Document #: P0148R0
Date: 2015-10-14
Authors: Pablo Halpern, phalpern@halpernwightsoftware.com
Dietmar Kühl, dkuhl@bloomberg.net

`memory_resource_ptr`: A Limited Smart Pointer for `memory_resource` Correctness

1 Abstract

The `memory_resource` polymorphic base class introduced in the Library Fundamentals Technical Specification (LFTS), N4529, provides a simpler model for customizable memory allocation than does the traditional allocator templates used by the STL containers. That said, it is still necessary to use polymorphic memory resources in an idiomatically correct way in order to achieve all of its goals, including interoperability with other components that expect such idiomatic consistency.

In the LFTS, raw pointers to `memory_resource` objects are passed to object constructors and returned from `get_memory_resource()` (e.g., for type-erased allocators, [memory.type.erased.allocator] in the LFTS). This proposal would replace those raw pointers with a simple smart pointer type, `memory_resource_ptr`, having a limited interface that gently encourages correct idiomatic use of `memory_resource`.

2 Goals of this paper

This paper is a late paper and is presented in advance of the Kona meeting for initial discussion and to see if the ideas herein are worth pursuing further. To that end, rough wording is presented, but not detailed formal wording.

Nevertheless, these ideas are simple enough that it is hoped that they can be incorporated either directly into the LFTS v2 or into C++17 if and when polymorphic memory resources are adopted into the standard working paper.

3 Motivation

The use of raw pointers to polymorphic memory resources makes their use more difficult and error prone than they need to be. The smart pointer proposed in this paper is a simple wrapper around a raw pointer that is intended to address the following issues:

3.1 Default value for a resource data member

Many classes will have a data member of type pointer-to-memory_resource and a constructor argument of the same type. The default constructor must initialize the member variable to `experimental::get_default_resource()`. Similarly, any constructor taking a default resource argument would need have an explicit default value of `experimental::get_default_resource()`.

The `memory_resource_ptr` class proposed here would have a default constructor that automatically sets it to `experimental::get_default_resource()`. Thus, a class `MyClass` might have constructors something like this:

```
MyClass() : m_mem_resource() { }  // Value initialization of member variable
MyClass(int i, memory_resource_ptr = {}); // optional argument easily defaulted
```

3.2 Null and uninitialized pointer values

It is almost never valid to have a null pointer to a memory resource. As described above, `memory_resource_ptr` can be value-initialized to the default memory resource, so null and uninitialized pointers would be difficult to create by accident. Code that takes a `memory_resource_ptr` can assume it is non-null and avoid gratuitous tests.

3.3 Inadvertent reseating of the memory resource

Idiomatically, neither move assignment nor copy assignment of an object using an allocator or memory resource should move or copy the allocator or memory resource. With rare exceptions, the memory resource used to construct an object should be the one used for its entire lifetime. Changing the resource can result in a mismatch between lifetime of the resource and the lifetime of the object that uses it. Also, assigning to an element of a container would result in breaking the homogenous use of a single allocator for all elements of that container, which is crucial to safely and efficiently applying algorithms like `sort` that swap elements within the container.

Raw pointers encourage blind moving or copying of member variables during assignment. In this proposal, therefore, `memory_resource_ptr` **does not have an assignment operator**, thus preventing accidental reseating during assignment. Instead, it provides a `reset` member function to provide deliberate reseating, which might be needed when `memory_resource_ptr` is used for a non-member (e.g., local) variable. The absence of assignment also makes `std::swap` non-functional, thus preventing accidental reseating via swap.

3.4 operator new

Given a memory resource, it is desirable to be able to allocate an object from that resource and construct it in a single step using `operator new`. We can support this by providing an overload of `operator new` that takes a pointer-to-`memory_resource` and uses it as the source of memory. However, consider the following simple code:

```
void f(memory_resource *mr, int v)
{
    MyClass *p = new(mr) MyClass(v);
    ...
}
```

Is the invocation of `operator new` intended to place the new object at `mr` (standard placement `new`) or is it intended to allocate memory *from* `*mr` (overload of `new` for memory resources). Using `memory_resource_ptr` would make this unambiguous, because a `memory_resource_ptr` is not convertible to `void*` and, thus, will not match the standard placement `new`:

```
void f(memory_resource_ptr mr, int v)
{
    MyClass *p = new(mr) MyClass(v);
    ...
}
```

3.5 Other idiomatic uses of `allocator_resource`

It is generally desirable for a class object to pass the memory resource provided to the constructor to sub-objects that take memory resource constructor arguments. The `memory_resource_ptr` class proposed here does not substantially make this job any easier, unfortunately, and other mechanisms (including static analysis by external tools) should be considered to make this process less error prone.

