Polymorphic Memory Resources - r2

Abstract
A significant impediment to effective memory management in C++ has been the inability to use allocators in non-generic contexts. In large software systems, most of the application program consists of non-generic procedural or object-oriented code that is compiled once and linked many times. Allocators in C++, however, have historically relied solely on compile-time polymorphism, and therefore have not been suitable for use in vocabulary types, which are passed through interfaces between separately-compiled modules, because the allocator type necessarily affects the type of the object that uses it. This proposal builds upon the improvements made to allocators in C++11 and describes a set of facilities for runtime polymorphic memory resources that interoperate with the existing compile-time polymorphic allocators. In addition, this proposal improves the interface and allocation semantics of some library classes, such as `std::function`, that use type erasure for allocators.

Contents
1 Proposal history ........................................................................................................................................2
  1.1 Target ...............................................................................................................................................2
  1.2 Changes from N3816 .....................................................................................................................2
  1.3 Changes from N3726 .....................................................................................................................3
  1.4 Changes from N3525 .....................................................................................................................3
2 Document Conventions ......................................................................................................................4
3 Motivation .............................................................................................................................................4
4 Usage Example .....................................................................................................................................5
5 Summary of Proposal ...........................................................................................................................8
  5.1 Namespace `std::pmr` ....................................................................................................................8
  5.2 Abstract base class `memory_resource` .......................................................................................9
  5.3 Class Template `polymorphic_allocator<T>` ............................................................................9
  5.4 Aliases for container classes .......................................................................................................9
  5.5 Class template `resource_adaptor<Alloc>` ..................................................................................10
  5.6 Function `new_delete_resource()` ............................................................................................10
  5.7 Function `null_memory_resource()` ............................................................................................10
  5.8 Functions `get_default_resource()` and `set_default_resource()` ........................................10
  5.9 Standard memory resources .......................................................................................................10
    5.9.1 Classes `synchronized_pool_resource` and `unsynchronized_pool_resource` ..................11
    5.9.2 Class `monotonic_buffer_resource` ......................................................................................11
1 Proposal history

1.1 Target

The original version of this proposal (N3525) was first discussed in the Library Evolution Working Group during the April 2013 meeting of WG21 in Bristol, UK. A revised version of the proposal, N3726, was brought back to the LEWG at the September 2013 meeting in Chicago. A straw poll of the LEWG both in Bristol and in Chicago indicated strong support for the concepts in this proposal and a decision was made to target these ideas for inclusion a forthcoming Library Fundamentals Technical Specification (TS).

1.2 Changes from N3816

- Removed a number of tuning parameters from pool_options.
- Added a diagram of an example implementation of pool resources.
- Changed formal wording in the “Changes to classes in the standard” section to conform to recent decisions for how such changes should be described in the TS.
• Numerous small changes in response to a detailed technical review by the library working group.

1.3 Changes from N3726
• Added rationale in a few places to justify design choices. Removed guidance requests that have been already been addressed by the LEWG.
• Reorganized the formal wording into two sections: new classes and changes to standard classes. The intention is that both would go into the TS, even though the second section is not a pure extension.
• Moved all new classes and nested namespaces into the std::experimental namespace.
• Changed allocator_resource to use the public-non-virtual-function-calls-protected-virtual-function idiom.
• Clarified wording for various do_allocate and do_deallocate functions. Specifically, changed alignment requirements to reflect the conditionally-supported nature of superalignment. Also clarified that do_deallocate must be called on a block that was allocated from the same (or equal) resource.
• Made resource_allocator_imp constructor explicit.
• Added a synchronized_pool_resource class in addition to the previously-described unsynchronized_pool_resource class.
• Added some algorithmic description to the pool resources classes and added a few tuning parameters so that users can determine when their use is appropriate.
• Removed threshold for monotonic_buffer_resource.
• Added noexcept to operator== and operator!=. Changes static const to constexpr.

1.4 Changes from N3525
• Simplified alignment requirements for memory_resource::allocate().
• Renamed the polyalloc namespace to pmr (Polymorphic Memory Resource).
• Simplified new_delete_resource and gave more leeway to the implementation.
• Added null_memory_resource() function.
• Borrowed some ideas from Mark Boyall’s N3575 and mixed them with some ideas from Bloomberg’s BSL project to yield the monotonic_buffer_resource and unsynchronized_pool_resource concrete manifestations of polymorphic memory resources.
Specified allocator behavior for promise and packaged_task.

