Umpire Documentation

Release 2.1.0

David Beckingsale

Jan 30, 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 -DCUDA_TOOLKIT_ROOT_DIR=/path/to/cuda ../
```

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

```
$ make
```

For more advanced configuration, see *Advanced Configuration*.

1.1.2 Installing Umpire

To install Umpire, just run:

```
$ make install
```

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

1.2 Basic Usage

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

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

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

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

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

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

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

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

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

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

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:

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

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

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

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

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

allocator.deallocate(data);
```

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

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

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

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

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

  allocator.deallocate(data);

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

  return 0;
}
```

2.2 Resources

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

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

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

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

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

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

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

```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"

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

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

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

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

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

  allocator.deallocate(data);

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

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

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

  return 0;
}
```

2.3 Operations

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

2.3.1 Copy

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

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

This example allocates the destination data using any valid Allocator.

2.3.2 Move

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

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

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

2.3.3 Memset

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

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

2.3.4 Reallocate

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

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

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

2.3.5 Listings

Copy Example Listing

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

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

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

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

  rm.copy(dest_data, source_data);

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

  dest_allocator.deallocate(dest_data);
}

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

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

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

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

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

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

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

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

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

  allocator.deallocate(data);

  return 0;
}
```

Move Example Listing

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

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

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

  return dest_data;
}

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

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

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

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

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

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

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

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

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

  rm.deallocate(data);
```

(continues on next page)

(continued from previous page)

```
  return 0;
}
```

Memset Example Listing

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

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

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

    rm.memset(data, 0);

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

    allocator.deallocate(data);
  }

  return 0;
}
```

Reallocate Example Listing

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

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

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

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

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

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

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

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

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

    allocator.deallocate(data);
  }

  return 0;
}
```

2.4 Dynamic Pools

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

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

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

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

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

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

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

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

  pooled_allocator.deallocate(data);
```

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

```
  allocate_and_deallocate_pool("HOST");

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

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

You can tune the way that *umpire::strategy::DynamicPool* allocates memory using two parameters: the initial size, and the minimum size. The initial size controls how large the first 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 allocator = rm.getAllocator(resource);

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

(continues on next page)

(continued from previous page)

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

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

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

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

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, 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)
{
  constexpr std::size_t SIZE = 1024;

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

  auto allocator = rm.getAllocator(resource);

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

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

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

  pooled_allocator.deallocate(data);

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

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

(continues on next page)

(continued from previous page)

```
  allocate_and_deallocate_pool("HOST");

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

  return 0;
}
```

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

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

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

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

  pooled_allocator.deallocate(data);

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

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

(continues on next page)

(continued from previous page)

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

  return 0;
}
```

2.5 Introspection

When writing code to run on computers with a complex memory hierarchy, one of the most difficult things can be keeping track of where each pointer has been allocated. Umpire's 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:

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

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

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

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

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

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

```
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2016-20, 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[] = {
```

(continues on next page)

(continued from previous page)

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

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

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

    auto found_allocator = rm.getAllocator(data);

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

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

    allocator.deallocate(data);
  }

  return 0;
}
```

2.6 Typed Allocators

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

A `umpire::TypedAllocator` is constructed from any existing Allocator, and provides the same interface as the normal *`umpire::Allocator`*. However, when you call allocate, this argument is the number of objects you want to allocate, 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};

  my_vector.resize(100);
```

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

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

#include "umpire/TypedAllocator.hpp"

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

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

  double* my_doubles = double_allocator.allocate(1024);

  double_allocator.deallocate(my_doubles, 1024);

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

  my_vector.resize(100);

  return 0;
}
```

2.7 Replay

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

2.7.1 Input Example

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

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

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

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

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

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

2.7.2 Running the Example

Running this program:

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

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

2.7.3 Replaying the session

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

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

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

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

2.8 C API: Allocators

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

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

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

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

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

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

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

  umpire_allocator_deallocate(&allocator, data);
```

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

2.9 C API: Resources

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

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

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

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

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

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

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

2.10 C API: Pools

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

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

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

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

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

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

2.11 FORTRAN API: Allocators

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

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

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

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

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

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

call allocator%deallocate(array)
```

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

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_STATISTICS	Off	Enable collection of memory statistics
ENABLE_TESTING	On	Build test executables
ENABLE_BENCHMARKS	On	Build benchmark programs
ENABLE_LOGGING	On	Enable Logging within Umpire
ENABLE_SLIC	Off	Enable SLIC logging
ENABLE_TOOLS	Off	Enable tools like replay
ENABLE_DOCS	Off	Build documentation (requires Sphinx and/or Doxygen)
ENABLE_C	Off	Build the C API
ENABLE_FORTRAN	Off	Build the Fortran API

These arguments are explained in more detail below:

- ENABLE_CUDA This option enables support for NVIDIA GPUs using the CUDA programming model. If Umpire is built without CUDA 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_STATISTICS This option enables collection of memory statistics. If Umpire is built with this option, the Conduit library will also be built.
- ENABLE_TESTING This option controls whether or not test executables will be built.
- ENABLE_BENCHMARKS This option will build the benchmark programs used to test performance.
- ENABLE_LOGGING This option enables usage of Logging services for Umpire
- ENABLE_SLIC This option enables usage of logging services provided by SLIC.
- ENABLE_TOOLS Enable development tools for Umpire (replay, etc.)

- ENABLE_DOCS Build user documentation (with Sphinx) and code documentation (with Doxygen)
- ENABLE_C Build the C API, this allows accessing Umpire Allocators and the ResourceManager through a C interface.
- ENABLE_FORTRAN Build the Fortran API.

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);
void* hold = pooled_allocator.allocate(64);
UMPIRE_USE_VAR(hold);
```

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

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

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

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

The complete example is included below:

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

(continues on next page)

(continued from previous page)

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

#include "umpire/util/Macros.hpp"

#include <iostream>

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

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

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


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


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

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

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

  return 0;
}
```

4.2 Disable Introspection

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

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

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/strategy/DynamicPool.hpp"

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

#include "umpire/util/Macros.hpp"

#include <iostream>

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

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

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

  void* data = pooled_allocator.allocate(1024);

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

(continues on next page)

(continued from previous page)

```
  pooled_allocator.deallocate(data);

  return 0;
}
```

4.3 Apply Memory Advice to a Pool

When using unified memory on systems with CUDA GPUs, various types of memory advice can be applied to modify how the CUDA runtime moves this memory around between the CPU and GPU. One type of advice that can be applied is "preferred location", and you can 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:

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

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

The complete example is included below:

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

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

#include "umpire/util/Macros.hpp"

#include <iostream>

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

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

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

(continues on next page)

(continued from previous page)

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

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

  UMPIRE_USE_VAR(pooled_allocator);

  return 0;
}
```

4.4 Apply Memory Advice with a Specific Device ID

When using unified memory on systems with CUDA GPUs, various types of memory advice can be applied to modify how the CUDA runtime moves this memory around between the CPU and GPU. When applying memory advice, a device ID can be used to 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

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

The complete example is included below:

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

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

#include "umpire/util/Exception.hpp"

#include <iostream>

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

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

(continues on next page)

(continued from previous page)

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

    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");
```

(continues on next page)

(continued from previous page)

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

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

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

  return 0;
}
```

4.6 Improving DynamicPool Performance with a Coalesce Heuristic

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

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

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

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

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

Creation of the heuristic function is accomplished by:

```
  //
```

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

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

(continues on next page)

(continued from previous page)

```
                              , 1024ul
                              , 16
```

The complete example is included below:

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

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

#include "umpire/util/Macros.hpp"

#include <iostream>

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    pooled_allocator.deallocate(a[i]);
    std::cout
      << "Pool has " << pooled_allocator.getActualSize() << " bytes of memory. "
      << pooled_allocator.getCurrentSize() << " bytes are used. "
      << dynamic_pool->getBlocksInPool() << " blocks are in the pool. "
      << dynamic_pool->getReleasableSize() << " bytes are 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 "umpire/Allocator.hpp"
#include "umpire/ResourceManager.hpp"

#include "umpire/strategy/NumaPolicy.hpp"

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

#include <iostream>

#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();
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

#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)
```

(continues on next page)

(continued from previous page)

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

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

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

  return 0;
}
```

4.8 Determining the Largest Block of Available Memory in Pool

The *umpire::strategy::DynamicPool* provides a `umpire::strategy::DynamicPool::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& rm = umpire::ResourceManager::getInstance();

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

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

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

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

The complete example is included below:

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

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

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

(continues on next page)

(continued from previous page)

```
#include <iostream>

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

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

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

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

  auto ptr = pool.allocate(1024);

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

  pool.deallocate(ptr);

  return 0;
}
```

4.9 Coalescing Pool Memory

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

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

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

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

The complete example is included below:

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

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

(continues on next page)

(continued from previous page)

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

#include <iostream>

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

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

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

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

  return 0;
}
```

4.10 Building a Pinned Memory Pool in FORTRAN

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

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

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

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

The complete example is included below:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Copyright (c) 2016-20, 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
```

(continues on next page)

(continued from previous page)

```
      type(UmpireAllocator) pinned_pool
      type(UmpireResourceManager) rm

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

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

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

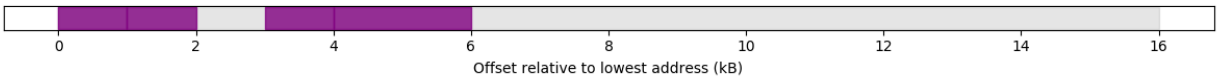
4.11 Visualizing Allocators

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

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

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

That script uses Python and Matplotlib to generate the following image



The complete example is included below:

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

#include <fstream>

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

  return 0;
}
```

4.12 Mixed Pool Creation and Algorithm Basics

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

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

The complete example is included below:

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

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

#include <iostream>
```

(continues on next page)

(continued from previous page)

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

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

  UMPIRE_USE_VAR(default_mixed_allocator);

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

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

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

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

  return 0;
}
```

4.13 Thread Safe Allocator

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

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

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

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

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

The complete example is included below:

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

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

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

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

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

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

  return 0;
}
```

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

5.2.1 Provided Strategies

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

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

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

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

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

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

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

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

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

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

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

5.3 Operations

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

5.3.1 Provided Operations

namespace op

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

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::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*

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.

5.4 Logging and Replay of Umpire Events

5.4.1 Logging

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

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

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

5.4.2 Replay

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

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

5.4.3 Running with Replay

To enable Umpire replay, one may execute as follows:

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

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

5.4.4 Interpretting Results - Version Event

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

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

Each line contains the following set of common elements:

kind Always set to `replay`

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

timestamp Set to the time when the event occurred.

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

payload Optional and varies upon event type

result Optional and varies upon event type

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

5.4.5 makeMemoryResource Event

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

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052595934822, "event": "allocate",
↪"payload": { "allocator_ref": "0x108a8730", "size": 0 }, "result": { "memory_ptr":
↪"0x200040000010" } }
{ "kind":"replay", "uid":27494, "timestamp":1558388052595939623, "event": "allocate",
↪"payload": { "allocator_ref": "0x108a8730", "size": 134 } }
{ "kind":"replay", "uid":27494, "timestamp":1558388052595943793, "event": "allocate",
↪"payload": { "allocator_ref": "0x108a8730", "size": 134 }, "result": { "memory_ptr
↪": "0x200040000010" } }
{ "kind":"replay", "uid":27494, "timestamp":1558388052595947408, "event": "allocate",
↪"payload": { "allocator_ref": "0x108a8730", "size": 774 } }
{ "kind":"replay", "uid":27494, "timestamp":1558388052595951548, "event": "allocate",
↪"payload": { "allocator_ref": "0x108a8730", "size": 774 }, "result": { "memory_ptr
↪": "0x2000400000a0" } }
```

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

5.4.6 makeAllocator Event

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

umpire::Allocator:

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052595954839, "event": "allocate",
↪"payload": { "allocator_ref": "0x108a8730", "size": 470 } }
{ "kind":"replay", "uid":27494, "timestamp":1558388052595958585, "event": "allocate",
↪"payload": { "allocator_ref": "0x108a8730", "size": 470 }, "result": { "memory_ptr
↪": "0x2000400003b0" } }
```

(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.

5.4.7 allocate Event

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

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052595961448, "event": "allocate",
↪"payload": { "allocator_ref": "0x108a8730", "size": 546 } }
{ "kind":"replay", "uid":27494, "timestamp":1558388052595964866, "event": "allocate",
↪"payload": { "allocator_ref": "0x108a8730", "size": 546 }, "result": { "memory_ptr
↪": "0x200040000590" } }
```

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

5.4.8 deallocate Event

```
{ "kind":"replay", "uid":27494, "timestamp":1558388052596397388, "event": "deallocate
↪", "payload": { "allocator_ref": "0x108a8730", "memory_ptr": "0x200040000b90" } }
```

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

5.4.9 Replaying the session

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

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

5.5 File I/O

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

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

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

The format of the filenames is:

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

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

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

Enums

- *Enum Platform*

Functions

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

Variables

- *Variable umpire::env_name*

Namespace umpire::alloc

Contents

Classes

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

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*
- *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*

Namespace umpire::resource

Contents

- *Classes*
- *Enums*

Classes

- *Struct MemoryResourceTypeHash*
- *Class CudaConstantMemoryResource*
- *Class CudaConstantMemoryResourceFactory*
- *Class CudaDeviceResourceFactory*
- *Class CudaPinnedMemoryResourceFactory*
- *Class CudaUnifiedMemoryResourceFactory*
- *Template Class DefaultMemoryResource*
- *Class HipConstantMemoryResource*
- *Class HipConstantMemoryResourceFactory*
- *Class HipDeviceResourceFactory*

- *Class HipPinnedMemoryResourceFactory*
- *Class HostResourceFactory*
- *Class MemoryResource*
- *Class MemoryResourceFactory*
- *Class MemoryResourceRegistry*
- *Class NullMemoryResource*
- *Class NullMemoryResourceFactory*

Enums

- *Enum MemoryResourceType*

Namespace umpire::strategy

Contents

Namespaces

- *Namespace umpire::strategy::mixins*

Classes

- *Struct FixedPool::Pool*
- *Class AllocationAdvisor*
- *Class AllocationPrefetcher*
- *Class AllocationStrategy*
- *Class AllocationTracker*
- *Class DynamicPoolList*
- *Class DynamicPoolMap*
- *Class FixedPool*
- *Class MixedPool*
- *Class MonotonicAllocationStrategy*

- *Class NamedAllocationStrategy*
- *Class NumaPolicy*
- *Class SizeLimiter*
- *Class SlotPool*
- *Class ThreadSafeAllocator*
- *Class ZeroByteHandler*

Functions

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

Typedefs

- *Typedef umpire::strategy::DynamicPool*

Variables

- *Variable umpire::strategy::bits_per_int*
- *Variable umpire::strategy::heuristic_percent_releasable*
- *Variable umpire::strategy::heuristic_percent_releasable_list*

