Umpire Documentation

Release 5.0.0

David Beckingsale

Dec 10, 2020

BASICS

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

CHAPTER

ONE

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 ../
```

By default, Umpire will only support host memory. Additional backends for device support can be enabled using the options detailed in *Advanced Configuration*. CMake will provide output about which compiler is being used and the values of other options. Once CMake has completed, Umpire can be built with Make:

```
$ make
```

For more advanced configuration options, 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
```

CHAPTER

TWO

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:

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

You can also use an existing allocator to build an additional allocator off of it:

```
auto addon_allocator = rm.getAllocator(allocator.getName());
```

This "add-on" allocator will also be built with the same memory resource. More information on memory resources is provided in the next section. Additionally, 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)));
```

```
allocator.deallocate(data);
```

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

```cpp
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, 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**)
{
  auto& rm = umpire::ResourceManager::getInstance();

  // _sphinx_tag_tut_get_allocator_start
  umpire::Allocator allocator = rm.getAllocator("HOST");
  // _sphinx_tag_tut_get_allocator_end

  constexpr std::size_t SIZE = 1024;

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

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

  //_sphinx_tag_tut_getAllocator_start
  auto addon_allocator = rm.getAllocator(allocator.getName());
  //_sphinx_tag_tut_getAllocator_end

  std::cout << "Created an add-on allocator of size " << addon_allocator.
↪getCurrentSize()
            << " using the " << allocator.getName() << " allocator." << std::endl;

  // _sphinx_tag_tut_deallocate_start
  allocator.deallocate(data);
  // _sphinx_tag_tut_deallocate_end

  std::cout << "...Memory 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 (defined by `MemoryResourceTraits::resource_type`), 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. We also have resources that represent global GPU memory ("DEVICE"), constant GPU memory ("DEVICE_CONST"), unified memory that can be accessed by the CPU or GPU ("UM"), host memory that can be accessed by the GPU ("PINNED"), and mmapped file memory ("FILE"). If an incorrect name is used or if the allocator was not set up correctly, the "UNKNOWN" resource name is returned.

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 since 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.

While using Umpire memory resources, it may be useful to query the memory resource currently associated with a particular allocator. For example, if we wanted to double check that our allocator is using the `device` resource, we can assert that `MemoryResourceTraits::resource_type::device` is equal to the return value of `allocator.getAllocationStrategy()->getTraits().resource`. The test code provided in `memory_resource_traits_tests.cpp` shows a complete example of how to query this information.

Note: In order to test some memory resources, you may need to configure your Umpire build to use a particular platform (a member of the `umpire::Allocator`, defined by `Platform.hpp`) that has access to that resource. See the Developer's Guide for more information.

Next, we will see an example of how to move data between resources using operations.

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, 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)
{
  auto& rm = umpire::ResourceManager::getInstance();

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

  constexpr std::size_t SIZE = 1024;

  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;
}
```

(continues on next page)

(continued from previous page)

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

#if defined(UMPIRE_ENABLE_DEVICE)
  allocate_and_deallocate("DEVICE");
#endif
#if defined(UMPIRE_ENABLE_UM)
  allocate_and_deallocate("UM");
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  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) 2016-20, 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 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)));

  // _sphinx_tag_tut_copy_start
  rm.copy(dest_data, source_data);
  // _sphinx_tag_tut_copy_end

  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;
```

(continues on next page)

(continued from previous page)

```
  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_DEVICE)
  copy_data(data, SIZE, "DEVICE");
#endif
#if defined(UMPIRE_ENABLE_UM)
  copy_data(data, SIZE, "UM");
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  copy_data(data, SIZE, "PINNED");
#endif

  allocator.deallocate(data);

  return 0;
}
```

Move Example Listing

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, 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 ";

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

(continues on next page)

(continued from previous page)

```
  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_DEVICE)
  data = move_data(data, "DEVICE");
#endif
#if defined(UMPIRE_ENABLE_UM)
  data = move_data(data, "UM");
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  data = move_data(data, "PINNED");
#endif

  rm.deallocate(data);

  return 0;
}
```

Memset Example Listing

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, 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**)
{
```

(continues on next page)

(continued from previous page)

```
  constexpr std::size_t SIZE = 1024;

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

  const std::string destinations[] = {
    "HOST"
#if defined(UMPIRE_ENABLE_DEVICE)
    ,
    "DEVICE"
#endif
#if defined(UMPIRE_ENABLE_UM)
    ,
    "UM"
#endif
#if defined(UMPIRE_ENABLE_PINNED)
    ,
    "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;

    // _sphinx_tag_tut_memset_start
    rm.memset(data, 0);
    // _sphinx_tag_tut_memset_end

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

    allocator.deallocate(data);
  }

  return 0;
}
```

Reallocate Example Listing

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, 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;
  constexpr std::size_t REALLOCATED_SIZE = 256;
```

(continues on next page)

(continued from previous page)

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

  const std::string destinations[] = {
    "HOST"
#if defined(UMPIRE_ENABLE_DEVICE)
    ,
    "DEVICE"
#endif
#if defined(UMPIRE_ENABLE_UM)
    ,
    "UM"
#endif
#if defined(UMPIRE_ENABLE_PINNED)
    ,
    "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 << "...";

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

    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 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.

Additionally, in the previous section on Allocators, we mentioned that you could build a new allocator off of an existing one using the `getAllocator` function. Here is another example of this, but using the strategy:

```
Umpire::Allocator addon_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>(
resource + "_addon_pool", rm.getAllocator(pooled_allocator.getName()));
```

The purpose of this example is to show that the `getAllocator` function can be used more than just to get an initial allocator. Another good use case for this function is grabbing each allocator while looping through a list of them. (Note that `addon_allocator` will be created with the same memory resource as `pooled_allocator` was.)

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)));
```

```
  pooled_allocator.deallocate(data);
```

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

```
  allocate_and_deallocate_pool("HOST");

#if defined(UMPIRE_ENABLE_DEVICE)
  allocate_and_deallocate_pool("DEVICE");
#endif
#if defined(UMPIRE_ENABLE_UM)
  allocate_and_deallocate_pool("UM");
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  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 underly 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 pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>(
      resource + "_pool", allocator, 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);
#if defined(UMPIRE_ENABLE_DEVICE)
  allocate_and_deallocate_pool("DEVICE", (1024 * 1024 * 1024), (1024 * 1024));
#endif
#if defined(UMPIRE_ENABLE_UM)
  allocate_and_deallocate_pool("UM", (1024 * 64), 1024);
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  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) 2016-20, 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)
{
  auto& rm = umpire::ResourceManager::getInstance();

  auto allocator = rm.getAllocator(resource);

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

  constexpr std::size_t SIZE = 1024;

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

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

  // _sphinx_tag_tut_deallocate_start
  pooled_allocator.deallocate(data);
  // _sphinx_tag_tut_deallocate_end

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

int main(int, char**)
{
  // _sphinx_tag_tut_anyallocator_start
  allocate_and_deallocate_pool("HOST");

#if defined(UMPIRE_ENABLE_DEVICE)
  allocate_and_deallocate_pool("DEVICE");
#endif
#if defined(UMPIRE_ENABLE_UM)
  allocate_and_deallocate_pool("UM");
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  allocate_and_deallocate_pool("PINNED");
#endif
  // _sphinx_tag_tut_anyallocator_end

  return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, 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);

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

  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**)
{
  // _sphinx_tag_tut_device_sized_pool_start
  allocate_and_deallocate_pool("HOST", 65536, 512);
#if defined(UMPIRE_ENABLE_DEVICE)
  allocate_and_deallocate_pool("DEVICE", (1024 * 1024 * 1024), (1024 * 1024));
#endif
#if defined(UMPIRE_ENABLE_UM)
  allocate_and_deallocate_pool("UM", (1024 * 64), 1024);
#endif
#if defined(UMPIRE_ENABLE_PINNED)
  allocate_and_deallocate_pool("PINNED", (1024 * 16), 1024);
#endif
  // _sphinx_tag_tut_device_sized_pool_end

  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 instrospection 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 or find out the associated *umpire::Platform*, which can help you decide where to operate on this 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;
```

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

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

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

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, 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"
#if defined(UMPIRE_ENABLE_DEVICE)
    ,
    "DEVICE"
#endif
#if defined(UMPIRE_ENABLE_UM)
    ,
    "UM"
#endif
#if defined(UMPIRE_ENABLE_PINNED)
    ,
    "PINNED"
#endif
  };

  for (auto& destination : destinations) {
```

(continues on next page)

(continued from previous page)

```
    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;

    // _sphinx_tag_tut_getallocator_start
    auto found_allocator = rm.getAllocator(data);
    // _sphinx_tag_tut_getallocator_end

    // _sphinx_tag_tut_getinfo_start
    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;
    // _sphinx_tag_tut_getinfo_end

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

    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, no 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};
```

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) 2016-20, 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/TypedAllocator.hpp"

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

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

  double* my_doubles = double_allocator.allocate(1024);

  double_allocator.deallocate(my_doubles, 1024);
  // _sphinx_tag_tut_typed_alloc_end

  // _sphinx_tag_tut_vector_alloc_start
  std::vector<double, umpire::TypedAllocator<double>> my_vector{
      double_allocator};
  // _sphinx_tag_tut_vector_alloc_end

  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 occured as part of the run that generated the log.

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

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

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:

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

CHAPTER

THREE

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_NUMA	Off	Enable NUMA support
ENABLE_FILE_RESOURCE	Off	Enable FILE support
ENABLE_TESTS	On	Build test executables
ENABLE_BENCHMARKS	On	Build benchmark programs
ENABLE_LOGGING	On	Enable Logging within Umpire
ENABLE_SLIC	Off	Enable SLIC logging
ENABLE_BACKTRACE	Off	Enable backtraces for allocations
ENABLE_BACKTRACE_SYMBOLS	Off	Enable symbol lookup for backtraces
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
ENABLE_PERFORMANCE_TESTS	Off	Build and run performance tests

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 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 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_FILE_RESOURCE This option will allow the build to make all File Memory Allocation files. If Umpire is built without FILE, CUDA or HIP support, then only the HOST allocator is available for use.
- ENABLE_TESTS 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_BACKTRACE This option enables collection of backtrace information for each allocation.
- ENABLE_BACKTRACE_SYMBOLS This option enables symbol information to be provided with backtraces. This requires -ldl to be specified for using programs.
- 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.
- ENABLE_PERFORMANCE_TESTS Build and run performance tests

CHAPTER

FOUR

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);
```

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);

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

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

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
```

(continues on next page)

(continued from previous page)

```
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/DynamicPool.hpp"
#include "umpire/util/Macros.hpp"

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 aligment)
  //
  // _sphinx_tag_tut_create_pool_start
  auto pooled_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>(
      "GPU_POOL", allocator, 4ul * 1024ul * 1024ul * 1024ul + 1);
  // _sphinx_tag_tut_create_pool_end

  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
  //
  // _sphinx_tag_tut_grow_pool_start
  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;
  // _sphinx_tag_tut_grow_pool_end

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

  return 0;
}
```

4.2 Disable Introspection

If you know that you won't be using any of Umpire's introspection capabalities 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:

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

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/DynamicPool.hpp"
#include "umpire/util/Macros.hpp"

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

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

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

  void* data = pooled_allocator.allocate(1024);
```

(continues on next page)

(continued from previous page)

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

  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 specificy 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:

```
  //
  // Create an allocator that applied "PREFFERED_LOCATION" advice to set the
  // GPU as the preferred location.
  //
  auto preferred_location_allocator =
      rm.makeAllocator<umpire::strategy::AllocationAdvisor>(
          "preferred_location_device", 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) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/AllocationAdvisor.hpp"
#include "umpire/strategy/DynamicPool.hpp"
#include "umpire/util/Macros.hpp"

int main(int, char**)
{
```

(continues on next page)

(continued from previous page)

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

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

  // _sphinx_tag_tut_pool_advice_start
  //
  // Create an allocator that applied "PREFFERED_LOCATION" advice to set the
  // GPU as the preferred location.
  //
  auto preferred_location_allocator =
      rm.makeAllocator<umpire::strategy::AllocationAdvisor>(
          "preferred_location_device", 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);
  // _sphinx_tag_tut_pool_advice_end

  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 specific which device the advice relates to. One type of advice that can be applied is "preferred location", and you can specificy 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

```
  //
  // Create an allocator that applied "PREFFERED_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);
```

The complete example is included below:

```cpp
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/AllocationAdvisor.hpp"
#include "umpire/util/Exception.hpp"

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

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

  // _sphinx_tag_tut_device_advice_start
  //
  // Create an allocator that applied "PREFFERED_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);

    // _sphinx_tag_tut_device_advice_end
    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) 2016-20, 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");

  //
  // 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");
  // _sphinx_tag_tut_move_host_to_managed_start
  double* um_data = static_cast<double*>(rm.move(host_data, um_allocator));
  // _sphinx_tag_tut_move_host_to_managed_end

  //
  // 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:

```
  //
  // 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::DynamicPool::percent_releasable(75);
```

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

```
  //
  // 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);
```

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, 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"
#include "umpire/util/Macros.hpp"

int main(int, char**)
```

(continues on next page)

(continued from previous page)

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

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

  // _sphinx_tag_tut_creat_heuristic_fun_start
  //
  // 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::DynamicPool::percent_releasable(75);
  // _sphinx_tag_tut_creat_heuristic_fun_end

  // _sphinx_tag_tut_use_heuristic_fun_start
  //
  // 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);
  // _sphinx_tag_tut_use_heuristic_fun_end

  //
  // 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) {
    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 releaseable. "
              << 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) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/NumaPolicy.hpp"
#include "umpire/util/Macros.hpp"
#include "umpire/util/numa.hpp"

#if defined(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();

  // 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) {
```

(continues on next page)

(continued from previous page)

```
    // 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");
    }
  }

#if defined(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)
    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::getLargestAvailableB` 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 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) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/DynamicPool.hpp"
#include "umpire/util/Exception.hpp"
#include "umpire/util/wrap_allocator.hpp"

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

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

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

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

  auto ptr = pool.allocate(1024);

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

(continues on next page)

(continued from previous page)

```
  // _sphinx_tag_tut_get_info_end

  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) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/strategy/AllocationTracker.hpp"
#include "umpire/strategy/DynamicPool.hpp"
#include "umpire/util/Exception.hpp"
#include "umpire/util/wrap_allocator.hpp"

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

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

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

  if (dynamic_pool) {
    // _sphinx_tag_tut_call_coalesce_start
    dynamic_pool->coalesce();
    // _sphinx_tag_tut_call_coalesce_end
```

(continues on next page)

(continued from previous page)

```
  } 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:

```
      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)
```

The complete example is included below:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Copyright (c) 2016-20, 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
      type(UmpireAllocator) pinned_pool
      type(UmpireResourceManager) rm

      ! _sphinx_tag_tut_pinned_fortran_start
      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)
      ! _sphinx_tag_tut_pinned_fortran_end

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

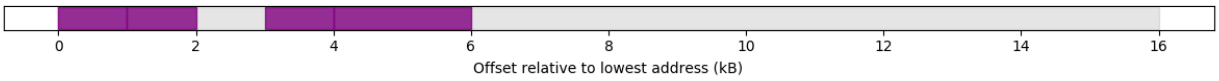
4.11 Visualizing Allocators

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:

```cpp
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <fstream>

#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/Umpire.hpp"
#include "umpire/strategy/DynamicPool.hpp"

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);

  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
  {
```

(continues on next page)

(continued from previous page)

```
    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) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>

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

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
```

(continues on next page)

(continued from previous page)

```
   * 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) 2016-20, 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"
#include "umpire/strategy/DynamicPool.hpp"
#include "umpire/strategy/ThreadSafeAllocator.hpp"

int main(int, char**)
{
  // _sphinx_tag_tut_thread_safe_start
  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);
  // _sphinx_tag_tut_thread_safe_end

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

  return 0;
}
```

4.14 Using File System Allocator (FILE)

Umpire supports the use of file based memory allocation. When `ENABLE_FILE_RESOURCE` is enabled, the environment variables `UMPIRE_MEMORY_FILE_DIR` can be used to determine where memory can be allocated from:

Variable	Default	Description
UMPIRE_MEMORY_FILE_DIR	./	Directory to create and allocate file based allocations

Requesting the allocation takes two steps: 1) getting a "FILE" allocator, 2) requesting the amount of memory to allocate.

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

  std::size_t* A = (std::size_t*)alloc.allocate(sizeof(size_t));
```

To deallocate:

```
  alloc.deallocate(A);
```

The complete example is included below:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, 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"

int main(int, char** argv)
{
  // _sphinx_tag_tut_file_allocate_start
  auto& rm = umpire::ResourceManager::getInstance();
  umpire::Allocator alloc = rm.getAllocator("FILE");

  std::size_t* A = (std::size_t*)alloc.allocate(sizeof(size_t));
  // _sphinx_tag_tut_file_allocate_end

  // _sphinx_tag_tut_file_deallocate_start
  alloc.deallocate(A);
  // _sphinx_tag_tut_file_deallocate_end

  return 0;
}
```

4.15 Using Burst Buffers On Lassen

On Lassen, 1) Download the latest version of Umpire 2) request a private node to build:

```
$ git clone --recursive https://github.com/LLNL/Umpire.git
$ lalloc 1 -stage storage=64
```

Note that `-stage storage=64` is needed in order to work with the Burst Buffers. 3) Additionally, the environment variable needs to set to `$BBPATH` :

```
$ export UMPIRE_MEMORY_FILE_DIR=$BBPATH/
```

4.15.1 Running File Resource Benchmarks

Continue building Umpire on 1 node, and set the `-DENABLE_FILE_RESOURCE=On` :

```
$ mkdir build && cd build
$ lrun -n 1 cmake -DENABLE_FILE_RESOURCE=On -DENABLE_OPENMP=On ../ && make
```

To run the built-in benchmarks in Umpire from the build run:

```
$ lrun -n 1 --threads=** ./bin/file_resource_benchmarks ##
```

** is a placeholder for the amount of threads wanted to run the benchmark on. ## stands for the number of array elements wanted to be passed through the benchmark. This number can range from 1-100,000,000,000.

