void \*, int, *size\_t*)

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

### Function Documentation

void **umpire\_resourcemanager\_memset\_with\_size** (*umpire\_resourcemanager \*self*, void \**ptr*, int *val*, *size\_t length*)

**Function umpire\_resourcemanager\_move**(*umpire\_resourcemanager \**, void \*, *umpire\_allocator*)

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.cpp

### Function Documentation

void \***umpire\_resourcemanager\_move** (*umpire\_resourcemanager \*self*, void \**src\_ptr*, *umpire\_allocator allocator*)

**Function umpire\_resourcemanager\_move**(*umpire\_resourcemanager \**, void \*, *umpire\_allocator*)

- Defined in file\_umpire\_interface\_c\_fortran\_wrapResourceManager.h

### Function Documentation

void \***umpire\_resourcemanager\_move** (*umpire\_resourcemanager \*self*, void \**src\_ptr*, *umpire\_allocator allocator*)



**Function `umpire_resource_manager_reallocate_default(umpire_resource_manager *, void *, size_t)`**

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

**Function Documentation**

```
void *umpire_resource_manager_reallocate_default (umpire_resource_manager *self, void
                                                *src_ptr, size_t size)
```

**Function `umpire_resource_manager_reallocate_default(umpire_resource_manager *, void *, size_t)`**

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

**Function Documentation**

```
void *umpire_resource_manager_reallocate_default (umpire_resource_manager *self, void
                                                *src_ptr, size_t size)
```

**Function `umpire_resource_manager_reallocate_with_allocator(umpire_resource_manager *, void *, size_t, umpire_allocator)`**

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.cpp`

**Function Documentation**

```
void *umpire_resource_manager_reallocate_with_allocator (umpire_resource_manager
                                                         *self, void *src_ptr, size_t size,
                                                         umpire_allocator allocator)
```

**Function `umpire_resource_manager_reallocate_with_allocator(umpire_resource_manager *, void *, size_t, umpire_allocator)`**

- Defined in file `umpire_interface_c_fortran_wrapResourceManager.h`

**Function Documentation**

```
void *umpire_resource_manager_reallocate_with_allocator (umpire_resource_manager
                                                         *self, void *src_ptr, size_t size,
                                                         umpire_allocator allocator)
```

**Function `umpire_resourcemanager_register_allocator(umpire_resourcemanager *, const char *, umpire_allocator)`**

- Defined in file `_umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

```
void umpire_resourcemanager_register_allocator (umpire_resourcemanager *self, const char *name, umpire_allocator allocator)
```

**Function `umpire_resourcemanager_register_allocator(umpire_resourcemanager *, const char *, umpire_allocator)`**

- Defined in file `_umpire_interface_c_fortran_wrapResourceManager.h`

### Function Documentation

```
void umpire_resourcemanager_register_allocator (umpire_resourcemanager *self, const char *name, umpire_allocator allocator)
```

**Function `umpire_resourcemanager_register_allocator_bufferify(umpire_resourcemanager *, const char *, int, umpire_allocator)`**

- Defined in file `_umpire_interface_c_fortran_wrapResourceManager.cpp`

### Function Documentation

```
void umpire_resourcemanager_register_allocator_bufferify (umpire_resourcemanager *self, const char *name, int Lname, umpire_allocator allocator)
```

**Function `umpire_resourcemanager_register_allocator_bufferify(umpire_resourcemanager *, const char *, int, umpire_allocator)`**

- Defined in file `_umpire_interface_c_fortran_wrapResourceManager.h`

### Function Documentation

```
void umpire_resourcemanager_register_allocator_bufferify (umpire_resourcemanager *self, const char *name, int Lname, umpire_allocator allocator)
```

**Function `umpire_SHROUD_memory_destructor(umpire_SHROUD_capsule_data *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_typesUmpire.h

**Function Documentation**

```
void umpire_SHROUD_memory_destructor (umpire_SHROUD_capsule_data *cap)
```

**Function `umpire_SHROUD_memory_destructor(umpire_SHROUD_capsule_data *)`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapUmpire.cpp

**Function Documentation**

```
void umpire_SHROUD_memory_destructor (umpire_SHROUD_capsule_data *cap)
```

**Function `umpire_ShroudCopyStringAndFree`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapAllocator.cpp

**Function Documentation**

**Warning:** doxygenfunction: Cannot find function “umpire\_ShroudCopyStringAndFree” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

**Function `umpire_strategy_mod::allocationadvisor_associated`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfUmpire\_strategy.f

**Function Documentation**

```
logical function umpire_strategy_mod::allocationadvisor_associated(obj obj)
```

**Function `umpire_strategy_mod::allocationadvisor_eq`**

- Defined in file\_umpire\_interface\_c\_fortran\_wrapfUmpire\_strategy.f

### Function Documentation

```
logical function umpire_strategy_mod::allocationadvisor_eq(a a, b b)
```

### Function `umpire_strategy_mod::allocationadvisor_get_instance`

- Defined in file `umpire_interface_c_fortran_wrapfUmpire_strategy.f`

### Function Documentation

```
type(c_ptr) function umpire_strategy_mod::allocationadvisor_get_instance(obj obj)
```

### Function `umpire_strategy_mod::allocationadvisor_ne`

- Defined in file `umpire_interface_c_fortran_wrapfUmpire_strategy.f`

### Function Documentation

```
logical function umpire_strategy_mod::allocationadvisor_ne(a a, b b)
```

### Function `umpire_strategy_mod::allocationadvisor_set_instance`

- Defined in file `umpire_interface_c_fortran_wrapfUmpire_strategy.f`

### Function Documentation

```
subroutine umpire_strategy_mod::allocationadvisor_set_instance (obj obj, cxxmem  
cxxmem)
```

### Function `umpire_strategy_mod::dynamicpool_associated`

- Defined in file `umpire_interface_c_fortran_wrapfUmpire_strategy.f`

### Function Documentation

```
logical function umpire_strategy_mod::dynamicpool_associated(obj obj)
```

**Function `umpire_strategy_mod::dynamicpool_eq`**

- Defined in `file_umpire_interface_c_fortran_wrapfUmpire_strategy.f`

**Function Documentation**

```
logical function umpire_strategy_mod::dynamicpool_eq(a a, b b)
```

**Function `umpire_strategy_mod::dynamicpool_get_instance`**

- Defined in `file_umpire_interface_c_fortran_wrapfUmpire_strategy.f`

**Function Documentation**

```
type(c_ptr) function umpire_strategy_mod::dynamicpool_get_instance(obj obj)
```

**Function `umpire_strategy_mod::dynamicpool_ne`**

- Defined in `file_umpire_interface_c_fortran_wrapfUmpire_strategy.f`

**Function Documentation**

```
logical function umpire_strategy_mod::dynamicpool_ne(a a, b b)
```

**Function `umpire_strategy_mod::dynamicpool_set_instance`**

- Defined in `file_umpire_interface_c_fortran_wrapfUmpire_strategy.f`

**Function Documentation**

```
subroutine umpire_strategy_mod::dynamicpool_set_instance (obj obj, cxxmem cxxmem)
```

**Function `umpire_strategy_mod::namedallocationstrategy_associated`**

- Defined in `file_umpire_interface_c_fortran_wrapfUmpire_strategy.f`

**Function Documentation**

```
logical function umpire_strategy_mod::namedallocationstrategy_associated(obj obj)
```

### Function `umpire_strategy_mod::namedallocationstrategy_eq`

- Defined in file `umpire_interface_c_fortran_wrapfUmpire_strategy.f`

#### Function Documentation

```
logical function umpire_strategy_mod::namedallocationstrategy_eq(a a, b b)
```

### Function `umpire_strategy_mod::namedallocationstrategy_get_instance`

- Defined in file `umpire_interface_c_fortran_wrapfUmpire_strategy.f`

#### Function Documentation

```
type(c_ptr) function umpire_strategy_mod::namedallocationstrategy_get_instance(obj obj)
```

### Function `umpire_strategy_mod::namedallocationstrategy_ne`

- Defined in file `umpire_interface_c_fortran_wrapfUmpire_strategy.f`

#### Function Documentation

```
logical function umpire_strategy_mod::namedallocationstrategy_ne(a a, b b)
```

### Function `umpire_strategy_mod::namedallocationstrategy_set_instance`

- Defined in file `umpire_interface_c_fortran_wrapfUmpire_strategy.f`

#### Function Documentation

```
subroutine umpire_strategy_mod::namedallocationstrategy_set_instance (obj obj,  
                                                                    cxxmem  
                                                                    cxxmem)
```

## 6.3.5 Variables

### Variable `genumpiresplicer::maxdims`

- Defined in file `umpire_interface_c_fortran_genumpiresplicer.py`

## Variable Documentation

```
int genumpiresplicer.maxdims = 3
```

## Variable genumpiresplicer::types

- Defined in file\_umpire\_interface\_c\_fortran\_genumpiresplicer.py

## Variable Documentation

```
tuple genumpiresplicer.types= ( ('int', 'integer(C_INT) '), ('long', 'integer(C_LONG)'))
```

## Variable s\_null\_resource\_name

- Defined in file\_umpire\_ResourceManager.cpp

## Variable Documentation

**Warning:** doxygenvariable: Cannot find variable “s\_null\_resource\_name” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

## Variable s\_umpire\_internal\_device\_constant\_memory

- Defined in file\_umpire\_resource\_HipConstantMemoryResource.cpp

## Variable Documentation

**Warning:** doxygenvariable: Cannot find variable “s\_umpire\_internal\_device\_constant\_memory” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

## Variable s\_zero\_byte\_pool\_name

- Defined in file\_umpire\_ResourceManager.cpp

## Variable Documentation

**Warning:** doxygenvariable: Cannot find variable “s\_zero\_byte\_pool\_name” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

## Variable umpire::env\_name

- Defined in file\_umpire\_Replay.cpp

## Variable Documentation

`const char *umpire::env_name = "UMPIRE_REPLAY"`

## Variable `umpire::strategy::bits_per_int`

- Defined in `file_umpire_strategy_FixedPool.cpp`

## Variable Documentation

`constexpr std::size_t umpire::strategy::bits_per_int = sizeof(int) * 8`

## Variable `umpire::strategy::heuristic_percent_releasable`

- Defined in `file_umpire_strategy_DynamicPoolHeuristic.hpp`

## Variable Documentation

`std::function<bool (const strategy::DynamicPoolMap&)> umpire::strategy::heuristic_percent_releasable`  
`int percentage` Return true if specified percentage of pool is releasable.

