P1083R4
Pablo Halpern <phalpern@halpernwightsoftware.com>
2022-01-01 14:56 EST
Target audience: LEWG/LWG

Move `resource_adaptor` from Library TS to the C++ WP

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.

Status

On Oct 5, 2021, a subgroup of LWG reviewed P1083R3 and found an issue in the way the max alignment supported by `pmr::resource_adaptor` was specified in the paper. There was general consensus that a `MaxAlign` template parameter would be preferable, but the change was considered to be of a design nature and therefore requires LEWG review. The R4 version of this paper contains the changes for LEWG review

History

Changes from R3 to R4

- Added *Design changes* section that describes changes after LWG review.
- Added `MaxType` as a second template parameter to `pmr::resource_adaptor`.
- Added the `max_align_v` constant, `aligned_type` metafunction, `aligned_raw_storage` class template, and `aligned_object_storage` class template.
- Made a few editorial changes to comply with LWG style.

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.

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

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.

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 rather than specifying an allocator as a template parameter. 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.

Design changes (for LEWG review)

The following design changes were made as a consequence of discussions in LWG on 5 October 2021. LWG felt that the scope of these changes warranted review by LEWG.

`MaxAlign` template argument: A `pmr::resource_adaptor` instance wraps an object having a type that meets the Allocator requirements. Its `do_allocate` virtual member function supplies aligned memory by invoking the `allocate` member function on the wrapped allocator. The only way to supply alignment information to the wrapped allocator is to rebind it for a `value_type` having the desired alignment but, because the alignment is specified to `pmr::resource_adaptor::allocate` at run time, the implementation must rebind its allocator for every possible alignment and dynamically choose the correct one. In order to keep the number of such rebound instantiations manageable and reduce the requirements on the allocator type, an upper limit (default `alignof(max_align_t)`) can be specified when instantiating `pmr::resource_adaptor`. This recent change was made after discussion with members of LWG, and with their encouragement.

(Optional) `constexpr` value `max_align_v`: The standard has a type, `std::max_align_t`, whose alignment is at least as great as that of every scalar type. I found that I was continually referring to the *value*, `alignof(std::max_align_t)`. In fact, *every single use* of `max_align_t` in the standard is as the operand of `alignof`. As a drive-by fix, therefore, this proposal introduces the constant `max_align_v` as a more straightforward spelling of `alignof(max_align_t)`. Note that the introduction of this constant is completely severable from the proposal if it is deemed undesirable. The name is also subject to bikeshedding (e.g., by removing the `_v`).

Alias template `std::aligned_type`: This alias is effectively a metafunction that resolves to a scalar type if possible, otherwise to a specialization of `aligned_raw_storage`. Its use in this specification allows `pmr::resource_adaptor` to work with minimalist allocators, including those that can be rebound only for scalar types. For over-aligned values, it uses `aligned_raw_storage`, below. Both `aligned_raw_storage` and `aligned_type` are declared in header `<memory>`, but LEWG could consider putting them somewhere else (e.g., in `<utility>`).

Class template `std::aligned_raw_storage`: When instantiated with an alignment greater than `max_align_v`, `std::aligned_type` could be defined vaguely in terms of an unspecified over-aligned type, but LWG wanted to be more precise so as to better describe the allowable set of allocators usable with `resource_adaptor`. The obvious choice of the over-aligned type would have been `std::aligned_storage`, but that template has been deprecated as a result of numerous flaws described in P1413. The class template `std::aligned_raw_storage` is intended to replace `std::aligned_storage` and correct the problems associated with it; specifically, it is not a metafunction, but a `struct` template, and it provides direct access to its data buffer, which can be validly cast to a pointer to any type having the specified alignment (or less). The relationship between size and alignment is specifically described in the wording, so programmers can rely on it. Finally, `aligned_raw_storage` provides a `type` member for backwards compatibility with the deprecated `aligned_storage` metafunction.

(Optional) Class template `std::aligned_object_storage`: The alignment parameter for `aligned_raw_storage`, described above, is specified as a number rather than as a type – as needed for low-level types like `pmr::resource_adaptor` – and the storage must be cast to the desired type before it's used. This primitive type practically screams for the introduction of an aligned storage type parameterized on the type of object you wish to store in it. Although not needed for this proposal, `aligned_object_storage` was included for this purpose.

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. A couple of general-purpose templates (`aligned_type`, `aligned_raw_storage`, and `aligned_object_storage`) are also added as pure library extensions.

Implementation Experience

A full implementation of the current proposal can be found in GitHub at https://github.com/phalpern/WG21-halpern/tree/P1083/P1083-resource_adaptor.

The version described in the Library Fundamentals TS has been implemented by multiple vendors in the `std::experimental::pmr` namespace.

Formal Wording

This proposal is based on the Library Fundamentals TS v2, N4617 and the October 2021 draft of the C++ WP, N4901.