Results should appear like:

```
Array Size:   1         Memory Size: 8e-06 MB
Total Arrays: 3                Total Memory Size: 2.4e-05 MB

HOST
```

(continues on next page)

(continued from previous page)

```
  Initialization:      0.0247461 MB/sec
  Initialization Time: 0.000969849 sec
  ---------------------------------------
  Copy:                0.890918 MB/sec
  Copy Time:           1.7959e-05 sec
  ---------------------------------------
  Scale:               0.928074 MB/sec
  Scale Time:          1.724e-05 sec
  ---------------------------------------
  Add:                 1.321 MB/sec
  Add Time:            1.8168e-05 sec
  ---------------------------------------
  Triad:               1.24102 MB/sec
  Triad Time:          1.9339e-05 sec
  ---------------------------------------
  Total Time:          0.00104323 sec

FILE
  Initialization:      0.210659 MB/sec
  Initialization Time: 0.000113928 sec
  ---------------------------------------
  Copy:                0.84091 MB/sec
  Copy Time:           1.9027e-05 sec
  ---------------------------------------
  Scale:               0.938086 MB/sec
  Scale Time:          1.7056e-05 sec
  ---------------------------------------
  Add:                 1.28542 MB/sec
  Add Time:            1.8671e-05 sec
  ---------------------------------------
  Triad:               1.54689 MB/sec
  Triad Time:          1.5515e-05 sec
  ---------------------------------------
  Total Time:          0.000184726 sec
```

This benchmark run similar to the STREAM Benchmark test and can also run a benchmark for the additional allocators like `UM` for CUDA and `DEVICE` for HIP.

CHAPTER

FIVE

FEATURES

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 Allocator Accessibility

The Umpire library provides a variety of *umpire::resource::MemoryResource* s which can be used to create *umpire::Allocator* s depending on what's available on your system. The resources are explained more on the Resources page.

Additionally, the platforms that Umpire supports is defined by the CAMP library. This means that there is also a selection of platforms for which an allocator can be associated with as well. For example, an Allocator created with the pinned memory resource can be used with the host, cuda, hip, or sycl platforms.

Because of these options, it can be difficult to trace not only which memory resource an allocator has been created with but also which allocators can be accessed by which platforms. Umpire has the memory resource trait, `resource_type`, to provide the ability to query which memory resource is associated with a particular allocator (See example here).

Additionally, Umpire has a function, `is_accessible(Platform p, Allocator a)`, that determines if a particular allocator is accessible by a particular platform (See example here). The `allocator_accessibility.cpp` test checks what platforms are available and confirms that all memory resources which should be accessible to that platform can actually be accessed and used.

For example, if a *umpire::Allocator*, `alloc`, is created with the host memory resource and we want to know if it should be accessible from the `omp_target` CAMP platform, then we can use the

`is_accessible(Platform::omp_target, alloc)` function and find that it should be accessible. The `allocator_access.cpp` file demonstrates this functionality for the *host* platform specifically.

5.2.1 Allocator Inaccessibility Configuration

On a different note, for those allocators that are deemed inaccessible, it may be useful to double check or confirm that the allocator can in fact NOT access memory on that given platform. In this case, the cmake flag, `ENABLE_INACCESSIBILITY_TESTS`, will need to be turned on.

5.2.2 Build and Run Configuration

To build and run these files, either use uberenv or the appropriate cmake flags for the desired platform and then run `ctest -T test -R allocator_accessibility_tests --output-on-failure` for the test code and `./bin/alloc_access` for the example code.

Note: The Developer's Guide shows how to configure Umpire with uberenv to build with different CAMP platforms.

Below, the `allocator_access.cpp` code is shown to demonstrate how this functionality can be used during development.

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>
#include <string>
#include "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"
#include "umpire/Umpire.hpp"

bool is_accessible_from_host(umpire::Allocator a)
{
  if(umpire::is_accessible(umpire::Platform::host, a)) {
    std::cout << "The allocator, " << a.getName()
              << ", is accessible." << std::endl;
    return true;
  } else {
    std::cout << "The allocator, " << a.getName()
              << ", is _not_ accessible." << std::endl << std::endl;
    return false;
  }
}

//////////////////////////////////////////////////////////////////////////////
↪///
//Depending on how Umpire has been set up, several different allocators could be␣
↪accessible
//from the host CAMP platform. This example will create a list of all currently␣
↪available
//allocators and then determine whether each can be accessed from the host platform.
//(To test other platforms, see allocator accessibility test.)
```

(continues on next page)

(continued from previous page)

```
//////////////////////////////////////////////////////////////////////////////////////
↪//
int main()
{
  auto& rm = umpire::ResourceManager::getInstance();

  std::vector<std::string> allNames = rm.getResourceNames();
  std::vector<umpire::Allocator> alloc;

  ////////////////////////////////////////////////
  //Create an allocator for each available type
  ////////////////////////////////////////////////
  std::cout << "Available allocators: ";
  for(auto a : allNames) {
    if (a.find("::") == std::string::npos) {
      alloc.push_back(rm.getAllocator(a));
      std::cout << a << " ";
    }
  }
  std::cout << std::endl;

  ////////////////////////////////////////////////
  //Test accessibility
  ////////////////////////////////////////////////
  std::cout << "Testing the available allocators for accessibility from the CAMP host␣
↪platform:" << std::endl;
  const int size = 100;
  for(auto a : alloc) {
    if(is_accessible_from_host(a)) {
      int* data = static_cast<int*>(a.allocate(size*sizeof(int)));
      for(int i = 0; i < size; i++) {
        data[i] = i * i;
      }
      UMPIRE_ASSERT(data[size-1] == (size-1) * (size-1) && "Inequality found in array␣
↪that should be accessible");
    }
  }

  return 0;
}
```

5.3 Backtrace

The Umpire library may be configured to provide using programs with backtrace information as part of Umpire thrown exception description strings.

Umpire may also be configured to collect and provide backtrace information for each Umpire provided memory allocation performed.

5.3.1 Build Configuration

Backtrace is enabled in Umpire builds with the following:

- **``cmake … -DENABLE_BACKTRACE=On …``** to backtrace capability in Umpire.
- **``cmake -DENABLE_BACKTRACE=On -DENABLE_BACKTRACE_SYMBOLS=On …``** to enable Umpire to display symbol information with backtrace. **Note:** Using programs will need to add the `-rdyanmic` and `-ldl` linker flags in order to properly link with this configuration of the Umpire library.

5.3.2 Runtime Configuration

For versions of the Umpire library that are backtrace enabled (from flags above), the user may expect the following.

Backtrace information will always be provided in the description strings of umpire generated exception throws.

Setting the environment variable `UMPIRE_BACKTRACE=On` will cause Umpire to record backtrace information for each memory allocation it provides.

Setting the environment variable `UMPIRE_LOG_LEVEL=Error` will cause to Umpire to log backtrace information for each of the leaked Umpire allocations found during application exit.

A programatic interface is also availble via the func::*umpire::print_allocator_records* free function.

An example for checking and displaying the information this information logged above may be found here:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, Lawrence Livermore National Security, LLC and Umpire
// project contributors. See the COPYRIGHT file for details.
//
// SPDX-License-Identifier: (MIT)
//////////////////////////////////////////////////////////////////////////////
#include <iostream>
#include <sstream>

#include "umpire/ResourceManager.hpp"
#include "umpire/Umpire.hpp"
#include "umpire/strategy/DynamicPool.hpp"

int main(int, char**)
{
  auto& rm = umpire::ResourceManager::getInstance();
  auto allocator = rm.getAllocator("HOST");
  auto pool_allocator = rm.makeAllocator<umpire::strategy::DynamicPool>(
      "host_dynamic_pool", allocator);

  allocator.allocate(16);
  allocator.allocate(32);
  allocator.allocate(64);

  pool_allocator.allocate(128);
  pool_allocator.allocate(256);
  pool_allocator.allocate(512);

  std::stringstream ss;
  umpire::print_allocator_records(allocator, ss);
  umpire::print_allocator_records(pool_allocator, ss);

  // Example #1 of 3 - Leaked allocations
```

(continues on next page)

(continued from previous page)

```
  //
  // If Umpire compiled with -DENABLE_BACKTRACE=On, then backtrace
  // information will be printed for each of the allocations made above.
  //
  // Otherwise, if Umpire was not compiled with -DENABLE_BACKTRACE=On,
  // then only the addresses and size information for each allocation will be
  // printed.
  //
  if (!ss.str().empty())
    std::cout << ss.str();

  // Example #2 of 3 - Umpire error exceptions
  //
  // When umpire throws an exception, a backtrace to the offending call will
  // be provided in the exception string.
  //
  void* bad_ptr = (void*)0xBADBADBAD;

  try {
    allocator.deallocate(bad_ptr); // Will cause a throw from umpire
  } catch (const std::exception& exc) {
    //
    // exc.what() string will also contain a backtrace
    //
    std::cout << "Exception thrown from Umpire:" << std::endl << exc.what();
  }

  // Example #3 of 3 - Leak detection
  //
  // When the program terminates, Umpire's resource manager will be
  // deconstructed.  During deconstruction, Umpire will log the size and
  // address, of each leaked allocation in each allocator.
  //
  // If Umpire was compiled with -DENABLE_BACKTRACE=On, backtrace
  // information will also be logged for each leaked allocation in each
  // allocator.
  //
  // To enable (and see) the umpire logs, set the environment variable
  // UMPIRE_LOG_LEVEL=Error.
  //
  return 0;
}
```

5.4 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.

5.5 Logging and Replay of Umpire Events

5.5.1 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.

5.5.2 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()*

5.5.3 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:

5.5.4 Interpretting 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.

5.5.5 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. Note that this could also be done with the *FILE* memory resource. The *result* is a reference to the object that was created within Umpire for that resource.

5.5.6 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" } }
```

(continues on next page)

(continued from previous page)

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

5.5.7 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.

5.5.8 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.

5.5.9 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
```

5.6 Operations

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

5.6.1 Provided Operations

namespace umpire::**op**

class CudaAdviseAccessedByOperation : **public** umpire::*op*::*MemoryOperation*
#include <umpire/op/CudaAdviseAccessedByOperation.hpp>

class CudaAdvisePreferredLocationOperation : **public** umpire::*op*::*MemoryOperation*
#include <umpire/op/CudaAdvisePreferredLocationOperation.hpp>

class CudaAdviseReadMostlyOperation : **public** umpire::*op*::*MemoryOperation*
#include <umpire/op/CudaAdviseReadMostlyOperation.hpp>

class CudaAdviseUnsetAccessedByOperation : **public** umpire::*op*::*MemoryOperation*
#include <umpire/op/CudaAdviseUnsetAccessedByOperation.hpp>

class CudaAdviseUnsetPreferredLocationOperation : **public** umpire::*op*::*MemoryOperation*
#include <umpire/op/CudaAdviseUnsetPreferredLocationOperation.hpp>

class CudaAdviseUnsetReadMostlyOperation : **public** umpire::*op*::*MemoryOperation*
#include <umpire/op/CudaAdviseUnsetReadMostlyOperation.hpp>

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

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

class CudaMemPrefetchOperation : **public** umpire::*op*::*MemoryOperation*
#include <umpire/op/CudaMemPrefetchOperation.hpp>

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 transfrom 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::CudaGetAttributeOperation< ATTRIBUTE >*, *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::OpenMPTargetCopyOperation*, *umpire::op::OpenMPTargetMemsetOperation*, *umpire::op::SyclCopyFromOperation*, *umpire::op::SyclCopyOperation*, *umpire::op::SyclCopyToOperation*, *umpire::op::SyclMemPrefetchOperation*, *umpire::op::SyclMemsetOperation*

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 OpenMPTargetCopyOperation : **public** umpire::*op*::*MemoryOperation*

#include <umpire/op/OpenMPTargetCopyOperation.hpp>

class OpenMPTargetMemsetOperation : **public** umpire::*op*::*MemoryOperation*

#include <umpire/op/OpenMPTargetMemsetOperation.hpp>

struct pair_hash

#include <umpire/op/MemoryOperationRegistry.hpp>

class SyclCopyFromOperation : **public** umpire::*op*::*MemoryOperation*

#include <umpire/op/SyclCopyFromOperation.hpp> Copy operation to move data between a Intel GPU and CPU memory.

class SyclCopyOperation : **public** umpire::*op*::*MemoryOperation*

#include <umpire/op/SyclCopyOperation.hpp> Copy operation to move data between two GPU addresses.

class SyclCopyToOperation : **public** umpire::*op*::*MemoryOperation*

#include <umpire/op/SyclCopyToOperation.hpp> Copy operation to move data between a Intel GPU and CPU memory.

class SyclMemPrefetchOperation : **public** umpire::*op*::*MemoryOperation*
#include <umpire/op/SyclMemPrefetchOperation.hpp>

class SyclMemsetOperation : **public** umpire::*op*::*MemoryOperation*
#include <umpire/op/SyclMemsetOperation.hpp> Memset on Intel GPU device memory.

5.7 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 alloctaion. 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::AlignedAllocator*, *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::QuickPool*, *umpire::strategy::SizeLimiter*, *umpire::strategy::SlotPool*, *umpire::strategy::ThreadSafeAllocator*, *umpire::strategy::ZeroByteHandler*

5.7.1 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 syncronizing access to the allocators interface.

CHAPTER

SIX

API

6.1 Class Hierarchy

6.2 File Hierarchy

6.3 Full API

6.3.1 Namespaces

Namespace genumpiresplicer

Contents

Functions

Variables

Namespace iso_c_binding

Namespace std

STL namespace.

Namespace umpire

Contents

Namespaces

Classes

Functions

Typedefs

- *Typedef umpire::Platform*

Variables

- *Variable umpire::env_name*

Namespace umpire::@87

Namespace umpire::alloc

Contents

Classes

- *Struct CudaMallocAllocator*
- *Struct CudaMallocManagedAllocator*
- *Struct CudaPinnedAllocator*
- *Struct HipMallocAllocator*
- *Struct HipMallocManagedAllocator*
- *Struct HipPinnedAllocator*
- *Struct MallocAllocator*
- *Struct OpenMPTargetAllocator*
- *Struct PosixMemalignAllocator*
- *Struct SyclMallocAllocator*
- *Struct SyclMallocManagedAllocator*
- *Struct SyclPinnedAllocator*

Namespace umpire::numa

Contents

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

- *Struct pair_hash*
- *Class CudaAdviseAccessedByOperation*
- *Class CudaAdvisePreferredLocationOperation*
- *Class CudaAdviseReadMostlyOperation*
- *Class CudaAdviseUnsetAccessedByOperation*
- *Class CudaAdviseUnsetPreferredLocationOperation*
- *Class CudaAdviseUnsetReadMostlyOperation*
- *Class CudaCopyFromOperation*
- *Class CudaCopyOperation*
- *Class CudaCopyToOperation*
- *Template Class CudaGetAttributeOperation*
- *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 OpenMPTargetCopyOperation*
- *Class OpenMPTargetMemsetOperation*
- *Class SyclCopyFromOperation*
- *Class SyclCopyOperation*
- *Class SyclCopyToOperation*
- *Class SyclMemPrefetchOperation*
- *Class SyclMemsetOperation*

Namespace umpire::resource

Contents

- *Classes*
- *Enums*
- *Functions*

Classes

- *Struct MemoryResourceTypeHash*
- *Class CudaConstantMemoryResource*
- *Class CudaConstantMemoryResourceFactory*
- *Class CudaDeviceMemoryResource*
- *Class CudaDeviceResourceFactory*
- *Class CudaPinnedMemoryResourceFactory*
- *Class CudaUnifiedMemoryResourceFactory*
- *Template Class DefaultMemoryResource*
- *Class FileMemoryResource*
- *Class FileMemoryResourceFactory*
- *Class HipConstantMemoryResource*
- *Class HipConstantMemoryResourceFactory*
- *Class HipDeviceMemoryResource*
- *Class HipDeviceResourceFactory*
- *Class HipPinnedMemoryResourceFactory*
- *Class HipUnifiedMemoryResourceFactory*

- *Class HostResourceFactory*
- *Class MemoryResource*
- *Class MemoryResourceFactory*
- *Class MemoryResourceRegistry*
- *Class NullMemoryResource*
- *Class NullMemoryResourceFactory*
- *Class OpenMPTargetResourceFactory*
- *Template Class SyclDeviceMemoryResource*
- *Class SyclDeviceResourceFactory*
- *Class SyclPinnedMemoryResourceFactory*
- *Class SyclUnifiedMemoryResourceFactory*

Enums

- *Enum MemoryResourceType*

Functions

- *Function umpire::resource::resource_to_device_id*
- *Function umpire::resource::resource_to_string*
- *Function umpire::resource::string_to_resource*

Namespace umpire::strategy

Contents

Namespaces

- *Namespace umpire::strategy::mixins*

Classes

- *Struct FixedPool::Pool*
- *Struct QuickPool::Chunk*
- *Class AlignedAllocator*
- *Class AllocationAdvisor*
- *Class AllocationPrefetcher*
- *Class AllocationStrategy*
- *Class AllocationTracker*
- *Class DynamicPoolList*
- *Class DynamicPoolMap*
- *Class FixedPool*
- *Class MixedPool*
- *Class MonotonicAllocationStrategy*
- *Class NamedAllocationStrategy*
- *Class NumaPolicy*
- *Class QuickPool*
- *Template Class QuickPool::pool_allocator*
- *Class SizeLimiter*
- *Class SlotPool*
- *Class ThreadSafeAllocator*
- *Class ZeroByteHandler*