When the specified percentage of the pool has been deallocated back to the pool, this heuristic will return true.

**Return** True if specified percentage of memory in pool is releasable.

### Parameters

- `percentage`: The integer percentage of releasable memory to actual memory used by the pool.

## Variable `umpire::strategy::heuristic_percent_releasable_list`

- Defined in `file_umpire_strategy_DynamicPoolHeuristic.hpp`

## Variable Documentation

`std::function<bool (const strategy::DynamicPoolList&)> umpire::strategy::heuristic_percent_releasable_list`  
`int percentage` Return true if specified percentage of pool is releasable.

When the specified percentage of the pool has been deallocated back to the pool, this heuristic will return true.

**Return** True if specified percentage of memory in pool is releasable.

### Parameters

- `percentage`: The integer percentage of releasable memory to actual memory used by the pool.



### Variable `umpire::util::defaultLevel`

- Defined in `file_umpire_util_Logger.cpp`

### Variable Documentation

`message::Level` `umpire::util::defaultLevel` = `message::Info`

### Variable `umpire::util::env_name`

- Defined in `file_umpire_util_Logger.cpp`

### Variable Documentation

`const char*` `umpire::util::env_name` = "UMPIRE\_LOG\_LEVEL"

### Variable `umpire::util::MessageLevelName`

- Defined in `file_umpire_util_Logger.cpp`

### Variable Documentation

`const char*` `umpire::util::MessageLevelName`[`message::Num_Levels`] = {"ERROR", , , }

### Variable `umpire_ver_2_found`

- Defined in `file_umpire_Umpire.cpp`

### Variable Documentation

**Warning:** doxygenvariable: Cannot find variable “`umpire_ver_2_found`” in doxygen xml output for project “`umpire`” from directory: `../doxygen/xml/`

## 6.3.6 Defines

### Define `_XOPEN_SOURCE_EXTENDED`

- Defined in `file_umpire_strategy_FixedPool.cpp`

## Define Documentation

**Warning:** doxygendefine: Cannot find define “\_XOPEN\_SOURCE\_EXTENDED” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

### Define `UMPIRE_Allocator_INL`

- Defined in file\_umpire\_Allocator.inl

## Define Documentation

`UMPIRE_Allocator_INL`

### Define `UMPIRE_ASSERT`

- Defined in file\_umpire\_util\_Macros.hpp

## Define Documentation

`UMPIRE_ASSERT` (condition)

### Define `UMPIRE_DefaultMemoryResource_INL`

- Defined in file\_umpire\_resource\_DefaultMemoryResource.inl

## Define Documentation

`UMPIRE_DefaultMemoryResource_INL`

### Define `UMPIRE_ERROR`

- Defined in file\_umpire\_util\_Macros.hpp

## Define Documentation

`UMPIRE_ERROR` (msg)

### Define `UMPIRE_LOG`

- Defined in file\_umpire\_util\_Macros.hpp

### Define Documentation

`UMPIRE_LOG` (lvl, msg)

### Define `UMPIRE_MemoryMap_INL`

- Defined in file\_umpire\_util\_MemoryMap.inl

### Define Documentation

`UMPIRE_MemoryMap_INL`

### Define `UMPIRE_NullMemoryResource_INL`

- Defined in file\_umpire\_resource\_NullMemoryResource.cpp

### Define Documentation

<p><b>Warning:</b> doxygendefine: Cannot find define “UMPIRE_NullMemoryResource_INL” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/</p>
---

### Define `UMPIRE_RECORD_STATISTIC`

- Defined in file\_umpire\_util\_Macros.hpp

### Define Documentation

`UMPIRE_RECORD_STATISTIC` (name, ...)

### Define `UMPIRE_REPLAY`

- Defined in file\_umpire\_Replay.hpp

### Define Documentation

**UMPIRE\_REPLAY** (msg)

### Define UMPIRE\_ResourceManager\_INL

- Defined in file\_umpire\_ResourceManager.inl

### Define Documentation

**UMPIRE\_ResourceManager\_INL**

### Define UMPIRE\_TypedAllocator\_INL

- Defined in file\_umpire\_TypedAllocator.inl

### Define Documentation

**UMPIRE\_TypedAllocator\_INL**

### Define UMPIRE\_UNUSED\_ARG

- Defined in file\_umpire\_util\_Macros.hpp

### Define Documentation

**UMPIRE\_UNUSED\_ARG** (x)

### Define UMPIRE\_USE\_VAR

- Defined in file\_umpire\_util\_Macros.hpp

### Define Documentation

**UMPIRE\_USE\_VAR** (x)

## 6.3.7 Typedefs

### Typedef umpire::strategy::DynamicPool

- Defined in file\_umpire\_strategy\_DynamicPool.hpp

### Typedef Documentation

`using` `umpire::strategy::DynamicPool = DynamicPoolMap`

### Typedef `umpire_allocator`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

### Typedef Documentation

`typedef struct s_umpire_allocator umpire_allocator`

### Typedef `umpire_resourcemanager`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

### Typedef Documentation

`typedef struct s_umpire_resourcemanager umpire_resourcemanager`

### Typedef `umpire_SHROUD_array`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

### Typedef Documentation

`typedef struct s_umpire_SHROUD_array umpire_SHROUD_array`

### Typedef `umpire_SHROUD_capsule_data`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

### Typedef Documentation

`typedef struct s_umpire_SHROUD_capsule_data umpire_SHROUD_capsule_data`

### Typedef `umpire_strategy_allocationadvisor`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

### Typedef Documentation

`typedef struct s_umpire_strategy_allocationadvisor umpire_strategy_allocationadvisor`

### Typedef `umpire_strategy_allocationprefetcher`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

### Typedef Documentation

`typedef struct s_umpire_strategy_allocationprefetcher umpire_strategy_allocationprefetcher`

### Typedef `umpire_strategy_dynamicpool`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

### Typedef Documentation

`typedef struct s_umpire_strategy_dynamicpool umpire_strategy_dynamicpool`

### Typedef `umpire_strategy_dynamicpoollist`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

### Typedef Documentation

`typedef struct s_umpire_strategy_dynamicpoollist umpire_strategy_dynamicpoollist`

### Typedef `umpire_strategy_fixedpool`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

### Typedef Documentation

`typedef struct s_umpire_strategy_fixedpool umpire_strategy_fixedpool`

### Typedef `umpire_strategy_namedallocationstrategy`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

### Typedef Documentation

```
typedef struct s_umpire_strategy_namedallocationstrategy umpire_strategy_namedallocationstrategy
```

### Typedef `umpire_strategy_threadsafeallocator`

- Defined in file `umpire_interface_c_fortran_typesUmpire.h`

### Typedef Documentation

```
typedef struct s_umpire_strategy_threadsafeallocator umpire_strategy_threadsafeallocator
```





## CONTRIBUTION GUIDE

This document is intended for developers who want to add new features or bugfixes to Umpire. It assumes you have some familiarity with git and GitHub. It will discuss what a good pull request (PR) looks like, and the tests that your PR must pass before it can be merged into Umpire.

### 7.1 Forking Umpire

If you aren't an Umpire developer at LLNL, then you won't have permission to push new branches to the repository. First, you should create a [fork](#). This will create a copy of the Umpire repository that you own, and will ensure you can push your changes up to GitHub and create pull requests.

#### 7.1.1 Developing a New Feature

New features should be based on the `develop` branch. When you want to create a new feature, first ensure you have an up-to-date copy of the `develop` branch:

```
$ git checkout develop
$ git pull origin develop
```

You can now create a new branch to develop your feature on:

```
$ git checkout -b feature/<name-of-feature>
```

Proceed to develop your feature on this branch, and add tests that will exercise your new code. If you are creating new methods or classes, please add Doxygen documentation.

Once your feature is complete and your tests are passing, you can push your branch to GitHub and create a PR.

#### 7.1.2 Developing a Bug Fix

First, check if the change you want to make has been fixed in `develop`. If so, we suggest you either start using the `develop` branch, or temporarily apply the fix to whichever version of Umpire you are using.