Namespace umpire::strategy::mixins

Contents

- *Classes*

Classes

- *Class Inspector*

Namespace umpire::util

Contents

Namespaces

Classes

Functions

- *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::@161

Namespace umpire::util::detail

Contents

Functions

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

Namespace umpire::util::message

Contents

Enums

Namespace umpire_mod

Contents

Functions

Namespace umpire_strategy_mod

Contents

Functions

- *Function umpire_strategy_mod::namedallocationstrategy_eq*
- *Function umpire_strategy_mod::namedallocationstrategy_get_instance*
- *Function umpire_strategy_mod::namedallocationstrategy_ne*
- *Function umpire_strategy_mod::namedallocationstrategy_set_instance*

6.3.2 Classes and Structs

Struct DynamicSizePool::Block

- Defined in file_umpire_strategy_DynamicSizePool.hpp

Nested Relationships

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

Struct Documentation

struct Block

Public Members

template<>
char ***data**

template<>
std::size_t **size**

template<>
std::size_t **blockSize**

template<>
Block ***next**

Struct FixedSizePool::Pool

- Defined in file_umpire_strategy_FixedSizePool.hpp

Nested Relationships

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

Struct Documentation

struct Pool

Public Members

template<>
unsigned char ***data**

template<>
unsigned int ***avail**

template<>
unsigned int **numAvail**

template<>
struct Pool ***next**

Struct s_umpire_allocator

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_allocator

Public Members

void ***addr**

int **idtor**

Struct s_umpire_resourcemanager

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_resourcemanager

Public Members

void ***addr**

int **idtor**

Struct s_umpire_SHROUD_array

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_SHROUD_array

Public Members

umpire_SHROUD_capsule_data **cxx**

const void ***cvoidp**

const char ***ccharp**

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

size_t **len**

size_t **size**

Struct s_umpire_SHROUD_capsule_data

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_SHROUD_capsule_data

Public Members

void ***addr**

int **idtor**

Struct s_umpire_strategy_allocationadvisor

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_strategy_allocationadvisor

Public Members

void *addr

int idtor

Struct s_umpire_strategy_allocationprefetcher

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_strategy_allocationprefetcher

Public Members

void *addr

int idtor

Struct s_umpire_strategy_dynamicpool

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_strategy_dynamicpool

Public Members

void *addr

int idtor

Struct s_umpire_strategy_dynamicpoollist

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_strategy_dynamicpoollist

Public Members

void *addr

int idtor

Struct s_umpire_strategy_fixedpool

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_strategy_fixedpool

Public Members

void *addr

int idtor

Struct s_umpire_strategy_namedallocationstrategy

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_strategy_namedallocationstrategy

Public Members

void *addr

int idtor

Struct s_umpire_strategy_threadsafeallocator

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Struct Documentation

struct s_umpire_strategy_threadsafeallocator

Public Members

void ***addr**

int **idtor**

Struct StdAllocator

- Defined in file_umpire_strategy_StdAllocator.hpp

Struct Documentation

struct StdAllocator

Public Static Functions

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

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

Struct CudaMallocAllocator

- Defined in file_umpire_alloc_CudaMallocAllocator.hpp

Struct Documentation

struct CudaMallocAllocator

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

Public Functions

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

Allocate bytes of memory using cudaMalloc.

Return Pointer to start of the allocation.

Parameters

- `bytes`: Number of bytes to allocate.

Exceptions

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

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

Deallocate memory using cudaFree.

Parameters

- `ptr`: Address to deallocate.

Exceptions

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

Struct CudaMallocManagedAllocator

- Defined in file_umpire_alloc_CudaMallocManagedAllocator.hpp

Struct Documentation

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

Public Functions

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

Return Pointer to start of the allocation.

Parameters

- `bytes`: Number of bytes to allocate.

Exceptions

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

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

Parameters

- `ptr`: Address to deallocate.

Exceptions

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

Struct CudaPinnedAllocator

- Defined in file_umpire_alloc_CudaPinnedAllocator.hpp

Struct Documentation

struct CudaPinnedAllocator

Public Functions

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

void __deallocate__(void **ptr*)

Struct HipMallocAllocator

- Defined in file_umpire_alloc_HipMallocAllocator.hpp

Struct Documentation

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

Public Functions

void *__allocate__(std::size_t *size*)
Allocate bytes of memory using hipMalloc.

Return Pointer to start of the allocation.

Parameters

- `bytes`: Number of bytes to allocate.

Exceptions

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

void __deallocate__(void **ptr*)
Deallocate memory using hipFree.

Parameters

- `ptr`: Address to deallocate.

Exceptions

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

Struct HipPinnedAllocator

- Defined in file_umpire_alloc_HipPinnedAllocator.hpp

Struct Documentation

struct HipPinnedAllocator

Public Functions

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

void __deallocate__(void **ptr*)

Struct MallocAllocator

- Defined in file_umpire_alloc_MallocAllocator.hpp

Struct Documentation

struct MallocAllocator
Uses malloc and free to allocate and deallocate CPU memory.

Public Functions

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

Return Pointer to start of the allocation.

Parameters

- `bytes`: Number of bytes to allocate.

Exceptions

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

void __deallocate__(void **ptr*)
Deallocate memory using free.

Parameters

- `ptr`: Address to deallocate.

Exceptions

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

Struct PosixMemalignAllocator

- Defined in file_umpire_alloc_PosixMemalignAllocator.hpp

Struct Documentation

struct PosixMemalignAllocator

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

Public Functions

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

Allocate bytes of memory using posix_memalign.

Return Pointer to start of the allocation.

Parameters

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

Exceptions

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

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

Deallocate memory using free.

Parameters

- `ptr`: Address to deallocate.

Exceptions

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

Struct MemoryResourceTraits

- Defined in file_umpire_util_MemoryResourceTraits.hpp

Struct Documentation

struct MemoryResourceTraits

Public Types

enum optimized_for

Values:

any

latency

bandwidth

access

enum vendor_type

Values:

UNKNOWN

AMD

IBM

INTEL

NVIDIA

enum memory_type

Values:

UNKNOWN

DDR

GDDR

HBM

NVME

Public Members

bool **unified** = false

std::size_t **size** = 0

vendor_type **vendor** = *vendor_type*::UNKNOWN

memory_type **kind** = *memory_type*::UNKNOWN

optimized_for **used_for** = *optimized_for*::any

Struct pair_hash

- Defined in file_umpire_op_MemoryOperationRegistry.hpp

Struct Documentation

struct pair_hash

Public Functions

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

Struct MemoryResourceTypeHash

- Defined in file_umpire_resource_MemoryResourceTypes.hpp

Struct Documentation

struct MemoryResourceTypeHash

Public Functions

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

Struct FixedPool::Pool

- Defined in file_umpire_strategy_FixedPool.hpp

Nested Relationships

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

Struct Documentation

struct Pool

Public Functions

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

Public Members

AllocationStrategy ***strategy**

char ***data**

int ***avail**

std::size_t **num_avail**

Template Struct RecordList::Block

- Defined in file_umpire_util_AllocationMap.hpp

Nested Relationships

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

Struct Documentation

template<typename **T**>
struct Block

Public Members

T **rec**

Block ***prev**

Struct AllocationRecord

- Defined in file_umpire_util_AllocationRecord.hpp

Struct Documentation

struct AllocationRecord

Public Members

void ***ptr**

std::size_t **size**

strategy::AllocationStrategy ***strategy**

Struct FixedMallocPool::Pool

- Defined in file_umpire_util_FixedMallocPool.hpp

Nested Relationships

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

Struct Documentation

struct Pool

Public Functions

Pool(**const** std::size_t *object_bytes*, **const** std::size_t *objects_per_pool*)

Public Members

unsigned char ***data**

unsigned char ***next**

unsigned int **num_initialized**

std::size_t **num_free**

Struct iterator_begin

- Defined in file_umpire_util_MemoryMap.hpp

Struct Documentation

struct iterator_begin

Struct iterator_end

- Defined in file_umpire_util_MemoryMap.hpp

Struct Documentation

struct iterator_end

Template Class DynamicSizePool

- Defined in file_umpire_strategy_DynamicSizePool.hpp

Nested Relationships

Nested Types

- *Struct DynamicSizePool::Block*

Class Documentation

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

Public Functions

DynamicSizePool (umpire::strategy::*AllocationStrategy* *_strat_, **const** std::size_t _*minInitialBytes* = (16 * 1024), **const** std::size_t _*minBytes* = 256)

~DynamicSizePool ()

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

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

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

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

std::size_t **getHighWatermark** () **const**

std::size_t **getBlocksInPool** () **const**

std::size_t **getLargestAvailableBlock** () **const**

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

std::size_t **getFreeBlocks** () **const**

std::size_t **getInUseBlocks** () **const**

void **coalesce** ()

void **release** ()

Protected Types

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

Protected Functions

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

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

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

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

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

std::size_t **freeReleasedBlocks** ()

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

void **freeAllBlocks** ()

Protected Attributes

BlockPool **blockPool**

struct *Block* ***usedBlocks**

struct *Block* ***freeBlocks**

std::size_t **totalBlocks**

std::size_t **totalBytes**

std::size_t **allocBytes**

std::size_t **minInitialBytes**

std::size_t **minBytes**

std::size_t **highWatermark**

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

struct Block

Public Members

template<>
char ***data**

template<>
std::size_t **size**

template<>
std::size_t **blockSize**

template<>
Block ***next**

Template Class FixedSizePool

- Defined in file_umpire_strategy_FixedSizePool.hpp

Nested Relationships

Nested Types

- *Struct FixedSizePool::Pool*

Class Documentation

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

Public Functions

FixedSizePool()

~FixedSizePool()

T ***allocate**()

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

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

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

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

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

Public Static Functions

static FixedSizePool &**getInstance**()

Protected Functions

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

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

Protected Attributes

struct *Pool* ***pool**

const std::size_t **numPerPool**

const std::size_t **totalPoolSize**

std::size_t **numBlocks**

struct Pool

Public Members

template<>
unsigned char ***data**

template<>
unsigned int ***avail**

template<>
unsigned int **numAvail**

template<>
struct Pool ***next**

Class Allocator

- Defined in file_umpire_Allocator.hpp

Class Documentation

class Allocator

Provides a unified interface to allocate and free data.

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

See *TypedAllocator*

Public Functions

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

Allocate bytes of memory.

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

Return Pointer to start of the allocation.

Parameters

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

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

Free the memory at ptr.

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

Parameters

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

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

std::size_t **getSize**(void **ptr*) **const**
Return number of bytes allocated for allocation.

Return number of bytes allocated for ptr

Parameters

- `ptr`: Pointer to allocation in question

std::size_t **getHighWatermark**() **const**
Return the memory high watermark for this *Allocator*.

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

Return Memory high watermark.

std::size_t **getCurrentSize**() **const**
Return the current size of this *Allocator*.

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

Return current size of *Allocator*.

std::size_t **getActualSize**() **const**
Return the actual size of this *Allocator*.

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

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

Return actual size of *Allocator*.

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

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

See *ResourceManager::getAllocator*

Return name of *Allocator*.

int **getId**() **const**
Get the integer ID of this *Allocator*.

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

See *ResourceManager::getAllocator*

Return integer id of *Allocator*.

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

Return Pointer to the AllocationStrategy.

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

Return Platform for this *Allocator*.

Allocator()

Friends

friend umpire::Allocator::::AllocatorTest

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

Class DeviceAllocator

- Defined in file_umpire_DeviceAllocator.hpp

Class Documentation

class DeviceAllocator
Lightweight allocator for use in GPU code.

Public Functions

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

Parameters

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

__host__ **~DeviceAllocator**()

__host__ __device__ umpire::DeviceAllocator::DeviceAllocator(const DeviceAllocator &

__device__ void * umpire::DeviceAllocator::allocate(size_t size)

Class CudaAdviseAccessedByOperation

- Defined in file_umpire_op_CudaAdviseAccessedByOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

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

Parameters

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

Exceptions

- *`util::Exception`*:

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

Class CudaAdvisePreferredLocationOperation

- Defined in file_umpire_op_CudaAdvisePreferredLocationOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class CudaAdvisePreferredLocationOperation : **public** umpire::*op*::*MemoryOperation*

Public Functions

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

Uses cudaMemAdvise to set 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`*:

Class CudaAdviseReadMostlyOperation

- Defined in file_umpire_op_CudaAdviseReadMostlyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

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

Parameters

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

Exceptions

- *`util::Exception`*:

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

Class CudaAdviseUnsetAccessedByOperation

- Defined in file_umpire_op_CudaAdviseUnsetAccessedByOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

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

Parameters

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

Exceptions

- *`util::Exception`*:

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

Class CudaAdviseUnsetPreferredLocationOperation

- Defined in file_umpire_op_CudaAdviseUnsetPreferredLocationOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

Uses cudaMemAdvise to set 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`*:

Class CudaAdviseUnsetReadMostlyOperation

- Defined in file_umpire_op_CudaAdviseUnsetReadMostlyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

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

Parameters

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

Exceptions

- *`util::Exception`*:

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

Class CudaCopyFromOperation

- Defined in file_umpire_op_CudaCopyFromOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Copy operation to move data from a 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`*:

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

Apply val to the first length bytes of src_ptr.

Parameters

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

Exceptions

- *`util::Exception`*:

Class CudaCopyOperation

- Defined in file_umpire_op_CudaCopyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

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

Parameters

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

Exceptions

- *`util::Exception`*:

Class CudaCopyToOperation

- Defined in file_umpire_op_CudaCopyToOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

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

Parameters

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

Exceptions

- *util::Exception*:

Class CudaMemPrefetchOperation

- Defined in file_umpire_op_CudaMemPrefetchOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

Parameters

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

Exceptions

- *`util::Exception`*:

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

Class CudaMemsetOperation

- Defined in file_umpire_op_CudaMemsetOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class CudaMemsetOperation : **public** umpire::*op*::*MemoryOperation*

Memset on NVIDIA device memory.

Public Functions

void **apply**(void **src_ptr*, util::*AllocationRecord* **ptr*, int *value*, std::size_t *length*)

Apply val to the first length bytes of src_ptr.

Uses cudaMemset to set first length bytes of src_ptr to value.

Parameters

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

Exceptions

- *`util::Exception`*:

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

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`*:

Class GenericReallocateOperation

- Defined in file_umpire_op_GenericReallocateOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class GenericReallocateOperation : **public** umpire::*op*::*MemoryOperation*

Generic reallocate operation to work on any current_ptr location.

Public Functions

void **transform**(void **current_ptr*, void ***new_ptr*, util::*AllocationRecord* **current_allocation*, util::*AllocationRecord* **new_allocation*, std::size_t *new_size*)

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`*:

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

Apply val to the first length bytes of src_ptr.