There were some design changes proposed during discussion at the April 2013 meeting in Bristol. Although I elected not to make a number of those changes, I did investigate each of them and, for those ideas that were rejected, I added rationale for why they are the way they are.

Wording improvements, especially in type-erased allocator section.

Complete description of aliases for containers using polymorphic allocators.

2 Document Conventions

For the parts of this document that refer to changes in existing standard classes, section names and numbers are relative to the October 2013 Working Draft, N3797.

Existing working paper text is indented and shown in dark blue. Edits to the working paper are shown with red strikeouts for deleted text and green underlining for inserted text within the indented blue original text. When describing the addition of entirely new sections, the underlining is omitted for ease of reading.

Comments and rationale mixed in with the proposed wording appears as shaded text.

Requests for committee opinions and guidance appear with light (yellow) shading. It is expected that changes resulting from such guidance will be minor and will not delay acceptance of this proposal in the same meeting at which it is presented.

3 Motivation

Back in 2005, I argued in N1850 that the C++03 allocator model hindered the usability of allocators for managing memory use by containers and other objects that allocate memory. Although N1850 conflated them, the proposals in that paper could be broken down into two separate principles:

1. The allocator used to construct a container should also be used to construct the elements within that container.

2. An object’s type should be independent of the allocator it uses to obtain memory.

In subsequent proposals, these principles were separated. The first principle eventually became known as the scoped allocator model and is embodied in the scoped_allocator_adaptor template in Section [allocator.adaptor] (20.12) of the 2011 standard (and the same section of the current WP).

Unfortunately, creating a scoped allocator model that was compatible with C++03 and acceptable to the committee, as well as fixing other flaws in the allocator section of the standard, proved a time-consuming task, and library changes implementing the second principle were not proposed in time for standardization in 2011.

This paper proposes new library facilities to address the second principle. Section 4.3 of N1850 (excerpted in the appendix of this paper) gives a detailed description of why it is undesirable to specify allocators as class template parameters. Key among the problems of allocator template parameters is that they inhibit the use of
**vocabulary types** by altering the type of specializations that would otherwise be the same. For example, `std::basic_string<char, char_traits<char>, Alloc1<char>>` and `std::basic_string<char, char_traits<char>, Alloc2<char>>` are different types in C++ even though they are both string types capable of representing the same set of (mathematical) values.

Some new vocabulary types introduced into the 2011 standard, including function, promise, and future use **type erasure** (see [jsmith](#)) as a way to get the benefits of allocators without the allocator contaminating their type. Type erasure is a powerful technique, but has its own flaws, such as that the allocators can be propagated outside of the scope in which they are valid and also that there is no way to query an object for its type-erased allocator. More importantly, even if type erasure were a completely general solution, it cannot be applied to existing container classes because they would break backwards compatibility with the existing interfaces and binary compatibility with existing implementations. Moreover, even for programmers creating their own classes, unconstrained by existing usage, type-erasure is a relatively complex and time-consuming technique and requires the creation of a polymorphic class hierarchy much like the `memory_resource` and `resource_adaptor` class hierarchy proposed for standardization below. Given that type erasure is expensive to implement and not general even when it is feasible, we must look to other solutions.

Fortunately, the changes to the allocator model made in 2011 (especially full support for stateful allocators and scoped allocators) make this problem with allocators relatively easy to solve in a more general way. The solution presented in this paper is to create a uniform memory allocation base class, `memory_resource`, suitable for use by template and non-template classes alike, and single allocator template, `polymorphic_allocator` that wraps a pointer to a `memory_resource` and which can be used ubiquitously for instantiating containers. The `polymorphic_allocator` will, as its name suggests, have polymorphic runtime behavior. Thus objects of the same type can have different effective allocators, achieving the goal of making an object’s type independent of the allocator it uses to obtain memory, and thereby allowing them to be interoperable when used with precompiled libraries.