If the bug is still unfixed, first make sure you have an up-to-date copy of the `develop` branch:

```
$ git checkout develop
$ git pull origin develop
```

Then create a new branch for your bugfix:

```
$ git checkout -b bugfix/<name-of-bug>
```

First, add a test that reproduces the bug you have found. Then develop your bugfix as normal, and ensure to make `test` to check your changes actually fix the bug.

Once you are finished, you can push your branch to GitHub, then create a PR.

### 7.1.3 Creating a Pull Request

You can create a new PR [here](#). Ensure that your PR base is the `develop` branch of Umpire.

Add a descriptive title explaining the bug you fixed or the feature you have added, and put a longer description of the changes you have made in the comment box.

Once your PR has been created, it will be run through our automated tests and also be reviewed by Umpire team members. Providing the branch passes both the tests and reviews, it will be merged into Umpire.

### 7.1.4 Tests

Umpire uses Bamboo for continuous integration tests. Our tests are automatically run against every new pull request, and passing all tests is a requirement for merging your PR. If you are developing a bugfix or a new feature, please add a test that checks the correctness of your new code. Umpire is used on a wide variety of systems with a number of configurations, and adding new tests helps ensure that all features work as expected across these environments.

Umpire's tests are all in the `test` directory and are split up by component.

**DEVELOPER GUIDE**



## D

- DynamicSizePool (C++ class), 79
- DynamicSizePool::~~DynamicSizePool (C++ function), 79
- DynamicSizePool::alignmentAdjust (C++ function), 79
- DynamicSizePool::allocate (C++ function), 79
- DynamicSizePool::allocateBlock (C++ function), 79
- DynamicSizePool::allocator (C++ member), 80
- DynamicSizePool::allocBytes (C++ member), 80
- DynamicSizePool::Block (C++ class), 64, 80
- DynamicSizePool::blockPool (C++ member), 80
- DynamicSizePool::BlockPool (C++ type), 79
- DynamicSizePool::coalesce (C++ function), 79
- DynamicSizePool::coalesceFreeBlocks (C++ function), 79
- DynamicSizePool::deallocate (C++ function), 79
- DynamicSizePool::DynamicSizePool (C++ function), 79
- DynamicSizePool::findUsableBlock (C++ function), 79
- DynamicSizePool::freeAllBlocks (C++ function), 79
- DynamicSizePool::freeBlocks (C++ member), 80
- DynamicSizePool::freeReleasedBlocks (C++ function), 79
- DynamicSizePool::getActualSize (C++ function), 79
- DynamicSizePool::getBlocksInPool (C++ function), 79
- DynamicSizePool::getCurrentSize (C++ function), 79
- DynamicSizePool::getFreeBlocks (C++ function), 79
- DynamicSizePool::getHighWatermark (C++ function), 79
- DynamicSizePool::getInUseBlocks (C++ function), 79
- DynamicSizePool::getLargestAvailableBlock (C++ function), 79
- DynamicSizePool::getReleasableSize (C++ function), 79
- DynamicSizePool::highWatermark (C++ member), 80
- DynamicSizePool::minBytes (C++ member), 80
- DynamicSizePool::minInitialBytes (C++ member), 80
- DynamicSizePool::release (C++ function), 79
- DynamicSizePool::releaseBlock (C++ function), 79
- DynamicSizePool::splitBlock (C++ function), 79
- DynamicSizePool::totalBlocks (C++ member), 80
- DynamicSizePool::totalBytes (C++ member), 80
- DynamicSizePool::usedBlocks (C++ member), 80
- DynamicSizePool<IA>::Block::blockSize (C++ member), 64, 80
- DynamicSizePool<IA>::Block::data (C++ member), 64, 80
- DynamicSizePool<IA>::Block::next (C++ member), 64, 80
- DynamicSizePool<IA>::Block::size (C++ member), 64, 80

## F

- FixedSizePool (C++ class), 81
- FixedSizePool::~~FixedSizePool (C++ function), 81
- FixedSizePool::allocate (C++ function), 81
- FixedSizePool::allocInPool (C++ function), 81
- FixedSizePool::deallocate (C++ function), 81
- FixedSizePool::FixedSizePool (C++ function), 81

FixedSizePool::getActualSize (C++ *function*), 81  
 FixedSizePool::getCurrentSize (C++ *function*), 81  
 FixedSizePool::getInstance (C++ *function*), 81  
 FixedSizePool::newPool (C++ *function*), 81  
 FixedSizePool::numBlocks (C++ *member*), 81  
 FixedSizePool::numPerPool (C++ *member*), 81  
 FixedSizePool::numPools (C++ *function*), 81  
 FixedSizePool::Pool (C++ *class*), 64, 81  
 FixedSizePool::pool (C++ *member*), 81  
 FixedSizePool::poolSize (C++ *function*), 81  
 FixedSizePool::totalPoolSize (C++ *member*), 81  
 FixedSizePool<T, MA, IA, NP>::Pool::avail (C++ *member*), 64, 82  
 FixedSizePool<T, MA, IA, NP>::Pool::data (C++ *member*), 64, 82  
 FixedSizePool<T, MA, IA, NP>::Pool::next (C++ *member*), 64, 82  
 FixedSizePool<T, MA, IA, NP>::Pool::numAvail (C++ *member*), 64, 82

**G**

genumpiresplicer.gen\_bounds() (*built-in function*), 177  
 genumpiresplicer.gen\_fortran() (*built-in function*), 177  
 genumpiresplicer.gen\_methods() (*built-in function*), 177

**O**

operator<< (C++ *function*), 84, 136

**S**

s\_umpire\_allocator (C++ *class*), 65  
 s\_umpire\_allocator::addr (C++ *member*), 65  
 s\_umpire\_allocator::idtor (C++ *member*), 65  
 s\_umpire\_resourcemanager (C++ *class*), 65  
 s\_umpire\_resourcemanager::addr (C++ *member*), 65  
 s\_umpire\_resourcemanager::idtor (C++ *member*), 65  
 s\_umpire\_SHROUD\_array (C++ *class*), 65  
 s\_umpire\_SHROUD\_array::addr (C++ *member*), 65  
 s\_umpire\_SHROUD\_array::csharp (C++ *member*), 65  
 s\_umpire\_SHROUD\_array::cvoidp (C++ *member*), 65  
 s\_umpire\_SHROUD\_array::cxx (C++ *member*), 65  
 s\_umpire\_SHROUD\_array::len (C++ *member*), 65  
 s\_umpire\_SHROUD\_array::size (C++ *member*), 65  
 s\_umpire\_SHROUD\_capsule\_data (C++ *class*), 66  
 s\_umpire\_SHROUD\_capsule\_data::addr (C++ *member*), 66  
 s\_umpire\_SHROUD\_capsule\_data::idtor (C++ *member*), 66  
 s\_umpire\_strategy\_allocationadvisor (C++ *class*), 66  
 s\_umpire\_strategy\_allocationadvisor::addr (C++ *member*), 66  
 s\_umpire\_strategy\_allocationadvisor::idtor (C++ *member*), 66  
 s\_umpire\_strategy\_allocationprefetcher (C++ *class*), 66  
 s\_umpire\_strategy\_allocationprefetcher::addr (C++ *member*), 66  
 s\_umpire\_strategy\_allocationprefetcher::idtor (C++ *member*), 66  
 s\_umpire\_strategy\_dynamicpool (C++ *class*), 67  
 s\_umpire\_strategy\_dynamicpool::addr (C++ *member*), 67  
 s\_umpire\_strategy\_dynamicpool::idtor (C++ *member*), 67  
 s\_umpire\_strategy\_dynamicpoollist (C++ *class*), 67  
 s\_umpire\_strategy\_dynamicpoollist::addr (C++ *member*), 67  
 s\_umpire\_strategy\_dynamicpoollist::idtor (C++ *member*), 67  
 s\_umpire\_strategy\_fixedpool (C++ *class*), 67  
 s\_umpire\_strategy\_fixedpool::addr (C++ *member*), 67  
 s\_umpire\_strategy\_fixedpool::idtor (C++ *member*), 67  
 s\_umpire\_strategy\_namedallocationstrategy (C++ *class*), 68  
 s\_umpire\_strategy\_namedallocationstrategy::addr (C++ *member*), 68  
 s\_umpire\_strategy\_namedallocationstrategy::idtor (C++ *member*), 68  
 s\_umpire\_strategy\_threadsafeallocator (C++ *class*), 68  
 s\_umpire\_strategy\_threadsafeallocator::addr (C++ *member*), 68  
 s\_umpire\_strategy\_threadsafeallocator::idtor

(C++ member), 68  
 shroud\_FccCopy (C++ function), 178  
 StdAllocator (C++ class), 68  
 StdAllocator::allocate (C++ function), 68  
 StdAllocator::deallocate (C++ function), 68