Functions

- *Function umpire::strategy::find_first_set*
- *Function umpire::strategy::operator<<*

Typedefs

- *Typedef umpire::strategy::DynamicPool*

Variables

- *Variable umpire::strategy::bits_per_int*

Namespace umpire::strategy::mixins

Contents

Classes

- *Class AlignedAllocation*
- *Class Inspector*

Namespace umpire::util

Contents

Namespaces

- *Namespace umpire::util::@195*
- *Namespace umpire::util::@210*
- *Namespace umpire::util::message*

Classes

- *Template Struct RecordList::Block*
- *Struct AllocationRecord*
- *Struct backtrace*
- *Template Struct backtracer*
- *Template Struct backtracer< trace_always >*
- *Template Struct backtracer< trace_optional >*
- *Struct FixedMallocPool::Pool*
- *Struct iterator_begin*
- *Struct iterator_end*
- *Struct trace_always*
- *Struct trace_optional*
- *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*

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::@195

Namespace umpire::util::@210

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_allocation_count*
- *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::get_backtrace*
- *Function umpire_mod::pointer_contains*
- *Function umpire_mod::pointer_overlaps*
- *Function umpire_mod::resourcemanager_add_alias*
- *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_id*
- *Function umpire_mod::resourcemanager_is_allocator_name*
- *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_quick_pool*
- *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*

- *Function umpire_mod::resourcemanager_remove_alias*

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 DynamicSizePool::**Block**

Public Members

char ***data**

std::size_t **size**

std::size_t **blockSize**

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 FixedSizePool::**Pool**

Public Members

unsigned char ***data**

unsigned int ***avail**

unsigned int **numAvail**

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 ***base**

const char ***ccharp**

union *s_umpire_SHROUD_array*::**[anonymous]** **addr**

int **type**

size_t **elem_len**

size_t **size**

int **rank**

long **shape**[7]

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_quickpool

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_strategy_quickpool

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

void *__allocate__(std::size_t *size*)

void __deallocate__(void **ptr*)

Struct CudaMallocAllocator

- Defined in file_umpire_alloc_CudaMallocAllocator.hpp

Struct Documentation

struct umpire::alloc::**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 umpire::alloc::**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.

bool __isAccessible__(*Platform p*)

Struct CudaPinnedAllocator

- Defined in file_umpire_alloc_CudaPinnedAllocator.hpp

Struct Documentation

struct umpire::alloc::**CudaPinnedAllocator**

Public Functions

void *__allocate__(std::size_t *bytes*)

void __deallocate__(void **ptr*)

bool __isAccessible__(*Platform p*)

Struct HipMallocAllocator

- Defined in file_umpire_alloc_HipMallocAllocator.hpp

Struct Documentation

__struct__ umpire::alloc::__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 HipMallocManagedAllocator

- Defined in file_umpire_alloc_HipMallocManagedAllocator.hpp

Struct Documentation

struct umpire::alloc::**HipMallocManagedAllocator**

Uses hipMallocManaged and hipFree to allocate and deallocate unified memory on AMD GPUs.

Public Functions

void ***allocate**(std::size_t *bytes*)

Allocate bytes of memory using hipMallocManaged.

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 be free'd.

bool **isAccessible**(*Platform p*)

Struct HipPinnedAllocator

- Defined in file_umpire_alloc_HipPinnedAllocator.hpp

Struct Documentation

struct umpire::alloc::**HipPinnedAllocator**

Public Functions

void ***allocate**(std::size_t *bytes*)

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

bool **isAccessible**(*Platform p*)

Struct MallocAllocator

- Defined in file_umpire_alloc_MallocAllocator.hpp

Struct Documentation

struct umpire::alloc::**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.

bool **isHostPageable**()

bool **isAccessible**(*Platform p*)

Struct OpenMPTargetAllocator

- Defined in file_umpire_alloc_OpenMPTargetAllocator.hpp

Struct Documentation

struct umpire::alloc::**OpenMPTargetAllocator**
Uses malloc and free to allocate and deallocate CPU memory.

Public Functions

OpenMPTargetAllocator(int *_device*)

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.

bool **isAccessible**(*Platform p*)

Public Members

int **device**

Struct PosixMemalignAllocator

- Defined in file_umpire_alloc_PosixMemalignAllocator.hpp

Struct Documentation

struct umpire::alloc::**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.

bool **isAccessible**(*Platform* _p_)

Struct SyclMallocAllocator

- Defined in file_umpire_alloc_SyclMallocAllocator.hpp

Struct Documentation

struct `umpire::alloc::`**SyclMallocAllocator**
Uses sycl's malloc and free to allocate and deallocate memory on Intel GPUs.

Public Functions

void *__allocate__(std::size_t _size_, **const** cl::sycl::queue &_queue_t_)
Allocate bytes of memory using SYCL malloc

Return Pointer to start of the allocation on device.

Parameters

- `size`: Number of bytes to allocate.
- `queue_t`: SYCL queue for providing information on device and context

Exceptions

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

void **deallocate**(void *_ptr_, **const** cl::sycl::queue &_queue_t_)
Deallocate memory using SYCL free.

Parameters

- `ptr`: Address to deallocate.
- `queue_t`: SYCL queue this pointer was asociated with

Exceptions

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

bool **isAccessible**(umpire::*Platform* _p_)

Struct SyclMallocManagedAllocator

- Defined in file_umpire_alloc_SyclMallocManagedAllocator.hpp

Struct Documentation

struct umpire::alloc::**SyclMallocManagedAllocator**
Uses sycl_shared and sycl_free to allocate and deallocate unified shared memory (USM) on Intel GPUs.

Public Functions

void ***allocate**(std::size_t *bytes*, **const** cl::sycl::queue &*queue_t*)
Allocate bytes of memory using cl::sycl::malloc_shared.

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 **usm_ptr*, **const** cl::sycl::queue &*queue_t*)
Deallocate memory using cl::sycl::free.

Parameters

- `usm_ptr`: Address to deallocate.

Exceptions

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

bool **isAccessible**(*Platform p*)

Struct SyclPinnedAllocator

- Defined in file_umpire_alloc_SyclPinnedAllocator.hpp

Struct Documentation

struct umpire::alloc::**SyclPinnedAllocator**
Uses sycl's malloc_host and free to allocate and deallocate pinned memory on host.

Public Functions

void ***allocate**(std::size_t *size*, **const** cl::sycl::queue &*queue_t*)
Allocate bytes of memory using SYCL malloc_host

Return Pointer to start of the allocation on host.

Parameters

- `size`: Number of bytes to allocate.
- `queue_t`: SYCL queue for providing information on device and context

Exceptions

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

void **deallocate**(void **ptr*, **const** cl::sycl::queue &*queue_t*)
Deallocate memory using SYCL free.

Parameters

- `ptr`: Address to deallocate.
- `queue_t`: SYCL queue this pointer was asociated with

Exceptions

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

bool **isAccessible**(*Platform p*)

Struct MemoryResourceTraits

- Defined in file_umpire_util_MemoryResourceTraits.hpp

Struct Documentation

struct umpire::**MemoryResourceTraits**

Public Types

enum optimized_for
Values:

enumerator any

enumerator latency

enumerator bandwidth

enumerator access

enum vendor_type
Values:

enumerator unknown

enumerator amd

enumerator ibm

enumerator intel

enumerator nvidia

enum memory_type

Values:

enumerator unknown

enumerator ddr

enumerator gddr

enumerator hbm

enumerator nvme

enum resource_type

Values:

enumerator unknown

enumerator host

enumerator device

enumerator device_const

enumerator pinned

enumerator um

enumerator file

Public Members

int **id**

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*

resource_type **resource** = *resource_type*::*unknown*

Struct pair_hash

- Defined in file_umpire_op_MemoryOperationRegistry.hpp

Struct Documentation

struct umpire::*op*::**pair_hash**

Public Functions

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

Struct MemoryResourceTypeHash

- Defined in file_umpire_resource_MemoryResourceTypes.hpp

Struct Documentation

struct umpire::resource::**MemoryResourceTypeHash**

Public Functions

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

Struct FixedPool::Pool

- Defined in file_umpire_strategy_FixedPool.hpp

Nested Relationships

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

Struct Documentation

struct umpire::strategy::*FixedPool*::**Pool**

Public Functions

Pool(*AllocationStrategy* **allocation_strategy*, **const** std::size_t *object_bytes*, **const** std::size_t *objects_per_pool*, **const** std::size_t *avail_bytes*)

Public Members

AllocationStrategy ***strategy**

char ***data**

int ***avail**

std::size_t **num_avail**

Struct QuickPool::Chunk

- Defined in file_umpire_strategy_QuickPool.hpp

Nested Relationships

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

Struct Documentation

struct umpire::strategy::*QuickPool*::**Chunk**

Public Functions

Chunk(void **ptr*, std::size_t *s*, std::size_t *cs*)

Public Members

void ***data** = {nullptr}

std::size_t **size** = {0}

std::size_t **chunk_size** = {0}

bool **free** = {true}

Chunk ***prev** = {nullptr}

Chunk ***next** = {nullptr}

SizeMap::iterator **size_map_it**

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 umpire::util::*AllocationMap*::*RecordList*::**Block**

Public Members

T **rec**

Block ***prev**

Struct AllocationRecord

- Defined in file_umpire_util_AllocationRecord.hpp

Struct Documentation

struct umpire::util::**AllocationRecord**

Public Functions

AllocationRecord(void **p*, std::size_t *s*, *strategy*::AllocationStrategy **strat*)

AllocationRecord()

Public Members

void ***ptr**

std::size_t **size**

strategy::AllocationStrategy ***strategy**

util::*backtrace* **allocation_backtrace**

Struct backtrace

- Defined in file_umpire_util_backtrace.hpp

Struct Documentation

struct umpire::util::**backtrace**

Public Members

std::vector<void*> **frames**

Template Struct backtracer

- Defined in file_umpire_util_backtrace.hpp

Struct Documentation

template<typename **TraceType** = *trace_optional*>
struct backtracer

Template Struct backtracer< trace_always >

- Defined in file_umpire_util_backtrace.inl

Struct Documentation

template<>
struct umpire::util::**backtracer**<*trace_always*>

Public Static Functions

void **get_backtrace**(*backtrace* &*bt*)

std::string **print**(**const** *backtrace* &*bt*)

Template Struct backtracer< trace_optional >

- Defined in file_umpire_util_backtrace.inl

Struct Documentation

template<>
struct umpire::util::**backtracer**<*trace_optional*>

Public Static Functions

void **get_backtrace**(*backtrace* &*bt*)

std::string **print**(**const** *backtrace* &*bt*)

Struct FixedMallocPool::Pool

- Defined in file_umpire_util_FixedMallocPool.hpp

Nested Relationships

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

Struct Documentation

struct umpire::util::*FixedMallocPool*::**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

Struct trace_always

- Defined in file_umpire_util_backtrace.hpp

Struct Documentation

struct trace_always

Struct trace_optional

- Defined in file_umpire_util_backtrace.hpp

Struct Documentation

struct trace_optional

Template Class DynamicSizePool

- Defined in file_umpire_strategy_DynamicSizePool.hpp

Nested Relationships

Nested Types

- *Struct DynamicSizePool::Block*

Inheritance Relationships

Base Type

- `private umpire::strategy::mixins::AlignedAllocation` (*Class AlignedAllocation*)

Class Documentation

template<class **IA** = *StdAllocator*>
class DynamicSizePool : **private** umpire::strategy::mixins::*AlignedAllocation*

Public Functions

DynamicSizePool(umpire::strategy::*AllocationStrategy* **const** std::size_t *first_minimum_pool_allocation_size* = (16 * 1024), **const** std::size_t *next_minimum_pool_allocation_size* = 256, **const** std::size_t *alignment* = 16)

DynamicSizePool(**const** *DynamicSizePool*&) = delete

~DynamicSizePool()

void ***allocate**(std::size_t *bytes*)

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

void **release**()

std::size_t **getActualSize**() **const**

std::size_t **getCurrentSize**() **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**()

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*)

void **allocateBlock**(**struct** *Block* *&*curr*, **struct** *Block* *&*prev*, 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*)

Protected Attributes

BlockPool **blockPool**

struct *Block* ***usedBlocks**

struct *Block* ***freeBlocks**

std::size_t **m_actual_bytes**

std::size_t **m_current_size** = {0}

std::size_t **m_first_minimum_pool_allocation_size**

std::size_t **m_next_minimum_pool_allocation_size**

bool **m_is_destructing** = {false}

Private Functions

std::size_t **aligned_round_up**(std::size_t *size*)
Round up the size to be an integral multple of configured alignment.

Return Size rounded up to be integral multiple of configured alignment

void ***aligned_allocate**(**const** std::size_t *size*)
Return an allocation of `size` bytes that is aligned on the configured alignment boundary.

void **aligned_deallocate**(void **ptr*)
Deallocate previously alligned allocation.

Private Members

strategy::AllocationStrategy ***m_allocator**

struct Block

Public Members

char ***data**

std::size_t **size**

std::size_t **blockSize**

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

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

unsigned char ***data**

unsigned int ***avail**

unsigned int **numAvail**

struct *Pool* ***next**

Class Allocator

- Defined in file_umpire_Allocator.hpp

Class Documentation

class umpire::**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 noexcept**
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 noexcept**
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 noexcept**
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*.

std::size_t **getAllocationCount**() **const noexcept**
Return the number of active allocations.

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

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

Return Pointer to the AllocationStrategy.

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

Return Platform for this *Allocator*.

Allocator() = default

Friends

friend class ::AllocatorTest

friend std::ostream &**operator<<**(std::ostream&, **const** *Allocator*&)

Class DeviceAllocator

- Defined in file_umpire_DeviceAllocator.hpp

Class Documentation

class umpire::**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__ DeviceAllocator (const DeviceAllocator &other)

__device__ void * 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 umpire::*op*::**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*)
Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

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

Exceptions

- *`util::Exception`*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class CudaAdvisePreferredLocationOperation

- Defined in file_umpire_op_CudaAdvisePreferredLocationOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**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 preffered 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*)
Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

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

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class CudaAdviseReadMostlyOperation

- Defined in file_umpire_op_CudaAdviseReadMostlyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**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*)
Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

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

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class CudaAdviseUnsetAccessedByOperation

- Defined in file_umpire_op_CudaAdviseUnsetAccessedByOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**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*)

Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

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

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class CudaAdviseUnsetPreferredLocationOperation

- Defined in file_umpire_op_CudaAdviseUnsetPreferredLocationOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**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 preffered 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*)

Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

- `src_ptr`: Pointer to source memory location.
- `dst_ptr`: Pointer to destinatino memory location.
- `src_allocation`: AllocationRecord of source.

- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

Exceptions

- *`util::Exception`*:

camp::resources::Event **`transform_async`**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **`apply_async`**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class CudaAdviseUnsetReadMostlyOperation

- Defined in file_umpire_op_CudaAdviseUnsetReadMostlyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

`class` `umpire::`*`op`*`::`**`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*)

Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

- `src_ptr`: Pointer to source memory location.

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

Exceptions

- *`util::Exception`*:

camp::resources::Event **`transform_async`**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **`apply_async`**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class CudaCopyFromOperation

- Defined in file_umpire_op_CudaCopyFromOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Copy operation to move data from a NVIDA 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*)

Transfrom 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 destinatino memory location.
- `src_allocation`: AllocationRecord of source.
- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

Exceptions

- *`util::Exception`*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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`*:

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class CudaCopyOperation

- Defined in file_umpire_op_CudaCopyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class `umpire::op::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*)
Transfrom 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 destinatino memory location.
- `src_allocation`: AllocationRecord of source.
- `dst_allocation`: AllocationRecord of destination.

- `length`: Number of bytes to transform.

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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*:

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class CudaCopyToOperation

- Defined in file_umpire_op_CudaCopyToOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**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*)
Transfrom 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 destinatino memory location.
- `src_allocation`: AllocationRecord of source.
- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

Exceptions

- *`util::Exception`*:

camp::resources::Event **`transform_async`**(void **src_ptr*, void ***dst_ptr*, umpire::util::*AllocationRecord* **src_allocation*, umpire::util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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`*:

camp::resources::Event **`apply_async`**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Template Class CudaGetAttributeOperation

- Defined in file_umpire_op_CudaGetAttributeOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

template<cudaMemRangeAttribute **`ATTRIBUTE`**>
`class` `umpire::`*`op`*`::CudaGetAttributeOperation` : **`public`** umpire::*op*::*MemoryOperation*

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

Public Functions

bool **check_apply**(void **src_ptr*, umpire::util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*) **override**

Uses cudaMemRangeGetAtribute to check attributes of a CUDA memory range.

Parameters

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

Exceptions

- *`util::Exception`*:

void **transform**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, 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 destinatino memory location.
- `src_allocation`: AllocationRecord of source.
- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

Exceptions

- *`util::Exception`*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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`*:

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class CudaMemPrefetchOperation

- Defined in file_umpire_op_CudaMemPrefetchOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**CudaMemPrefetchOperation** : **public** umpire::*op*::*MemoryOperation*

Public Functions

void **apply**(void **src_ptr*, umpire::util::*AllocationRecord* **src_allocation*, int *value*, 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*)
Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

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

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class CudaMemsetOperation

- Defined in file_umpire_op_CudaMemsetOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class `umpire::op::`**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`*:

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **ptr*, int *value*, std::size_t *length*, camp::resources::Resource &*ctx*)

void **transform**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, 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 destinatino memory location.
- `src_allocation`: AllocationRecord of source.
- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class GenericReallocateOperation

- Defined in file_umpire_op_GenericReallocateOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**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*)

Transfrom 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 destinatino memory location.
- `src_allocation`: AllocationRecord of source.
- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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`*:

camp::resources::Event **`apply_async`**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource *&ctx*)

Class HipCopyFromOperation

- Defined in file_umpire_op_HipCopyFromOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

`class` `umpire::`*`op`*`::`**`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*)

Transfrom 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 destinatino memory location.
- `src_allocation`: AllocationRecord of source.
- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

Exceptions

- *`util::Exception`*:

camp::resources::Event **`transform_async`**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource *&ctx*)

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`*:

camp::resources::Event **apply_async**(void *src_ptr, util::*AllocationRecord* *src_allocation, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class HipCopyOperation

- Defined in file_umpire_op_HipCopyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class `umpire::`*`op`*`::`**`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*)
Transfrom 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 destinatino memory location.
- `src_allocation`: AllocationRecord of source.
- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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*:

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class HipCopyToOperation

- Defined in file_umpire_op_HipCopyToOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**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*)

Transfrom 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 destinatino memory location.
- src_allocation: AllocationRecord of source.

- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

Exceptions

- *`util::Exception`*:

camp::resources::Event **`transform_async`**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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`*:

camp::resources::Event **`apply_async`**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class HipMemsetOperation

- Defined in file_umpire_op_HipMemsetOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

`class` `umpire::`*`op`*`::`**`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*)
Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

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

Exceptions

- *`util::Exception`*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class HostCopyOperation

- Defined in file_umpire_op_HostCopyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**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*)

Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

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

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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*:

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class HostMemsetOperation

- Defined in file_umpire_op_HostMemsetOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**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*)

Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

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

Exceptions

- *`util::Exception`*:

camp::resources::Event **transform_async**(void **src_ptr*, void ****dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class HostReallocateOperation

- Defined in file_umpire_op_HostReallocateOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class `umpire::op::`**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*)

Transfrom 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 destinatino memory location.
- `src_allocation`: AllocationRecord of source.
- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

Exceptions

- *`util::Exception`*:

camp::resources::Event **transform_async**(void **src_ptr*, void ****dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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`*:

camp::resources::Event **`apply_async`**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

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::CudaGetAttributeOperation< ATTRIBUTE >` (*Template Class CudaGetAttributeOperation*)
- `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::OpenMPTargetCopyOperation (*Class OpenMPTargetCopyOperation*)
- public umpire::op::OpenMPTargetMemsetOperation (*Class OpenMPTargetMemsetOperation*)
- public umpire::op::SyclCopyFromOperation (*Class SyclCopyFromOperation*)
- public umpire::op::SyclCopyOperation (*Class SyclCopyOperation*)
- public umpire::op::SyclCopyToOperation (*Class SyclCopyToOperation*)
- public umpire::op::SyclMemPrefetchOperation (*Class SyclMemPrefetchOperation*)
- public umpire::op::SyclMemsetOperation (*Class SyclMemsetOperation*)

Class Documentation

class umpire::*op*::**MemoryOperation**

Base class of an operation on memory.

Neither the transfrom 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::CudaGetAttributeOperation< ATTRIBUTE >*, *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::OpenMPTargetCopyOperation*, *umpire::op::OpenMPTargetMemsetOperation*, *umpire::op::SyclCopyFromOperation*, *umpire::op::SyclCopyOperation*, *umpire::op::SyclCopyToOperation*, *umpire::op::SyclMemPrefetchOperation*, *umpire::op::SyclMemsetOperation*

Public Functions

~MemoryOperation() = default

void **transform**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, 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 destinatino memory location.
- src_allocation: AllocationRecord of source.
- dst_allocation: AllocationRecord of destination.

- `length`: Number of bytes to transform.

Exceptions

- *`util::Exception`*:

camp::resources::Event **`transform_async`**(void **src_ptr*, void ****dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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`*:

camp::resources::Event **`apply_async`**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class MemoryOperationRegistry

- Defined in file_umpire_op_MemoryOperationRegistry.hpp

Class Documentation

`class` `umpire::op::`**`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.

std::shared_ptr<umpire::*op*::*MemoryOperation*> **find**(**const** std::string &*name*, std::pair<*Platform*, *Platform*> *platforms*)

void **registerOperation**(**const** std::string &*name*, std::pair<*Platform*, *Platform*> *platforms*, std::shared_ptr<*MemoryOperation*> &&*operation*) **noexcept**

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*&) = delete

MemoryOperationRegistry &**operator=**(**const** *MemoryOperationRegistry*&) = delete

~MemoryOperationRegistry() = default

Public Static Functions

MemoryOperationRegistry &**getInstance**() **noexcept**

Get the *MemoryOperationRegistry* singleton instance.

Protected Functions

MemoryOperationRegistry() **noexcept**

Class NumaMoveOperation

- Defined in file_umpire_op_NumaMoveOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**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 destinatino memory location.
- src_allocation: AllocationRecord of source.
- dst_allocation: AllocationRecord of destination.
- length: Number of bytes to transform.

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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*:

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class OpenMPTargetCopyOperation

- Defined in file_umpire_op_OpenMPTargetCopyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**OpenMPTargetCopyOperation** : **public** umpire::*op*::*MemoryOperation*

Public Functions

OpenMPTargetCopyOperation() = default

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

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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`*:

camp::resources::Event **`apply_async`**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource *&ctx*)

Class OpenMPTargetMemsetOperation

- Defined in file_umpire_op_OpenMPTargetMemsetOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

void **`apply`**(void **src_ptr*, umpire::util::*AllocationRecord* **src_allocation*, int *value*, 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*)
Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

- `src_ptr`: Pointer to source memory location.
- `dst_ptr`: Pointer to destinatino memory location.
- `src_allocation`: AllocationRecord of source.
- `dst_allocation`: AllocationRecord of destination.

- `length`: Number of bytes to transform.

Exceptions

- *`util::Exception`*:

camp::resources::Event **`transform_async`**(void **src_ptr*, void ****dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **`apply_async`**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class SyclCopyFromOperation

- Defined in file_umpire_op_SyclCopyFromOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Copy operation to move data between a Intel GPU and CPU memory.

Public Functions

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

Transfrom length bytes of memory from src_ptr to dst_ptr.

Uses SYCL memcpy to move data when src_ptr is on Intel GPU and dst_ptr is on CPU

Parameters

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

Exceptions

- *`util::Exception`*:

camp::resources::Event **`transform_async`**(void **src_ptr*, void ****dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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`*:

camp::resources::Event **apply_async**(void *src_ptr, util::*AllocationRecord* *src_allocation, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class SyclCopyOperation

- Defined in file_umpire_op_SyclCopyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class `umpire::`*`op`*`::SyclCopyOperation` : **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*)
Transfrom length bytes of memory from src_ptr to dst_ptr.

Uses DPCPP's USM memcpy to move data when both src_ptr and dst_ptr are on Intel GPUs.

Parameters

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

Exceptions

- *util::Exception*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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*:

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class SyclCopyToOperation

- Defined in file_umpire_op_SyclCopyToOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class `umpire::op::`**SyclCopyToOperation** : **public** umpire::*op*::*MemoryOperation*
Copy operation to move data between a Intel GPU and CPU memory.

Public Functions

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

Uses SYCL memcpy to move data when src_ptr on CPU to dst_ptr on Intel GPU

Parameters

- `src_ptr`: Pointer to source memory location.
- `dst_ptr`: Pointer to destinatino memory location.
- `src_allocation`: AllocationRecord of source.

- `dst_allocation`: AllocationRecord of destination.
- `length`: Number of bytes to transform.

Exceptions

- *`util::Exception`*:

camp::resources::Event **`transform_async`**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

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`*:

camp::resources::Event **`apply_async`**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class SyclMemPrefetchOperation

- Defined in file_umpire_op_SyclMemPrefetchOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

void **apply**(void **src_ptr*, umpire::util::*AllocationRecord* **src_allocation*, int *value*, 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*)
Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

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

Exceptions

- *`util::Exception`*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class SyclMemsetOperation

- Defined in file_umpire_op_SyclMemsetOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class umpire::*op*::**SyclMemsetOperation** : **public** umpire::*op*::*MemoryOperation*

Memset on Intel GPU 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 SYCL memset 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*)

Transfrom length bytes of memory from src_ptr to dst_ptr.

Parameters

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

Exceptions

- *`util::Exception`*:

camp::resources::Event **transform_async**(void **src_ptr*, void ***dst_ptr*, util::*AllocationRecord* **src_allocation*, util::*AllocationRecord* **dst_allocation*, std::size_t *length*, camp::resources::Resource &*ctx*)

camp::resources::Event **apply_async**(void **src_ptr*, util::*AllocationRecord* **src_allocation*, int *val*, std::size_t *length*, camp::resources::Resource &*ctx*)

Class Replay

- Defined in file_umpire_Replay.hpp

Class Documentation

class umpire::**Replay**

Public Functions

void **logMessage**(**const** std::string &*message*)

bool **replayLoggingEnabled**()

uint64_t **replayUid**()

Public Static Functions

Replay ***getReplayLogger**()

std::string **printReplayAllocator**(void)

template<typename **T**, typename ...**Args**>
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 umpire::resource::**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 noexcept**
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 noexcept**
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.

bool **isAccessibleFrom**(*Platform* _p_) **noexcept**

Platform **getPlatform**() **noexcept**
Get the Platform assocatiated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

Return Platform associated with this *MemoryResource*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getActualSize**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

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 CudaDeviceMemoryResource

- Defined in file_umpire_resource_CudaDeviceMemoryResource.hpp

Inheritance Relationships

Base Type

- `public umpire::resource::MemoryResource` (*Class MemoryResource*)

Class Documentation

class umpire::resource::**CudaDeviceMemoryResource** : **public** umpire::resource::*MemoryResource*

Concrete *MemoryResource* object that uses the template _allocator to allocate and deallocate memory.

Public Functions

CudaDeviceMemoryResource(*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 noexcept**

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 noexcept**

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.

bool **isAccessibleFrom**(*Platform p*) **noexcept**

Platform **getPlatform**() **noexcept**

Get the Platform assocatiated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

Return Platform associated with this *MemoryResource*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()

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

std::size_t **getActualSize**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

alloc::*CudaMallocAllocator* **m_allocator**

Platform **m_platform**

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

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, accesible 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 `umpire::resource::`**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*)

DefaultMemoryResource(*Platform platform*, **const** std::string &*name*, int *id*, *MemoryResource-Traits traits*, *_allocator alloc*)

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** **noexcept**
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 noexcept**
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.

bool **isAccessibleFrom**(*Platform p*) **noexcept**

Platform **getPlatform**() **noexcept**
Get the Platform assocatiated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

Return Platform associated with this *MemoryResource*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getActualSize**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

_allocator **m_allocator**

Platform **m_platform**

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class FileMemoryResource

- Defined in file_umpire_resource_FileMemoryResource.hpp

Inheritance Relationships

Base Type

- `public umpire::resource::MemoryResource` (*Class MemoryResource*)

Class Documentation

class `umpire::resource::`**FileMemoryResource** : **public** umpire::resource::*MemoryResource*

File Memory allocator.

This *FileMemoryResource* uses mmap to create a file mapping in order to use as additional memory. To create this mapping the function needs to take in the size of memory wanted for the allocation. The set location for the allocation by defult is ./ but can be assigned using enviroment variable "UMPIRE_MEMORY_FILE_DIR"

The return should be a pointer location. The same pointer location can be used for the deallocation. Deallocation uses munmap and removes the file associated with the pointer location.

Public Functions

FileMemoryResource(*Platform platform*, **const** std::string &*name*, int *id*, *MemoryResourceTraits traits*)

Construct a new *FileMemoryResource*.

Parameters

- `platform`: Platform of this instance of the *FileMemoryResource*.
- `name`: Name of this instance of the *FileMemoryResource*.
- `id`: Id of this instance of the *FileMemoryResource*.
- `traits`: Traits of this instance of the *FileMemoryResource*.

~FileMemoryResource()

Dallocates and removes all files created by the code meant for allocations.

void ***allocate**(std::size_t *bytes*)
Creates the allocation of size bytes using mmap.

Does the allocation as follows: 1) Find output file directory for mmap files using UMPIRE_MEMORY_FILE_DIR 2) Create name and create the file index using open 3) Setting Size Of Map File. Size is scaled to a page length on the system. 4) Truncate file using ftruncate64 5) Map file index with mmap 6) Store information about the allocated file into m_size_map

Return void* Since you are only reciving a pointer location of any size non spcaific to a type you will have to cast it to the desired type if needed.

Parameters

- `bytes`: The requested amount of bytes the user wants to use. Can not be zero or greater than avalable amount of bytes available.
- `UMPIRE_MEMORY_FILE_DIR`: Used to specify where memory is going to be allocated from

void **deallocate**(void **ptr*)
Deallocates file connected to the pointer.

Using m_size_map, the pointer is looked up and the file name and size can be returned. With this munmap can be called to deallocated the correct file.

Parameters

- `ptr`: Pointer location used to look up its information in m_size_map

std::size_t **getCurrentSize**() **const noexcept**
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 noexcept**
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.

bool **isAccessibleFrom**(*Platform p*) **noexcept**

Platform **getPlatform**() **noexcept**
Get the Platform assocatiated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

Return Platform associated with this *MemoryResource*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getActualSize**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Public Static Attributes

int **s_file_counter** = {0}

Protected Attributes

Platform **m_platform**

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class FileMemoryResourceFactory

- Defined in file_umpire_resource_FileMemoryResourceFactory.hpp

Inheritance Relationships

Base Type

- `public umpire::resource::MemoryResourceFactory` (*Class MemoryResourceFactory*)

Class Documentation

class FileMemoryResourceFactory : **public** umpire::resource::*MemoryResourceFactory*
Factory class to construct a *MemoryResource*.

Class HipConstantMemoryResource

- Defined in file_umpire_resource_HipConstantMemoryResource.hpp

Inheritance Relationships

Base Type

- `public umpire::resource::MemoryResource` (*Class MemoryResource*)

Class Documentation

class `umpire::resource::`**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 noexcept**
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 noexcept**
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.

bool **isAccessibleFrom**(*Platform p*) **noexcept**

Platform **getPlatform**() **noexcept**
Get the Platform assocatiated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

Return Platform associated with this *MemoryResource*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getActualSize**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

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 HipDeviceMemoryResource

- Defined in file_umpire_resource_HipDeviceMemoryResource.hpp

Inheritance Relationships

Base Type

- `public umpire::resource::MemoryResource` (*Class MemoryResource*)

Class Documentation

class `umpire::resource::`**HipDeviceMemoryResource** : **public** umpire::resource::*MemoryResource*
Concrete *MemoryResource* object that uses the template _allocator to allocate and deallocate memory.

Public Functions

HipDeviceMemoryResource(*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 noexcept**
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 noexcept**
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.

bool **isAccessibleFrom**(*Platform p*) **noexcept**

Platform **getPlatform**() **noexcept**
Get the Platform assocatiated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

Return Platform associated with this *MemoryResource*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getActualSize**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

alloc::*HipMallocAllocator* **m_allocator**

Platform **m_platform**

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

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 HipUnifiedMemoryResourceFactory

- Defined in file_umpire_resource_HipUnifiedMemoryResourceFactory.hpp

Inheritance Relationships

Base Type

- `public umpire::resource::MemoryResourceFactory` (*Class MemoryResourceFactory*)

Class Documentation

class HipUnifiedMemoryResourceFactory : **public** umpire::resource::*MemoryResourceFactory*
Factory class to construct a *MemoryResource* that uses AMD "unified" memory, accesible from both the CPU and AMD GPUs.

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::CudaDeviceMemoryResource` (*Class CudaDeviceMemoryResource*)
- `public umpire::resource::DefaultMemoryResource< _allocator >` (*Template Class DefaultMemoryResource*)
- `public umpire::resource::FileMemoryResource` (*Class FileMemoryResource*)
- `public umpire::resource::HipConstantMemoryResource` (*Class HipConstantMemoryResource*)
- `public umpire::resource::HipDeviceMemoryResource` (*Class HipDeviceMemoryResource*)
- `public umpire::resource::NullMemoryResource` (*Class NullMemoryResource*)
- `public umpire::resource::SyclDeviceMemoryResource< _allocator >` (*Template Class SyclDeviceMemoryResource*)

Class Documentation

class umpire::resource::**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::CudaDeviceMemoryResource*, *umpire::resource::DefaultMemoryResource< _allocator >*, *umpire::resource::FileMemoryResource*, *umpire::resource::HipConstantMemoryResource*, *umpire::resource::HipDeviceMemoryResource*, *umpire::resource::NullMemoryResource*, *umpire::resource::SyclDeviceMemoryResource< _allocator >*

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).

~MemoryResource() = default

void ***allocate**(std::size_t *bytes*) **override** = 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.

void **deallocate**(void **ptr*) **override** = 0

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 noexcept override** = 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*.

std::size_t **getHighWatermark**() **const noexcept override** = 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.