### 4 Usage Example

Suppose we are processing a series of shopping lists, where a shopping list is a container of strings, and storing them in a collection (a list) of shopping lists. Each shopping list being processed uses a bounded amount of memory that is needed for a short period of time, while the collection of shopping lists uses an unbounded amount of memory and will exist for a longer period of time. For efficiency, we can use a more time-efficient memory allocator based on a finite buffer for the temporary shopping lists. However, this time-efficient allocator is not appropriate for the longer lived collection of shopping lists. This example shows how those temporary shopping lists, using a time-efficient allocator, can be used to populate the long lived collection of shopping lists, using a general purpose allocator, something that would be annoyingly difficult without the polymorphic allocators in this proposal.
First, we define a class, ShoppingList, that contains a vector of strings. It is not a template, so it has no Allocator template argument. Instead, it uses memory_resource as a way to allow clients to control its memory allocation:

```cpp
#include <polymorphic_allocator>
#include <vector>
#include <string>

class ShoppingList {
    // Define a vector of strings using polymorphic allocators. polymorphic_allocator is scoped,
    // so every element of the vector will use the same allocator as the vector itself.
    typedef std::pmr::string string_type;
    typedef std::pmr::vector<string_type> strvec_type;

    strvec_type m_strvec;

public:
    // This type makes uses_allocator<ShoppingList, memory_resource*>::value true.
    typedef std::pmr::memory_resource* allocator_type;

    // Construct with optional memory_resource. If alloc is not specified, uses pmr::get_default_resource().
    ShoppingList(allocator_type alloc = nullptr) : m_strvec(alloc) { }

    ShoppingList(const ShoppingList& other) = default;
    ShoppingList(std::allocator_arg_t, allocator_type a, const ShoppingList& other)
        : m_strvec(other, a) { }

    allocator_type get_allocator() const
        { return m_strvec.get_allocator().resource(); } 

    void add_item(const string_type& item) { m_strvec.push_back(item); } 
    ...
};

bool operator==(const ShoppingList &a, const ShoppingList &b);
```

There was some discussion in LEWG as to whether it was appropriate to use allocator_type as an alias for something that is not, strictly speaking, an allocator. At the time, I sympathized with this objection and set out to see what the ripple effect would be if a different typedef name were chosen in cases where a class uses memory_resource directly. Unfortunately, the ripple effect is too large, in my opinion, to justify this change. In particular, every class function or constructor that propagates its allocator to a member or element would need to be reworded to use argument names like allocator_or_resource and descriptions with duplicate wording based on whether an allocator or resource pointer were passed in. In effect, we would be undoing in English what we so carefully created in the interface, which is the nearly complete interchangeability of allocators and memory resource pointers.
Next, we create an allocator resource, FixedBufferResource, that allocates memory from a fixed-size buffer supplied at construction. The FixedBufferResource is not responsible for reclaiming this externally managed buffer, and consequently its deallocate method and destructor are no-ops. This makes allocations and deallocations very fast, and is useful when building up an object of a bounded size that will be destroyed all at once (such as one of the short lived shopping lists in this example).

```cpp
class FixedBufferResource : public std::pmr::memory_resource
{
    void* m_next_alloc;
    std::size_t m_remaining;

public:
    FixedBufferResource(void* buffer, std::size_t size)
        : m_next_alloc(buffer), m_remaining(size) { }

protected:
    virtual void* do_allocate(std::size_t sz, std::size_t alignment)
    {
        if (std::align(alignment, sz, m_next_alloc, m_remaining))
        {
            void* ret = m_next_alloc;
            m_next_alloc = static_cast<char*>(m_next_alloc) + sz;
            return ret;
        }
        else
            throw std::bad_alloc();
    }

    virtual void do_deallocate(void*, std::size_t, std::size_t) { }

    virtual bool do_is_equal(std::pmr::memory_resource& other) const noexcept
    {
        return this == &other;
    }
};
```

Now, we use the ShoppingList and FixedBufferResource defined above to demonstrate processing a short-lived shopping list into a collection of shopping lists. We define a collection of shopping lists, folder, that will use the default allocator. The temporary shopping list temporaryShoppingList will use the FixedBufferResource to allocator memory, since the items being added to the list are of a fixed size.