**U**

umpire::alloc::AmAllocAllocator (C++ class), 69  
 umpire::alloc::AmAllocAllocator::allocate (C++ function), 69  
 umpire::alloc::AmAllocAllocator::deallocate (C++ function), 69  
 umpire::alloc::AmPinnedAllocator (C++ class), 69  
 umpire::alloc::AmPinnedAllocator::allocate (C++ function), 70  
 umpire::alloc::AmPinnedAllocator::deallocate (C++ function), 70  
 umpire::alloc::CudaMallocAllocator (C++ class), 70  
 umpire::alloc::CudaMallocAllocator::allocate (C++ function), 70  
 umpire::alloc::CudaMallocAllocator::deallocate (C++ function), 70  
 umpire::alloc::CudaMallocManagedAllocator (C++ class), 71  
 umpire::alloc::CudaMallocManagedAllocator::allocate (C++ function), 71  
 umpire::alloc::CudaMallocManagedAllocator::deallocate (C++ function), 71  
 umpire::alloc::CudaPinnedAllocator (C++ class), 71  
 umpire::alloc::CudaPinnedAllocator::allocate (C++ function), 72  
 umpire::alloc::CudaPinnedAllocator::deallocate (C++ function), 72  
 umpire::alloc::HipMallocAllocator (C++ class), 72  
 umpire::alloc::HipMallocAllocator::allocate (C++ function), 72  
 umpire::alloc::HipMallocAllocator::deallocate (C++ function), 72  
 umpire::alloc::HipPinnedAllocator (C++ class), 73  
 umpire::alloc::HipPinnedAllocator::allocate (C++ function), 73  
 umpire::alloc::HipPinnedAllocator::deallocate (C++ function), 73  
 umpire::alloc::MallocAllocator (C++ class), 73  
 umpire::alloc::MallocAllocator::allocate (C++ function), 73  
 umpire::alloc::MallocAllocator::deallocate (C++ function), 73  
 umpire::alloc::PosixMemalignAllocator (C++ class), 74  
 umpire::alloc::PosixMemalignAllocator::allocate (C++ function), 74  
 umpire::alloc::PosixMemalignAllocator::deallocate (C++ function), 74  
 umpire::Allocator (C++ class), 43, 82  
 umpire::Allocator::allocate (C++ function), 82  
 umpire::Allocator::Allocator (C++ function), 84  
 umpire::Allocator::deallocate (C++ function), 82  
 umpire::Allocator::getActualSize (C++ function), 83  
 umpire::Allocator::getAllocationStrategy (C++ function), 83  
 umpire::Allocator::getCurrentSize (C++ function), 83  
 umpire::Allocator::getHighWatermark (C++ function), 83  
 umpire::Allocator::getId (C++ function), 83  
 umpire::Allocator::getName (C++ function), 83  
 umpire::Allocator::getPlatform (C++ function), 84  
 umpire::Allocator::getSize (C++ function), 83  
 umpire::Allocator::release (C++ function), 82  
 umpire::cpu (C++ enumerator), 175  
 umpire::cpu\_vendor\_type (C++ function), 178  
 umpire::cuda (C++ enumerator), 175  
 umpire::DeviceAllocator (C++ class), 84  
 umpire::DeviceAllocator::~DeviceAllocator (C++ function), 84  
 umpire::DeviceAllocator::DeviceAllocator (C++ function), 84  
 umpire::env\_name (C++ member), 236  
 umpire::error (C++ function), 178  
 umpire::finalize (C++ function), 179  
 umpire::free (C++ function), 179  
 umpire::get\_allocator\_records (C++ function), 179  
 umpire::get\_major\_version (C++ function), 180  
 umpire::get\_minor\_version (C++ function), 180  
 umpire::get\_page\_size (C++ function), 180  
 umpire::get\_patch\_version (C++ function), 180  
 umpire::get\_rc\_version (C++ function), 180

umpire::hip (C++ *enumerator*), 175  
 umpire::initialize (C++ *function*), 181  
 umpire::log (C++ *function*), 181  
 umpire::malloc (C++ *function*), 181  
 umpire::MemoryResourceTraits (C++ *class*), 74  
 umpire::MemoryResourceTraits::access (C++ *enumerator*), 74  
 umpire::MemoryResourceTraits::AMD (C++ *enumerator*), 75  
 umpire::MemoryResourceTraits::any (C++ *enumerator*), 74  
 umpire::MemoryResourceTraits::bandwidth (C++ *enumerator*), 74  
 umpire::MemoryResourceTraits::DDR (C++ *enumerator*), 75  
 umpire::MemoryResourceTraits::GDDR (C++ *enumerator*), 75  
 umpire::MemoryResourceTraits::HBM (C++ *enumerator*), 75  
 umpire::MemoryResourceTraits::IBM (C++ *enumerator*), 75  
 umpire::MemoryResourceTraits::INTEL (C++ *enumerator*), 75  
 umpire::MemoryResourceTraits::kind (C++ *member*), 75  
 umpire::MemoryResourceTraits::latency (C++ *enumerator*), 74  
 umpire::MemoryResourceTraits::memory\_type (C++ *enum*), 75  
 umpire::MemoryResourceTraits::NVIDIA (C++ *enumerator*), 75  
 umpire::MemoryResourceTraits::NVME (C++ *enumerator*), 75  
 umpire::MemoryResourceTraits::optimized\_umpire (C++ *enum*), 74  
 umpire::MemoryResourceTraits::size (C++ *member*), 75  
 umpire::MemoryResourceTraits::unified (C++ *member*), 75  
 umpire::MemoryResourceTraits::UNKNOWN (C++ *enumerator*), 75  
 umpire::MemoryResourceTraits::used\_for (C++ *member*), 75  
 umpire::MemoryResourceTraits::vendor (C++ *member*), 75  
 umpire::MemoryResourceTraits::vendor\_type (C++ *enum*), 74  
 umpire::none (C++ *enumerator*), 175  
 umpire::numa::get\_allocatable\_nodes (C++ *function*), 181  
 umpire::numa::get\_device\_nodes (C++ *function*), 182  
 umpire::numa::get\_host\_nodes (C++ *function*), 182  
 umpire::numa::get\_location (C++ *function*), 182  
 umpire::numa::move\_to\_node (C++ *function*), 182  
 umpire::numa::preferred\_node (C++ *function*), 182  
 umpire::op (C++ *type*), 44  
 umpire::op::CudaAdviseAccessedByOperation (C++ *class*), 85  
 umpire::op::CudaAdviseAccessedByOperation::apply (C++ *function*), 85  
 umpire::op::CudaAdvisePreferredLocationOperation (C++ *class*), 86  
 umpire::op::CudaAdvisePreferredLocationOperation::apply (C++ *function*), 86  
 umpire::op::CudaAdviseReadMostlyOperation (C++ *class*), 87  
 umpire::op::CudaAdviseReadMostlyOperation::apply (C++ *function*), 87  
 umpire::op::CudaAdviseUnsetAccessedByOperation (C++ *class*), 88  
 umpire::op::CudaAdviseUnsetAccessedByOperation::apply (C++ *function*), 88  
 umpire::op::CudaAdviseUnsetPreferredLocationOperation (C++ *class*), 89  
 umpire::op::CudaAdviseUnsetPreferredLocationOperation::apply (C++ *function*), 89  
 umpire::op::CudaAdviseUnsetReadMostlyOperation (C++ *class*), 90  
 umpire::op::CudaAdviseUnsetReadMostlyOperation::apply (C++ *function*), 90  
 umpire::op::CudaCopyFromOperation (C++ *class*), 44, 91  
 umpire::op::CudaCopyFromOperation::transform (C++ *function*), 91  
 umpire::op::CudaCopyOperation (C++ *class*), 44, 92  
 umpire::op::CudaCopyOperation::transform (C++ *function*), 92  
 umpire::op::CudaCopyToOperation (C++ *class*), 44, 93  
 umpire::op::CudaCopyToOperation::transform (C++ *function*), 93  
 umpire::op::CudaMemPrefetchOperation (C++ *class*), 94  
 umpire::op::CudaMemPrefetchOperation::apply (C++ *function*), 94  
 umpire::op::CudaMemsetOperation (C++ *class*), 45, 95  
 umpire::op::CudaMemsetOperation::apply (C++ *function*), 95  
 umpire::op::GenericReallocateOperation (C++ *class*), 45, 96