Parameters

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

Exceptions

- *`util::Exception`*:

Class HipCopyFromOperation

- Defined in file_umpire_op_HipCopyFromOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

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

Parameters

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

Exceptions

- *`util::Exception`*:

Class HipCopyOperation

- Defined in file_umpire_op_HipCopyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

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

Parameters

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

Exceptions

- *util::Exception*:

Class HipCopyToOperation

- Defined in file_umpire_op_HipCopyToOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class HipCopyToOperation : **public** umpire::*op*::*MemoryOperation*
Copy operation to move data from CPU to AMD GPU memory.

Public Functions

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

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

Parameters

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

Exceptions

- *`util::Exception`*:

Class HipMemsetOperation

- Defined in file_umpire_op_HipMemsetOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

Uses hipMemset to set first length bytes of src_ptr to value.

Parameters

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

Exceptions

- *`util::Exception`*:

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

Class HostCopyOperation

- Defined in file_umpire_op_HostCopyOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class HostCopyOperation : **public** umpire::*op*::*MemoryOperation*
Copy memory between two allocations in CPU memory.

Public Functions

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

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

Parameters

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

Exceptions

- *`util::Exception`*:

Class HostMemsetOperation

- Defined in file_umpire_op_HostMemsetOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class HostMemsetOperation : **public** umpire::*op*::*MemoryOperation*

Memset an allocation in CPU memory.

Public Functions

void **apply**(void **src_ptr*, util::*AllocationRecord* **allocation*, int *value*, std::size_t *length*)

Apply val to the first length bytes of src_ptr.

Uses std::memset to set the first length bytes of src_ptr to value.

Parameters

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

Exceptions

- *`util::Exception`*:

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

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`*:

Class HostReallocateOperation

- Defined in file_umpire_op_HostReallocateOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class HostReallocateOperation : **public** umpire::*op*::*MemoryOperation*

Reallocate data in CPU memory.

Public Functions

void **transform**(void **current_ptr*, void ***new_ptr*, util::*AllocationRecord* **current_allocation*, util::*AllocationRecord* **new_allocation*, std::size_t *new_size*)

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`*:

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

Apply val to the first length bytes of src_ptr.

Parameters

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

Exceptions

- *`util::Exception`*:

Class MemoryOperation

- Defined in file_umpire_op_MemoryOperation.hpp

Inheritance Relationships

Derived Types

- `public umpire::op::CudaAdviseAccessedByOperation` (*Class CudaAdviseAccessedByOperation*)
- `public umpire::op::CudaAdvisePreferredLocationOperation` (*Class CudaAdvisePreferredLocationOperation*)
- `public umpire::op::CudaAdviseReadMostlyOperation` (*Class CudaAdviseReadMostlyOperation*)
- `public umpire::op::CudaAdviseUnsetAccessedByOperation` (*Class CudaAdviseUnsetAccessedByOperation*)
- `public umpire::op::CudaAdviseUnsetPreferredLocationOperation` (*Class CudaAdviseUnsetPreferredLocationOperation*)
- `public umpire::op::CudaAdviseUnsetReadMostlyOperation` (*Class CudaAdviseUnsetReadMostlyOperation*)
- `public umpire::op::CudaCopyFromOperation` (*Class CudaCopyFromOperation*)
- `public umpire::op::CudaCopyOperation` (*Class CudaCopyOperation*)
- `public umpire::op::CudaCopyToOperation` (*Class CudaCopyToOperation*)
- `public umpire::op::CudaMemPrefetchOperation` (*Class CudaMemPrefetchOperation*)
- `public umpire::op::CudaMemsetOperation` (*Class CudaMemsetOperation*)
- `public umpire::op::GenericReallocateOperation` (*Class GenericReallocateOperation*)
- `public umpire::op::HipCopyFromOperation` (*Class HipCopyFromOperation*)
- `public umpire::op::HipCopyOperation` (*Class HipCopyOperation*)
- `public umpire::op::HipCopyToOperation` (*Class HipCopyToOperation*)
- `public umpire::op::HipMemsetOperation` (*Class HipMemsetOperation*)
- `public umpire::op::HostCopyOperation` (*Class HostCopyOperation*)
- `public umpire::op::HostMemsetOperation` (*Class HostMemsetOperation*)
- `public umpire::op::HostReallocateOperation` (*Class HostReallocateOperation*)
- `public umpire::op::NumaMoveOperation` (*Class NumaMoveOperation*)

Class Documentation

class 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::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*

Public Functions

virtual ~MemoryOperation()

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`*:

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

Apply val to the first length bytes of src_ptr.

Parameters

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

Exceptions

- *`util::Exception`*:

Class MemoryOperationRegistry

- Defined in file_umpire_op_MemoryOperationRegistry.hpp

Class Documentation

class MemoryOperationRegistry

The *MemoryOperationRegistry* serves as a registry for *MemoryOperation* objects. It is a singleton class, typically accessed through the *ResourceManager*.

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

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

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

- "COPY"
- "MEMSET"
- "REALLOCATE"

See *MemoryOperation*

See AllocationStrategy

Public Functions

std::shared_ptr<umpire::*op*::*MemoryOperation*> **find**(**const** std::string *&name*, strategy::*AllocationStrategy* **source_allocator*, strategy::*AllocationStrategy* **dst_allocator*)

Function to find a *MemoryOperation* object.

Finds the *MemoryOperation* object that matches the given name and AllocationStrategy objects. If the requested *MemoryOperation* is not found, this method will throw an Exception.

Parameters

- `name`: Name of operation.
- `src_allocator`: AllocationStrategy of the source allocation.
- `dst_allocator`: AllocationStrategy of the destination allocation.

Exceptions

- *`umpire::util::Exception`*: if the requested *MemoryOperation* is not found.

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

Add a new *MemoryOperation* to the registry.

This object will register the provided *MemoryOperation*, making it available for later retrieval using MemoryOperation::find

Parameters

- `name`: Name of the operation.

- `platforms`: pair of Platforms for the source and destination.
- `operation`: pointer to the *MemoryOperation*.

MemoryOperationRegistry(**const** *MemoryOperationRegistry*&)

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

~MemoryOperationRegistry()

Public Static Functions

MemoryOperationRegistry &**getInstance**()
Get the *MemoryOperationRegistry* singleton instance.

Protected Functions

MemoryOperationRegistry()

Class NumaMoveOperation

- Defined in file_umpire_op_NumaMoveOperation.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class NumaMoveOperation : **public** umpire::*op*::*MemoryOperation*
Relocate a pointer to a different NUMA node.

Public Functions

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

Parameters

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

Exceptions

- *util::Exception*:

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

Parameters

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

Exceptions

- *util::Exception*:

Class Replay

- Defined in file_umpire_Replay.hpp

Class Documentation

class Replay

Public Functions

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

bool **replayLoggingEnabled**()

uint64_t **replayUid**()

Public Static Functions

Replay ***getReplayLogger**()

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

template<typename **T**, typename ...**Args**>
static std::string **printReplayAllocator**(*T* &&*firstArg*, *Args*&&... *args*)

Class CudaConstantMemoryResource

- Defined in file_umpire_resource_CudaConstantMemoryResource.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class CudaConstantMemoryResource : **public** umpire::resource::*MemoryResource*

Public Functions

CudaConstantMemoryResource(**const** std::string &*name*, int *id*, *MemoryResourceTraits traits*)

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

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

Return Pointer to start of allocation.

Parameters

- `bytes`: Number of bytes to allocate.

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

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

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const**
Return the current size of this *MemoryResource*.

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

Return current total size of active allocations in this *MemoryResource*.

std::size_t **getHighWatermark**() **const**
Return the memory high watermark for this *MemoryResource*.

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

Return Memory high watermark.

Platform **getPlatform**()
Get the Platform 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**()

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

Class CudaConstantMemoryResourceFactory

- Defined in file_umpire_resource_CudaConstantMemoryResourceFactory.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Class CudaDeviceResourceFactory

- Defined in file_umpire_resource_CudaDeviceResourceFactory.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Class CudaPinnedMemoryResourceFactory

- Defined in file_umpire_resource_CudaPinnedMemoryResourceFactory.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class CudaPinnedMemoryResourceFactory : **public** umpire::resource::*MemoryResourceFactory*

Class CudaUnifiedMemoryResourceFactory

- Defined in file_umpire_resource_CudaUnifiedMemoryResourceFactory.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class CudaUnifiedMemoryResourceFactory : **public** umpire::resource::*MemoryResourceFactory*
Factory class to construct a *MemoryResource* that uses NVIDIA "unified" memory, 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 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*, *MemoryResourceTraits traits*)

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

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

Return Pointer to start of allocation.

Parameters

- `bytes`: Number of bytes to allocate.

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

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

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const**
Return the current size of this *MemoryResource*.

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

Return current total size of active allocations in this *MemoryResource*.

std::size_t **getHighWatermark**() **const**
Return the memory high watermark for this *MemoryResource*.

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

Return Memory high watermark.

Platform **getPlatform**()
Get the Platform 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**()

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

_allocator **m_allocator**

Platform **m_platform**

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

Class HipConstantMemoryResource

- Defined in file_umpire_resource_HipConstantMemoryResource.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class HipConstantMemoryResource : **public** umpire::resource::*MemoryResource*

Public Functions

HipConstantMemoryResource(**const** std::string &*name*, int *id*, *MemoryResourceTraits traits*)

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

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

Return Pointer to start of allocation.

Parameters

- `bytes`: Number of bytes to allocate.

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

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

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const**
Return the current size of this *MemoryResource*.

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

Return current total size of active allocations in this *MemoryResource*.

std::size_t **getHighWatermark**() **const**
Return the memory high watermark for this *MemoryResource*.

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

Return Memory high watermark.

Platform **getPlatform**()
Get the Platform 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**()

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

Class HipConstantMemoryResourceFactory

- Defined in file_umpire_resource_HipConstantMemoryResourceFactory.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Class HipDeviceResourceFactory

- Defined in file_umpire_resource_HipDeviceResourceFactory.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Class HipPinnedMemoryResourceFactory

- Defined in file_umpire_resource_HipPinnedMemoryResourceFactory.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class HipPinnedMemoryResourceFactory : **public** umpire::resource::*MemoryResourceFactory*

Class HostResourceFactory

- Defined in file_umpire_resource_HostResourceFactory.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class HostResourceFactory : **public** umpire::resource::*MemoryResourceFactory*
Factory class to construct a *MemoryResource* that uses CPU memory.

Class MemoryResource

- Defined in file_umpire_resource_MemoryResource.hpp

Inheritance Relationships

Base Type

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

Derived Types

- `public umpire::resource::CudaConstantMemoryResource` (*Class CudaConstantMemoryResource*)
- `public umpire::resource::DefaultMemoryResource< _allocator >` (*Template Class DefaultMemoryResource*)
- `public umpire::resource::HipConstantMemoryResource` (*Class HipConstantMemoryResource*)
- `public umpire::resource::NullMemoryResource` (*Class NullMemoryResource*)

Class Documentation

class MemoryResource : **public** umpire::strategy::*AllocationStrategy*

Base class to represent the available hardware resources for memory allocation in the system.

Objects of this inherit from *strategy::AllocationStrategy*, allowing them to be used directly.

Subclassed by *umpire::resource::CudaConstantMemoryResource*, *umpire::resource::DefaultMemoryResource< _allocator >*, *umpire::resource::HipConstantMemoryResource*, *umpire::resource::NullMemoryResource*

Public Functions

MemoryResource(**const** std::string *&name*, int *id*, *MemoryResourceTraits traits*)

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

Parameters

- `name`: Name of the *MemoryResource*.
- `id`: ID of the *MemoryResource* (must be unique).

virtual ~MemoryResource()

virtual void ***allocate**(std::size_t *bytes*) = 0

Allocate bytes of memory.

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

Return Pointer to start of allocation.

Parameters

- `bytes`: Number of bytes to allocate.

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

Free the memory at ptr.

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

Parameters

- `ptr`: Pointer to free.

virtual std::size_t **getCurrentSize**() **const** = 0

Return the current size of this *MemoryResource*.

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

Return current total size of active allocations in this *MemoryResource*.

virtual std::size_t **getHighWatermark**() **const** = 0

Return the memory high watermark for this *MemoryResource*.

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

Return Memory high watermark.

virtual *Platform* **getPlatform**() = 0
Get the Platform 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**()

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

Class MemoryResourceFactory

- Defined in file_umpire_resource_MemoryResourceFactory.hpp

Inheritance Relationships

Derived Types

- `public umpire::resource::CudaConstantMemoryResourceFactory` (*Class CudaConstantMemoryResourceFactory*)
- `public umpire::resource::CudaDeviceResourceFactory` (*Class CudaDeviceResourceFactory*)
- `public umpire::resource::CudaPinnedMemoryResourceFactory` (*Class CudaPinnedMemoryResourceFactory*)

- `public umpire::resource::CudaUnifiedMemoryResourceFactory` (*Class CudaUnifiedMemoryResourceFactory*)
- `public umpire::resource::HipConstantMemoryResourceFactory` (*Class HipConstantMemoryResourceFactory*)
- `public umpire::resource::HipDeviceResourceFactory` (*Class HipDeviceResourceFactory*)
- `public umpire::resource::HipPinnedMemoryResourceFactory` (*Class HipPinnedMemoryResourceFactory*)
- `public umpire::resource::HostResourceFactory` (*Class HostResourceFactory*)
- `public umpire::resource::NullMemoryResourceFactory` (*Class NullMemoryResourceFactory*)

Class Documentation

class MemoryResourceFactory

Abstract factory class for constructing *MemoryResource* objects.

Concrete implementations of this class are used by the *MemoryResourceRegistry* to construct *MemoryResource* objects.

See *MemoryResourceRegistry*

Subclassed by *umpire::resource::CudaConstantMemoryResourceFactory*, *umpire::resource::CudaDeviceResourceFactory*, *umpire::resource::CudaPinnedMemoryResourceFactory*, *umpire::resource::CudaUnifiedMemoryResourceFactory*, *umpire::resource::HipConstantMemoryResourceFactory*, *umpire::resource::HipDeviceResourceFactory*, *umpire::resource::HipPinnedMemoryResourceFactory*, *umpire::resource::HostResourceFactory*, *umpire::resource::NullMemoryResourceFactory*

Public Functions

virtual ~MemoryResourceFactory()

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

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

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

Parameters

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

Class MemoryResourceRegistry

- Defined in file_umpire_resource_MemoryResourceRegistry.hpp

Class Documentation

class MemoryResourceRegistry

Public Functions

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

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

MemoryResourceRegistry(**const** *MemoryResourceRegistry*&)

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

~MemoryResourceRegistry()

Public Static Functions

MemoryResourceRegistry &**getInstance**()

Class NullMemoryResource

- Defined in file_umpire_resource_NullMemoryResource.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class NullMemoryResource : **public** umpire::resource::*MemoryResource*

Public Functions

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

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

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

Return Pointer to start of allocation.

Parameters

- `bytes`: Number of bytes to allocate.