Platform **getPlatform**() **noexcept override** = 0
Get the Platform assocatiated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

Return Platform associated with this *MemoryResource*.

bool **isAccessibleFrom**(*Platform p*) **noexcept** = 0

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getActualSize**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

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::FileMemoryResourceFactory` (*Class FileMemoryResourceFactory*)
- `public umpire::resource::HipConstantMemoryResourceFactory` (*Class HipConstantMemoryResourceFactory*)
- `public umpire::resource::HipDeviceResourceFactory` (*Class HipDeviceResourceFactory*)
- `public umpire::resource::HipPinnedMemoryResourceFactory` (*Class HipPinnedMemoryResourceFactory*)
- `public umpire::resource::HipUnifiedMemoryResourceFactory` (*Class HipUnifiedMemoryResourceFactory*)
- `public umpire::resource::HostResourceFactory` (*Class HostResourceFactory*)
- `public umpire::resource::NullMemoryResourceFactory` (*Class NullMemoryResourceFactory*)
- `public umpire::resource::OpenMPTargetResourceFactory` (*Class OpenMPTargetResourceFactory*)
- `public umpire::resource::SyclDeviceResourceFactory` (*Class SyclDeviceResourceFactory*)
- `public umpire::resource::SyclPinnedMemoryResourceFactory` (*Class SyclPinnedMemoryResourceFactory*)
- `public umpire::resource::SyclUnifiedMemoryResourceFactory` (*Class SyclUnifiedMemoryResourceFactory*)

Class Documentation

class umpire::resource::**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::FileMemoryResourceFactory*, *umpire::resource::HipConstantMemoryResourceFactory*, *umpire::resource::HipDeviceResourceFactory*, *umpire::resource::HipPinnedMemoryResourceFactory*, *umpire::resource::HipUnifiedMemoryResourceFactory*, *umpire::resource::HostResourceFactory*, *umpire::resource::NullMemoryResourceFactory*, *umpire::resource::OpenMPTargetResourceFactory*, *umpire::resource::SyclDeviceResourceFactory*, *umpire::resource::SyclPinnedMemoryResourceFactory*, *umpire::resource::SyclUnifiedMemoryResourceFactory*

Public Functions

~MemoryResourceFactory() = default

bool **isValidMemoryResourceFor**(**const** std::string &*name*) **noexcept** = 0

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*.
- `traits`: Traits for the *MemoryResource*

std::unique_ptr<resource::*MemoryResource*> **create**(**const** std::string &*name*, int *id*, *MemoryResourceTraits* *traits*) = 0

Construct a *MemoryResource* with the given name and id.

Parameters

- `name`: Name of the *MemoryResource*.
- `id`: ID of the *MemoryResource*.
- `traits`: Traits for the *MemoryResource*

MemoryResourceTraits **getDefaultTraits**() = 0

Class MemoryResourceRegistry

- Defined in file_umpire_resource_MemoryResourceRegistry.hpp

Class Documentation

class umpire::resource::**MemoryResourceRegistry**

Public Functions

const std::vector<std::string> &**getResourceNames**() **noexcept**

std::unique_ptr<resource::*MemoryResource*> **makeMemoryResource**(**const** std::string &*name*, int *id*)

std::unique_ptr<resource::*MemoryResource*> **makeMemoryResource**(**const** std::string &*name*, int *id*, *MemoryResourceTraits traits*)

void **registerMemoryResource**(std::unique_ptr<*MemoryResourceFactory*> &&*factory*)

MemoryResourceTraits **getDefaultTraitsForResource**(**const** std::string &*name*)

MemoryResourceRegistry(**const** *MemoryResourceRegistry*&) = delete

MemoryResourceRegistry &**operator=**(**const** *MemoryResourceRegistry*&) = delete

~MemoryResourceRegistry() = default

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 umpire::resource::**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 noexcept**
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 noexcept**
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.

bool **isAccessibleFrom**(*Platform p*) **noexcept**

Platform **getPlatform**() **noexcept**
Get the Platform assocatiated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

Return Platform associated with this *MemoryResource*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getActualSize**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

Platform **m_platform**

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

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 OpenMPTargetResourceFactory

- Defined in file_umpire_resource_OpenMPTargetMemoryResourceFactory.hpp

Inheritance Relationships

Base Type

- `public umpire::resource::MemoryResourceFactory` (*Class MemoryResourceFactory*)

Class Documentation

class `umpire::resource::`**OpenMPTargetResourceFactory** : **public** umpire::resource::*MemoryResourceFactory*
Factory class for constructing *MemoryResource* objects that use GPU memory.

Public Functions

bool **isValidMemoryResourceFor**(**const** std::string &*name*) **noexcept final override**

std::unique_ptr<resource::*MemoryResource*> **create**(**const** std::string &*name*, int *id*) **final override**
Construct a *MemoryResource* with the given name and id.

Parameters

- `name`: Name of the *MemoryResource*.
- `id`: ID of the *MemoryResource*.
- `traits`: Traits for the *MemoryResource*

std::unique_ptr<resource::*MemoryResource*> **create**(**const** std::string &*name*, int *id*, *MemoryResourceTraits* *traits*) **final override**
Construct a *MemoryResource* with the given name and id.

Parameters

- `name`: Name of the *MemoryResource*.
- `id`: ID of the *MemoryResource*.
- `traits`: Traits for the *MemoryResource*

MemoryResourceTraits **getDefaultTraits**() **final override**

Template Class SyclDeviceMemoryResource

- Defined in file_umpire_resource_SyclDeviceMemoryResource.hpp

Inheritance Relationships

Base Type

- `public umpire::resource::MemoryResource` (*Class MemoryResource*)

Class Documentation

template<typename **_allocator**>
class `umpire::resource::` **SyclDeviceMemoryResource** : **public** umpire::resource::*MemoryResource*
Concrete *MemoryResource* object that uses the template _allocator to allocate and deallocate memory.

Public Functions

SyclDeviceMemoryResource(*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 noexcept**
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 noexcept**
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.

bool **isAccessibleFrom**(*Platform* *p*) **noexcept**

Platform **getPlatform**() **noexcept**
Get the Platform assocatiated with this *MemoryResource*.

This function is pure virtual and must be implemented by the inheriting class.

Return Platform associated with this *MemoryResource*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getActualSize**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

_allocator **m_allocator**

Platform **m_platform**

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class SyclDeviceResourceFactory

- Defined in file_umpire_resource_SyclDeviceResourceFactory.hpp

Inheritance Relationships

Base Type

- `public umpire::resource::MemoryResourceFactory` (*Class MemoryResourceFactory*)

Class Documentation

`class SyclDeviceResourceFactory` : **`public`** umpire::resource::*MemoryResourceFactory*
Factory class for constructing *MemoryResource* objects that use Intel's GPU memory.

Class SyclPinnedMemoryResourceFactory

- Defined in file_umpire_resource_SyclPinnedMemoryResourceFactory.hpp

Inheritance Relationships

Base Type

- `public umpire::resource::MemoryResourceFactory` (*Class MemoryResourceFactory*)

Class Documentation

`class SyclPinnedMemoryResourceFactory` : **`public`** umpire::resource::*MemoryResourceFactory*

Class SyclUnifiedMemoryResourceFactory

- Defined in file_umpire_resource_SyclUnifiedMemoryResourceFactory.hpp

Inheritance Relationships

Base Type

- `public umpire::resource::MemoryResourceFactory` (*Class MemoryResourceFactory*)

Class Documentation

class SyclUnifiedMemoryResourceFactory : **public** umpire::resource::*MemoryResourceFactory*

Factory class to construct a *MemoryResource* that uses Intel's "unified shared" memory (USM), accesible from both the CPU and Intel GPUs.

Class ResourceManager

- Defined in file_umpire_ResourceManager.hpp

Class Documentation

class umpire::**ResourceManager**

Public Functions

void **initialize**()

Initialize the *ResourceManager*.

This will create all registered MemoryResource objects

std::vector<std::string> **getAllocatorNames**() **const noexcept**

Get the names of all available *Allocator* objects.

std::vector<int> **getAllocatorIds**() **const noexcept**

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*.

std::vector<std::string> **getResourceNames**()

Get the names for existing Resources.

The Memory Resource Registry dynamically populates available memory resource types based on what's available. This function returns those names so they can be used to determine allocator accessibility.

Return The available resource names.

void **setDefaultAllocator**(*Allocator allocator*) **noexcept**
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*.

Allocator **makeResource**(**const** std::string &*name*)

Allocator **makeResource**(**const** std::string &*name*, *MemoryResourceTraits traits*)

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.

void **addAlias**(**const** std::string &*name*, *Allocator allocator*)
Add an *Allocator* alias.

After this call, allocator can be retrieved by calling getAllocator(name).

The same *Allocator* can have multiple aliases.

Parameters

- `name`: Name to alias *Allocator* with.
- `allocator`: *Allocator* to register.

void **removeAlias**(**const** std::string &*name*, *Allocator allocator*)
Remove an *Allocator* alias.

After calling, allocator can no longer be accessed by calling getAllocator(name). If allocator is not registered under name, an error will be thrown.

If one of the default resource names (e.g. HOST) is used, an error will be thrown.

Parameters

- `name`: Name to deregister *Allocator* with.
- `allocator`: *Allocator* to deregister.

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*) **noexcept**

bool **isAllocator**(int *id*) **noexcept**

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.

camp::resources::Event **copy**(void **dst_ptr*, void **src_ptr*, camp::resources::Resource &*ctx*, std::size_t *size* = 0)

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*)

int **getNumDevices**() **const**

~ResourceManager()

ResourceManager(**const** *ResourceManager*&) = delete

ResourceManager &**operator=**(**const** *ResourceManager*&) = delete

Public Static Functions

ResourceManager &**getInstance**()

Class AlignedAllocator

- Defined in file_umpire_strategy_AlignedAllocator.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

class umpire::strategy::**AlignedAllocator** : **public** umpire::strategy::*AllocationStrategy*

Public Functions

AlignedAllocator(**const** std::string &*name*, int *id*, *Allocator* *allocator*, std::size_t *alignment* = 16)

void ***allocate**(std::size_t *bytes*) **override**
Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void **ptr*) **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

Platform **getPlatform**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getCurrentSize**() **const noexcept**
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 noexcept**
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 noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

strategy::*AllocationStrategy* ***m_allocator**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class AllocationAdvisor

- Defined in file_umpire_strategy_AllocationAdvisor.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

class `umpire::strategy::`**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*) **override**

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void **ptr*) **override**

Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

Platform **getPlatform**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getCurrentSize**() **const noexcept**
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 noexcept**
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 noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class AllocationPrefetcher

- Defined in file_umpire_strategy_AllocationPrefetcher.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

class `umpire::strategy::`**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*) **override**

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void **ptr*) **override**

Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

Platform **getPlatform**() **noexcept override**

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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getCurrentSize**() **const noexcept**
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 noexcept**
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 noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class AllocationStrategy

- Defined in file_umpire_strategy_AllocationStrategy.hpp

Inheritance Relationships

Derived Types

- `public umpire::resource::MemoryResource` (*Class MemoryResource*)
- `public umpire::strategy::AlignedAllocator` (*Class AlignedAllocator*)
- `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::QuickPool` (*Class QuickPool*)
- `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 `umpire::strategy::`**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::AlignedAllocator*, *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::QuickPool*, *umpire::strategy::SizeLimiter*, *umpire::strategy::SlotPool*, *umpire::strategy::ThreadSafeAllocator*, *umpire::strategy::ZeroByteHandler*

Public Functions

AllocationStrategy(**const** std::string &*name*, int *id*, *AllocationStrategy* **parent*) **noexcept**
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.

~AllocationStrategy() = default

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*) = 0
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 noexcept**
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 noexcept**
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 noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

Platform **getPlatform**() **noexcept** = 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**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

MemoryResourceTraits **getTraits**() **const noexcept**

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Friends

friend 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 `umpire::strategy::`**AllocationTracker** : **public** umpire::strategy::*AllocationStrategy*, **private** umpire::strate

Public Functions

AllocationTracker(std::unique_ptr<*AllocationStrategy*> &&*allocator*) **noexcept**

void ***allocate**(std::size_t *bytes*) **override**
Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void **ptr*) **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

void **release**() **override**
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getCurrentSize**() **const noexcept override**
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 noexcept override**
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 noexcept override**
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 **getAllocationCount**() **const noexcept override**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

Platform **getPlatform**() **noexcept override**
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**()

MemoryResourceTraits **getTraits**() **const noexcept override**

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

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**

std::size_t **m_allocation_count**

Class DynamicPoolList

- Defined in file_umpire_strategy_DynamicPoolList.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

class `umpire::strategy::`**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*&)>

Public Functions

DynamicPoolList(**const** std::string &*name*, int *id*, *Allocator allocator*, **const** std::size_t *first_minimum_pool_allocation_size* = (512 * 1024 * 1024), **const** std::size_t *next_minimum_pool_allocation_size* = (1 * 1024 * 1024), **const** std::size_t *alignment* = 16, *CoalesceHeuristic should_coalesce* = *percent_releasable*(100)) **noexcept**

Construct a new *DynamicPoolList*.

Parameters

- `name`: Name of this instance of the *DynamicPoolList*.
- `id`: Id of this instance of the *DynamicPoolList*.
- `allocator`: Allocation resource that pool uses
- `first_minimum_pool_allocation_size`: Minimum size the pool initially allocates
- `next_minimum_pool_allocation_size`: The minimum size of all future allocations.
- `align_bytes`: Number of bytes with which to align allocation sizes (power-of-2)

- `do_heuristic`: Heuristic for when to perform coalesce operation

DynamicPoolList(**const** *DynamicPoolList*&) = delete

void ***allocate**(size_t *bytes*) **override**

void **deallocate**(void **ptr*) **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

void **release**() **override**
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getActualSize**() **const noexcept override**
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 **getCurrentSize**() **const noexcept override**
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.

Platform **getPlatform**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept final override**

std::size_t **getReleasableSize**() **const noexcept**
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 noexcept**
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 noexcept**
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**() **noexcept**

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.

std::size_t **getHighWatermark**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Public Static Functions

CoalesceHeuristic **percent_releasable**(int *percentage*)

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class DynamicPoolMap

- Defined in file_umpire_strategy_DynamicPoolMap.hpp

Inheritance Relationships

Base Types

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)
- `private umpire::strategy::mixins::AlignedAllocation` (*Class AlignedAllocation*)

Class Documentation

class `umpire::strategy::`**DynamicPoolMap** : **public** umpire::strategy::*AllocationStrategy*, **private** umpire::strategy::

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*&)>

Public Functions

DynamicPoolMap(**const** std::string &*name*, int *id*, *Allocator allocator*, **const** std::size_t *first_minimum_pool_allocation_size* = (512 * 1024 * 1024), **const** std::size_t *min_alloc_size* = (1 * 1024 * 1024), **const** std::size_t *align_bytes* = 16, *CoalesceHeuristic should_coalesce* = *percent_releasable*(100)) **noexcept**

Construct a new *DynamicPoolMap*.

Parameters

- `name`: Name of this instance of the *DynamicPoolMap*
- `id`: Unique identifier for this instance
- `allocator`: Allocation resource that pool uses
- `first_minimum_pool_allocation_size`: Size the pool initially allocates
- `next_minimum_pool_allocation_size`: The minimum size of all future allocations
- `align_bytes`: Number of bytes with which to align allocation sizes (power-of-2)
- `should_coalesce`: Heuristic for when to perform coalesce operation

~DynamicPoolMap()

DynamicPoolMap(**const** *DynamicPoolMap*&) = delete

void *__allocate__(std::size_t *bytes*) __override__
Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void __deallocate__(void **ptr*) __override__
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

void __release__() __override__
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t __getActualSize__() __const noexcept override__
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 __getCurrentSize__() __const noexcept override__
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.

Platform __getPlatform__() __noexcept override__
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*.

MemoryResourceTraits __getTraits__() __const noexcept override__

std::size_t __getReleasableSize__() __const noexcept__
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 noexcept__
Return the number of free memory blocks that the pools holds.

std::size_t __getInUseBlocks__() __const noexcept__
Return the number of used memory blocks that the pools holds.

std::size_t **getBlocksInPool**() **const noexcept**
Return the number of memory blocks both leased to application and internal free memory that the pool holds.

std::size_t **getLargestAvailableBlock**() **noexcept**
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.

std::size_t **getHighWatermark**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Public Static Functions

CoalesceHeuristic **percent_releasable**(int *percentage*)

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Private Functions

std::size_t **aligned_round_up**(std::size_t *size*)
Round up the size to be an integral multple of configured alignment.

Return Size rounded up to be integral multiple of configured alignment

void ***aligned_allocate**(**const** std::size_t *size*)
Return an allocation of `size` bytes that is aligned on the configured alignment boundary.

void **aligned_deallocate**(void **ptr*)
Deallocate previously alligned allocation.

Private Members

strategy::*AllocationStrategy* ***m_allocator**

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 `umpire::strategy::`**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) **noexcept**
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*&) = delete

void ***allocate**(std::size_t *bytes* = 0) **final** **override**
Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void **ptr*) **final** **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const** **noexcept** **final** **override**
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** **noexcept** **final** **override**
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** **noexcept** **final** **override**
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**() **noexcept final override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept final override**

bool **pointerIsFromPool**(void **ptr*) **const noexcept**

std::size_t **numPools**() **const noexcept**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class MixedPool

- Defined in file_umpire_strategy_MixedPool.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

class `umpire::strategy::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 << LastFixed) bytes.

Public Functions

MixedPool(**const** std::string &*name*, int *id*, *Allocator* *allocator*, std::size_t *smallest_fixed_obj_size* = (1 << 8), std::size_t *largest_fixed_obj_size* = (1 << 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* *should_coalesce* = *DynamicPoolMap*::*percent_releasable*(100)) **noexcept**

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)
- `should_coalesce`: Heuristic callback function (for the dynamic pool)

void ***allocate**(std::size_t *bytes*) **override**

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void *_ptr_) **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

void **release**() **override**
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getCurrentSize**() **const noexcept override**
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 noexcept override**
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 noexcept override**
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**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

std::size_t **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent*** () **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class AlignedAllocation

- Defined in file_umpire_strategy_mixins_AlignedAllocation.hpp

Inheritance Relationships

Derived Types

- `private DynamicSizePool< IA >` (*Template Class DynamicSizePool*)
- `private umpire::strategy::DynamicPoolMap` (*Class DynamicPoolMap*)
- `private umpire::strategy::QuickPool` (*Class QuickPool*)

Class Documentation

class `umpire::strategy::mixins::`**AlignedAllocation**
Subclassed by *DynamicSizePool< IA >*, *umpire::strategy::DynamicPoolMap*, *umpire::strategy::QuickPool*

Public Functions

AlignedAllocation() = delete

AlignedAllocation(std::size_t *alignment*, *strategy*::AllocationStrategy **strategy*)

std::size_t **aligned_round_up**(std::size_t *size*)
Round up the size to be an integral multple of configured alignment.