umpire::op::GenericReallocateOperation::transform (C++ class),  
 (C++ function), 96  
 umpire::op::HipCopyFromOperation (C++ class), 45, 97  
 umpire::op::HipCopyFromOperation::transform (C++ function), 97  
 umpire::op::HipCopyOperation (C++ class), 45, 98  
 umpire::op::HipCopyOperation::transform (C++ function), 98  
 umpire::op::HipCopyToOperation (C++ class), 45, 99  
 umpire::op::HipCopyToOperation::transform (C++ function), 99  
 umpire::op::HipMemsetOperation (C++ class), 45, 100  
 umpire::op::HipMemsetOperation::apply (C++ function), 100  
 umpire::op::HostCopyOperation (C++ class), 45, 101  
 umpire::op::HostCopyOperation::transform (C++ function), 101  
 umpire::op::HostMemsetOperation (C++ class), 45, 102  
 umpire::op::HostMemsetOperation::apply (C++ function), 102  
 umpire::op::HostReallocateOperation (C++ class), 45, 103  
 umpire::op::HostReallocateOperation::transform (C++ function), 103  
 umpire::op::MemoryOperation (C++ class), 45, 105  
 umpire::op::MemoryOperation::~MemoryOperation (C++ function), 105  
 umpire::op::MemoryOperation::apply (C++ function), 91–93, 96–99, 101, 103, 105, 108, 109  
 umpire::op::MemoryOperation::transform (C++ function), 85–90, 94, 95, 100, 102, 105, 109  
 umpire::op::MemoryOperationRegistry (C++ class), 45, 106  
 umpire::op::MemoryOperationRegistry::~MemoryOperationRegistry (C++ function), 107  
 umpire::op::MemoryOperationRegistry::find (C++ function), 106  
 umpire::op::MemoryOperationRegistry::getInstance (C++ function), 107  
 umpire::op::MemoryOperationRegistry::MemoryOperationRegistry (C++ class), 107  
 umpire::op::MemoryOperationRegistry::operator= (C++ class), 113 (C++ function), 107  
 umpire::op::MemoryOperationRegistry::registerOperation (C++ class), 113 (C++ function), 106  
 umpire::op::NumbaMoveOperation (C++ class), 46, 107  
 umpire::op::NumbaMoveOperation::transform (C++ function), 107  
 umpire::op::pair\_hash (C++ class), 75  
 umpire::op::pair\_hash::operator() (C++ function), 75  
 umpire::op::RocmCopyOperation (C++ class), 46, 108  
 umpire::op::RocmCopyOperation::transform (C++ function), 108  
 umpire::op::RocmMemsetOperation (C++ class), 46, 109  
 umpire::op::RocmMemsetOperation::apply (C++ function), 109  
 umpire::operator<< (C++ function), 183  
 umpire::Platform (C++ enum), 175  
 umpire::print\_allocator\_records (C++ function), 183  
 umpire::Replay (C++ class), 110  
 umpire::replay (C++ function), 184  
 umpire::Replay::getReplayLogger (C++ function), 110  
 umpire::Replay::logMessage (C++ function), 110  
 umpire::Replay::printReplayAllocator (C++ function), 110  
 umpire::Replay::replayLoggingEnabled (C++ function), 110  
 umpire::Replay::replayUid (C++ function), 110  
 umpire::resource::Constant (C++ enumerator), 176  
 umpire::resource::CudaConstantMemoryResource (C++ class), 111  
 umpire::resource::CudaConstantMemoryResource::allocate (C++ function), 111  
 umpire::resource::CudaConstantMemoryResource::CudaConstantMemoryResource (C++ function), 111  
 umpire::resource::CudaConstantMemoryResource::deallocate (C++ function), 111  
 umpire::resource::CudaConstantMemoryResource::getCudaDeviceId (C++ function), 111  
 umpire::resource::CudaConstantMemoryResource::getHostAddress (C++ function), 111  
 umpire::resource::CudaConstantMemoryResource::getPinnedMemoryAddress (C++ function), 111  
 umpire::resource::CudaConstantMemoryResourceFactory (C++ class), 113  
 umpire::resource::CudaDeviceResourceFactory (C++ class), 113  
 umpire::resource::CudaPinnedMemoryResourceFactory (C++ class), 113  
 umpire::resource::CudaUnifiedMemoryResourceFactory (C++ class), 113

(C++ class), 114

umpire::resource::DefaultMemoryResource (C++ class), 114

umpire::resource::DefaultMemoryResource::allocate(C++ function), 114

umpire::resource::DefaultMemoryResource::deallocate(C++ function), 112, 115, 117, 121, 124

umpire::resource::DefaultMemoryResource::DefaultMemoryResource(C++ function), 114

umpire::resource::DefaultMemoryResource::GetCurrentPlatform(C++ function), 120

umpire::resource::DefaultMemoryResource::getHighWatermark(C++ function), 115

umpire::resource::DefaultMemoryResource::getPlatform(C++ function), 122

umpire::resource::DefaultMemoryResource::m\_allocate(C++ member), 116

umpire::resource::DefaultMemoryResource::m\_platform(C++ member), 116

umpire::resource::Device (C++ enumerator), 176

umpire::resource::HipConstantMemoryResource (C++ class), 116

umpire::resource::HipConstantMemoryResource::allocate(C++ function), 116

umpire::resource::HipConstantMemoryResource::deallocate(C++ function), 116

umpire::resource::HipConstantMemoryResource::getHighWatermark(C++ function), 116

umpire::resource::HipConstantMemoryResource::getPlatform(C++ function), 117

umpire::resource::HipConstantMemoryResource::getPlatform(C++ function), 117

umpire::resource::HipConstantMemoryResource::HipConstantMemoryResource(C++ function), 116

umpire::resource::HipConstantMemoryResourceFactory (C++ class), 76

umpire::resource::HipDeviceResourceFactory (C++ class), 118

umpire::resource::HipPinnedMemoryResourceFactory (C++ class), 119

umpire::resource::Host (C++ enumerator), 176

umpire::resource::HostResourceFactory (C++ class), 119

umpire::resource::MemoryResource (C++ class), 120

umpire::resource::MemoryResource::~MemoryResource(C++ function), 120

umpire::resource::MemoryResource::allocate(C++ function), 120

umpire::resource::MemoryResource::deallocate(C++ function), 120

umpire::resource::MemoryResource::GetCurrentPlatform(C++ function), 120

umpire::resource::MemoryResource::getHighWatermark(C++ function), 120

umpire::resource::MemoryResource::getPlatform(C++ function), 120

umpire::resource::MemoryResource::getTraits(C++ function), 121

umpire::resource::MemoryResource::m\_traits(C++ member), 121

umpire::resource::MemoryResource::MemoryResource(C++ function), 120

umpire::resource::MemoryResourceFactory (C++ class), 122

umpire::resource::MemoryResourceFactory::~MemoryResourceFactory(C++ function), 122

umpire::resource::MemoryResourceFactory::create(C++ function), 122

umpire::resource::MemoryResourceFactory::isValidMemoryResource(C++ function), 122

umpire::resource::MemoryResourceRegistry (C++ class), 123

umpire::resource::MemoryResourceRegistry::~MemoryResourceRegistry(C++ function), 123

umpire::resource::MemoryResourceRegistry::GetInstance(C++ function), 123

umpire::resource::MemoryResourceRegistry::makeMemoryResource(C++ function), 123

umpire::resource::MemoryResourceRegistry::MemoryResourceRegistry(C++ function), 123

umpire::resource::MemoryResourceRegistry::operator=(C++ function), 123

umpire::resource::MemoryResourceRegistry::registerMemoryResource(C++ function), 123

umpire::resource::MemoryResourceType (C++ enumeration), 123

umpire::resource::MemoryResourceTypeHash (C++ class), 76

umpire::resource::MemoryResourceTypeHash::operator=(C++ function), 76

umpire::resource::NullMemoryResource (C++ class), 123

umpire::resource::NullMemoryResource::allocate(C++ function), 123

umpire::resource::NullMemoryResource::deallocate(C++ function), 124

umpire::resource::NullMemoryResource::getCurrentSize(C++ function), 124

umpire::resource::NullMemoryResource::getHighWatermark(C++ function), 124

umpire::resource::NullMemoryResource::getPlatform(C++ function), 124

umpire::resource::NullMemoryResource::m\_platform(C++ member), 125

umpire::resource::NullMemoryResource::NullMemoryResource(C++ function), 123

umpire::resource::NullMemoryResourceFactory (C++ function), 127  
 (C++ class), 125      umpire::ResourceManager::registerAllocator  
 umpire::resource::Pinned (C++ enumerator), (C++ function), 127  
 176      umpire::ResourceManager::ResourceManager  
 umpire::resource::RocmDeviceResourceFactory (C++ function), 129  
 (C++ class), 126      umpire::ResourceManager::setDefaultAllocator  
 umpire::resource::RocmPinnedMemoryResourceFactory (C++ function), 127  
 (C++ class), 126      umpire::rocm (C++ enumerator), 175  
 umpire::resource::Unified (C++ enumerator), 176      umpire::strategy::AllocationAdvisor  
 (C++ class), 44, 130  
 umpire::ResourceManager (C++ class), 126      umpire::strategy::AllocationAdvisor::allocate  
 umpire::ResourceManager::~~ResourceManager (C++ function), 130  
 (C++ function), 129      umpire::strategy::AllocationAdvisor::AllocationAdvisor  
 umpire::ResourceManager::copy (C++ function), 130  
 (C++ function), 128      umpire::strategy::AllocationAdvisor::deallocate  
 umpire::ResourceManager::deallocate (C++ function), 130  
 (C++ function), 129      umpire::strategy::AllocationAdvisor::getCurrentSize  
 umpire::ResourceManager::deregisterAllocation (C++ function), 131  
 (C++ function), 127      umpire::strategy::AllocationAdvisor::getHighWatermark  
 umpire::ResourceManager::findAllocationRecord (C++ function), 131  
 (C++ function), 128      umpire::strategy::AllocationAdvisor::getPlatform  
 umpire::ResourceManager::getAllocator (C++ function), 131  
 (C++ function), 126, 127      umpire::strategy::AllocationPrefetcher  
 umpire::ResourceManager::getAllocatorIds (C++ class), 132  
 (C++ function), 126      umpire::strategy::AllocationPrefetcher::allocate  
 umpire::ResourceManager::getAllocatorNames (C++ function), 132  
 (C++ function), 126      umpire::strategy::AllocationPrefetcher::AllocationPrefetcher  
 umpire::ResourceManager::getDefaultAllocator (C++ function), 132  
 (C++ function), 127      umpire::strategy::AllocationPrefetcher::deallocate  
 umpire::ResourceManager::getInstance (C++ function), 132  
 (C++ function), 130      umpire::strategy::AllocationPrefetcher::getCurrentSize  
 umpire::ResourceManager::getOperation (C++ function), 132  
 (C++ function), 129      umpire::strategy::AllocationPrefetcher::getHighWatermark  
 umpire::ResourceManager::getSize (C++ function), 132  
 function), 129      umpire::strategy::AllocationPrefetcher::getPlatform  
 umpire::ResourceManager::hasAllocator (C++ function), 132  
 (C++ function), 127      umpire::strategy::AllocationStrategy  
 umpire::ResourceManager::initialize (C++ class), 43, 134  
 (C++ function), 126      umpire::strategy::AllocationStrategy::~~AllocationStrategy  
 umpire::ResourceManager::isAllocator (C++ function), 134  
 (C++ function), 127      umpire::strategy::AllocationStrategy::allocate  
 umpire::ResourceManager::isAllocatorRegistered (C++ function), 134, 140  
 (C++ function), 128      umpire::strategy::AllocationStrategy::AllocationStrategy  
 umpire::ResourceManager::makeAllocator (C++ function), 134  
 (C++ function), 127      umpire::strategy::AllocationStrategy::deallocate  
 umpire::ResourceManager::memset (C++ function), 134  
 function), 128      umpire::strategy::AllocationStrategy::getActualSize  
 umpire::ResourceManager::move (C++ function), 112, 115, 117, 121, 124, 131,  
 function), 129      133, 135, 149, 151, 153, 154, 156, 158  
 umpire::ResourceManager::operator= (C++ function), 135  
 function), 129      umpire::strategy::AllocationStrategy::getCurrentSize  
 umpire::ResourceManager::reallocate (C++ function), 128, 129      (C++ function), 135  
 (C++ function), 128, 129      umpire::strategy::AllocationStrategy::getHighWatermark  
 umpire::ResourceManager::registerAllocator (C++ function), 135  
 (C++ function), 127      umpire::strategy::AllocationStrategy::getId