Note that the memory-resource library is designed so that the ShoppingList constructor accepts a pointer to a memory_resource rather than a reference to a memory_resource. It was noted that one common practice is to use references rather than pointers in situations where a null pointer is out of contract. However, there is a more compelling practice of avoiding constructors that take objects by reference and store their addresses. We also want to avoid passing non-const
references, as that, too, is usually considered bad practice (except in overloaded operators).

```cpp
std::pmr::list<ShoppingList> folder; // Default allocator resource
{
    char buffer[1024];
    FixedBufferResource buf_rsrc(&buffer, 1024);
    ShoppingCartList temporaryShoppingList(&buf_rsrc);
    assert(&buf_rsrc == temporaryShoppingList.get_allocator());

    temporaryShoppingList.add_item("salt");
    temporaryShoppingList.add_item("pepper");

    if (processShoppingList(temporaryShoppingList)) {
        folder.push_back(temporaryShoppingList);
        assert(std::pmr::get_default_resource() ==
            folder.back().get_allocator());
    }
}
```

// temporaryShoppingList, buf_rsrc, and buffer go out of scope

Notice that the shopping lists within `folder` use the default allocator resource whereas the shopping list `temporaryShoppingList` uses the short-lived but very fast `buf_rsrc`. Despite using different allocators, you can insert `temporaryShoppingList` into `folder` because they have the same `ShoppingList` type. Also, while `ShoppingList` uses `memory_resource` directly, `std::pmr::list`, `std::pmr::vector`, and `std::pmr::string` all use `polymorphic_allocator`. The resource passed to the `ShoppingList` constructor is propagated to the vector and each string within that `ShoppingList`. Similarly, the resource used to construct `folder` is propagated to the constructors of the `ShoppingLists` that are inserted into the list (and to the strings within those `ShoppingLists`). The `polymorphic_allocator` template is designed to be almost interchangeable with a pointer to `memory_resource`, thus producing a “bridge” between the template-policy style of allocator and the polymorphic-base-class style of allocator.

### 5 Summary of Proposal

#### 5.1 Namespace `std::pmr`

All new components introduced in this proposal are in a new namespace, `pmr`, nested within namespace `std`.

The name, `pmr`, and all other identifiers introduced in this proposal are subject to change. If this proposal is accepted, we can have the bicycle-shed discussion of names. If you think of a better name, send a suggestion to the email address at the top of this paper.
5.2 Abstract base class memory_resource

An abstract base class, memory_resource, describes a memory resource from which blocks can be allocated and deallocated. It provides functions allocate(), deallocate(), and is_equal(), which call pure virtual functions do_allocate(), do_deallocate(), and do_is_equal(), respectively. Derived classes of memory_resource contain the machinery for actually allocating and deallocating memory. Note that memory_resource, not being a template, operates at the level of raw bytes rather than objects. The caller is responsible for constructing objects into the allocated memory and destroying the objects before deallocating the memory.

5.3 Class Template polymorphic_allocator<T>

An instance of polymorphic_allocator<T> is a wrapper around a memory_resource pointer that gives it a C++11 allocator interface. It is this adaptor that achieves the goal of separating an object’s type from its allocator, especially for existing templates that have an allocator template parameter. Two objects x and y of type list<int, polymorphic_allocator<int>> have the same type, but may use different memory resources.

Polymorphic allocators use scoped allocator semantics. Thus, a container containing other containers or strings can be built to use the same memory resource throughout if polymorphic allocators are used ubiquitously.

5.4 Aliases for container classes

There would be an alias in the pmr namespace for each standard container (except array). The alias would not take an allocator parameter but instead would use polymorphic_allocator<T> as the allocator. For example, the <vector> header would contain the following declaration:

```cpp
namespace std {
    namespace pmr {

        template <class T>
        using vector<T> = std::vector<T, polymorphic_allocator<T>>;

    } // namespace pmr
} // namespace std
```

Thus, std::pmr::vector<int> would be a vector that uses a polymorphic allocator. Consistent use of his aliases would allow std::pmr::vector<int> to be used as a vocabulary type, interoperable with all other instances of std::pmr::vector<int>.

Within the LEWG, there was extensive discussion of the desirability of creating same-name aliases within a nested namespace. Proponents argued that the name std::pmr::vector would be cleaner and better accepted than pmr_vector or std::pmr::pmr_vector. Opponents claimed that users were likely to run into ambiguities if both using std; and using std::pmr; were present (though such an
ambiguity would be noisy and thus easy to fix). A straw poll was strongly in favor of leaving the aliases as proposed here (and warning users not to put using std::pmr in their code).