Return Size rounded up to be integral multiple of configured alignment

void ***aligned_allocate**(**const** std::size_t *size*)
Return an allocation of `size` bytes that is aligned on the configured alignment boundary.

void **aligned_deallocate**(void **ptr*)
Deallocate previously alligned allocation.

Protected Attributes

strategy::*AllocationStrategy* ***m_allocator**

Class Inspector

- Defined in file_umpire_strategy_mixins_Inspector.hpp

Inheritance Relationships

Derived Type

- `private umpire::strategy::AllocationTracker` (*Class AllocationTracker*)

Class Documentation

class `umpire::strategy::mixins::`**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**

std::size_t **m_allocation_count**

Class MonotonicAllocationStrategy

- Defined in file_umpire_strategy_MonotonicAllocationStrategy.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

class umpire::strategy::**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*) **override**
Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void **ptr*) **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const noexcept override**
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 noexcept override**
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**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getActualSize**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class NamedAllocationStrategy

- Defined in file_umpire_strategy_NamedAllocationStrategy.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

class `umpire::strategy::`**NamedAllocationStrategy** : **public** umpire::strategy::*AllocationStrategy*

Public Functions

NamedAllocationStrategy(**const** std::string &*name*, int *id*, *Allocator* *allocator*)

void ***allocate**(std::size_t *bytes*) **override**
Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void **ptr*) **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

Platform **getPlatform**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getCurrentSize**() **const noexcept**
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 noexcept**
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 noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

strategy::*AllocationStrategy* ***m_allocator**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class NumaPolicy

- Defined in file_umpire_strategy_NumaPolicy.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

class `umpire::strategy::`**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*) **override**
Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void **ptr*) **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

Platform **getPlatform**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

int **getNode**() **const noexcept**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getCurrentSize**() **const noexcept**
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 noexcept**
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 noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class QuickPool

- Defined in file_umpire_strategy_QuickPool.hpp

Nested Relationships

Nested Types

- *Struct QuickPool::Chunk*
- *Template Class QuickPool::pool_allocator*

Inheritance Relationships

Base Types

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)
- `private umpire::strategy::mixins::AlignedAllocation` (*Class AlignedAllocation*)

Class Documentation

class umpire::strategy::**QuickPool** : **public** umpire::strategy::*AllocationStrategy*, **private** umpire::strategy::mixins::

Public Types

using Pointer = void*

using CoalesceHeuristic = std::function<bool (**const** strategy::*QuickPool*&)>

Public Functions

QuickPool(**const** std::string &*name*, int *id*, *Allocator allocator*, **const** std::size_t *first_minimum_pool_allocation_size* = (512 * 1024 * 1024), **const** std::size_t *next_minimum_pool_allocation_size* = (1 * 1024 * 1024), **const** std::size_t *alignment* = 16, *CoalesceHeuristic should_coalesce* = *percent_releasable*(100)) **noexcept**

Construct a new *QuickPool*.

Parameters

- `name`: Name of this instance of the *QuickPool*
- `id`: Unique identifier for this instance
- `allocator`: Allocation resource that pool uses
- `first_minimum_pool_allocation_size`: Size the pool initially allocates
- `next_minimum_pool_allocation_size`: The minimum size of all future allocations
- `alignment`: Number of bytes with which to align allocation sizes (power-of-2)
- `should_coalesce`: Heuristic for when to perform coalesce operation

~QuickPool()

QuickPool(**const** *QuickPool*&) = delete

void ***allocate**(std::size_t *bytes*) **override**

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void **ptr*) **override**

Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

void **release**() **override**

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

std::size_t **getActualSize**() **const noexcept override**
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 **getReleasableSize**() **const noexcept**

Platform **getPlatform**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

std::size_t **getBlocksInPool**() **const noexcept**
Return the number of memory blocks both leased to application and internal free memory that the pool holds.

std::size_t **getLargestAvailableBlock**() **noexcept**
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**() **noexcept**

void **do_coalesce**() **noexcept**

std::size_t **getCurrentSize**() **const noexcept**
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 noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Public Static Functions

CoalesceHeuristic **percent_releasable**(int *percentage*)

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Private Functions

std::size_t **aligned_round_up**(std::size_t *size*)
Round up the size to be an integral multple of configured alignment.

Return Size rounded up to be integral multiple of configured alignment

void ***aligned_allocate**(**const** std::size_t *size*)
Return an allocation of `size` bytes that is aligned on the configured alignment boundary.

void **aligned_deallocate**(void **ptr*)
Deallocate previously alligned allocation.

Private Members

strategy::*AllocationStrategy* ***m_allocator**

Template Class QuickPool::pool_allocator

- Defined in file_umpire_strategy_QuickPool.hpp

Nested Relationships

This class is a nested type of *Class QuickPool*.

Class Documentation

template<typename **Value**>
class umpire::strategy::*QuickPool*::**pool_allocator**

Public Types

using value_type = *Value*

using size_type = std::size_t

using difference_type = std::ptrdiff_t

Public Functions

pool_allocator()

template<typename **U**>
pool_allocator(**const** *pool_allocator*<*U*> &*other*)
BUG: Only required for MSVC.

Value ***allocate**(std::size_t *n*)

void **deallocate**(*Value* **data*, std::size_t)

Public Members

util::*FixedMallocPool* **pool**

Class SizeLimiter

- Defined in file_umpire_strategy_SizeLimiter.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

class umpire::strategy::**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*) **override**
Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void **ptr*) **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

Platform **getPlatform**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getCurrentSize**() **const noexcept**
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 noexcept**
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 noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class SlotPool

- Defined in file_umpire_strategy_SlotPool.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

class `umpire::strategy::`**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*) **override**
Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void *_ptr_) **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const noexcept override**
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 noexcept override**
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**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getActualSize**() **const noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent** () **const noexcept***
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class ThreadSafeAllocator

- Defined in file_umpire_strategy_ThreadSafeAllocator.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

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

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

Public Functions

ThreadSafeAllocator (**const** std::string &*name*, int *id*, *Allocator allocator*)

void ***allocate** (std::size_t *bytes*) **override***
Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate** (void **ptr*) **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

Platform **getPlatform**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

void **release**()
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getCurrentSize**() **const noexcept**
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 noexcept**
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 noexcept**
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 **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

strategy::*AllocationStrategy* ***m_allocator**

std::mutex **m_mutex**

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Class ZeroByteHandler

- Defined in file_umpire_strategy_ZeroByteHandler.hpp

Inheritance Relationships

Base Type

- `public umpire::strategy::AllocationStrategy` (*Class AllocationStrategy*)

Class Documentation

class `umpire::strategy::`**ZeroByteHandler** : **public** umpire::strategy::*AllocationStrategy*

Public Functions

ZeroByteHandler(std::unique_ptr<*AllocationStrategy*> &&*allocator*) **noexcept**

void ***allocate**(std::size_t *bytes*) **override**
Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

void **deallocate**(void **ptr*) **override**
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

void **release**() **override**
Release any and all unused memory held by this *AllocationStrategy*.

std::size_t **getCurrentSize**() **const noexcept override**
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 noexcept override**
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 noexcept override**
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**() **noexcept override**
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*.

MemoryResourceTraits **getTraits**() **const noexcept override**

strategy::*AllocationStrategy* ***getAllocationStrategy**()

std::size_t **getAllocationCount**() **const noexcept**
Get the total number of active allocations by this allocator.

Return The total number of active allocations this object has allocated.

const std::string &**getName**() **noexcept**
Get the name of this *AllocationStrategy*.

Return The name of this *AllocationStrategy*.

int **getId**() **noexcept**
Get the id of this *AllocationStrategy*.

Return The id of this *AllocationStrategy*.

AllocationStrategy ***getParent**() **const noexcept**
Traces where the allocator came from.

Return Pointer to the parent *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

AllocationStrategy ***m_parent**

Template Class TypedAllocator

- Defined in file_umpire_TypedAllocator.hpp

Class Documentation

template<typename **T**>
class umpire::**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 class 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 umpire::util::**AllocationMap**

Public Types

using Map = *MemoryMap*<*RecordList*>

Public Functions

AllocationMap()

AllocationMap(**const** *AllocationMap*&) = delete

void **insert**(void **ptr*, *AllocationRecord record*)

const *AllocationRecord* ***find**(void **ptr*) **const**

AllocationRecord ***find**(void **ptr*)

const *AllocationRecord* ***findRecord**(void **ptr*) **const noexcept**

AllocationRecord ***findRecord**(void **ptr*) **noexcept**

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**

ConstIterator **begin**() **const**

ConstIterator **end**() **const**

class ConstIterator

Public Types

using iterator_category = std::forward_iterator_tag

using value_type = *AllocationRecord*

using difference_type = std::ptrdiff_t

using pointer = *value_type**

using reference = *value_type*&

Public Functions

ConstIterator(**const** *AllocationMap* **map*, *iterator_begin*)

ConstIterator(**const** *AllocationMap* **map*, *iterator_end*)

ConstIterator(**const** *ConstIterator*&) = default

const *AllocationRecord* &**operator***()

const *AllocationRecord* ***operator->**()

ConstIterator &**operator++**()

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*.

Class Documentation

class umpire::util::*AllocationMap*::**ConstIterator**

Public Types

using iterator_category = std::forward_iterator_tag

using value_type = *AllocationRecord*

using difference_type = std::ptrdiff_t

using pointer = *value_type**

using reference = *value_type*&

Public Functions

ConstIterator(**const** *AllocationMap* **map*, *iterator_begin*)

ConstIterator(**const** *AllocationMap* **map*, *iterator_end*)

ConstIterator(**const** *ConstIterator*&) = default

const *AllocationRecord* &**operator***()

const *AllocationRecord* ***operator->**()

ConstIterator &**operator++**()

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 umpire::util::*AllocationMap*::**RecordList**

Public Types

using RecordBlock = *Block*<*AllocationRecord*>

Public Functions

RecordList(*AllocationMap* &*map*, *AllocationRecord* *record*)

~RecordList()

void **push_back**(**const** *AllocationRecord* &*rec*)

AllocationRecord **pop_back**()

ConstIterator **begin**() **const**

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 Types

using iterator_category = std::forward_iterator_tag

using value_type = *AllocationRecord*

using difference_type = std::ptrdiff_t

using pointer = *value_type**

using reference = *value_type*&

Public Functions

ConstIterator()

ConstIterator(**const** *RecordList* **list*, *iterator_begin*)

ConstIterator(**const** *RecordList* **list*, *iterator_end*)

ConstIterator(**const** *ConstIterator*&) = default

const *AllocationRecord* &**operator***()

const *AllocationRecord* ***operator->**()

ConstIterator &**operator++**()

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*.

Class Documentation

class umpire::util::*AllocationMap*::*RecordList*::**ConstIterator**

Public Types

using iterator_category = std::forward_iterator_tag

using value_type = *AllocationRecord*

using difference_type = std::ptrdiff_t

using pointer = *value_type**

using reference = *value_type*&

Public Functions

ConstIterator()

ConstIterator(**const** *RecordList* **list*, *iterator_begin*)

ConstIterator(**const** *RecordList* **list*, *iterator_end*)

ConstIterator(**const** *ConstIterator*&) = default

const *AllocationRecord* &**operator***()

const *AllocationRecord* ***operator->**()

ConstIterator &**operator++**()

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 umpire::util::**Exception** : **public** std::exception

Public Functions

Exception(**const** std::string &*msg*, **const** std::string &*file*, int *line*)

~Exception() = default

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 umpire::util::**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 noexcept**

Class Logger

- Defined in file_umpire_util_Logger.hpp

Class Documentation

class umpire::util::**Logger**

Public Functions

void **setLoggingMsgLevel**(message::*Level level*) **noexcept**

void **logMessage**(*message*::Level *level*, **const** std::string &*message*, **const** std::string &*fileName*, int *line*) **noexcept**

bool **logLevelEnabled**(message::*Level level*)

~Logger() **noexcept** = default

Logger(**const** *Logger*&) = delete

Logger &**operator=**(**const** *Logger*&) = delete

Public Static Functions

void **initialize**()

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 umpire::util::**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

using Key = void*

using Value = *V*

using KeyValuePair = std::pair<*Key*, *Value**>

using Iterator = *Iterator_*<false>

using ConstIterator = *Iterator_*<true>

Public Functions

MemoryMap()

~MemoryMap()

MemoryMap(**const** *MemoryMap*&) = delete

std::pair<*Iterator*, bool> **insert**(*Key ptr*, **const** *Value* &*val*) **noexcept**

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*) **noexcept**

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*) **noexcept**

Emplaces a new value at ptr in the map, forwarding args to the placement new constructor.

Return See alternative version.

Iterator **findOrBefore**(*Key ptr*) **noexcept**

Find a value at ptr.

Return iterator into map at ptr or preceeding position.

ConstIterator **findOrBefore**(*Key ptr*) **const noexcept**

Iterator **find**(*Key ptr*) **noexcept**
Find a value at ptr.

Return iterator into map at ptr or *end()* if not found.

ConstIterator **find**(*Key ptr*) **const noexcept**

ConstIterator **begin**() **const**
Iterator to first value or *end()* if empty.

Iterator **begin**()

ConstIterator **end**() **const**
Iterator to one-past-last value.

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**() **noexcept**
Clear all entris from the map.

std::size_t **size**() **const noexcept**
Return number of entries in the map.

template<typename ...**Args**>
std::pair<**typename** *MemoryMap*<*V*>::Iterator, bool> **doInsert**(*Key ptr*, *Args*&&... *args*) **noexcept**

template<typename **P**>
std::pair<**typename** *MemoryMap*<*V*>::Iterator, bool> **insert**(*P* &&*pair*) **noexcept**

template<typename ...**Args**>
std::pair<**typename** *MemoryMap*<*V*>::Iterator, bool> **insert**(*Key ptr*, *Args*&&... *args*) **noexcept**

Friends

friend class Iterator_

template<bool **Const** = false>
class Iterator_

Public Types

using iterator_category = std::forward_iterator_tag

using value_type = *Value*

using difference_type = std::ptrdiff_t

using pointer = *value_type**

using reference = *value_type*&

using Map = **typename** std::conditional<*Const*, **const** *MemoryMap*<*Value*>, *MemoryMap*<*Value*>>::type

using ValuePtr = **typename** std::conditional<*Const*, **const** *Value**, *Value**>::type

using Content = std::pair<*Key*, *ValuePtr*>

using Reference = **typename** std::conditional<*Const*, **const** *Content*&, *Content*&>::type

using Pointer = **typename** std::conditional<*Const*, **const** *Content**, *Content**>::type

Public Functions

Iterator_(*Map *map*, *Key ptr*)

Iterator_(*Map *map*, *iterator_begin*)

Iterator_(*Map *map*, *iterator_end*)

template<bool **OtherConst**>
Iterator_(**const** *Iterator_*<*OtherConst*> &*other*)

Reference **operator***()

Pointer **operator->**()

*Iterator_* &**operator++**()

*Iterator_* **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*.

Class Documentation

template<bool **Const** = false>
class umpire::util::*MemoryMap*::**Iterator_**

Public Types

using iterator_category = std::forward_iterator_tag

using value_type = Value

using difference_type = std::ptrdiff_t

using pointer = *value_type**

using reference = *value_type*&

using Map = **typename** std::conditional<*Const*, **const** MemoryMap<Value>, MemoryMap<Value>>::type

using ValuePtr = **typename** std::conditional<*Const*, **const** Value*, Value*>::type

using Content = std::pair<Key, *ValuePtr*>

using Reference = **typename** std::conditional<*Const*, **const** *Content*&, *Content*&>::type

using Pointer = **typename** std::conditional<*Const*, **const** *Content**, *Content**>::type

Public Functions

Iterator_(*Map* **map*, Key *ptr*)

Iterator_(*Map* **map*, *iterator_begin*)

Iterator_(*Map* **map*, *iterator_end*)

template<bool **OtherConst**>
Iterator_(**const** *Iterator_*<*OtherConst*> &*other*)

Reference **operator***()

Pointer **operator->**()

*Iterator_* &**operator++**()

*Iterator_* **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 umpire::util::**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 umpire::util::**OutputBuffer** : **public** streambuf