(C++ function), 112, 115, 117, 121, 124, 131, 133, 135, 137, 140, 143, 145, 147, 150, 151, 153, 155, 156, 158, 160

umpire::strategy::AllocationStrategy::getName (C++ function), 138

(C++ function), 112, 115, 117, 121, 124, 131, 133, 135, 137, 140, 142, 145, 147, 150, 151, 153, 155, 156, 158, 160

umpire::strategy::AllocationStrategy::getPlatform (C++ function), 135

umpire::strategy::AllocationStrategy::m\_id (C++ member), 112, 116, 117, 121, 125, 131, 133, 135, 137, 140, 143, 145, 147, 150, 151, 153, 155, 157, 158, 160

umpire::strategy::AllocationStrategy::m\_name (C++ member), 112, 116, 117, 121, 125, 131, 133, 135, 137, 140, 143, 145, 147, 150, 151, 153, 155, 157, 158, 160

umpire::strategy::AllocationStrategy::release (C++ function), 139

(C++ function), 112, 115, 117, 121, 124, 131, 133, 134, 145, 149, 151, 153, 154, 156, 158

umpire::strategy::AllocationTracker (C++ class), 136

umpire::strategy::AllocationTracker::allocate (C++ function), 136

umpire::strategy::AllocationTracker::AllocateStrategy (C++ function), 136

umpire::strategy::AllocationTracker::deallocate (C++ function), 136

umpire::strategy::AllocationTracker::getActualSize (C++ function), 137

umpire::strategy::AllocationTracker::getAllocationStrategy (C++ function), 137

umpire::strategy::AllocationTracker::getCurrentSize (C++ function), 136

umpire::strategy::AllocationTracker::getHighWatermark (C++ function), 136

umpire::strategy::AllocationTracker::getPlatform (C++ function), 137

umpire::strategy::AllocationTracker::release (C++ function), 136

umpire::strategy::bits\_per\_int (C++ member), 236

umpire::strategy::DynamicPool (C++ type), 241

umpire::strategy::DynamicPool::allocate (C++ function), 141

umpire::strategy::DynamicPoolList (C++ class), 138

umpire::strategy::DynamicPoolList::allocate (C++ function), 139

umpire::strategy::DynamicPoolList::coalesce (C++ function), 140

umpire::strategy::DynamicPoolList::CoalesceHeuristic (C++ type), 138

umpire::strategy::DynamicPoolList::deallocate (C++ function), 139

umpire::strategy::DynamicPoolList::DynamicPoolList (C++ function), 139

umpire::strategy::DynamicPoolList::getActualSize (C++ function), 139

umpire::strategy::DynamicPoolList::getBlocksInPool (C++ function), 139

umpire::strategy::DynamicPoolList::getCurrentSize (C++ function), 139

umpire::strategy::DynamicPoolList::getHighWatermark (C++ function), 139

umpire::strategy::DynamicPoolList::getLargestAvailable (C++ function), 139

umpire::strategy::DynamicPoolList::getPlatform (C++ function), 139

umpire::strategy::DynamicPoolList::getReleasableSize (C++ function), 139

umpire::strategy::DynamicPoolList::release (C++ function), 139

umpire::strategy::DynamicPoolMap (C++ class), 140

umpire::strategy::DynamicPoolMap::~DynamicPoolMap (C++ function), 141

umpire::strategy::DynamicPoolMap::coalesce (C++ function), 142

umpire::strategy::DynamicPoolMap::CoalesceHeuristic (C++ type), 141

umpire::strategy::DynamicPoolMap::deallocate (C++ function), 141

umpire::strategy::DynamicPoolMap::DynamicPoolMap (C++ function), 141

umpire::strategy::DynamicPoolMap::getActualSize (C++ function), 142

umpire::strategy::DynamicPoolMap::getBlocksInPool (C++ function), 142

umpire::strategy::DynamicPoolMap::getCurrentSize (C++ function), 141

umpire::strategy::DynamicPoolMap::getFreeBlocks (C++ function), 142

umpire::strategy::DynamicPoolMap::getHighWatermark (C++ function), 142

umpire::strategy::DynamicPoolMap::getInUseBlocks (C++ function), 142

umpire::strategy::DynamicPoolMap::getLargestAvailable (C++ function), 142

umpire::strategy::DynamicPoolMap::getPlatform (C++ function), 142

umpire::strategy::DynamicPoolMap::getReleasableSize (C++ function), 142

umpire::strategy::DynamicPoolMap::Pointer (C++ type), 141

umpire::strategy::DynamicPoolMap::release (C++ function), 141

umpire::strategy::find\_first\_set (C++ function), 176, 184  
 umpire::strategy::FixedPool (C++ class), 44, 143  
 umpire::strategy::FixedPool::~FixedPool (C++ function), 144  
 umpire::strategy::FixedPool::allocate (C++ function), 144  
 umpire::strategy::FixedPool::deallocate (C++ function), 144  
 umpire::strategy::FixedPool::FixedPool (C++ function), 144  
 umpire::strategy::FixedPool::getActualSize (C++ function), 144  
 umpire::strategy::FixedPool::getCurrentSize (C++ function), 144  
 umpire::strategy::FixedPool::getHighWatermark (C++ function), 144  
 umpire::strategy::FixedPool::getPlatform (C++ function), 145  
 umpire::strategy::FixedPool::numPools (C++ function), 145  
 umpire::strategy::FixedPool::pointerIsFromPool (C++ function), 145  
 umpire::strategy::FixedPool::Pool (C++ class), 76  
 umpire::strategy::FixedPool::Pool::avail (C++ member), 76  
 umpire::strategy::FixedPool::Pool::data (C++ member), 76  
 umpire::strategy::FixedPool::Pool::num\_available (C++ member), 76  
 umpire::strategy::FixedPool::Pool::Pool (C++ function), 76  
 umpire::strategy::FixedPool::Pool::strategy (C++ member), 76  
 umpire::strategy::heuristic\_percent\_release (C++ function), 184  
 umpire::strategy::heuristic\_percent\_release (C++ member), 236  
 umpire::strategy::heuristic\_percent\_release (C++ function), 184  
 umpire::strategy::heuristic\_percent\_release (C++ member), 236  
 umpire::strategy::MixedPool (C++ class), 146  
 umpire::strategy::MixedPool::allocate (C++ function), 146  
 umpire::strategy::MixedPool::deallocate (C++ function), 146  
 umpire::strategy::MixedPool::getActualSize (C++ function), 147  
 umpire::strategy::MixedPool::getCurrentSize (C++ function), 147  
 umpire::strategy::MixedPool::getHighWatermark (C++ function), 147  
 umpire::strategy::MixedPool::getPlatform (C++ function), 147  
 umpire::strategy::MixedPool::MixedPool (C++ function), 146  
 umpire::strategy::MixedPool::release (C++ function), 146  
 umpire::strategy::mixins::Inspector (C++ class), 148  
 umpire::strategy::mixins::Inspector::deregisterAll (C++ function), 137, 148  
 umpire::strategy::mixins::Inspector::Inspector (C++ function), 148  
 umpire::strategy::mixins::Inspector::m\_current\_size (C++ member), 138, 148  
 umpire::strategy::mixins::Inspector::m\_high\_watermark (C++ member), 138, 148  
 umpire::strategy::mixins::Inspector::registerAllocations (C++ function), 137, 148  
 umpire::strategy::MonotonicAllocationStrategy (C++ class), 44, 149  
 umpire::strategy::MonotonicAllocationStrategy::~MonotonicAllocationStrategy (C++ function), 149  
 umpire::strategy::MonotonicAllocationStrategy::allocate (C++ function), 149  
 umpire::strategy::MonotonicAllocationStrategy::deallocate (C++ function), 149  
 umpire::strategy::MonotonicAllocationStrategy::getCurrentSize (C++ function), 149  
 umpire::strategy::MonotonicAllocationStrategy::getHighWatermark (C++ function), 149  
 umpire::strategy::MonotonicAllocationStrategy::MonotonicAllocationStrategy (C++ function), 149  
 umpire::strategy::NamedAllocationStrategy (C++ class), 150  
 umpire::strategy::NamedAllocationStrategy::allocate (C++ function), 150  
 umpire::strategy::NamedAllocationStrategy::deallocate (C++ function), 150  
 umpire::strategy::NamedAllocationStrategy::getCurrentSize (C++ function), 151  
 umpire::strategy::NamedAllocationStrategy::getHighWatermark (C++ function), 151  
 umpire::strategy::NamedAllocationStrategy::getPlatform (C++ function), 151  
 umpire::strategy::NamedAllocationStrategy::m\_allocated (C++ member), 151  
 umpire::strategy::NamedAllocationStrategy::NamedAllocationStrategy (C++ function), 150  
 umpire::strategy::NumaPolicy (C++ class), 152