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

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

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const**
Return the current size of this *MemoryResource*.

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

Return current total size of active allocations in this *MemoryResource*.

std::size_t **getHighWatermark**() **const**
Return the memory high watermark for this *MemoryResource*.

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

Return Memory high watermark.

Platform **getPlatform**()
Get the Platform 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**()

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

Platform **m_platform**

MemoryResourceTraits **m_traits**

std::string **m_name**

int **m_id**

Class NullMemoryResourceFactory

- Defined in file_umpire_resource_NullMemoryResourceFactory.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Class ResourceManager

- Defined in file_umpire_ResourceManager.hpp

Class Documentation

class ResourceManager

Public Functions

void **initialize**()
Initialize the *ResourceManager*.

This will create all registered MemoryResource objects

std::vector<std::string> **getAllocatorNames**() **const**
Get the names of all available *Allocator* objects.

std::vector<int> **getAllocatorIds**() **const**
Get the ids of all available *Allocator* objects.

Allocator **getAllocator**(**const** std::string &*name*)
Get the *Allocator* with the given name.

Allocator **getAllocator**(**const** char **name*)

Allocator **getAllocator**(resource::*MemoryResourceType resource_type*)
Get the default *Allocator* for the given resource_type.

Allocator **getAllocator**(int *id*)
Get the *Allocator* with the given ID.

Allocator **getDefaultAllocator**()
Get the default *Allocator*.

The default *Allocator* is used whenever an *Allocator* is required and one is not provided, or cannot be inferred.

Return The default *Allocator*.

void **setDefaultAllocator**(*Allocator allocator*)
Set the default *Allocator*.

The default *Allocator* is used whenever an *Allocator* is required and one is not provided, or cannot be inferred.

Parameters

- `allocator`: The *Allocator* to use as the default.

template<typename **Strategy**, bool **introspection** = true, typename ...**Args**>
Allocator **makeAllocator**(**const** std::string &*name*, *Args*&&... *args*)
Construct a new *Allocator*.

void **registerAllocator**(**const** std::string &*name*, *Allocator allocator*)
Register an *Allocator* with the *ResourceManager*.

After registration, the *Allocator* can be retrieved by calling getAllocator(name).

The same *Allocator* can be registered under multiple names.

Parameters

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

Allocator **getAllocator**(void **ptr*)
Get the *Allocator* used to allocate ptr.

Return *Allocator* for the given ptr.

Parameters

- `ptr`: Pointer to find the *Allocator* for.

bool **isAllocator**(**const** std::string &*name*)

bool **hasAllocator**(void **ptr*)
Does the given pointer have an associated *Allocator*.

Return True if the pointer has an associated *Allocator*.

void **registerAllocation**(void **ptr*, util::*AllocationRecord* *record*)
register an allocation with the manager.

util::*AllocationRecord* **deregisterAllocation**(void **ptr*)
de-register the address ptr with the manager.

Return the allocation record removed from the manager.

const util::*AllocationRecord* ***findAllocationRecord**(void **ptr*) **const**
Find the allocation record associated with an address ptr.

Return the record if found, or throws an exception if not found.

bool **isAllocatorRegistered**(**const** std::string *&name*)
Check whether the named *Allocator* exists.

void **copy**(void **dst_ptr*, void **src_ptr*, std::size_t *size* = 0)
Copy size bytes of data from src_ptr to dst_ptr.

Both the src_ptr and dst_ptr addresses must be allocated by Umpire. They can be offset from any Umpire-managed base address.

The dst_ptr must be large enough to accommodate size bytes of data.

Parameters

- `dst_ptr`: Destination pointer.
- `src_ptr`: Source pointer.
- `size`: Size in bytes.

void **memset**(void **ptr*, int *val*, std::size_t *length* = 0)
Set the first length bytes of ptr to the value val.

Parameters

- `ptr`: Pointer to data.
- `val`: Value to set.
- `length`: Number of bytes to set to val.

void ***reallocate**(void **current_ptr*, std::size_t *new_size*)
Reallocate current_ptr to new_size.

If current_ptr is nullptr, then the default allocator will be used to allocate data. The default allocator may be set with a call to *setDefaultAllocator(Allocator allocator)*.

Parameters

- `current_ptr`: Source pointer to reallocate.
- `new_size`: New size of pointer.

NOTE 1: This is not thread safe NOTE 2: If the allocator for which current_ptr is intended is different from the default allocator, then all subsequent reallocate calls will result in allocations from the default allocator which may not be the intended behavior.

If new_size is 0, then the current_ptr will be deallocated if it is not a nullptr, and a zero-byte allocation will be returned.

Return Reallocated pointer.

void ***reallocate**(void *_current_ptr_, std::size_t _new_size_, _Allocator_ _allocator_)
Reallocate current_ptr to new_size.

If current_ptr is null, then allocator will be used to allocate the data.

Parameters

- `current_ptr`: Source pointer to reallocate.
- `new_size`: New size of pointer.
- `allocator`: *Allocator* to use if current_ptr is null.

If new_size is 0, then the current_ptr will be deallocated if it is not a nullptr, and a zero-byte allocation will be returned.

Return Reallocated pointer.

void ***move**(void *_src_ptr_, _Allocator_ _allocator_)
Move src_ptr to memory from allocator.

Return Pointer to new location of data.

Parameters

- `src_ptr`: Pointer to move.
- `allocator`: *Allocator* to use to allocate new memory for moved data.

void **deallocate**(void *_ptr_)
Deallocate any pointer allocated by an Umpire-managed resource.

Parameters

- `ptr`: Pointer to deallocate.

std::size_t **getSize**(void *_ptr_) **const**
Get the size in bytes of the allocation for the given pointer.

Return Size of allocation in bytes.

Parameters

- `ptr`: Pointer to find size of.

std::shared_ptr<_op_::_MemoryOperation_> **getOperation**(**const** std::string &_operation_name_, _Allocator_ _src_allocator_, _Allocator_ _dst_allocator_)

~ResourceManager()

ResourceManager(**const** _ResourceManager_&)

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

Public Static Functions

static *ResourceManager* &**getInstance**()

Class AllocationAdvisor

- Defined in file_umpire_strategy_AllocationAdvisor.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class AllocationAdvisor : **public** umpire::strategy::*AllocationStrategy*

Applies the given MemoryOperation to every allocation.

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

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

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

Public Functions

AllocationAdvisor(**const** std::string &*name*, int *id*, *Allocator* *allocator*, **const** std::string &*advice_operation*, int *device_id* = 0)

AllocationAdvisor(**const** std::string &*name*, int *id*, *Allocator* *allocator*, **const** std::string &*advice_operation*, *Allocator* *accessing_allocator*, int *device_id* = 0)

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

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const**
Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getHighWatermark**() **const**
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Class AllocationPrefetcher

- Defined in file_umpire_strategy_AllocationPrefetcher.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class AllocationPrefetcher : **public** umpire::strategy::*AllocationStrategy*
Apply the appropriate "PREFETCH" operation to every allocation.

Public Functions

AllocationPrefetcher(**const** std::string &*name*, int *id*, *Allocator allocator*, int *device_id* = 0)

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

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const**
Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getHighWatermark**() **const**
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Class AllocationStrategy

- Defined in file_umpire_strategy_AllocationStrategy.hpp

Inheritance Relationships

Derived Types

- `public umpire::resource::MemoryResource` (*Class MemoryResource*)
- `public umpire::strategy::AllocationAdvisor` (*Class AllocationAdvisor*)
- `public umpire::strategy::AllocationPrefetcher` (*Class AllocationPrefetcher*)
- `public umpire::strategy::AllocationTracker` (*Class AllocationTracker*)
- `public umpire::strategy::DynamicPoolList` (*Class DynamicPoolList*)

- public umpire::strategy::DynamicPoolMap (*Class DynamicPoolMap*)
- public umpire::strategy::FixedPool (*Class FixedPool*)
- public umpire::strategy::MixedPool (*Class MixedPool*)
- public umpire::strategy::MonotonicAllocationStrategy (*Class MonotonicAllocationStrategy*)
- public umpire::strategy::NamedAllocationStrategy (*Class NamedAllocationStrategy*)
- public umpire::strategy::NumaPolicy (*Class NumaPolicy*)
- public umpire::strategy::SizeLimiter (*Class SizeLimiter*)
- public umpire::strategy::SlotPool (*Class SlotPool*)
- public umpire::strategy::ThreadSafeAllocator (*Class ThreadSafeAllocator*)
- public umpire::strategy::ZeroByteHandler (*Class ZeroByteHandler*)

Class Documentation

class AllocationStrategy

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

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

Public Functions

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

Construct a new *AllocationStrategy* object.

All *AllocationStrategy* objects must will have a unique name and id. This uniqueness is enforced by the *ResourceManager*.

Parameters

- name: The name of this *AllocationStrategy* object.
- id: The id of this *AllocationStrategy* object.

virtual ~AllocationStrategy()

virtual void ***allocate**(std::size_t *bytes*) = 0

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- bytes: Number of bytes to allocate.

virtual void **deallocate**(void *_ptr_) = 0
Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

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

virtual std::size_t **getCurrentSize**() **const** = 0
Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

virtual std::size_t **getHighWatermark**() **const** = 0
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

virtual *Platform* **getPlatform**() = 0
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Friends

std::ostream &**operator<<**(std::ostream &*os*, **const** AllocationStrategy &*strategy*)

Class AllocationTracker

- Defined in file_umpire_strategy_AllocationTracker.hpp

Inheritance Relationships

Base Types

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

Class Documentation

class AllocationTracker : **public** umpire::strategy::*AllocationStrategy*, **private** umpire::strategy::mixins::*Inspector*

Public Functions

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

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

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Parameters

- `ptr`: Pointer to free.

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

std::size_t **getCurrentSize**() **const**
Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getHighWatermark**() **const**
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

strategy::*AllocationStrategy* ***getAllocationStrategy**()

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Private Functions

void **registerAllocation**(void *ptr, std::size_t *size*, *strategy*::AllocationStrategy **strategy*)

util::*AllocationRecord* **deregisterAllocation**(void *ptr, *strategy*::AllocationStrategy **strategy*)

Private Members

std::size_t **m_current_size**

std::size_t **m_high_watermark**

Class DynamicPoolList

- Defined in file_umpire_strategy_DynamicPoolList.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class DynamicPoolList : **public** umpire::strategy::*AllocationStrategy*

Simple dynamic pool for allocations.

This *AllocationStrategy* uses Simpool to provide pooling for allocations of any size. The behavior of the pool can be controlled by two parameters: the initial allocation size, and the minimum allocation size.

The initial size controls how large the first piece of memory allocated is, and the minimum size controls the lower bound on all future chunk allocations.

Public Types

using CoalesceHeuristic = std::function<bool(**const** strategy::*DynamicPoolList*&)>

Callback Heuristic to trigger coalesce of free blocks in pool.

The registered heuristic callback function will be called immediately after a deallocation() has completed from the pool.

Public Functions

DynamicPoolList(**const** std::string &*name*, int *id*, *Allocator allocator*, **const** std::size_t *min_initial_alloc_size* = (512 * 1024 * 1024), **const** std::size_t *min_alloc_size* = (1 * 1024 * 1024), *CoalesceHeuristic coalesce_heuristic* = *heuristic_percent_releasable_list*(100))

Construct a new *DynamicPoolList*.

Parameters

- `name`: Name of this instance of the *DynamicPoolList*.
- `id`: Id of this instance of the *DynamicPoolList*.
- `min_initial_alloc_size`: The minimum size of the first allocation the pool will make.
- `min_alloc_size`: The minimum size of all future allocations.
- `coalesce_heuristic`: Heuristic callback function.

void *__allocate__(size_t *bytes*)

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

Parameters

- `ptr`: Pointer to free.

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

std::size_t __getCurrentSize__() __const__
Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t __getActualSize__() __const__
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

std::size_t __getHighWatermark__() __const__
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

Platform __getPlatform__()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

std::size_t __getReleasableSize__() __const__
Get the number of bytes that may be released back to resource.

A memory pool has a set of blocks that have no allocations against them. If the size of the set is greater than one, then the pool will have a number of bytes that may be released back to the resource or coalesced into a larger block.

Return The total number of bytes that are releasable

std::size_t **getBlocksInPool**() **const**
Get the number of memory blocks that the pool has.

Return The total number of blocks that are allocated by the pool

std::size_t **getLargestAvailableBlock**() **const**
Get the largest allocatable number of bytes from pool before the pool will grow.

return The largest number of bytes that may be allocated without causing pool growth

void **coalesce**()

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

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Class DynamicPoolMap

- Defined in file_umpire_strategy_DynamicPoolMap.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class DynamicPoolMap : **public** umpire::strategy::*AllocationStrategy*

Simple dynamic pool for allocations.

This *AllocationStrategy* uses Simpool to provide pooling for allocations of any size. The behavior of the pool can be controlled by two parameters: the initial allocation size, and the minimum allocation size.

The initial size controls how large the first piece of memory allocated is, and the minimum size controls the lower bound on all future chunk allocations.

Public Types

using Pointer = void *

using CoalesceHeuristic = std::function<bool (**const** strategy::*DynamicPoolMap*&)>

Callback heuristic to trigger coalesce of free blocks in pool.

The registered heuristic callback function will be called immediately after a deallocation() has completed from the pool.

Public Functions

DynamicPoolMap(**const** std::string *&name*, int *id*, *Allocator allocator*, **const** std::size_t *initial_alloc_size* = (512 * 1024 * 1024), **const** std::size_t *min_alloc_size* = (1 * 1024 * 1024), **const** std::size_t *align_bytes* = 16, *CoalesceHeuristic coalesce_heuristic* = *heuristic_percent_releasable*(100))

Construct a new *DynamicPoolMap*.

Parameters

- `name`: Name of this instance of the *DynamicPoolMap*
- `id`: Unique identifier for this instance
- `initial_alloc_bytes`: Size the pool initially allocates
- `min_alloc_bytes`: The minimum size of all future allocations
- `coalesce_heuristic`: Heuristic callback function
- `align_bytes`: Number of bytes with which to align allocation sizes

~DynamicPoolMap()

Destructs the *DynamicPoolMap*.

DynamicPoolMap(**const** *DynamicPoolMap*&)

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

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Parameters

- `ptr`: Pointer to free.

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

std::size_t **getCurrentSize**() **const**
Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

std::size_t **getHighWatermark**() **const**
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

std::size_t **getReleasableSize**() **const**
Returns the number of bytes of unallocated data held by this pool that could be immediately released back to the resource.

A memory pool has a set of blocks that are not leased out to the application as allocations. Allocations from the resource begin as a single chunk, but these could be split, and only the first chunk can be deallocated back to the resource immediately.

Return The total number of bytes that are immediately releasable.

std::size_t **getFreeBlocks**() **const**
Return the number of free memory blocks that the pools holds.

std::size_t **getInUseBlocks**() **const**
Return the number of used memory blocks that the pools holds.