Public Functions

OutputBuffer() = default

~OutputBuffer()

void **setConsoleStream**(std::ostream *_stream_)

void **setFileStream**(std::ostream *_stream_)

int **overflow**(int _ch_) **override**

int **sync**() **override**

6.3.3 Enums

Enum MemoryResourceType

- Defined in file_umpire_resource_MemoryResourceTypes.hpp

Enum Documentation

enum umpire::resource::**MemoryResourceType**
Values:

enumerator Host

enumerator Device

enumerator Unified

enumerator Pinned

enumerator Constant

enumerator File

enumerator Unknown

Enum Level

- Defined in file_umpire_util_Logger.hpp

Enum Documentation

enum umpire::util::message::**Level**
Values:

enumerator Error

enumerator Warning

enumerator Info

enumerator Debug

enumerator Num_Levels

6.3.4 Functions

Function find_first_set

- Defined in file_umpire_strategy_FixedSizePool.hpp

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "find_first_set" with arguments (int) in doxygen xml output for project "umpire" from directory: ../doxygen/xml/. Potential matches:

```
- int 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 ShroudStrToArray(umpire_SHROUD_array *, const std::string *, int)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.cpp

Function Documentation

Warning: doxygenfunction: Cannot find function “ShroudStrToArray” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

Function ShroudStrToArray(umpire_SHROUD_array *, const std::string *, int)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.cpp

Function Documentation

Warning: doxygenfunction: Cannot find function “ShroudStrToArray” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

Function umpire::cpu_vendor_type

- Defined in file_umpire_util_detect_vendor.cpp

Function Documentation

MemoryResourceTraits::*vendor_type* `umpire::cpu_vendor_type() noexcept`

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_backtrace

- Defined in file_umpire_Umpire.cpp

Function Documentation

std::string umpire::**get_backtrace**(void *_ptr_)

Get the backtrace associated with the allocation of ptr.

The string may be empty if backtraces are not enabled.

Function umpire::get_device_memory_usage

- Defined in file_umpire_Umpire.cpp

Function Documentation

std::size_t `umpire::get_device_memory_usage`(int *device_id*)
Get memory usage of device device_id, using appropriate underlying vendor API.

Function umpire::get_leaked_allocations

- Defined in file_umpire_Umpire.cpp

Function Documentation

std::vector<util::*AllocationRecord*> `umpire::get_leaked_allocations`(*Allocator allocator*)
Get all the leaked (active) allocations associated with 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_process_memory_usage

- Defined in file_umpire_Umpire.cpp

Function Documentation

std::size_t umpire::**get_process_memory_usage**()
Get memory usage of the current process (uses underlying system-dependent calls)

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::is_accessible

- Defined in file_umpire_Umpire.cpp

Function Documentation

bool umpire::**is_accessible**(*Platform* *p*, *Allocator* *a*)

Check whether or not an *Allocator* is accessible from a given platform.

This function describes which allocators should be accessible from which CAMP platforms. Information on platform/allocator accessibility can be found at https://umpire.readthedocs.io/en/develop/features/allocator_accessibility.html

Parameters

- camp::Platform: p
- *umpire::Allocator*: a

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

Warning: doxygenfunction: Unable to resolve multiple matches for function "umpire::operator<<" with arguments (std::ostream&, const Allocator&) in doxygen xml output for project "umpire" from directory: ../doxygen/xml/. Potential matches:

```
- std::ostream &operator<<(std::ostream &os, const Allocator &allocator)
- std::ostream &operator<<(std::ostream &out, umpire::Allocator &alloc)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::DynamicPoolList::CoalesceHeuristic&)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::DynamicPoolMap::CoalesceHeuristic&)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::QuickPool::CoalesceHeuristic&)
```

Function umpire::operator<<(std::ostream&, umpire::Allocator&)

- Defined in file_umpire_Replay.cpp

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "umpire::operator<<" with arguments (std::ostream&, umpire::Allocator&) in doxygen xml output for project "umpire" from directory: ../doxygen/xml/. Potential matches:

```
- std::ostream &operator<<(std::ostream &os, const Allocator &allocator)
- std::ostream &operator<<(std::ostream &out, umpire::Allocator &alloc)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::DynamicPoolList::CoalesceHeuristic&)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::DynamicPoolMap::CoalesceHeuristic&)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::QuickPool::CoalesceHeuristic&)
```

Function umpire::operator<<(std::ostream&, umpire::strategy::DynamicPoolMap::CoalesceHeuristic&)

- Defined in file_umpire_Replay.cpp

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "umpire::operator<<" with arguments (std::ostream&, umpire::strategy::DynamicPoolMap::CoalesceHeuristic&) in doxygen xml output for project "umpire" from directory: ../doxygen/xml/. Potential matches:

```
- std::ostream &operator<<(std::ostream &os, const Allocator &allocator)
- std::ostream &operator<<(std::ostream &out, umpire::Allocator &alloc)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::DynamicPoolList::CoalesceHeuristic&)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::DynamicPoolMap::CoalesceHeuristic&)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::QuickPool::CoalesceHeuristic&)
```

Function umpire::operator<<(std::ostream&, umpire::strategy::DynamicPoolList::CoalesceHeuristic&)

- Defined in file_umpire_Replay.cpp

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "umpire::operator<<" with arguments (std::ostream&, umpire::strategy::DynamicPoolList::CoalesceHeuristic&) in doxygen xml output for project "umpire" from directory: ../doxygen/xml/. Potential matches:

```
- std::ostream &operator<<(std::ostream &os, const Allocator &allocator)
- std::ostream &operator<<(std::ostream &out, umpire::Allocator &alloc)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::DynamicPoolList::CoalesceHeuristic&)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::DynamicPoolMap::CoalesceHeuristic&)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::QuickPool::CoalesceHeuristic&)
```

Function umpire::operator<<(std::ostream&, umpire::strategy::QuickPool::CoalesceHeuristic&)

- Defined in file_umpire_Replay.cpp

Function Documentation

Warning: doxygenfunction: Unable to resolve multiple matches for function "umpire::operator<<" with arguments (std::ostream&, umpire::strategy::QuickPool::CoalesceHeuristic&) in doxygen xml output for project "umpire" from directory: ../doxygen/xml/. Potential matches:

```
- std::ostream &operator<<(std::ostream &os, const Allocator &allocator)
- std::ostream &operator<<(std::ostream &out, umpire::Allocator &alloc)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::DynamicPoolList::CoalesceHeuristic&)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::DynamicPoolMap::CoalesceHeuristic&)
- std::ostream &operator<<(std::ostream &out,␣
↪umpire::strategy::QuickPool::CoalesceHeuristic&)
```

Function umpire::pointer_contains

- Defined in file_umpire_Umpire.cpp

Function Documentation

bool `umpire::pointer_contains`(void *_left_, void *_right_)

Check whether the left allocation contains the right.

right is contained by left if right is greater than left, and right+size is greater than left+size.

Parameters

- `left`: Pointer to left allocation
- `right`: Poniter to right allocation

Function umpire::pointer_overlaps

- Defined in file_umpire_Umpire.cpp

Function Documentation

bool `umpire::pointer_overlaps`(void *_left_, void *_right_)

Check whether the right allocation overlaps the left.

right will overlap left if the right is greater than left, but less than left+size, and right+size is strictly greater than left+size.

Parameters

- `left`: Pointer to left allocation
- `right`: Poniter to right allocation

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::resource::resource_to_device_id

- Defined in file_umpire_resource_MemoryResourceTypes.hpp

Function Documentation

int umpire::resource::**resource_to_device_id**(**const** std::string &*resource*)

Function umpire::resource::resource_to_string

- Defined in file_umpire_resource_MemoryResourceTypes.hpp

Function Documentation

std::string umpire::resource::**resource_to_string**(*MemoryResourceType type*)

Function umpire::resource::string_to_resource

- Defined in file_umpire_resource_MemoryResourceTypes.hpp

Function Documentation

MemoryResourceType umpire::resource::**string_to_resource**(**const** std::string &*resource*)

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::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

int umpire::util::**case_insensitive_match**(**const** std::string *s1*, **const** std::string *s2*)

Function umpire::util::directory_exists

- Defined in file_umpire_util_io.cpp

Function Documentation

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

Warning: doxygenfunction: Unable to resolve multiple matches for function "umpire::util::do_wrap" with arguments (std::unique_ptr<Base>&&) in doxygen xml output for project "umpire" from directory: ../doxygen/xml/. Potential matches:

```
- template<typename Base, typename Strategy, typename ...Strategies> std::unique_ptr
↪<Base> do_wrap(std::unique_ptr<Base> &&allocator)
- template<typename Base> std::unique_ptr<Base> 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

Warning: doxygenfunction: Unable to resolve multiple matches for function "umpire::util::do_wrap" with arguments (std::unique_ptr<Base>&&) in doxygen xml output for project "umpire" from directory: ../doxygen/xml/. Potential matches:

```
- template<typename Base, typename Strategy, typename ...Strategies> std::unique_ptr
↪<Base> do_wrap(std::unique_ptr<Base> &&allocator)
- template<typename Base> std::unique_ptr<Base> do_wrap(std::unique_ptr<Base> &&
↪allocator)
```

Function umpire::util::file_exists

- Defined in file_umpire_util_io.cpp

Function Documentation

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

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 - (largest free block) / (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_allocation_count(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.cpp

Function Documentation

size_t **umpire_allocator_get_allocation_count**(*umpire_allocator* **self*)

Function umpire_allocator_get_allocation_count(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.h

Function Documentation

size_t **umpire_allocator_get_allocation_count**(*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_get_backtrace_bufferify(void *, umpire_SHROUD_array *)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.cpp

Function Documentation

void **umpire_get_backtrace_bufferify** (void *ptr, *umpire_SHROUD_array* *DSHF_rv)

Function umpire_get_backtrace_bufferify(void *, umpire_SHROUD_array *)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.h

Function Documentation

void **umpire_get_backtrace_bufferify**(void *ptr, *umpire_SHROUD_array* *DSHF_rv)

Function umpire_get_device_memory_usage(int)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.cpp

Function Documentation

size_t **umpire_get_device_memory_usage**(int *device_id*)

Function umpire_get_device_memory_usage(int)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.h

Function Documentation

size_t **umpire_get_device_memory_usage**(int *device_id*)

Function umpire_get_process_memory_usage(void)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.cpp

Function Documentation

size_t **umpire_get_process_memory_usage**(void)

Function umpire_get_process_memory_usage(void)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.h

Function Documentation

size_t **umpire_get_process_memory_usage**(void)

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, 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, array, 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, array, 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, array, 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, array, 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, array, 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, array, 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, array, 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, array, 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, array, 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, array, 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, array, 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, array, 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, array, 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, array, 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, array, 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, array, dims)

Function umpire_mod::allocator_associated

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

logical function umpire_mod::allocator_associated (obj)

Function umpire_mod::allocator_deallocate

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::**allocator_deallocate**(obj, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, array)
```

Function umpire_mod::allocator_delete

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine `umpire_mod::allocator_delete(obj)`

Function umpire_mod::allocator_eq

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
logical function umpire_mod::allocator_eq (a, 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)
```

Function umpire_mod::allocator_get_allocation_count

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
integer(c_size_t) function umpire_mod::allocator_get_allocation_count (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)
```

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)
```

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)
```

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)
```

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)
```

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, ptr)
```

Function umpire_mod::allocator_ne

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

logical function umpire_mod::allocator_ne (a, b)

Function umpire_mod::allocator_release

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine `umpire_mod::`**`allocator_release`**(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, cxxmem)

Function umpire_mod::get_backtrace

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

character(len=:) function, allocatable umpire_mod::get_backtrace (ptr)

Function umpire_mod::pointer_contains

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

logical function umpire_mod::pointer_contains (left, right)

Function umpire_mod::pointer_overlaps

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

logical function umpire_mod::pointer_overlaps (left, right)

Function umpire_mod::resourcemanager_add_alias

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::**resourcemanager_add_alias**(obj, name, allocator)

Function umpire_mod::resourcemanager_associated

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

logical function umpire_mod::resourcemanager_associated (obj)

Function umpire_mod::resourcemanager_copy_all

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::**resourcemanager_copy_all**(obj, src_ptr, dst_ptr)

Function umpire_mod::resourcemanager_copy_with_size

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::**resourcemanager_copy_with_size**(obj, src_ptr, dst_ptr, size)

Function umpire_mod::resourcemanager_deallocate

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::**resourcemanager_deallocate**(obj, ptr)

Function umpire_mod::resourcemanager_eq

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

logical function umpire_mod::resourcemanager_eq (a, b)

Function umpire_mod::resourcemanager_get_allocator_by_id

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

type(umpireallocator) function umpire_mod::resourcemanager_get_allocator_by_id (obj, id)

Function umpire_mod::resourcemanager_get_allocator_by_name

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

type(umpireallocator) function umpire_mod::resourcemanager_get_allocator_by_name (obj, name

Function umpire_mod::resourcemanager_get_allocator_for_ptr

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

type(umpireallocator) function umpire_mod::resourcemanager_get_allocator_for_ptr (obj, ptr)

Function umpire_mod::resourcemanager_get_instance

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
type(umpireresourcemanager) function umpire_mod::resourcemanager_get_instance ()
```

Function umpire_mod::resourcemanager_get_size

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
integer(c_size_t) function umpire_mod::resourcemanager_get_size (obj, ptr)
```

Function umpire_mod::resourcemanager_has_allocator

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
logical function umpire_mod::resourcemanager_has_allocator (obj, ptr)
```

Function umpire_mod::resourcemanager_is_allocator_id

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
logical function umpire_mod::resourcemanager_is_allocator_id (obj, id)
```

Function umpire_mod::resourcemanager_is_allocator_name

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
logical function umpire_mod::resourcemanager_is_allocator_name (obj, name)
```

Function umpire_mod::resourcemanager_make_allocator_advisor

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
type(umpireallocator) function umpire_mod::resourcemanager_make_allocator_advisor (obj, nam
```

Function umpire_mod::resourcemanager_make_allocator_fixed_pool

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
type(umpireallocator) function umpire_mod::resourcemanager_make_allocator_fixed_pool (obj,
```

Function umpire_mod::resourcemanager_make_allocator_list_pool

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
type(umpireallocator) function umpire_mod::resourcemanager_make_allocator_list_pool (obj, n
```

Function umpire_mod::resourcemanager_make_allocator_named

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
type(umpireallocator) function umpire_mod::resourcemanager_make_allocator_named (obj, name,
```

Function umpire_mod::resourcemanager_make_allocator_pool

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
type(umpireallocator) function umpire_mod::resourcemanager_make_allocator_pool (obj, name,
```

Function umpire_mod::resourcemanager_make_allocator_prefetcher

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

type(umpireallocator) function umpire_mod::resourcemanager_make_allocator_prefetcher (obj,

Function umpire_mod::resourcemanager_make_allocator_quick_pool

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

type(umpireallocator) function umpire_mod::resourcemanager_make_allocator_quick_pool (obj,

Function umpire_mod::resourcemanager_make_allocator_thread_safe

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

type(umpireallocator) function umpire_mod::resourcemanager_make_allocator_thread_safe (obj

Function umpire_mod::resourcemanager_memset_all

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::**resourcemanager_memset_all**(obj, ptr, val)

Function umpire_mod::resourcemanager_memset_with_size

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::**resourcemanager_memset_with_size**(obj, ptr, val, length)

Function umpire_mod::resourcemanager_move

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
type(c_ptr) function umpire_mod::resourcemanager_move (obj, src_ptr, allocator)
```

Function umpire_mod::resourcemanager_ne

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
logical function umpire_mod::resourcemanager_ne (a, b)
```

Function umpire_mod::resourcemanager_reallocate_default

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
type(c_ptr) function umpire_mod::resourcemanager_reallocate_default (obj, src_ptr, size)
```

Function umpire_mod::resourcemanager_reallocate_with_allocator

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
type(c_ptr) function umpire_mod::resourcemanager_reallocate_with_allocator (obj, src_ptr, s
```

Function umpire_mod::resourcemanager_register_allocator

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine `umpire_mod::`**`resourcemanager_register_allocator`**(obj, name, allocator)

Function umpire_mod::resourcemanager_remove_alias

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine `umpire_mod::`**`resourcemanager_remove_alias`**(obj, name, allocator)

Function umpire_pointer_contains(void *, void *)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.cpp

Function Documentation

bool **`umpire_pointer_contains`**(void **left*, void **right*)

Function umpire_pointer_contains(void *, void *)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.h

Function Documentation

bool **`umpire_pointer_contains`**(void **left*, void **right*)

Function umpire_pointer_overlaps(void *, void *)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.cpp

Function Documentation

bool **`umpire_pointer_overlaps`**(void **left*, void **right*)

Function umpire_pointer_overlaps(void *, void *)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.h

Function Documentation

bool **`umpire_pointer_overlaps`**(void **left*, void **right*)

Function umpire_resourcemanager_add_alias(umpire_resourcemanager *, const char *, umpire_allocator)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

void **umpire_resourcemanager_add_alias**(*umpire_resourcemanager* *self, **const** char *name, *umpire_allocator* allocator)

Function umpire_resourcemanager_add_alias(umpire_resourcemanager *, const char *, umpire_allocator)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

void **umpire_resourcemanager_add_alias**(*umpire_resourcemanager* *self, **const** char *name, *umpire_allocator* allocator)

Function umpire_resourcemanager_add_alias_bufferify(umpire_resourcemanager *, const char *, int, umpire_allocator)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

void **umpire_resourcemanager_add_alias_bufferify**(*umpire_resourcemanager* *self, **const** char *name, int Lname, *umpire_allocator* allocator)

Function umpire_resourcemanager_add_alias_bufferify(umpire_resourcemanager *, const char *, int, umpire_allocator)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

void **umpire_resourcemanager_add_alias_bufferify**(*umpire_resourcemanager* *self, **const** char *name, int Lname, *umpire_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_resourcemanager_get_allocator_by_name(umpire_resourcemanager *, const char *, umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

umpire_allocator ***umpire_resourcemanager_get_allocator_by_name**(*umpire_resourcemanager* **self*, **const** char **name*, *umpire_allocator* **SHC_rv*)

Function umpire_resourcemanager_get_allocator_by_name(umpire_resourcemanager *, const char *, umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

umpire_allocator ***umpire_resourcemanager_get_allocator_by_name**(*umpire_resourcemanager* **self*, **const** char **name*, *umpire_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.cpp

Function Documentation

umpire_allocator ***umpire_resourcemanager_get_allocator_by_name_bufferify**(*umpire_resourcemanager* **self*, **const** char **name*, int *Lname*, *umpire_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*, *umpire_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_id(umpire_resourcemanager *, int)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

bool **umpire_resourcemanager_is_allocator_id**(*umpire_resourcemanager* *self, int *id*)

Function umpire_resourcemanager_is_allocator_id(umpire_resourcemanager *, int)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

bool **umpire_resourcemanager_is_allocator_id**(*umpire_resourcemanager* *self, int *id*)

Function umpire_resourcemanager_is_allocator_name(umpire_resourcemanager *, const char *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

bool **umpire_resourcemanager_is_allocator_name**(*umpire_resourcemanager* *self, **const** char *name)