In section 17.2.1 [cstddef.syn] of the C++WP, add the following definition sometime after the declaration of `max_align_t` in header `<cstddef>`:

```
constexpr size_t max_align_v = alignof(max_align_t);
```

In section 20.10.2 [memory.syn], add the following declarations to `<memory>` (probably near the top):

```
template <size_t Align, size_t Sz = Align> struct aligned_raw_storage;
template <typename T> struct aligned_object_storage;
template <size_t Align> using aligned_type = see below;
```

Prior to section 20.10.3, add the description of these new templates:

20.10.? Aligned storage [aligned.storage]

20.10.?.1 Aligned raw storage [aligned.raw.storage]

```
namespace std {
  template <size_t Align, size_t Sz = Align>
  struct aligned_raw_storage
  {
    static constexpr size_t alignment = Align;
    static constexpr size_t size      = (Sz + Align - 1) & ~(Align - 1);
    using type = aligned_raw_storage;

    constexpr       void* data()       noexcept { return buffer; }
    constexpr const void* data() const noexcept { return buffer; }

    alignas(alignment) byte buffer[size];
  };
}
```

Mandates: `Align` is a power of 2, `Sz > 0`

An instantiation of template `aligned_raw_storage` is a standard-layout trivial type that provides storage having the specified alignment and size, where the size is rounded up to the nearest multiple of the alignment.

20.10.?.2 Aligned storage for object [aligned.object.storage]

```
namespace std {
  template <typename T>
  struct aligned_object_storage : aligned_raw_storage<alignof(T), sizeof(T)>
  {
    using type       = aligned_object_storage;
    using value_type = T;

    constexpr T&       object()       { return *static_cast<T *>(this->data()); }
    constexpr const T& object() const { return *static_cast<const T*>(this->data()); }
  };
}
```

An instantiation of template `aligned_object_storage` is a standard-layout trivial type that provides storage of suitable size and alignment to store an object of type `T`.

20.10.?.3 Aligned type [aligned.type]

`template <size_t Align> using aligned_type =` *see below*;

Mandates: `Align` is a power of 2.

If there exists a scalar type, `T`, such that `alignof(T) == Align` and `sizeof(T) == Align`, then `aligned_type<Align>` is an alias for `T`; otherwise, it is an alias for `aligned_raw_storage<Align, Align>`. If more than one scalar meets the requirements for `T`, the one chosen is implementation defined, but consistent for all instantiations of `aligned_type` with that alignment.

In section 20.12.1 [mem.res.syn], add the following declaration immediately after the declaration of `operator!=(const polymorphic_allocator...)`:

```
// 20.12.? resource adaptorfor a given alignment.
// The name resource-adaptor-imp is for exposition only.
template <class Allocator, size_t MaxAlign> class resource-adaptor-imp;
template <class Allocator, size_t MaxAlign = max_align_v>
  using resource_adaptor = resource-adaptor-imp<
    typename allocator_traits<Allocator>::template rebind_alloc<byte>,
    MaxAlign>;
```

Insert before section 20.12.5 [mem.res.pool] of the C++ WP, the following section, taken with modifications from section 8.7 of the LFTS v2:

20.12.? Alias template resource__adaptor [memory.resource.adaptor]

20.12.?.1 `resource_adaptor` [memory.resource.adaptor.overview]

An instance of `resource_adaptor<Allocator, MaxAlign>` is an adaptor that wraps a `memory_resource` interface around `Allocator`. [*Note*: The type of `resource_adaptor<X, N>` is independent of `X::value_type`. – *end note*] In addition to the *Cpp17Allocator* requirements (§15.5.3.5), the `Allocator` parameter to `resource_adaptor` is further constrained as follows:

- `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<aligned_type<N>>::allocate` and `allocator_traits<Allocator>::template rebind_traits<aligned_type<N>>::deallocate` shall be well-formed for all `N`, such that `N` is a power of 2 less than or equal to `MaxAlign`, no diagnostic required.

```
// The name "resource-adaptor-imp" is for exposition only.
template <class Allocator, size_t MaxAlign>
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&) noexcept = default;
  resource-adaptor-imp(resource-adaptor-imp&&) noexcept = default;

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

20.12.?.2 `resource-adaptor-imp` constructors [memory.resource.adaptor.ctor]

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

Effects: Initializes `m_alloc` with `a2`.

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

Effects: Initializes `m_alloc` with `std::move(a2)`.

20.12.?.3 `resource-adaptor-imp` member functions [memory.resource.adaptor.mem]

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

Let `CA` be an integral constant expression such that `CA == alignment`, is `true`, let `U` be the type `aligned_type<CA>`, and let `n` be `(bytes + sizeof(U) - 1) / sizeof(U)`.

Preconditions: `alignment` is a power of two.

Returns: `allocator_traits<Allocator>::template rebind_traits<U>::allocate(m_alloc, n)`

Throws: nothing unless the underlying allocator throws.

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

Let `CA` be an integral constant expression such that `CA == alignment`, is `true`, let `U` be the type `aligned_type<CA>`, and let `n` be `(bytes + sizeof(U) - 1) / sizeof(U)`.

Preconditions: given a memory resource `r` such that `this->is_equal(r)` is `true`, `p` was returned from a prior call to `r.allocate(bytes, alignment)` and the storage at `p` has not yet been deallocated.

Effects: `allocator_traits<Allocator>::template rebind_traits<U>::deallocate(m_alloc, p, n)`

`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`.

References

N4901. Working Draft, Standard for Programming Language C++, Thomas Köppe, editor, 2021-10-23.

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.