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

std::size_t **getLargestAvailableBlock**()
Get the largest allocatable number of bytes from pool before the pool will grow.

return The largest number of bytes that may be allocated without causing pool growth

void **coalesce**()
Merge as many free records as possible, release all possible free blocks, then reallocate a chunk to keep the actual size the same.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Class FixedPool

- Defined in file_umpire_strategy_FixedPool.hpp

Nested Relationships

Nested Types

- *Struct FixedPool::Pool*

Inheritance Relationships

Base Type

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

Class Documentation

class FixedPool : **public** umpire::strategy::*AllocationStrategy*

Pool for fixed size allocations.

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

Public Functions

FixedPool(**const** std::string &*name*, int *id*, *Allocator allocator*, **const** std::size_t *object_bytes*, **const** std::size_t *objects_per_pool* = 64 * sizeof(int) * 8)

Constructs a *FixedPool*.

Parameters

- `name`: The allocator name for reference later in *ResourceManager*
- `id`: The allocator id for reference later in *ResourceManager*
- `allocator`: Used for data allocation. It uses std::malloc for internal tracking of these allocations.
- `object_bytes`: The fixed size (in bytes) for each allocation
- `objects_per_pool`: Number of objects in each sub-pool internally. Performance likely improves if this is large, at the cost of memory usage. This does not have to be a multiple of sizeof(int)*8, but it will also likely improve performance if so.

~FixedPool()

FixedPool(**const** *FixedPool*&)

void ***allocate**(std::size_t *bytes* = 0)

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

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

Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getHighWatermark**() **const**
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

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

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

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

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Class MixedPool

- Defined in file_umpire_strategy_MixedPool.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class MixedPool : **public** umpire::strategy::*AllocationStrategy*

A faster pool that pulls from a series of pools.

Pool implementation using a series of FixedPools for small sizes, and a DynamicPool for sizes larger than (1 << 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* *dynamic_coalesce_heuristic* = *heuristic_percent_releasable*(100))

Creates a *MixedPool* of one or more fixed pools and a dynamic pool for large allocations.

Parameters

- `name`: Name of the pool
- `id`: Unique identifier for lookup later in *ResourceManager*
- `allocator`: Underlying allocator
- `smallest_fixed_obj_size`: Smallest fixed pool object size in bytes
- `largest_fixed_obj_size`: Largest fixed pool object size in bytes
- `max_initial_fixed_pool_size`: Largest initial size of any fixed pool
- `fixed_size_multiplier`: Fixed pool object size increase factor
- `dynamic_initial_alloc_size`: Size the dynamic pool initially allocates
- `dynamic_min_alloc_bytes`: Minimum size of all future allocations in the dynamic pool
- `dynamic_align_bytes`: Size with which to align allocations (for the dynamic pool)
- `coalesce_heuristic`: Heuristic callback function (for the dynamic pool)

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

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Parameters

- `ptr`: Pointer to free.

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

std::size_t **getCurrentSize**() **const**
Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

std::size_t **getHighWatermark**() **const**
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Class Inspector

- Defined in file_umpire_strategy_mixins_Inspector.hpp

Inheritance Relationships

Derived Type

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

Class Documentation

class Inspector
Subclassed by *umpire::strategy::AllocationTracker*

Public Functions

Inspector()

void **registerAllocation**(void **ptr*, std::size_t *size*, *strategy*::AllocationStrategy **strategy*)

util::*AllocationRecord* **deregisterAllocation**(void **ptr*, *strategy*::AllocationStrategy **strategy*)

Protected Attributes

std::size_t **m_current_size**

std::size_t **m_high_watermark**

Class MonotonicAllocationStrategy

- Defined in file_umpire_strategy_MonotonicAllocationStrategy.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

MonotonicAllocationStrategy(**const** std::string &*name*, int *id*, *Allocator* *allocator*, std::size_t *capacity*)

~MonotonicAllocationStrategy()

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

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const**
Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getHighWatermark**() **const**
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Class NamedAllocationStrategy

- Defined in file_umpire_strategy_NamedAllocationStrategy.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

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

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

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const**
Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getHighWatermark**() **const**
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

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

std::string **m_name**

int **m_id**

Class NumaPolicy

- Defined in file_umpire_strategy_NumaPolicy.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class NumaPolicy : **public** umpire::strategy::*AllocationStrategy*

Use NUMA interface to locate memory to a specific NUMA node.

This *AllocationStrategy* provides a method of ensuring memory sits on a specific NUMA node. This can be used either for optimization, or for moving memory between the host and devices.

Public Functions

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

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

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

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

Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getHighWatermark**() **const**

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

int **getNode**() **const**

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Class SizeLimiter

- Defined in file_umpire_strategy_SizeLimiter.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class SizeLimiter : **public** umpire::strategy::*AllocationStrategy*

An allocator with a limited total size.

Using this *AllocationStrategy* with another can be a good way to limit the total size of allocations made on a particular resource or from a particular context.

Public Functions

SizeLimiter(**const** std::string &*name*, int *id*, *Allocator* *allocator*, std::size_t *size_limit*)

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

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

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

Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getHighWatermark**() **const**

Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

Platform **getPlatform**()

Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

void **release**()

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Class SlotPool

- Defined in file_umpire_strategy_SlotPool.hpp

Inheritance Relationships

Base Type

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

Class Documentation

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

Public Functions

SlotPool(**const** std::string &*name*, int *id*, *Allocator allocator*, std::size_t *slots*)

~SlotPool()

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

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Parameters

- `ptr`: Pointer to free.

std::size_t **getCurrentSize**() **const**
Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getHighWatermark**() **const**
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Class ThreadSafeAllocator

- Defined in file_umpire_strategy_ThreadSafeAllocator.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class ThreadSafeAllocator : **public** umpire::strategy::*AllocationStrategy*

Make an *Allocator* thread safe.

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

Public Functions

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

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

Allocate bytes of memory.

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Free the memory at ptr.

Parameters

- `ptr`: Pointer to free.

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

Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getHighWatermark**() **const**
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

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

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

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

std::mutex **m_mutex**

std::string **m_name**

int **m_id**

Class ZeroByteHandler

- Defined in file_umpire_strategy_ZeroByteHandler.hpp

Inheritance Relationships

Base Type

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

Class Documentation

class ZeroByteHandler : **public** umpire::strategy::*AllocationStrategy*

Public Functions

ZeroByteHandler(std::unique_ptr<*AllocationStrategy*> &&*allocator*)

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

Return Pointer to start of allocated bytes.

Parameters

- `bytes`: Number of bytes to allocate.

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

Parameters

- `ptr`: Pointer to free.

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

std::size_t **getCurrentSize**() **const**
Get current (total) size of the allocated memory.

This is the total size of all allocation currently 'live' that have been made by this *AllocationStrategy* object.

Return Current total size of allocations.

std::size_t **getHighWatermark**() **const**
Get the high watermark of the total allocated size.

This is equivalent to the highest observed value of getCurrentSize.

Return High watermark allocation size.

std::size_t **getActualSize**() **const**
Get the current amount of memory allocated by this allocator.

Note that this can be larger than *getCurrentSize()*, particularly if the *AllocationStrategy* implements some kind of pooling.

Return The total size of all the memory this object has allocated.

Platform **getPlatform**()
Get the platform associated with this *AllocationStrategy*.

The Platform distinguishes the appropriate place to execute operations on memory allocated by this *AllocationStrategy*.

Return The platform associated with this *AllocationStrategy*.

strategy::*AllocationStrategy* ***getAllocationStrategy**()

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

Return The name of this *AllocationStrategy*.

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

Return The id of this *AllocationStrategy*.

Protected Attributes

std::string **m_name**

int **m_id**

Template Class TypedAllocator

- Defined in file_umpire_TypedAllocator.hpp

Class Documentation

template<typename **T**>
class TypedAllocator
Allocator for objects of type T.

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

Public Types

typedef T **value_type**

Public Functions

TypedAllocator(*Allocator allocator*)
Construct a new *TypedAllocator* that will use allocator to allocate data.

Parameters

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

template<typename **U**>
TypedAllocator(**const** *TypedAllocator*<*U*> &*other*)

T ***allocate**(std::size_t *size*)

void **deallocate**(T **ptr*, std::size_t *size*)
Deallocate ptr, the passed size is ignored.

Parameters

- `ptr`: Pointer to deallocate
- `size`: Size of allocation (ignored).

Friends

friend umpire::TypedAllocator::TypedAllocator

Class AllocationMap

- Defined in file_umpire_util_AllocationMap.hpp

Nested Relationships

Nested Types

- *Class AllocationMap::ConstIterator*
- *Class AllocationMap::RecordList*
- *Template Struct RecordList::Block*
- *Class RecordList::ConstIterator*

Class Documentation

class AllocationMap

Public Types

using Map = MemoryMap<*RecordList*>

Public Functions

AllocationMap()

AllocationMap(**const** *AllocationMap*&)

void **insert**(void **ptr*, *AllocationRecord record*)

const *AllocationRecord* ***find**(void **ptr*) **const**

AllocationRecord ***find**(void **ptr*)

const *AllocationRecord* ***findRecord**(void **ptr*) **const**

AllocationRecord ***findRecord**(void **ptr*)

AllocationRecord **remove**(void **ptr*)

bool **contains**(void **ptr*) **const**

void **clear**()

std::size_t **size**() **const**

void **print**(**const** std::function<bool) **const** *AllocationRecord*&
> &&*predicate*, std::ostream &*os* = std::cout **const**

void **printAll**(std::ostream &*os* = std::cout) **const**

AllocationMap::*ConstIterator* **begin**() **const**

AllocationMap::*ConstIterator* **end**() **const**

class ConstIterator : **public** std::iterator<std::forward_iterator_tag, *AllocationRecord*>

Public Functions

ConstIterator(**const** *AllocationMap *map*, *iterator_begin*)

ConstIterator(**const** *AllocationMap *map*, *iterator_end*)

ConstIterator(**const** *ConstIterator*&)

const *AllocationRecord* &**operator***()

const *AllocationRecord* ***operator->**()

AllocationMap::*ConstIterator* &**operator++**()

AllocationMap::*ConstIterator* **operator++**(int)

bool **operator==**(**const** *ConstIterator* &*other*) **const**

bool **operator!=**(**const** *ConstIterator* &*other*) **const**

Class AllocationMap::ConstIterator

- Defined in file_umpire_util_AllocationMap.hpp

Nested Relationships

This class is a nested type of *Class AllocationMap*.

Inheritance Relationships

Base Type

- `public std::iterator< std::forward_iterator_tag, AllocationRecord >`

Class Documentation

class ConstIterator : **public** std::iterator<std::forward_iterator_tag, *AllocationRecord*>

Public Functions

ConstIterator(**const** *AllocationMap* **map*, *iterator_begin*)

ConstIterator(**const** *AllocationMap* **map*, *iterator_end*)

ConstIterator(**const** *ConstIterator*&)

const *AllocationRecord* &**operator***()

const *AllocationRecord* ***operator->**()

AllocationMap::*ConstIterator* &**operator++**()

AllocationMap::*ConstIterator* **operator++**(int)

bool **operator==**(**const** *ConstIterator* &*other*) **const**

bool **operator!=**(**const** *ConstIterator* &*other*) **const**

Class AllocationMap::RecordList

- Defined in file_umpire_util_AllocationMap.hpp

Nested Relationships

This class is a nested type of *Class AllocationMap*.

Nested Types

- *Template Struct RecordList::Block*
- *Class RecordList::ConstIterator*

Class Documentation

class RecordList

Public Types

using RecordBlock = Block<*AllocationRecord*>

Public Functions

RecordList(*AllocationMap* &*map*, *AllocationRecord* *record*)

~RecordList()

void **push_back**(**const** *AllocationRecord* &*rec*)

AllocationRecord **pop_back**()

AllocationMap::*RecordList*::*ConstIterator* **begin**() **const**

AllocationMap::*RecordList*::*ConstIterator* **end**() **const**

std::size_t **size**() **const**

bool **empty**() **const**

AllocationRecord ***back**()

const *AllocationRecord* ***back**() **const**

template<typename **T**>
struct Block

Public Members

T **rec**

Block ***prev**

class ConstIterator : **public** std::iterator<std::forward_iterator_tag, *AllocationRecord*>

Public Functions

ConstIterator()

ConstIterator(**const** *RecordList* *list, *iterator_begin*)

ConstIterator(**const** *RecordList* *list, *iterator_end*)

ConstIterator(**const** *ConstIterator*&)

const *AllocationRecord* &**operator***()

const *AllocationRecord* ***operator->**()

AllocationMap::*RecordList*::*ConstIterator* &**operator++**()

AllocationMap::*RecordList*::*ConstIterator* **operator++**(int)

bool **operator==**(**const** *ConstIterator* &*other*) **const**

bool **operator!=**(**const** *ConstIterator* &*other*) **const**

Class RecordList::ConstIterator

- Defined in file_umpire_util_AllocationMap.hpp

Nested Relationships

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

Inheritance Relationships

Base Type

- `public std::iterator< std::forward_iterator_tag, AllocationRecord >`

Class Documentation

class ConstIterator : **public** std::iterator<std::forward_iterator_tag, *AllocationRecord*>

Public Functions

ConstIterator()

ConstIterator(**const** *RecordList* *list, *iterator_begin*)

ConstIterator(**const** *RecordList* *list, *iterator_end*)

ConstIterator(**const** *ConstIterator*&)

const *AllocationRecord* &**operator***()

const *AllocationRecord* ***operator->**()

AllocationMap::*RecordList*::*ConstIterator* &**operator++**()

AllocationMap::*RecordList*::*ConstIterator* **operator++**(int)

bool **operator==**(**const** *ConstIterator* &*other*) **const**

bool **operator!=**(**const** *ConstIterator* &*other*) **const**

Class Exception

- Defined in file_umpire_util_Exception.hpp

Inheritance Relationships

Base Type

- `public std::exception`

Class Documentation

class Exception : **public** std::exception

Public Functions

Exception(**const** std::string &*msg*, **const** std::string &*file*, int *line*)

virtual ~Exception()

std::string **message**() **const**

const char ***what**() **const**

Class FixedMallocPool

- Defined in file_umpire_util_FixedMallocPool.hpp

Nested Relationships

Nested Types

- *Struct FixedMallocPool::Pool*

Class Documentation

class FixedMallocPool

Pool for fixed size allocations using *malloc()*

Another version of this class exists in umpire::strategy, but this version does not rely on *Allocator* and all the memory tracking statistics, so it is useful for building objects in umpire::util.