Function umpire_resourcemanager_is_allocator_name(umpire_resourcemanager *, const char *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

bool **umpire_resourcemanager_is_allocator_name**(*umpire_resourcemanager* **self*, **const** char **name*)

Function umpire_resourcemanager_is_allocator_name_bufferify(umpire_resourcemanager *, const char *, int)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

bool **umpire_resourcemanager_is_allocator_name_bufferify**(*umpire_resourcemanager* **self*, **const** char **name*, int *Lname*)

Function umpire_resourcemanager_is_allocator_name_bufferify(umpire_resourcemanager *, const char *, int)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

bool **umpire_resourcemanager_is_allocator_name_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*, *umpire_allocator allocator*, **const** char **advice_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*, *umpire_allocator allocator*, **const** char **advice_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*, *umpire_allocator allocator*, **const** char **advice_op*, int *Ladvice_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.h

Function Documentation

umpire_allocator ***umpire_resourcemanager_make_allocator_bufferify_advisor**(*umpire_resourcemanager* **self*, **const** char **name*, int *Lname*, *umpire_allocator* *allocator*, **const** char **advice_op*, int *Ladvice_op*, int *device_id*, *umpire_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*, *umpire_allocator* *allocator*, size_t *object_size*, *umpire_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*, *umpire_allocator* *allocator*, size_t *object_size*, *umpire_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*, *umpire_allocator* *allocator*, size_t *initial_size*, size_t *block*, *umpire_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, *umpire_allocator* allocator, size_t initial_size, size_t block, *umpire_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, *umpire_allocator* allocator, *umpire_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*, *umpire_allocator* *allocator*, *umpire_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*, *umpire_allocator* *allocator*, size_t *initial_size*, size_t *block*, *umpire_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*, *umpire_allocator* *allocator*, size_t *initial_size*, size_t *block*, *umpire_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*, *umpire_allocator* *allocator*, int *device_id*, *umpire_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*, *umpire_allocator allocator*, int *device_id*, *umpire_allocator* **SHC_rv*)

Function umpire_resourcemanager_make_allocator_bufferify_quick_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_quick_pool**(*umpire_resourcemanager* *self, **const** char *name, int Lname, *umpire_allocator* allocator, size_t initial_size, size_t block, *umpire_allocator* *SHC_rv)

Function umpire_resourcemanager_make_allocator_bufferify_quick_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_quick_pool**(*umpire_resourcemanager* *self, **const** char *name, int Lname, *umpire_allocator* allocator, size_t initial_size, size_t block, *umpire_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*, *umpire_allocator allocator*, *umpire_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.h

Function Documentation

umpire_allocator ***umpire_resourcemanager_make_allocator_bufferify_thread_safe**(*umpire_resourcemanager* **self*, **const** char **name*, int *Lname*, *umpire_allocator allocator*, *umpire_allocator* **SHC_rv*)

Function umpire_resourcemanager_make_allocator_fixed_pool(umpire_resourcemanager *, const char *, umpire_allocator, size_t, umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

umpire_allocator ***umpire_resourcemanager_make_allocator_fixed_pool**(*umpire_resourcemanager* *self*, **const** char *name*, *umpire_allocator* *allocator*, size_t *object_size*, *umpire_allocator* *SHC_rv*)

Function umpire_resourcemanager_make_allocator_fixed_pool(umpire_resourcemanager *, 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*, *umpire_allocator* *allocator*, size_t *object_size*, *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.cpp

Function Documentation

umpire_allocator ***umpire_resourcemanager_make_allocator_list_pool**(*umpire_resourcemanager* *self*, **const** char *name*, *umpire_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*, *umpire_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*, *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.h

Function Documentation

umpire_allocator ***umpire_resourcemanager_make_allocator_named**(*umpire_resourcemanager* **self*, **const** char **name*, *umpire_allocator* *allocator*, *umpire_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* *allocator*, size_t *initial_size*, size_t *block*, *umpire_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.h

Function Documentation

umpire_allocator ***umpire_resourcemanager_make_allocator_pool**(*umpire_resourcemanager* **self*, **const** char **name*, *umpire_allocator* *allocator*, size_t *initial_size*, size_t *block*, *umpire_allocator* **SHC_rv*)

Function umpire_resourcemanager_make_allocator_prefetcher(umpire_resourcemanager *, const char *, umpire_allocator, int, umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

umpire_allocator ***umpire_resourcemanager_make_allocator_prefetcher**(*umpire_resourcemanager* **self*, **const** char **name*, *umpire_allocator* *allocator*, int *device_id*, *umpire_allocator* **SHC_rv*)

Function umpire_resourcemanager_make_allocator_prefetcher(umpire_resourcemanager *, const char *, umpire_allocator, int, umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

umpire_allocator ***umpire_resourcemanager_make_allocator_prefetcher**(*umpire_resourcemanager* *self*, **const** char *name*, *umpire_allocator* *allocator*, int *device_id*, *umpire_allocator* *SHC_rv*)

Function umpire_resourcemanager_make_allocator_quick_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_quick_pool**(*umpire_resourcemanager* *self*, **const** char *name*, *umpire_allocator* *allocator*, size_t *initial_size*, size_t *block*, *umpire_allocator* *SHC_rv*)

Function umpire_resourcemanager_make_allocator_quick_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_quick_pool**(*umpire_resourcemanager* *self*, **const** char *name*, *umpire_allocator* *allocator*, size_t *initial_size*, size_t *block*, *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*, *umpire_allocator* *allocator*, *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.h

Function Documentation

umpire_allocator ***umpire_resourcemanager_make_allocator_thread_safe**(*umpire_resourcemanager* **self*, **const** char **name*, *umpire_allocator* *allocator*, *umpire_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_resourcemanager_reallocate_default(umpire_resourcemanager *, void *, size_t)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

void *__umpire_resourcemanager_reallocate_default__(*umpire_resourcemanager* **self*, void **src_ptr*, size_t *size*)

Function umpire_resourcemanager_reallocate_default(umpire_resourcemanager *, void *, size_t)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

void *__umpire_resourcemanager_reallocate_default__(*umpire_resourcemanager* **self*, void **src_ptr*, size_t *size*)

Function umpire_resourcemanager_reallocate_with_allocator(umpire_resourcemanager *, void *, size_t, umpire_allocator)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

void *__umpire_resourcemanager_reallocate_with_allocator__(*umpire_resourcemanager* **self*, void **src_ptr*, size_t *size*, *umpire_allocator* *allocator*)

Function umpire_resourcemanager_reallocate_with_allocator(umpire_resourcemanager *, void *, size_t, umpire_allocator)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

void *__umpire_resourcemanager_reallocate_with_allocator__(*umpire_resourcemanager* **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_resourcemanager_remove_alias(umpire_resourcemanager *, const char *, umpire_allocator)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

void **umpire_resourcemanager_remove_alias**(*umpire_resourcemanager* *self, **const** char *name, *umpire_allocator* allocator)

Function umpire_resourcemanager_remove_alias(umpire_resourcemanager *, const char *, umpire_allocator)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

void **umpire_resourcemanager_remove_alias**(*umpire_resourcemanager* *self, **const** char *name, *umpire_allocator* allocator)

Function umpire_resourcemanager_remove_alias_bufferify(umpire_resourcemanager *, const char *, int, umpire_allocator)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

void **umpire_resourcemanager_remove_alias_bufferify**(*umpire_resourcemanager* *self, **const** char *name, int Lname, *umpire_allocator* allocator)

Function umpire_resourcemanager_remove_alias_bufferify(umpire_resourcemanager *, const char *, int, umpire_allocator)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

void **umpire_resourcemanager_remove_alias_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_utilUmpire.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)
```

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, 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)

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, 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, 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)

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, 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)

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, 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, 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)

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, 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)

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, 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, cxxmem)

6.3.5 Variables

Variable genumpiresplicer::maxdims

- Defined in file_umpire_interface_c_fortran_genumpiresplicer.py

Variable Documentation

genumpiresplicer.**maxdims = 3**

Variable genumpiresplicer::types

- Defined in file_umpire_interface_c_fortran_genumpiresplicer.py

Variable Documentation

```
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::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", "WARNING", "INFO", "DEBU

Variable umpire_ver_5_found

- Defined in file_umpire_Umpire.cpp

Variable Documentation

Warning: doxygenvariable: Cannot find variable "umpire_ver_5_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 SH_TYPE_BOOL

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_BOOL

Define SH_TYPE_CHAR

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_CHAR

Define SH_TYPE_CPTR

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_CPTR

Define SH_TYPE_DOUBLE

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_DOUBLE

Define SH_TYPE_DOUBLE_COMPLEX

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_DOUBLE_COMPLEX

Define SH_TYPE_FLOAT

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_FLOAT

Define SH_TYPE_FLOAT_COMPLEX

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_FLOAT_COMPLEX

Define SH_TYPE_INT

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_INT

Define SH_TYPE_INT16_T

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_INT16_T

Define SH_TYPE_INT32_T

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_INT32_T

Define SH_TYPE_INT64_T

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_INT64_T

Define SH_TYPE_INT8_T

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_INT8_T

Define SH_TYPE_LONG

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_LONG

Define SH_TYPE_LONG_DOUBLE

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_LONG_DOUBLE

Define SH_TYPE_LONG_DOUBLE_COMPLEX

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_LONG_DOUBLE_COMPLEX

Define SH_TYPE_LONG_LONG

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_LONG_LONG

Define SH_TYPE_OTHER

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_OTHER

Define SH_TYPE_SHORT

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_SHORT

Define SH_TYPE_SIGNED_CHAR

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_SIGNED_CHAR

Define SH_TYPE_SIZE_T

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_SIZE_T

Define SH_TYPE_STRUCT

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_STRUCT

Define SH_TYPE_UINT16_T

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_UINT16_T

Define SH_TYPE_UINT32_T

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_UINT32_T

Define SH_TYPE_UINT64_T

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_UINT64_T

Define SH_TYPE_UINT8_T

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_UINT8_T

Define SH_TYPE_UNSIGNED_INT

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_UNSIGNED_INT

Define SH_TYPE_UNSIGNED_LONG

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_UNSIGNED_LONG

Define SH_TYPE_UNSIGNED_LONG_LONG

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_UNSIGNED_LONG_LONG

Define SH_TYPE_UNSIGNED_SHORT

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Define Documentation

SH_TYPE_UNSIGNED_SHORT

Define UMPIRE_aligned_allocation_INL

- Defined in file_umpire_strategy_mixins_AlignedAllocation.inl

Define Documentation

UMPIRE_aligned_allocation_INL

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_Backtrace_INL

- Defined in file_umpire_util_backtrace.inl

Define Documentation

UMPIRE_Backtrace_INL

Define UMPIRE_CudaGetAttributeOperation_INL

- Defined in file_umpire_op_CudaGetAttributeOperation.cpp

Define Documentation

Warning: doxygendefine: Cannot find define "UMPIRE_CudaGetAttributeOperation_INL" in doxygen xml output for project "umpire" from directory: ../doxygen/xml/

Define UMPIRE_DefaultMemoryResource_INL

- Defined in file_umpire_resource_DefaultMemoryResource.inl

Define Documentation

UMPIRE_DefaultMemoryResource_INL

Define UMPIRE_DEPRECATE

- Defined in file_umpire_util_Macros.hpp

Define Documentation

UMPIRE_DEPRECATE(*Msg*)

Define UMPIRE_DEPRECATE_ALIAS

- Defined in file_umpire_util_Macros.hpp

Define Documentation

UMPIRE_DEPRECATE_ALIAS(*Msg*)

Define UMPIRE_ERROR

- Defined in file_umpire_util_Macros.hpp

Define Documentation

UMPIRE_ERROR(*msg*)

Define UMPIRE_INVALID_ALLOCATOR_ID

- Defined in file_umpire_interface_umpire.h

Define Documentation

UMPIRE_INVALID_ALLOCATOR_ID

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

Warning: doxygendefine: Cannot find define "UMPIRE_NullMemoryResource_INL" in doxygen xml output for project "umpire" from directory: ../doxygen/xml/

Define UMPIRE_POISON_MEMORY_REGION

- Defined in file_umpire_util_memory_sanitizers.hpp

Define Documentation

UMPIRE_POISON_MEMORY_REGION(*allocator*, *ptr*, *size*)

Define UMPIRE_RECORD_BACKTRACE

- Defined in file_umpire_util_Macros.hpp

Define Documentation

UMPIRE_RECORD_BACKTRACE(*record*)

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_SyclDeviceMemoryResource_INL

- Defined in file_umpire_resource_SyclDeviceMemoryResource.inl

Define Documentation

UMPIRE_SyclDeviceMemoryResource_INL

Define UMPIRE_TypedAllocator_INL

- Defined in file_umpire_TypedAllocator.inl

Define Documentation

UMPIRE_TypedAllocator_INL

Define UMPIRE_UNPOISON_MEMORY_REGION

- Defined in file_umpire_util_memory_sanitizers.hpp

Define Documentation

UMPIRE_UNPOISON_MEMORY_REGION(*allocator*, *ptr*, *size*)

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::Platform

- Defined in file_umpire_util_Platform.hpp

Typedef Documentation

using umpire::**Platform** = camp::resources::Platform

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_quickpool

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_strategy_quickpool* **umpire_strategy_quickpool**

Typedef umpire_strategy_threadsafeallocator

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_strategy_threadsafeallocator* **umpire_strategy_threadsafeallocator**

CHAPTER

SEVEN

CONTRIBUTION GUIDE

This document is intented 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 deveolper 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 and Gitlab 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.

CHAPTER

EIGHT

DEVELOPER GUIDE

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.

8.1 Continuous Integration

8.1.1 Gitlab CI

Umpire shares its Gitlab CI workflow with other projects. The documentation is therefore **`shared`_**.

8.2 Uberenv

Umpire shares its Uberenv workflow with other projects. The documentation is therefore **`shared`_**.

This page will provides some Umpire specific examples to illustrate the workflow described in the documentation.

8.2.1 Before to start

First of all, it is worth noting that Umpire does not have dependencies, except for CMake, which is most of the time installed externally.

That does not make the workflow useless: Uberenv will drive Spack which will generate a host-config file with the toolchain (including cuda if activated) and the options or variants pre-configured.

Machine specific configuration

```
$ ls -c1 scripts/uberenv/spack_configs
blueos_3_ppc64le_ib
darwin
toss_3_x86_64_ib
blueos_3_ppc64le_ib_p9
config.yaml
```

Umpire has been configured for `toss_3_x86_64_ib` and other systems.

Vetted specs

```
$ ls -c1 .gitlab/*jobs.yml
.gitlab/lassen-jobs.yml
.gitlab/quartz-jobs.yml
```

CI contains jobs for quartz.

```
$ git grep -h "SPEC" .gitlab/quartz-jobs.yml | grep "gcc"
    SPEC: "%gcc@4.9.3"
    SPEC: "%gcc@6.1.0"
    SPEC: "%gcc@7.1.0"
    SPEC: "%gcc@7.3.0"
    SPEC: "%gcc@8.1.0"
```

We now have a list of the specs vetted on `quartz/toss_3_x86_64_ib`.

Note: In practice, one should check if the job is not *allowed to fail*, or even deactivated.

MacOS case

In Umpire, the Spack configuration for MacOS contains the default compilers depending on the OS version (*compilers.yaml*), and a commented section to illustrate how to add *CMake* as an external package. You may install CMake with homebrew, for example.

8.2.2 Using Uberenv to generate the host-config file

We have seen that we can safely use *gcc@8.1.0* on quartz. Let us ask for the default configuration first, and then produce static libs, have OpenMP support and run the benchmarks:

```
$ python scripts/uberenv/uberenv.py --spec="%gcc@8.1.0"
$ python scripts/uberenv/uberenv.py --spec="%gcc@8.1.0~shared+openmp tests=benchmarks"
```

Each will generate a CMake cache file, e.g.:

```
hc-quartz-toss_3_x86_64_ib-gcc@8.1.0-fjcjwd6ec3uen5rh6msdqujydsj74ubf.cmake
```

8.2.3 Using host-config files to build Umpire

```
$ mkdir build && cd build
$ cmake -C <path_to>/<host-config>.cmake ..
$ cmake --build -j .
$ ctest --output-on-failure -T test
```

It is also possible to use this configuration with the CI script outside of CI:

```
$ HOST_CONFIG=<path_to>/<host-config>.cmake scripts/gitlab/build_and_test.sh
```

8.2.4 Using Uberenv to configure and run Leak Sanitizer

During development, it may be beneficial to regularly check for memory leaks. This will help avoid the possibility of having many memory leaks showing up all at once during the CI tests later on. The Leak Sanitizer can easily be configured from the root directory with:

```
$ srun -ppdebug -N1 --exclusive python scripts/uberenv/uberenv.py --spec="%clang@9.0.
↪0 cxxflags=-fsanitize=address"
$ cd build
$ cmake -C <path_to>/hc-quartz-toss_3_x86_64_ib-clang@9.0.0.cmake ..
$ cmake --build -j
$ ASAN_OPTIONS=detect_leaks=1 make test
```

Note: The host config file (i.e., `hc-quartz-...cmake`) can be reused in order to rebuild with the same configuration if needed.

This will configure a build with Clang 9.0.0 and the Leak Sanitizer. If there is a leak in one of the tests, it can be useful to gather more information about what happened and more details about where it happened. One way to do this is to run:

```
$ ASAN_OPTIONS=detect_leaks=1 ctest -T test --output-on-failure
```

Additionally, the Leak Sanitizer can be run on one specific test (in this example, the "replay" tests) with:

```
$ ASAN_OPTIONS=detect_leaks=1 ctest -T test -R replay --output-on-failure
```

Depending on the output given when running the test with the Leak Sanitizer, it may be useful to use `addr2line -e <./path_to/executable> <address_of_leak>` to see the exact line the output is referring to.

INDEX

B

D

F

G

M

S

T

U