umpire::strategy::NumaPolicy::allocate (C++ function), 152  
 umpire::strategy::NumaPolicy::deallocate (C++ function), 152  
 umpire::strategy::NumaPolicy::getCurrentSize (C++ function), 152  
 umpire::strategy::NumaPolicy::getHighWatermark (C++ function), 152  
 umpire::strategy::NumaPolicy::getNode (C++ function), 153  
 umpire::strategy::NumaPolicy::getPlatform (C++ function), 152  
 umpire::strategy::NumaPolicy::NumaPolicy (C++ function), 152  
 umpire::strategy::operator<< (C++ function), 185  
 umpire::strategy::SizeLimiter (C++ class), 154  
 umpire::strategy::SizeLimiter::allocate (C++ function), 154  
 umpire::strategy::SizeLimiter::deallocate (C++ function), 154  
 umpire::strategy::SizeLimiter::getCurrentSize (C++ function), 154  
 umpire::strategy::SizeLimiter::getHighWatermark (C++ function), 154  
 umpire::strategy::SizeLimiter::getPlatform (C++ function), 154  
 umpire::strategy::SizeLimiter::SizeLimiter (C++ function), 154  
 umpire::strategy::SlotPool (C++ class), 44, 155  
 umpire::strategy::SlotPool::~~SlotPool (C++ function), 155  
 umpire::strategy::SlotPool::allocate (C++ function), 155  
 umpire::strategy::SlotPool::deallocate (C++ function), 156  
 umpire::strategy::SlotPool::getCurrentSize (C++ function), 156  
 umpire::strategy::SlotPool::getHighWatermark (C++ function), 156  
 umpire::strategy::SlotPool::getPlatform (C++ function), 156  
 umpire::strategy::SlotPool::SlotPool (C++ function), 155  
 umpire::strategy::ThreadSafeAllocator (C++ class), 44, 157  
 umpire::strategy::ThreadSafeAllocator::allocate (C++ function), 157  
 umpire::strategy::ThreadSafeAllocator::deallocate (C++ function), 157  
 umpire::strategy::ThreadSafeAllocator::getCurrentSize (C++ function), 157  
 umpire::strategy::ThreadSafeAllocator::getHighWatermark (C++ function), 157  
 umpire::strategy::ThreadSafeAllocator::getPlatform (C++ function), 158  
 umpire::strategy::ThreadSafeAllocator::m\_allocator (C++ member), 158  
 umpire::strategy::ThreadSafeAllocator::m\_mutex (C++ member), 158  
 umpire::strategy::ThreadSafeAllocator::ThreadSafeAllocator (C++ function), 157  
 umpire::strategy::ZeroByteHandler (C++ class), 159  
 umpire::strategy::ZeroByteHandler::allocate (C++ function), 159  
 umpire::strategy::ZeroByteHandler::deallocate (C++ function), 159  
 umpire::strategy::ZeroByteHandler::getActualSize (C++ function), 159  
 umpire::strategy::ZeroByteHandler::getAllocationSize (C++ function), 160  
 umpire::strategy::ZeroByteHandler::getCurrentSize (C++ function), 159  
 umpire::strategy::ZeroByteHandler::getHighWatermark (C++ function), 159  
 umpire::strategy::ZeroByteHandler::getPlatform (C++ function), 159  
 umpire::strategy::ZeroByteHandler::release (C++ function), 159  
 umpire::strategy::ZeroByteHandler::ZeroByteHandler (C++ function), 159  
 umpire::TypedAllocator (C++ class), 43, 160  
 umpire::TypedAllocator::allocate (C++ function), 161  
 umpire::TypedAllocator::deallocate (C++ function), 161  
 umpire::TypedAllocator::TypedAllocator (C++ function), 161  
 umpire::TypedAllocator::value\_type (C++ type), 160  
 umpire::util::AllocationMap (C++ class), 161  
 umpire::util::AllocationMap::AllocationMap (C++ function), 162  
 umpire::util::AllocationMap::begin (C++ function), 162  
 umpire::util::AllocationMap::clear (C++ function), 162  
 umpire::util::AllocationMap::ConstIterator (C++ class), 162, 163  
 umpire::util::AllocationMap::ConstIterator::ConstIterator (C++ function), 162, 163  
 umpire::util::AllocationMap::ConstIterator::operator (C++ function), 162, 163  
 umpire::util::AllocationMap::ConstIterator::operator (C++ function), 162, 163



umpire::util::FixedMallocPool::FixedMallocPool *function*), 170  
     (C++ *function*), 167  
 umpire::util::FixedMallocPool::numPools  
     (C++ *function*), 167  
 umpire::util::FixedMallocPool::Pool  
     (C++ *class*), 78  
 umpire::util::FixedMallocPool::Pool::data  
     (C++ *member*), 78  
 umpire::util::FixedMallocPool::Pool::next  
     (C++ *member*), 78  
 umpire::util::FixedMallocPool::Pool::num\_free  
     (C++ *member*), 78  
 umpire::util::FixedMallocPool::Pool::num\_initialized  
     (C++ *member*), 78  
 umpire::util::FixedMallocPool::Pool::Pool  
     (C++ *function*), 78  
 umpire::util::flush\_files (C++ *function*),  
     187  
 umpire::util::initialize\_io (C++ *function*),  
     187  
 umpire::util::iterator\_begin (C++ *class*),  
     78  
 umpire::util::iterator\_end (C++ *class*), 78  
 umpire::util::Logger (C++ *class*), 167  
 umpire::util::Logger::~~Logger (C++ *func-*  
     *tion*), 167  
 umpire::util::Logger::~finalize (C++ *func-*  
     *tion*), 168  
 umpire::util::Logger::~getActiveLogger  
     (C++ *function*), 168  
 umpire::util::Logger::~initialize (C++  
     *function*), 168  
 umpire::util::Logger::~Logger (C++ *func-*  
     *tion*), 167  
 umpire::util::Logger::~LogLevelEnabled  
     (C++ *function*), 167  
 umpire::util::Logger::~logMessage (C++  
     *function*), 167  
 umpire::util::Logger::~operator= (C++  
     *function*), 167  
 umpire::util::Logger::~setLoggingMsgLevel  
     (C++ *function*), 167  
 umpire::util::make\_unique (C++ *function*),  
     187  
 umpire::util::make\_unique\_filename (C++  
     *function*), 187  
 umpire::util::MemoryMap (C++ *class*), 168  
 umpire::util::MemoryMap::~~MemoryMap  
     (C++ *function*), 169  
 umpire::util::MemoryMap::begin (C++ *func-*  
     *tion*), 169  
 umpire::util::MemoryMap::clear (C++ *func-*  
     *tion*), 170  
 umpire::util::MemoryMap::doInsert (C++  
     umpire::util::MemoryMap::end (C++ *func-*  
         *tion*), 169  
     umpire::util::MemoryMap::erase (C++ *func-*  
         *tion*), 169  
     umpire::util::MemoryMap::find (C++ *func-*  
         *tion*), 169  
     umpire::util::MemoryMap::findOrBefore  
         (C++ *function*), 169  
     umpire::util::MemoryMap::insert (C++  
         *function*), 169, 170  
     umpire::util::MemoryMap::Iterator\_ (C++  
         *class*), 170, 172  
     umpire::util::MemoryMap::MemoryMap (C++  
         *function*), 169  
     umpire::util::MemoryMap::removeLast  
         (C++ *function*), 170  
     umpire::util::MemoryMap::size (C++ *func-*  
         *tion*), 170  
     umpire::util::MemoryMap<V>::ConstIterator  
         (C++ *type*), 168  
     umpire::util::MemoryMap<V>::Iterator  
         (C++ *type*), 168  
     umpire::util::MemoryMap<V>::Iterator\_::Iterator\_  
         (C++ *function*), 171, 172  
     umpire::util::MemoryMap<V>::Iterator\_::operator!=  
         (C++ *function*), 171, 173  
     umpire::util::MemoryMap<V>::Iterator\_::operator\*  
         (C++ *function*), 171, 172  
     umpire::util::MemoryMap<V>::Iterator\_::operator++  
         (C++ *function*), 171–173  
     umpire::util::MemoryMap<V>::Iterator\_::operator==  
         (C++ *function*), 171, 173  
     umpire::util::MemoryMap<V>::Iterator\_::operator->  
         (C++ *function*), 171, 172  
     umpire::util::MemoryMap<V>::Iterator\_<Const>::Conte  
         (C++ *type*), 170, 172  
     umpire::util::MemoryMap<V>::Iterator\_<Const>::Map  
         (C++ *type*), 170, 172  
     umpire::util::MemoryMap<V>::Iterator\_<Const>::Point  
         (C++ *type*), 170, 172  
     umpire::util::MemoryMap<V>::Iterator\_<Const>::Refer  
         (C++ *type*), 170, 172  
     umpire::util::MemoryMap<V>::Iterator\_<Const>::Value  
         (C++ *type*), 170, 172  
     umpire::util::MemoryMap<V>::Key (C++  
         *type*), 168  
     umpire::util::MemoryMap<V>::KeyValuePair  
         (C++ *type*), 168  
     umpire::util::MemoryMap<V>::Value (C++  
         *type*), 168  
     umpire::util::message::Debug (C++ *enumer-*  
         *ator*), 176  
     umpire::util::message::Error (C++ *enumer-*