Public Functions

FixedMallocPool(**const** std::size_t *object_bytes*, **const** std::size_t *objects_per_pool* = 1024 * 1024)

~FixedMallocPool()

void ***allocate**(std::size_t *bytes* = 0)

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

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

Class Logger

- Defined in file_umpire_util_Logger.hpp

Class Documentation

class Logger

Public Functions

void **setLoggingMsgLevel**(message::*Level level*)

void **logMessage**(*message*::Level *level*, **const** std::string &*message*, **const** std::string &*fileName*, int *line*)

bool **logLevelEnabled**(message::*Level level*)

~Logger()

Logger(**const** *Logger*&)

Logger &**operator=**(**const** *Logger*&)

Public Static Functions

static void **initialize**()

static void **finalize**()

Logger ***getActiveLogger**()

Template Class MemoryMap

- Defined in file_umpire_util_MemoryMap.hpp

Nested Relationships

Nested Types

- *Template Class MemoryMap::Iterator_*

Class Documentation

template<typename **V**>
class MemoryMap

A fast replacement for std::map<void*,Value> for a generic Value.

This uses *FixedMallocPool* and Judy arrays and provides forward const and non-const iterators.

Public Types

template<>
using Key = void *

template<>
using Value = V

template<>
using KeyValuePair = std::pair<*Key*, *Value* *>

template<>
using Iterator = Iterator_<false>

template<>
using ConstIterator = Iterator_<true>

Public Functions

MemoryMap()

~MemoryMap()

MemoryMap(**const** *MemoryMap*&)

std::pair<**typename** MemoryMap<V>::Iterator, bool> **insert**(Key *ptr*, **const** Value &*val*)
Insert Value at ptr in the map if ptr does not exist. Uses copy constructor on Value once.

Return Pair of iterator position into map and boolean value whether entry was added. The iterator will be set to *end()* if no insertion was made.

template<typename **P**>
std::pair<Iterator, bool> **insert**(*P* &&*pair*)
Insert a key-value pair if pair.first does not exist as a key. Must have first and second fields. Calls the first version.

Return See alternative version.

template<typename ...**Args**>
std::pair<Iterator, bool> **insert**(Key *ptr*, *Args*&&... *args*)
Emplaces a new value at ptr in the map, forwarding args to the placement new constructor.

Return See alternative version.

MemoryMap<V>::Iterator **findOrBefore**(Key *ptr*)
Find a value at ptr.

Return iterator into map at ptr or preceeding position.

MemoryMap<V>::ConstIterator **findOrBefore**(Key *ptr*) **const**

MemoryMap<V>::Iterator **find**(Key *ptr*)
Find a value at ptr.

Return iterator into map at ptr or *end()* if not found.

MemoryMap<V>::ConstIterator **find**(Key *ptr*) **const**

MemoryMap<V>::ConstIterator **begin**() **const**
Iterator to first value or *end()* if empty.

MemoryMap<V>::Iterator **begin**()

MemoryMap<V>::ConstIterator **end**() **const**
Iterator to one-past-last value.

MemoryMap<V>::Iterator **end**()

void **erase**(Key *ptr*)
Remove an entry from the map.

void **erase**(Iterator *iter*)

void **erase**(ConstIterator *oter*)

void **removeLast**()
Remove/deallocate the last found entry.

WARNING: Use this with caution, only directly after using a method above. *erase(Key)* is safer, but requires an additional lookup.

void **clear**()
Clear all entris from the map.

std::size_t **size**() **const**
Return number of entries in the map.

template<typename ...**Args**>
std::pair<**typename** MemoryMap<V>::Iterator, bool> **doInsert**(Key *ptr*, *Args*&&... *args*)

template<typename **P**>
std::pair<**typename** MemoryMap<V>::Iterator, bool> **insert**(*P* &&*pair*)

template<typename ...**Args**>
std::pair<**typename** MemoryMap<V>::Iterator, bool> **insert**(Key *ptr*, *Args*&&... *args*)

Friends

friend umpire::util::MemoryMap::Iterator_

template<bool **Const** = false>
class Iterator_ : **public** std::iterator<std::forward_iterator_tag, Value>

Public Types

template<>
template<>
using Map = **typename** std::conditional<Const, **const** MemoryMap<*Value*>, MemoryMap<*Value*>>::type

template<>
template<>
using ValuePtr = **typename** std::conditional<Const, **const** *Value* *, *Value* *>::type

template<>
template<>
using Content = std::pair<*Key*, *ValuePtr*>

template<>
template<>
using Reference = **typename** std::conditional<Const, **const** *Content*&, *Content*&>::type

template<>
template<>
using Pointer = **typename** std::conditional<Const, **const** *Content* *, *Content* *>::type

Public Functions

template<>
Iterator_(Map *map, *Key* *ptr*)

template<>
Iterator_(Map *map, *iterator_begin*)

template<>
Iterator_(Map *map, *iterator_end*)

template<bool **OtherConst**>
Iterator_(**const** *Iterator_*<OtherConst> &*other*)

template<>
MemoryMap<V>::template Iterator_<Const>::*Reference* **operator***()

template<>
MemoryMap<V>::template Iterator_<Const>::*Pointer* **operator->**()

template<>
MemoryMap<V>::template Iterator_<Const> &**operator++**()

template<>
MemoryMap<V>::template Iterator_<Const> **operator++**(int)

template<bool **OtherConst**>
bool **operator==**(**const** Iterator_<OtherConst> &*other*) **const**

template<bool **OtherConst**>
bool **operator!=**(**const** Iterator_<OtherConst> &*other*) **const**

template<bool **OtherConst**>
bool **operator==**(**const** MemoryMap<V>::Iterator_<OtherConst> &*other*) **const**

template<bool **OtherConst**>
bool **operator!=**(**const** MemoryMap<V>::Iterator_<OtherConst> &*other*) **const**

Template Class MemoryMap::Iterator_

- Defined in file_umpire_util_MemoryMap.hpp

Nested Relationships

This class is a nested type of *Template Class MemoryMap*.

Inheritance Relationships

Base Type

- `public std::iterator< std::forward_iterator_tag, Value >`

Class Documentation

template<bool **Const** = false>
class Iterator_ : **public** std::iterator<std::forward_iterator_tag, Value>

Public Types

template<>
template<>
using Map = **typename** std::conditional<Const, **const** MemoryMap<*Value*>, MemoryMap<*Value*>>::type

template<>
template<>
using ValuePtr = **typename** std::conditional<Const, **const** *Value* *, *Value* *>::type

template<>
template<>
using Content = std::pair<*Key*, *ValuePtr*>

template<>
template<>
using Reference = **typename** std::conditional<Const, **const** *Content*&, *Content*&>::type

template<>
template<>
using Pointer = **typename** std::conditional<Const, **const** *Content* *, *Content* *>::type

Public Functions

template<>
Iterator_(Map **map*, *Key ptr*)

template<>
Iterator_(Map **map*, *iterator_begin*)

template<>
Iterator_(Map **map*, *iterator_end*)

template<bool **OtherConst**>
Iterator_(**const** *Iterator_*<OtherConst> &*other*)

template<>
MemoryMap<V>::template Iterator_<Const>::*Reference* **operator***()

template<>
MemoryMap<V>::template Iterator_<Const>::*Pointer* **operator->**()

template<>

MemoryMap<V>::template Iterator_<Const> &**operator++**()

template<>
MemoryMap<V>::template Iterator_<Const> **operator++**(int)

template<bool **OtherConst**>
bool **operator==**(**const** Iterator_<OtherConst> &*other*) **const**

template<bool **OtherConst**>
bool **operator!=**(**const** Iterator_<OtherConst> &*other*) **const**

template<bool **OtherConst**>
bool **operator==**(**const** MemoryMap<V>::Iterator_<OtherConst> &*other*) **const**

template<bool **OtherConst**>
bool **operator!=**(**const** MemoryMap<V>::Iterator_<OtherConst> &*other*) **const**

Class MPI

- Defined in file_umpire_util_MPI.hpp

Class Documentation

class MPI

Public Static Functions

void **initialize**()

void **finalize**()

int **getRank**()

int **getSize**()

void **sync**()

void **logMpiInfo**()

bool **isInitialized**()

Class OutputBuffer

- Defined in file_umpire_util_OutputBuffer.hpp

Inheritance Relationships

Base Type

- `public streambuf`

Class Documentation

class OutputBuffer : **public** streambuf

Public Functions

OutputBuffer()

~OutputBuffer()

void **setConsoleStream**(std::ostream *_stream_)

void **setFileStream**(std::ostream *_stream_)

int **overflow**(int _ch_)

int **sync**()

Class Statistic

- Defined in file_umpire_util_Statistic.hpp

Class Documentation

class Statistic

Public Functions

~Statistic()

void **recordStatistic**(conduit::Node &&_n_)

void **printData**(std::ostream &_stream_)

Protected Functions

Statistic(**const** std::string &_name_)

Class StatisticsDatabase

- Defined in file_umpire_util_StatisticsDatabase.hpp

Class Documentation

class StatisticsDatabase

Public Functions

std::shared_ptr<*Statistic*> **getStatistic**(**const** std::string &*name*)

void **printStatistics**(std::ostream &*stream*)

Public Static Functions

StatisticsDatabase ***getDatabase**()

6.3.3 Enums

Enum Platform

- Defined in file_umpire_util_Platform.hpp

Enum Documentation

enum umpire::**Platform**

Values:

none

cpu

cuda

hip

Enum MemoryResourceType

- Defined in file_umpire_resource_MemoryResourceTypes.hpp

Enum Documentation

enum umpire::resource::**MemoryResourceType**
Values:

Host

Device

Unified

Pinned

Constant

Enum Level

- Defined in file_umpire_util_Logger.hpp

Enum Documentation

enum umpire::util::message::**Level**
Values:

Error

Warning

Info

Debug

Num_Levels

6.3.4 Functions

Function find_first_set

- Defined in file_umpire_strategy_FixedSizePool.hpp

Function Documentation

int umpire::strategy::**find_first_set**(int *i*)

Function genumpiresplicer::gen_bounds

- Defined in file_umpire_interface_c_fortran_genumpiresplicer.py

Function Documentation

genumpiresplicer.**gen_bounds**()

Function genumpiresplicer::gen_fortran

- Defined in file_umpire_interface_c_fortran_genumpiresplicer.py

Function Documentation

genumpiresplicer.**gen_fortran**()

Function genumpiresplicer::gen_methods

- Defined in file_umpire_interface_c_fortran_genumpiresplicer.py

Function Documentation

genumpiresplicer.**gen_methods**()

Function round_up

- Defined in file_umpire_strategy_DynamicPoolMap.cpp

Function Documentation

Warning: doxygenfunction: Cannot find function "round_up" in doxygen xml output for project "umpire" from directory: ../doxygen/xml/

Function shroud_c_loc

- Defined in file_umpire_interface_c_fortran_shroudrt.cpp

Function Documentation

Warning: doxygenfunction: Cannot find function "shroud_c_loc" in doxygen xml output for project "umpire" from directory: ../doxygen/xml/

Function shroud_c_loc_

- Defined in file_umpire_interface_c_fortran_shroudrt.cpp

Function Documentation

Warning: doxygenfunction: Cannot find function “shroud_c_loc_” in doxygen xml output for project “umpire” from directory: ../doxygen/xml/

Function shroud_FccCopy(char *, int, const char *)

- Defined in file_umpire_interface_c_fortran_shroudrt.cpp

Function Documentation

void **shroud_FccCopy**(char *a, int la, **const** char *s)

Function shroud_FccCopy(char *, int, const char *)

- Defined in file_umpire_interface_c_fortran_shroudrt.hpp

Function Documentation

void **shroud_FccCopy**(char *a, int la, **const** char *s)

Function umpire::cpu_vendor_type

- Defined in file_umpire_util_detect_vendor.cpp

Function Documentation

MemoryResourceTraits::*vendor_type* umpire::**cpu_vendor_type**()

Function umpire::error

- Defined in file_umpire_util_io.cpp

Function Documentation

std::ostream &umpire::**error**()

Function umpire::finalize

- Defined in file_umpire_Umpire.hpp

Function Documentation

void `umpire::finalize`()

Function umpire::free

- Defined in file_umpire_Umpire.hpp

Function Documentation

void `umpire::free`(void *_ptr_)

Free any memory allocated with Umpire.

This method is a convenience wrapper around calls to the *ResourceManager*, it can be used to free allocations from any MemorySpace. *

Parameters

- `ptr`: Address to free.

Function umpire::get_allocator_records

- Defined in file_umpire_Umpire.cpp

Function Documentation

std::vector<util::*AllocationRecord*> `umpire::get_allocator_records`(*Allocator* *allocator*)

Returns vector of AllocationRecords created by the allocator.

Parameters

- `allocator`: source *Allocator*.

Function umpire::get_major_version

- Defined in file_umpire_Umpire.hpp

Function Documentation

int umpire::**get_major_version**()

Function umpire::get_minor_version

- Defined in file_umpire_Umpire.hpp

Function Documentation

int umpire::**get_minor_version**()

Function umpire::get_page_size

- Defined in file_umpire_util_numa.cpp

Function Documentation

long umpire::**get_page_size**()

Function umpire::get_patch_version

- Defined in file_umpire_Umpire.hpp

Function Documentation

int umpire::**get_patch_version**()

Function umpire::get_rc_version

- Defined in file_umpire_Umpire.hpp

Function Documentation

std::string umpire::**get_rc_version**()

Function umpire::initialize

- Defined in file_umpire_Umpire.hpp

Function Documentation

void umpire::**initialize**()

Function umpire::log

- Defined in file_umpire_util_io.cpp

Function Documentation

std::ostream &umpire::**log**()

Function umpire::malloc

- Defined in file_umpire_Umpire.hpp

Function Documentation

void *umpire::**malloc**(std::size_t *size*)

Allocate memory in the default space, with the default allocator.

This method is a convenience wrapper around calls to the *ResourceManager* to allocate memory in the default MemorySpace.

Parameters

- size: Number of bytes to allocate.

Function umpire::numa::get_allocatable_nodes

- Defined in file_umpire_util_numa.cpp

Function Documentation

std::vector<int> umpire::numa::**get_allocatable_nodes**()

Function umpire::numa::get_device_nodes

- Defined in file_umpire_util_numa.cpp

Function Documentation

std::vector<int> umpire::numa::**get_device_nodes**()

Function umpire::numa::get_host_nodes

- Defined in file_umpire_util_numa.cpp

Function Documentation

std::vector<int> umpire::numa::**get_host_nodes**()

Function umpire::numa::get_location

- Defined in file_umpire_util_numa.cpp

Function Documentation

int umpire::numa::**get_location**(void *_ptr_)

Function umpire::numa::move_to_node

- Defined in file_umpire_util_numa.cpp

Function Documentation

void umpire::numa::**move_to_node**(void *_ptr_, std::size_t _bytes_, int _node_)

Function umpire::numa::preferred_node

- Defined in file_umpire_util_numa.cpp

Function Documentation

int umpire::numa::**preferred_node**()

Function umpire::operator<<(std::ostream&, const Allocator&)

- Defined in file_umpire_Allocator.cpp

Function Documentation

std::ostream &umpire::**operator<<**(std::ostream &*os*, **const** *Allocator* &*allocator*)

Function umpire::operator<<(std::ostream&, umpire::Allocator&)

- Defined in file_umpire_Replay.cpp

Function Documentation

std::ostream &umpire::**operator<<**(std::ostream &*out*, umpire::*Allocator* &*alloc*)

Function umpire::operator<<(std::ostream&, umpire::strategy::DynamicPoolMap::CoalesceHeuristic&)

- Defined in file_umpire_Replay.cpp

Function Documentation

std::ostream &umpire::**operator<<**(std::ostream &*out*, umpire::strategy::*DynamicPoolMap*::*CoalesceHeuristic*&)

Function umpire::operator<<(std::ostream&, umpire::strategy::DynamicPoolList::CoalesceHeuristic&)

- Defined in file_umpire_Replay.cpp

Function Documentation

std::ostream &umpire::**operator<<**(std::ostream &*out*, umpire::strategy::*DynamicPoolList*::*CoalesceHeuristic*&)

Function umpire::print_allocator_records

- Defined in file_umpire_Umpire.cpp

Function Documentation

void umpire::**print_allocator_records**(*Allocator allocator*, std::ostream &*os* = std::cout)
Print the allocations from a specific allocator in a human-readable format.