Bloomberg has promoted the idiom that copy constructors should not propagate the memory resource pointer from the copied-from object to the copied-to object but that, instead, the copied-to object should use the default resource (unless another resource is provided using the "extended" copy constructor). We considered giving `memory_resource_ptr` an unusual copy constructor that would leave the copied-to object pointing to the default resource. We decided that such a copy constructor is not in keeping with the spirit of copy construction – that the copy should compare equal to the original. This issue is addressed in the "two separate classes" alternative, described below. In this proposal, we do nothing to help enforce the idiom of not propagating the memory resource on copy construction. Again, static analysis tools can help here.

4 Alternatives considered

4.1 Status quo

We could continue to pass `memory_resource` addresses around as raw pointers. However, the authors of this proposal think that we can make it easier for those who are not experts in the use of allocators to use memory resources correctly if this simple wrapper class were provided.

4.2 Two separate classes

The `memory_resource_ptr` class proposed here has two different major use cases, with slightly different needs:

A **member_variable** of type `memory_resource_ptr` should not be assignable or swappable, for the reasons discussed above. Moreover, it is arguable that it should not have a copy constructor either (though it can have a move constructor) so that a programmer would need to consider carefully whether the resource pointer should be propagated from the original object to the copy (Bloomberg recommends that it should not be).

Conversely, a **non-member variable** such as a local variable could be assigned, swapped, and copy-constructed without causing problems to the idiom. This use case absolutely needs a public copy constructor so that it can be passed and returned by value. It benefits from `memory_resource_ptr`'s default constructor, but no other features.

For this reason, we considered having a different type for each of the two use cases. Our decision of a single type for this proposal was based on a desire to keep things minimal and to avoid adding confusion in one place (which class do I use?) even if it were to avoid some confusion elsewhere (why can't I assign or swap these members)?

4.3 Reference-like wrapper instead of pointer-like wrapper

We also considered making the wrapper behave as a *reference* to a `memory_resource` rather than a *pointer* to a memory resource. While this idea has merit, it complicates things in a few ways:

- The smart pointer concept is well understood and is easier to explain than a smart reference.
- The wrapper would need to provide a duplicate interface to `memory_resource`, resulting in a bigger interface and more opportunity for skew to develop between them. (Operator dot could potentially help here.)
- We want to make it very clear to the programmer that she is manipulating a pointer to a (non-owned) resource, and not obscure that fact with something that looks like a value class, but is in fact a reference-like class.

1 Implementation experience

The actual `memory_resource_ptr` class is trivial to implement and, in fact an entire implementation exists in the exposition of the proposal below. Portions of the proposed interface have been used in experimental code with good results, but a larger experiment is in order to judge usability and effectiveness for the intended goals.

2 Proposal

Add the following non-owning smart pointer class to `<experimental/memory_resource>` (for brevity, the requirements and effects of all members are described as either comments or inline code):

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

class memory_resource_ptr {
    memory_resource *m_resource;  // For exposition only

  public:
    memory_resource_ptr() noexcept
      : m_resource(get_default_resource().get()) { }

    memory_resource_ptr(nullptr_t) noexcept
      : m_resource(get_default_resource().get()) { }

    memory_resource_ptr(memory_resource *p) noexcept
      : m_resource(p ? p : get_default_resource().get()) { }
        //  This constructor is deliberately not 'explicit'. Should it be?
    memory_resource_ptr(const memory_resource_ptr&) noexcept
        = default;
    ~memory_resource_ptr() noexcept = default;

    memory_resource_ptr& operator=(const memory_resource_ptr&)
        = delete;

    void reset(memory_resource *p) noexcept
      { m_resource = p ? p : get_default_resource().get(); }
    void reset(nullptr_t) noexcept
      { m_resource = get_default_resource().get(); }
    void reset(memory_resource_ptr p) noexcept
      { m_resource = p.m_resource; }

    memory_resource *get() const noexcept { return m_resource; }
    memory_resource *operator->() const noexcept
      { return m_resource; }
    memory_resource& operator*()  const noexcept
      { return *m_resource; }
};

bool operator==(memory_resource_ptr a, memory_resource_ptr b) {
    return a.get() == b.get();
}

bool operator!=(memory_resource_ptr a, memory_resource_ptr b) {
    return a.get() != b.get();
}

}
}
}
}

void *operator new(std::size_t sz,
                   std::experimental::pmr::memory_resource_ptr mrp)
{
    return mrp->allocate(sz);
}

void operator delete(void *p, std::size_t sz,
                     std::experimental::pmr::memory_resource_ptr mrp)
{
    mrp->deallocate(p, sz);
}
```