ator), 176

umpire::util::message::Info (C++ enumerator), 176

umpire::util::message::Level (C++ enum), 176

umpire::util::message::Num\_Levels (C++ enumerator), 176

umpire::util::message::Warning (C++ enumerator), 176

umpire::util::MessageLevelName (C++ member), 237

umpire::util::MPI (C++ class), 173

umpire::util::MPI::finalize (C++ function), 173

umpire::util::MPI::getRank (C++ function), 173

umpire::util::MPI::getSize (C++ function), 173

umpire::util::MPI::initialize (C++ function), 173

umpire::util::MPI::isInitialized (C++ function), 173

umpire::util::MPI::logMpiInfo (C++ function), 173

umpire::util::MPI::sync (C++ function), 173

umpire::util::OutputBuffer (C++ class), 174

umpire::util::OutputBuffer::~~OutputBuffer (C++ function), 174

umpire::util::OutputBuffer::OutputBuffer (C++ function), 174

umpire::util::OutputBuffer::overflow (C++ function), 174

umpire::util::OutputBuffer::setConsoleStream (C++ function), 174

umpire::util::OutputBuffer::setFileStream (C++ function), 174

umpire::util::OutputBuffer::sync (C++ function), 174

umpire::util::relative\_fragmentation (C++ function), 188

umpire::util::Statistic (C++ class), 174

umpire::util::Statistic::~~Statistic (C++ function), 174

umpire::util::Statistic::printData (C++ function), 174

umpire::util::Statistic::recordStatistic (C++ function), 174

umpire::util::Statistic::Statistic (C++ function), 174

umpire::util::StatisticsDatabase (C++ class), 175

umpire::util::StatisticsDatabase::getDataBase (C++ function), 175

umpire::util::StatisticsDatabase::getStatistics (C++ function), 175

umpire::util::StatisticsDatabase::printStatistics (C++ function), 175

umpire::util::unwrap\_allocation\_strategy (C++ function), 188

umpire::util::unwrap\_allocator (C++ function), 188

umpire::util::wrap\_allocator (C++ function), 188

umpire\_allocator (C++ type), 241

umpire\_allocator\_allocate (C++ function), 189

umpire\_allocator\_deallocate (C++ function), 189

umpire\_allocator\_delete (C++ function), 189, 190

umpire\_allocator\_get\_actual\_size (C++ function), 190

umpire\_allocator\_get\_current\_size (C++ function), 190

umpire\_allocator\_get\_high\_watermark (C++ function), 191

umpire\_allocator\_get\_id (C++ function), 191

umpire\_allocator\_get\_name (C++ function), 191, 192

umpire\_allocator\_get\_name\_bufferify (C++ function), 192

umpire\_allocator\_get\_size (C++ function), 192, 193

UMPIRE\_Allocator\_INL (C macro), 238

umpire\_allocator\_release (C++ function), 193

UMPIRE\_ASSERT (C macro), 238

UMPIRE\_DefaultMemoryResource\_INL (C macro), 238

UMPIRE\_ERROR (C macro), 238

UMPIRE\_LOG (C macro), 239

UMPIRE\_MemoryMap\_INL (C macro), 239

umpire\_mod::allocator\_deallocate (C++ function), 197

umpire\_mod::allocator\_delete (C++ function), 200

umpire\_mod::allocator\_release (C++ function), 202

umpire\_mod::allocator\_set\_instance (C++ function), 202

umpire\_mod::resourcemanager\_copy\_all (C++ function), 203

umpire\_mod::resourcemanager\_copy\_with\_size (C++ function), 203

umpire\_mod::resourcemanager\_deallocate (C++ function), 203

umpire\_mod::resourcemanager\_memset\_all (C++ function), 206

umpire\_mod::resourcemanager\_memset\_with\_size (C++ function), 206

(C++ function), 207

umpire\_mod::resourcemanager\_register\_allocator (C++ function), 208

UMPIRE\_RECORD\_STATISTIC (C macro), 239

UMPIRE\_REPLAY (C macro), 240

umpire\_resourcemanager (C++ type), 241

umpire\_resourcemanager\_copy\_all (C++ function), 208

umpire\_resourcemanager\_copy\_with\_size (C++ function), 208, 209

umpire\_resourcemanager\_deallocate (C++ function), 209

umpire\_resourcemanager\_get\_allocator\_by\_id (C++ function), 209

umpire\_resourcemanager\_get\_allocator\_by\_name (C++ function), 210

umpire\_resourcemanager\_get\_allocator\_by\_name\_bufferify (C++ function), 210, 211

umpire\_resourcemanager\_get\_allocator\_for\_ptr (C++ function), 211

umpire\_resourcemanager\_get\_instance (C++ function), 212

umpire\_resourcemanager\_get\_size (C++ function), 212

umpire\_resourcemanager\_has\_allocator (C++ function), 213

UMPIRE\_ResourceManager\_INL (C macro), 240

umpire\_resourcemanager\_is\_allocator (C++ function), 213

umpire\_resourcemanager\_is\_allocator\_bufferify (C++ function), 213, 214

umpire\_resourcemanager\_make\_allocator\_advisor (C++ function), 214

umpire\_resourcemanager\_make\_allocator\_bufferify (C++ function), 215, 216

umpire\_resourcemanager\_make\_allocator\_bufferify\_fixedpool (C++ function), 216, 217

umpire\_resourcemanager\_make\_allocator\_bufferify\_listpool (C++ function), 218

umpire\_resourcemanager\_make\_allocator\_bufferify\_named (C++ function), 219

umpire\_resourcemanager\_make\_allocator\_bufferify\_named\_allocator (C++ function), 220

umpire\_resourcemanager\_make\_allocator\_bufferify\_named\_thread\_safe (C++ function), 221, 222

umpire\_resourcemanager\_make\_allocator\_bufferify\_thread\_safe (C++ function), 222, 223

umpire\_resourcemanager\_make\_allocator\_fixed\_pool (C++ function), 223, 224

umpire\_resourcemanager\_make\_allocator\_list\_pool (C++ function), 224

umpire\_resourcemanager\_make\_allocator\_named (C++ function), 225

umpire\_resourcemanager\_make\_allocator\_pool (C++ function), 225, 226

umpire\_resourcemanager\_make\_allocator\_prefetcher (C++ function), 226

umpire\_resourcemanager\_make\_allocator\_thread\_safe (C++ function), 227

umpire\_resourcemanager\_memset\_all (C++ function), 227, 228

umpire\_resourcemanager\_memset\_with\_size (C++ function), 228

umpire\_resourcemanager\_move (C++ function), 228

umpire\_resourcemanager\_reallocate\_default (C++ function), 229

umpire\_resourcemanager\_reallocate\_with\_allocator (C++ function), 229

umpire\_resourcemanager\_register\_allocator (C++ function), 230

umpire\_resourcemanager\_register\_allocator\_bufferify (C++ function), 230

umpire\_SHROUD\_array (C++ type), 241

umpire\_SHROUD\_capsule\_data (C++ type), 241

umpire\_SHROUD\_memory\_destructor (C++ function), 231

umpire\_strategy\_allocationadvisor (C++ type), 242

umpire\_strategy\_allocationprefetcher (C++ type), 242

umpire\_strategy\_dynamicpool (C++ type), 242

umpire\_strategy\_dynamicpoollist (C++ type), 242

umpire\_strategy\_fixedpool (C++ type), 242

umpire\_strategy\_mod::allocationadvisor\_set\_instance (C++ function), 232

umpire\_strategy\_mod::dynamicpool\_set\_instance (C++ function), 233

umpire\_strategy\_mod::namedallocationstrategy\_set\_instance (C++ function), 234

umpire\_strategy\_mod::namedallocationstrategy (C++ type), 243

umpire\_strategy\_threadsafeallocator (C++ type), 243

UMPIRE\_TypeAllocator\_INL (C macro), 240

UMPIRE\_UNUSED\_ARG (C macro), 240

UMPIRE\_USE\_VAR (C macro), 240