Parameters

- `allocator`: source *Allocator*.
- `os`: output stream

Function umpire::replay

- Defined in file_umpire_util_io.cpp

Function Documentation

std::ostream &umpire::**replay**()

Function umpire::strategy::find_first_set

- Defined in file_umpire_strategy_FixedPool.cpp

Function Documentation

int umpire::strategy::**find_first_set**(int *i*)

Function umpire::strategy::heuristic_percent_releasable

- Defined in file_umpire_strategy_DynamicPoolHeuristic.cpp

Function Documentation

std::function<bool (**const** strategy::*DynamicPoolMap*&)> umpire::strategy::**heuristic_percent_releasable**
int *percentage*

Function umpire::strategy::heuristic_percent_releasable_list

- Defined in file_umpire_strategy_DynamicPoolHeuristic.cpp

Function Documentation

std::function<bool (**const** strategy::*DynamicPoolList*&)> umpire::strategy::**heuristic_percent_releasable_list**
int *percentage*

Function umpire::strategy::operator<<

- Defined in file_umpire_strategy_AllocationStrategy.cpp

Function Documentation

std::ostream &umpire::strategy::**operator<<**(std::ostream &*os*, **const** *AllocationStrategy* &*strategy*)

Function umpire::util::case_insensitive_match

- Defined in file_umpire_util_Logger.cpp

Function Documentation

static int umpire::util::**case_insensitive_match**(**const** std::string *s1*, **const** std::string *s2*)

Function umpire::util::detail::add_entry(conduit::Node&)

- Defined in file_umpire_util_statistic_helper.hpp

Function Documentation

conduit::Node umpire::util::detail::**add_entry**(conduit::Node &*n*)

Template Function umpire::util::detail::add_entry(conduit::Node&, T, U)

- Defined in file_umpire_util_statistic_helper.hpp

Function Documentation

template<typename **T**, typename **U**>
conduit::Node umpire::util::detail::**add_entry**(conduit::Node &*n*, *T* *k*, *U* *v*)

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

- Defined in file_umpire_util_statistic_helper.hpp

Function Documentation

template<typename **T**, typename **U**, typename ...**Args**>
conduit::Node umpire::util::detail::**add_entry**(conduit::Node &*n*, *T* *k*, *U* *v*, *Args*... *args*)

Template Function umpire::util::detail::record_statistic

- Defined in file_umpire_util_statistic_helper.hpp

Function Documentation

template<typename ...**Args**>
void umpire::util::detail::**record_statistic**(**const** std::string &*name*, *Args*&&... *args*)

Function umpire::util::directory_exists

- Defined in file_umpire_util_io.cpp

Function Documentation

static bool umpire::util::**directory_exists**(**const** std::string &*file*)

Template Function umpire::util::do_wrap(std::unique_ptr<Base>&&)

- Defined in file_umpire_util_wrap_allocator.hpp

Function Documentation

template<typename **Base**>
std::unique_ptr<*Base*> umpire::util::**do_wrap**(std::unique_ptr<*Base*> &&*allocator*)

Template Function umpire::util::do_wrap(std::unique_ptr<Base>&&)

- Defined in file_umpire_util_wrap_allocator.hpp

Function Documentation

template<typename **Base**>
std::unique_ptr<*Base*> umpire::util::**do_wrap**(std::unique_ptr<*Base*> &&*allocator*)

Function umpire::util::file_exists

- Defined in file_umpire_util_io.cpp

Function Documentation

static bool umpire::util::**file_exists**(**const** std::string &*file*)

Function umpire::util::flush_files

- Defined in file_umpire_util_io.cpp

Function Documentation

void umpire::util::**flush_files**()
Synchronize all stream buffers to their respective output sequences. This function is usually called by exception generating code like UMPIRE_ERROR.

Function umpire::util::initialize_io

- Defined in file_umpire_util_io.cpp

Function Documentation

void umpire::util::**initialize_io**(**const** bool *enable_log*, **const** bool *enable_replay*)
Initialize the streams. This method is called when ResourceManger is initialized. Do not call this manually.

Template Function umpire::util::make_unique

- Defined in file_umpire_util_make_unique.hpp

Function Documentation

template<typename **T**, typename ...**Args**>
constexpr std::unique_ptr<*T*> umpire::util::**make_unique**(*Args*&&... *args*)

Function umpire::util::make_unique_filename

- Defined in file_umpire_util_io.cpp

Function Documentation

static std::string umpire::util::**make_unique_filename**(**const** std::string &*base_dir*, **const** std::string &*name*, **const** int *pid*, **const** std::string &*extension*)

Function umpire::util::relative_fragmentation

- Defined in file_umpire_util_allocation_statistics.cpp

Function Documentation

float umpire::util::**relative_fragmentation**(std::vector<util::*AllocationRecord*> &*recs*)

Compute the relative fragmentation of a set of allocation records.

Fragmentation = 1 - (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_current_size(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.cpp

Function Documentation

size_t **umpire_allocator_get_current_size**(*umpire_allocator* **self*)

Function umpire_allocator_get_current_size(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.h

Function Documentation

size_t **umpire_allocator_get_current_size**(*umpire_allocator* **self*)

Function umpire_allocator_get_high_watermark(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.cpp

Function Documentation

size_t **umpire_allocator_get_high_watermark**(*umpire_allocator* **self*)

Function umpire_allocator_get_high_watermark(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.h

Function Documentation

size_t **umpire_allocator_get_high_watermark**(*umpire_allocator* **self*)

Function umpire_allocator_get_id(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.cpp

Function Documentation

size_t **umpire_allocator_get_id**(*umpire_allocator* **self*)

Function umpire_allocator_get_id(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.h

Function Documentation

size_t **umpire_allocator_get_id**(*umpire_allocator* **self*)

Function umpire_allocator_get_name(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.cpp

Function Documentation

const char ***umpire_allocator_get_name**(*umpire_allocator* **self*)

Function umpire_allocator_get_name(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.h

Function Documentation

const char ***umpire_allocator_get_name**(*umpire_allocator* **self*)

Function umpire_allocator_get_name_bufferify(umpire_allocator *, umpire_SHROUD_array *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.cpp

Function Documentation

void **umpire_allocator_get_name_bufferify**(*umpire_allocator* **self*, *umpire_SHROUD_array* **DSHF_rv*)

Function umpire_allocator_get_name_bufferify(umpire_allocator *, umpire_SHROUD_array *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.h

Function Documentation

void **umpire_allocator_get_name_bufferify**(*umpire_allocator* **self*, *umpire_SHROUD_array* **DSHF_rv*)

Function umpire_allocator_get_size(umpire_allocator *, void *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.cpp

Function Documentation

size_t **umpire_allocator_get_size**(*umpire_allocator* **self*, void **ptr*)

Function umpire_allocator_get_size(umpire_allocator *, void *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.h

Function Documentation

size_t **umpire_allocator_get_size**(*umpire_allocator* *self, void *ptr)

Function umpire_allocator_release(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.cpp

Function Documentation

void **umpire_allocator_release**(*umpire_allocator* *self)

Function umpire_allocator_release(umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapAllocator.h

Function Documentation

void **umpire_allocator_release**(*umpire_allocator* *self)

Function umpire_mod::allocator_allocate

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
type(c_ptr) function umpire_mod::allocator_allocate(obj obj, bytes bytes)
```

Function umpire_mod::allocator_allocate_double_array_1d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_double_array_1d(this this, array array, dims dims
```

Function umpire_mod::allocator_allocate_double_array_2d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_double_array_2d(this this, array array, dims dims
```

Function umpire_mod::allocator_allocate_double_array_3d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_double_array_3d(this this, array array, dims dims
```

Function umpire_mod::allocator_allocate_double_array_4d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_double_array_4d(this this, array array, dims dims
```

Function umpire_mod::allocator_allocate_float_array_1d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_float_array_1d(this this, array array, dims dims)
```

Function umpire_mod::allocator_allocate_float_array_2d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_float_array_2d(this this, array array, dims dims)
```

Function umpire_mod::allocator_allocate_float_array_3d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_float_array_3d(this this, array array, dims dims)
```

Function umpire_mod::allocator_allocate_float_array_4d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_float_array_4d(this this, array array, dims dims)
```

Function umpire_mod::allocator_allocate_int_array_1d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_int_array_1d(this this, array array, dims dims)
```

Function umpire_mod::allocator_allocate_int_array_2d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_int_array_2d(this this, array array, dims dims)
```

Function umpire_mod::allocator_allocate_int_array_3d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_int_array_3d(this this, array array, dims dims)
```

Function umpire_mod::allocator_allocate_int_array_4d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_int_array_4d(this this, array array, dims dims)
```

Function umpire_mod::allocator_allocate_long_array_1d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_long_array_1d(this this, array array, dims dims)
```

Function umpire_mod::allocator_allocate_long_array_2d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_long_array_2d(this this, array array, dims dims)
```

Function umpire_mod::allocator_allocate_long_array_3d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_long_array_3d(this this, array array, dims dims)
```

Function umpire_mod::allocator_allocate_long_array_4d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_allocate_long_array_4d(this this, array array, dims dims)
```

Function umpire_mod::allocator_associated

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

logical function umpire_mod::allocator_associated(obj obj)

Function umpire_mod::allocator_deallocate

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine `umpire_mod::`**`allocator_deallocate`**(*obj obj*, *ptr ptr*)

Function umpire_mod::allocator_deallocate_double_array_1d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::allocator_deallocate_double_array_1d(this this, array array)

Function umpire_mod::allocator_deallocate_double_array_2d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::allocator_deallocate_double_array_2d(this this, array array)

Function umpire_mod::allocator_deallocate_double_array_3d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::allocator_deallocate_double_array_3d(this this, array array)

Function umpire_mod::allocator_deallocate_double_array_4d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_deallocate_double_array_4d(this this, array array)
```

Function umpire_mod::allocator_deallocate_float_array_1d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_deallocate_float_array_1d(this this, array array)
```

Function umpire_mod::allocator_deallocate_float_array_2d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_deallocate_float_array_2d(this this, array array)
```

Function umpire_mod::allocator_deallocate_float_array_3d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_deallocate_float_array_3d(this this, array array)
```

Function umpire_mod::allocator_deallocate_float_array_4d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_deallocate_float_array_4d(this this, array array)
```

Function umpire_mod::allocator_deallocate_int_array_1d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_deallocate_int_array_1d(this this, array array)
```

Function umpire_mod::allocator_deallocate_int_array_2d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_deallocate_int_array_2d(this this, array array)
```

Function umpire_mod::allocator_deallocate_int_array_3d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_deallocate_int_array_3d(this this, array array)
```

Function umpire_mod::allocator_deallocate_int_array_4d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_deallocate_int_array_4d(this this, array array)
```

Function umpire_mod::allocator_deallocate_long_array_1d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
subroutine umpire_mod::allocator_deallocate_long_array_1d(this this, array array)
```

Function umpire_mod::allocator_deallocate_long_array_2d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::allocator_deallocate_long_array_2d(this this, array array)

Function umpire_mod::allocator_deallocate_long_array_3d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::allocator_deallocate_long_array_3d(this this, array array)

Function umpire_mod::allocator_deallocate_long_array_4d

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::allocator_deallocate_long_array_4d(this this, array array)

Function umpire_mod::allocator_delete

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::**allocator_delete**(*obj obj*)

Function umpire_mod::allocator_eq

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

logical function umpire_mod::allocator_eq(a a, b b)

Function umpire_mod::allocator_get_actual_size

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
integer(c_size_t) function umpire_mod::allocator_get_actual_size(obj obj)
```

Function umpire_mod::allocator_get_current_size

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
integer(c_size_t) function umpire_mod::allocator_get_current_size(obj obj)
```

Function umpire_mod::allocator_get_high_watermark

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
integer(c_size_t) function umpire_mod::allocator_get_high_watermark(obj obj)
```

Function umpire_mod::allocator_get_id

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
integer(c_size_t) function umpire_mod::allocator_get_id(obj obj)
```

Function umpire_mod::allocator_get_instance

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
type(c_ptr) function umpire_mod::allocator_get_instance(obj obj)
```

Function umpire_mod::allocator_get_name

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

character(len=:) function, allocatable umpire_mod::allocator_get_name(obj obj)

Function umpire_mod::allocator_get_size

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

integer(c_size_t) function umpire_mod::allocator_get_size(obj obj, ptr ptr)

Function umpire_mod::allocator_ne

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

logical function umpire_mod::allocator_ne(a a, b b)

Function umpire_mod::allocator_release

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine `umpire_mod`**::allocator_release**(*obj obj*)

Function umpire_mod::allocator_set_instance

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine `umpire_mod`**::allocator_set_instance**(*obj obj*, *cxxmem cxxmem*)

Function umpire_mod::resourcemanager_associated

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

logical function umpire_mod::resourcemanager_associated(obj 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 obj*, *src_ptr src_ptr*, *dst_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 obj*, *src_ptr src_ptr*, *dst_ptr dst_ptr*, *size size*)

Function umpire_mod::resourcemanager_deallocate

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::**resourcemanager_deallocate**(*obj obj*, *ptr ptr*)

Function umpire_mod::resourcemanager_eq

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

logical function umpire_mod::resourcemanager_eq(a a, b 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 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 obj, n
```

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 obj, p
```

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 obj, ptr 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 obj, ptr ptr)
```

Function umpire_mod::resourcemanager_is_allocator

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
logical function umpire_mod::resourcemanager_is_allocator(obj obj, name 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 obj,
```

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 ob
```

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 obj
```

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 obj, na

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 obj, nam

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 ob

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 c

Function umpire_mod::resourcemanager_memset_all

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine umpire_mod::**resourcemanager_memset_all** (*obj obj*, *ptr ptr*, *val 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* *obj*, *ptr* *ptr*, *val* *val*, *length* *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 obj, src_ptr src_ptr, allocator a
```

Function umpire_mod::resourcemanager_ne

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

