P1083r3 | Move resource_adaptor from Library TS to the C++ WP

Pablo Halpern phalpern@halpernwightsoftware.com

2019-06-14 | Target audience: LWG

1 Abstract

When the polymorphic allocator infrastructure was moved from the Library Fundamentals TS to the C++17 working draft, `pmr::resource_adaptor` was left behind. The decision not to move `pmr::resource_adaptor` was deliberately conservative, but the absence of `resource_adaptor` in the standard is a hole that must be plugged for a smooth transition to the ubiquitous use of `polymorphic_allocator`, as proposed in P0339 and P0987. This paper proposes that `pmr::resource_adaptor` be moved from the LFTS and added to the C++20 working draft.

2 History

2.1 Changes from R2 to R3 (in Kona and pre-Cologne)

- Changed `resource-adaptor-imp` to kabob case.
- Removed special member functions (copy/move ctors, etc.) and let them be auto-generated.
- Added a requirement that the `Allocator` template parameter must support rebinding to any non-class, non-over-aligned type. This allows the implementation of `do_allocate` to dispatch to a suitably rebound copy of the allocator as needed to support any native alignment argument.

2.2 Changes from R1 to R2 (in San Diego)

- Paper was forwarded from LEWG to LWG on Tuesday, 2018-10-06
- Copied the formal wording from the LFTS directly into this paper
- Minor wording changes as per initial LWG review
- Rebased to the October 2018 draft of the C++ WP

2.3 Changes from R0 to R1 (pre-San Diego)

- Added a note for LWG to consider clarifying the alignment requirements for `resource_adaptor<A>::do_allocate()`.
- Changed rebind type from `char` to `byte`.
- Rebased to July 2018 draft of the C++ WP.

3 Motivation

It is expected that more and more classes, especially those that would not otherwise be templates, will use `pmr::polymorphic_allocator<byte>` to allocate memory. In order to pass an allocator to one of these classes, the allocator must either already be a polymorphic allocator, or must be adapted from a non-polymorphic allocator. The process of adaptation is facilitated by `pmr::resource_adaptor`, which is a simple class template, has been in the LFTS for a long time, and has been fully implemented. It is therefore a low-risk, high-benefit component to add to the C++ WP.

4   Impact on the standard

`pmr::resource_adaptor` is a pure library extension requiring no changes to the core language nor to any existing classes in the standard library.

5   Formal Wording

This proposal is based on the Library Fundamentals TS v2, N4617 and the March 2019 draft of the C++ WP, N4810.

In section 19.12.1 [mem.res.syn] of the C++ WP, add the following declaration immediately after the declaration of `operator!=(const polymorphic_allocator...):`

```
// 19.12.x resource adaptor
// The name resource-adaptor-imp is for exposition only.
template <class Allocator> class resource-adaptor-imp;

template <class Allocator>
  using resource_adaptor = resource-adaptor-imp<
    typename allocator_traits<Allocator>::template rebind_alloc<byte>>;
```

Insert between sections 19.12.3 [mem.poly.allocator.class] and 19.12.4 [mem.res.global] of the C++ WP, the following section, taken with modifications from section 8.7 of the LFTS v2:

19.12.x template alias resource__adaptor [memory.resource.adaptor]

19.12.x.1 resource__adaptor [memory.resource.adaptor.overview]

An instance of `resource_adaptor<Allocator>` is an adaptor that wraps a `memory_resource` interface around `Allocator`. `resource_adaptor<X<T>>` and `resource_adaptor<X<U>>` are the same type for any allocator template `X` and types `T` and `U`. In addition to the *Cpp17Allocator* requirements (§15.5.3.5), the `Allocator` parameter to `resource_adaptor` shall meet the following additional requirements:

- `typename allocator_traits<Allocator>::pointer` shall denote the type `allocator_traits< Allocator>::value_type*`.

- `typename allocator_traits<Allocator>::const_pointer` shall denote the type to `allocator_traits< Allocator>::value_type const*`.

- `typename allocator_traits<Allocator>::void_pointer` shall denote the type `void*`.

- `typename allocator_traits<Allocator>::const_void_pointer` shall denote the type `void const*`.

- Calls to `allocator_traits<Allocator>::template rebind_traits<T>::allocate` and `allocator_traits< Allocator>::template rebind_traits<T>::deallocate` shall be well-formed for all non-class, non-over-aligned types `T`; no diagnostic required.

```
// The name resource-adaptor-imp is for exposition only.
template <class Allocator>
class resource-adaptor-imp : public memory_resource {
  Allocator m_alloc; // exposition only

public:
  using allocator_type = Allocator;

  resource-adaptor-imp() = default;
  resource-adaptor-imp(const resource-adaptor-imp&) = default;
  resource-adaptor-imp(resource-adaptor-imp&&) = default;
```

Pablo Halpern

```
    explicit resource-adaptor-imp(const Allocator& a2);
    explicit resource-adaptor-imp(Allocator&& a2);

    resource-adaptor-imp& operator=(const resource-adaptor-imp&) = default;

    allocator_type get_allocator() const { return m_alloc; }

protected:
  void* do_allocate(size_t bytes, size_t alignment) override;
  void do_deallocate(void* p, size_t bytes, size_t alignment) override;
  bool do_is_equal(const memory_resource& other) const noexcept override;
};
```

19.12.x.2 resource-adaptor-imp constructors [memory.resource.adaptor.ctor]

```
explicit resource-adaptor-imp(const Allocator& a2);
```

>   *Effects*: Initializes m_alloc with a2.

```
explicit resource-adaptor-imp(Allocator&& a2);
```

>   *Effects*: Initializes m_alloc with std::move(a2).

19.12.x.3 resource-adaptor-imp member functions [memory.resource.adaptor.mem]

```
void* do_allocate(size_t bytes, size_t alignment);
```

>   *Expects:* alignment is a power of two.

>   *Returns:* a pointer to allocated storage obtained by calling the allocate member function on a suitably rebound copy of m_alloc such that the expected size and alignment of the allocated memory are at least bytes and alignment, respectively. If the rebound Allocator supports over-aligned storage, then resource_adaptor<Allocator> should also support over-aligned storage.

>   *Throws:* nothing unless the underlying allocator throws.

```
void do_deallocate(void* p, size_t bytes, size_t alignment);
```

>   *Expects*: p has been returned from a prior call to allocate(bytes, alignment) on a memory resource equal to *this, and the storage at p shall not yet have been deallocated.

>   *Effects*: Returns memory to the allocator using m_alloc.deallocate.

```
bool do_is_equal(const memory_resource& other) const noexcept;
```

>   Let p be dynamic_cast<const resource-adaptor-imp*>(&other).

>   *Returns*: false if p is null; otherwise the value of m_alloc == p->m_alloc.

6   References

N4810; Working Draft, Standard for Programming Language C++, Richard Smith, editor, 2019-03-15.

N4617: Programming Languages - C++ Extensions for Library Fundamentals, Version 2, 2016-11-28.

P0339: polymorphic_allocator<> as a vocabulary type, Pablo Halpern, 2018-04-02.

P0987: polymorphic_allocator instead of type-erasure, Pablo Halpern, 2018-04-02.

Pablo Halpern