P1083r2 | Move resource_adaptor from Library TS to the C++ WP

Pablo Halpern phalpern@halpernwightsoftware.com

2018-11-13 | 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 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

2.2 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 October 2018 draft of the C++ WP, N4778.

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 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`. To ensure that `resource_adaptor<X<T>>` and `resource_adaptor<X<U>>` are the same type for any allocator template `X` and types `T` and `U`, `resource_adaptor<Allocator>` is rendered as an alias to a class template such that Allocator is rebound to a `byte` value type in every specialization of the class template. The requirements on this class template are defined below. The name *resource_adaptor_imp* is for exposition only and is not normative, but the definitions of the members of that class, whatever its name, *are* normative. 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 be identical to `typename allocator_traits<Allocator>::value_type*`.
- `typename allocator_traits<Allocator>::const_pointer` shall be identical to `typename allocator_traits<Allocator>::value_type const*`.
- `typename allocator_traits<Allocator>::void_pointer` shall be identical to `void*`.
- `typename allocator_traits<Allocator>::const_void_pointer` shall be identical to `void const*`.

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

  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` shall be 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` shall have 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

N4778: Working Draft, Standard for Programming Language C++, Richard Smith, editor, 2018-10-08.

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.