```
logical function umpire_mod::resourcemanager_ne(a a, b 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 obj, src_ptr src_pt
```

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 obj, src_ptr
```

Function umpire_mod::resourcemanager_register_allocator

- Defined in file_umpire_interface_c_fortran_wrapfumpire.f

Function Documentation

subroutine `umpire_mod::`**`resourcemanager_register_allocator`**(*obj obj*, *name name*, *allocator allocator*)

Function umpire_resourcemanager_copy_all(umpire_resourcemanager *, void *, void *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

void **`umpire_resourcemanager_copy_all`**(*umpire_resourcemanager* **self*, void **src_ptr*, void **dst_ptr*)

Function umpire_resourcemanager_copy_all(umpire_resourcemanager *, void *, void *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

void **`umpire_resourcemanager_copy_all`**(*umpire_resourcemanager* **self*, void **src_ptr*, void **dst_ptr*)

Function umpire_resourcemanager_copy_with_size(umpire_resourcemanager *, void *, void *, size_t)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

void **`umpire_resourcemanager_copy_with_size`**(*umpire_resourcemanager* **self*, void **src_ptr*, void **dst_ptr*, size_t *size*)

Function umpire_resourcemanager_copy_with_size(umpire_resourcemanager *, void *, void *, size_t)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

void **umpire_resourcemanager_copy_with_size**(*umpire_resourcemanager* *self, void *src_ptr, void *dst_ptr, size_t size)

Function umpire_resourcemanager_deallocate(umpire_resourcemanager *, void *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

void **umpire_resourcemanager_deallocate**(*umpire_resourcemanager* *self, void *ptr)

Function umpire_resourcemanager_deallocate(umpire_resourcemanager *, void *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

void **umpire_resourcemanager_deallocate**(*umpire_resourcemanager* *self, void *ptr)

Function umpire_resourcemanager_get_allocator_by_id(umpire_resourcemanager *, const int, umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

umpire_allocator ***umpire_resourcemanager_get_allocator_by_id**(*umpire_resourcemanager* *self, **const** int id, *umpire_allocator* *SHC_rv)

Function umpire_resourcemanager_get_allocator_by_id(umpire_resourcemanager *, const int, umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

umpire_allocator ***umpire_resourcemanager_get_allocator_by_id**(*umpire_resourcemanager* *self, **const** int id, *umpire_allocator* *SHC_rv)

Function umpire_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(umpire_resourcemanager *, const char *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

bool **umpire_resourcemanager_is_allocator**(*umpire_resourcemanager* *self, **const** char *name)

Function umpire_resourcemanager_is_allocator(umpire_resourcemanager *, const char *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

bool **umpire_resourcemanager_is_allocator**(*umpire_resourcemanager* *self, **const** char *name)

Function umpire_resourcemanager_is_allocator_bufferify(umpire_resourcemanager *, const char *, int)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

bool **umpire_resourcemanager_is_allocator_bufferify**(*umpire_resourcemanager* *self, **const** char *name, int *Lname*)

Function umpire_resourcemanager_is_allocator_bufferify(umpire_resourcemanager *, const char *, int)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.h

Function Documentation

bool **umpire_resourcemanager_is_allocator_bufferify**(*umpire_resourcemanager* *self, **const** char *name, int Lname)

Function umpire_resourcemanager_make_allocator_advisor(umpire_resourcemanager *, const char *, umpire_allocator, const char *, int, umpire_allocator *)

- Defined in file_umpire_interface_c_fortran_wrapResourceManager.cpp

Function Documentation

umpire_allocator ***umpire_resourcemanager_make_allocator_advisor**(*umpire_resourcemanager* *self, **const** char *name, *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_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_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_SHROUD_memory_destructor(umpire_SHROUD_capsule_data *)

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Function Documentation

void **umpire_SHROUD_memory_destructor**(*umpire_SHROUD_capsule_data* *cap)

Function umpire_SHROUD_memory_destructor(umpire_SHROUD_capsule_data *)

- Defined in file_umpire_interface_c_fortran_wrapUmpire.cpp

Function Documentation

void **umpire_SHROUD_memory_destructor**(*umpire_SHROUD_capsule_data* *cap)

Function umpire_ShroudCopyStringAndFree

- Defined in file_umpire_interface_c_fortran_wrapAllocator.cpp

Function Documentation

Warning: doxygenfunction: Cannot find function "umpire_ShroudCopyStringAndFree" in doxygen xml output for project "umpire" from directory: ../doxygen/xml/

Function umpire_strategy_mod::allocationadvisor_associated

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

logical function umpire_strategy_mod::allocationadvisor_associated(obj obj)

Function umpire_strategy_mod::allocationadvisor_eq

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

logical function umpire_strategy_mod::allocationadvisor_eq(a a, b b)

Function umpire_strategy_mod::allocationadvisor_get_instance

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

type(c_ptr) function umpire_strategy_mod::allocationadvisor_get_instance(obj obj)

Function umpire_strategy_mod::allocationadvisor_ne

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

logical function umpire_strategy_mod::allocationadvisor_ne(a a, b b)

Function umpire_strategy_mod::allocationadvisor_set_instance

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

subroutine umpire_strategy_mod::**allocationadvisor_set_instance**(*obj obj*, *cxxmem cxxmem*)

Function umpire_strategy_mod::dynamicpool_associated

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

logical function umpire_strategy_mod::dynamicpool_associated(obj obj)

Function umpire_strategy_mod::dynamicpool_eq

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

logical function umpire_strategy_mod::dynamicpool_eq(a a, b b)

Function umpire_strategy_mod::dynamicpool_get_instance

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

type(c_ptr) function umpire_strategy_mod::dynamicpool_get_instance(obj obj)

Function umpire_strategy_mod::dynamicpool_ne

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

logical function umpire_strategy_mod::dynamicpool_ne(a a, b b)

Function umpire_strategy_mod::dynamicpool_set_instance

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

subroutine `umpire_strategy_mod::`**`dynamicpool_set_instance`**(*obj obj*, *cxxmem cxxmem*)

Function umpire_strategy_mod::namedallocationstrategy_associated

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

logical function umpire_strategy_mod::namedallocationstrategy_associated(obj obj)

Function umpire_strategy_mod::namedallocationstrategy_eq

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

logical function umpire_strategy_mod::namedallocationstrategy_eq(a a, b b)

Function umpire_strategy_mod::namedallocationstrategy_get_instance

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

type(c_ptr) function umpire_strategy_mod::namedallocationstrategy_get_instance(obj obj)

Function umpire_strategy_mod::namedallocationstrategy_ne

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

logical function umpire_strategy_mod::namedallocationstrategy_ne(a a, b b)

Function umpire_strategy_mod::namedallocationstrategy_set_instance

- Defined in file_umpire_interface_c_fortran_wrapfUmpire_strategy.f

Function Documentation

subroutine umpire_strategy_mod::**namedallocationstrategy_set_instance**(*obj* *obj*, *cxxmem* *cxxmem*)

6.3.5 Variables

Variable genumpiresplicer::maxdims

- Defined in file_umpire_interface_c_fortran_genumpiresplicer.py

Variable Documentation

```
int genumpiresplicer.maxdims =  3
```

Variable genumpiresplicer::types

- Defined in file_umpire_interface_c_fortran_genumpiresplicer.py

Variable Documentation

```
tuple genumpiresplicer.types=  (    ( 'int', 'integer(C_INT)' ),    ( 'long', 'integer(C_LO
```

Variable s_null_resource_name

- Defined in file_umpire_ResourceManager.cpp

Variable Documentation

Warning: doxygenvariable: Cannot find variable "s_null_resource_name" in doxygen xml output for project "umpire" from directory: ../doxygen/xml/

Variable s_umpire_internal_device_constant_memory

- Defined in file_umpire_resource_HipConstantMemoryResource.cpp

Variable Documentation

Warning: doxygenvariable: Cannot find variable "s_umpire_internal_device_constant_memory" in doxygen xml output for project "umpire" from directory: ../doxygen/xml/

Variable s_zero_byte_pool_name

- Defined in file_umpire_ResourceManager.cpp

Variable Documentation

Warning: doxygenvariable: Cannot find variable "s_zero_byte_pool_name" in doxygen xml output for project "umpire" from directory: ../doxygen/xml/

Variable umpire::env_name

- Defined in file_umpire_Replay.cpp

Variable Documentation

const char *umpire::**env_name** = "UMPIRE_REPLAY"

Variable umpire::strategy::bits_per_int

- Defined in file_umpire_strategy_FixedPool.cpp

Variable Documentation

constexpr std::size_t umpire::strategy::**bits_per_int** = sizeof(int) * 8

Variable umpire::strategy::heuristic_percent_releasable

- Defined in file_umpire_strategy_DynamicPoolHeuristic.hpp

Variable Documentation

std::function<bool (**const** strategy::*DynamicPoolMap*&)> umpire::strategy::**heuristic_percent_releasable**
int *percentage*Return true if specified percentage of pool is releasable.

When the specified percentage of the pool has been deallocated back to the pool, this heuristic will return true.

Return True if specified percentage of memory in pool is releasable.

Parameters

- percentage: The integer percentage of releasable memory to actual memory used by the pool.

Variable umpire::strategy::heuristic_percent_releasable_list

- Defined in file_umpire_strategy_DynamicPoolHeuristic.hpp

Variable Documentation

std::function<bool (**const** strategy::*DynamicPoolList*&)> umpire::strategy::**heuristic_percent_releasable_list**
int *percentage*Return true if specified percentage of pool is releasable.

When the specified percentage of the pool has been deallocated back to the pool, this heuristic will return true.

Return True if specified percentage of memory in pool is releasable.

Parameters

- percentage: The integer percentage of releasable memory to actual memory used by the pool.

Variable umpire::util::defaultLevel

- Defined in file_umpire_util_Logger.cpp

Variable Documentation

message::*Level* `umpire::util::`**`defaultLevel`** = message::*Info*

Variable umpire::util::env_name

- Defined in file_umpire_util_Logger.cpp

Variable Documentation

`const` char *`umpire::util::`**`env_name`** = "UMPIRE_LOG_LEVEL"

Variable umpire::util::MessageLevelName

- Defined in file_umpire_util_Logger.cpp

Variable Documentation

`const` char *`umpire::util::`**`MessageLevelName`**[`message::`**`Num_Levels`**] = {"ERROR", , , }

Variable umpire_ver_2_found

- Defined in file_umpire_Umpire.cpp

Variable Documentation

Warning: doxygenvariable: Cannot find variable "umpire_ver_2_found" in doxygen xml output for project "umpire" from directory: ../doxygen/xml/

6.3.6 Defines

Define _XOPEN_SOURCE_EXTENDED

- Defined in file_umpire_strategy_FixedPool.cpp

Define Documentation

Warning: doxygendefine: Cannot find define "_XOPEN_SOURCE_EXTENDED" in doxygen xml output for project "umpire" from directory: ../doxygen/xml/

Define UMPIRE_Allocator_INL

- Defined in file_umpire_Allocator.inl

Define Documentation

UMPIRE_Allocator_INL

Define UMPIRE_ASSERT

- Defined in file_umpire_util_Macros.hpp

Define Documentation

UMPIRE_ASSERT(condition)

Define UMPIRE_DefaultMemoryResource_INL

- Defined in file_umpire_resource_DefaultMemoryResource.inl

Define Documentation

UMPIRE_DefaultMemoryResource_INL

Define UMPIRE_ERROR

- Defined in file_umpire_util_Macros.hpp

Define Documentation

UMPIRE_ERROR(msg)

Define UMPIRE_LOG

- Defined in file_umpire_util_Macros.hpp

Define Documentation

UMPIRE_LOG(lvl, msg)

Define UMPIRE_MemoryMap_INL

- Defined in file_umpire_util_MemoryMap.inl

Define Documentation

UMPIRE_MemoryMap_INL

Define UMPIRE_NullMemoryResource_INL

- Defined in file_umpire_resource_NullMemoryResource.cpp

Define Documentation

Warning: doxygendefine: Cannot find define "UMPIRE_NullMemoryResource_INL" in doxygen xml output for project "umpire" from directory: ../doxygen/xml/

Define UMPIRE_RECORD_STATISTIC

- Defined in file_umpire_util_Macros.hpp

Define Documentation

UMPIRE_RECORD_STATISTIC(name, ...)

Define UMPIRE_REPLAY

- Defined in file_umpire_Replay.hpp

Define Documentation

UMPIRE_REPLAY(msg)

Define UMPIRE_ResourceManager_INL

- Defined in file_umpire_ResourceManager.inl

Define Documentation

UMPIRE_ResourceManager_INL

Define UMPIRE_TypedAllocator_INL

- Defined in file_umpire_TypedAllocator.inl

Define Documentation

UMPIRE_TypedAllocator_INL

Define UMPIRE_UNUSED_ARG

- Defined in file_umpire_util_Macros.hpp

Define Documentation

UMPIRE_UNUSED_ARG(x)

Define UMPIRE_USE_VAR

- Defined in file_umpire_util_Macros.hpp

Define Documentation

UMPIRE_USE_VAR(x)

6.3.7 Typedefs

Typedef umpire::strategy::DynamicPool

- Defined in file_umpire_strategy_DynamicPool.hpp

Typedef Documentation

using umpire::strategy::**DynamicPool** = *DynamicPoolMap*

Typedef umpire_allocator

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_allocator* **umpire_allocator**

Typedef umpire_resourcemanager

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_resourcemanager* **umpire_resourcemanager**

Typedef umpire_SHROUD_array

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_SHROUD_array* **umpire_SHROUD_array**

Typedef umpire_SHROUD_capsule_data

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_SHROUD_capsule_data* **umpire_SHROUD_capsule_data**

Typedef umpire_strategy_allocationadvisor

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_strategy_allocationadvisor* **umpire_strategy_allocationadvisor**

Typedef umpire_strategy_allocationprefetcher

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_strategy_allocationprefetcher* **umpire_strategy_allocationprefetcher**

Typedef umpire_strategy_dynamicpool

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_strategy_dynamicpool* **umpire_strategy_dynamicpool**

Typedef umpire_strategy_dynamicpoollist

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_strategy_dynamicpoollist* **umpire_strategy_dynamicpoollist**

Typedef umpire_strategy_fixedpool

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_strategy_fixedpool* **umpire_strategy_fixedpool**

Typedef umpire_strategy_namedallocationstrategy

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_strategy_namedallocationstrategy* **umpire_strategy_namedallocationstrategy**

Typedef umpire_strategy_threadsafeallocator

- Defined in file_umpire_interface_c_fortran_typesUmpire.h

Typedef Documentation

typedef struct *s_umpire_strategy_threadsafeallocator* **umpire_strategy_threadsafeallocator**

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 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

INDEX

D

F

G

O

S

U