5.5 **Class template resource_adaptor<Alloc>**

An instance of `resource_adaptor<Alloc>` is a wrapper around a C++11 allocator type that gives it a `memory_resource` interface. In a sense, it is the complementary adaptor to `polymorphic_allocator<T>`. The adapted allocator, `Alloc`, is required to use normal (raw) pointers, rather than shared-memory pointers or pointers to some other kind of weird memory. (I have floated the term, *Euclidean Allocator*, to describe allocators such as these 😊.) The `resource_adaptor` template is actually an alias template designed such that `resource_adaptor<X<T>>` and `resource_adaptor<X<U>>` are the same type for all parameters `T` and `U`.

5.6 **Function new_delete_resource()**

Returns a pointer to a memory resource that forwards all calls to `allocate()` and `deallocate()` to global `operator new()` and `operator delete()`, respectively. Every call to this function returns the same value. Since the resource is stateless, all instances of such memory resources would be equivalent and there is never a need for more than one instance in a program.

5.7 **Function null_memory_resource()**

Returns a pointer to a memory resource that always fails with a `bad_alloc` exception when `allocate()` is called. This function is useful for setting the end of a `chain` of memory resource, where one memory resource depends on another. In cases where the first memory resource is not expected to exhaust its own pool of memory, the null memory resource can be used to avoid accidentally allocating memory from the heap. This function is also useful for testing, in situations such as the small-object optimization, where an allocator must be supplied, but is not expected to be used.

5.8 **Functions get_default_resource() and set_default_resource()**

Namespace-scoped functions `get_default_resource()` and `set_default_resource()` are used to get and set a specific memory resource to be used by certain classes when an explicit resource is not specified to the class’s constructor. The ability to change the default resource used when constructing an object is extremely useful for testing and can also be useful for other purposes such as preventing DoS attacks by limiting the maximum size of an allocation.

If `set_default_resource()` is never called, the “default default” memory resource is `new_delete_resource()`.

5.9 **Standard memory resources**

A new library facility for using different types of allocators is useful only to the extent that such allocators actually exist. This proposal, therefore, includes a few memory
resource classes that have broad usefulness in our experience. In the future, we may propose additional resource classes for standardization, including a resource for testing the memory allocation behavior of allocator-aware classes.

5.9.1 Classes synchronized_pool_resource and unsynchronized_pool_resource

The synchronized_pool_resource and unsynchronized_pool_resource classes are general-purpose resources that own the allocated storage and free it on destruction, even if deallocate is not called for some or all of the allocated blocks. Efficiency is obtained by allocating memory in chunks from an “upstream” allocator (often the default allocator) and by maximizing storage locality among separate allocations. A logical data structure would be a set of object pools, but the actual choice of data structure and algorithm is left to the QOI.

5.9.2 Class monotonic_buffer_resource

The monotonic_buffer_resource class is designed for very fast memory allocations in situations where memory is used to build up a few objects and then is released all at once when those objects go out of scope. Like unsynchronized_pool_resource, it owns its memory and it is intended for single-threaded operation. The “monotonic” in its name refers to the fact that its use of memory increases monotonically because its deallocate() member is a no-op. By ignoring deallocation calls, this type of memory resource can use extremely simple data structures that do not require keeping track of individual allocated blocks. In addition, the user can provide it an initial buffer from which to allocate memory. In many applications, this buffer can reside on the stack, providing even more efficient allocation for small amounts of memory.

A particularly good use for a monotonic_buffer_resource is to provide memory for a local variable of container or string type. For example, the following code concatenates two strings, looks for the word “hello” in the concatenated string, and then discards the concatenated string after the word is found or not found. The concatenated string is expected to be no more than 80 bytes long, so the code is optimized for these short strings using a small monotonic_buffer_resource (but will still work, using the default allocator as a backup resource, if the concatenated string is over 80 bytes long):

```cpp
bool find_hello(const std::pmr::string s1, const std::pmr::string s2) {
    char buffer[80];
    monotonic_buffer_resource m(buffer, 80);
    std::pmr::string s(&m);
    s.reserve(s1.length() + s2.length());
    s += s1;
    s += s2;
    return s.find("hello") != pmr::string::npos;
    // s goes out of scope, then m and buffer go out of scope
}
```
5.10 Idiom for type-Erased Allocators

Type-erased allocators, which are used by std::function, std::promise, and std::packaged_task are already implemented internally using polymorphic wrappers. In this proposal, the implicit use of polymorphic wrappers is made explicit (reified). When one of these types is constructed, the caller may supply either a C++11 allocator or a pointer to memory_resource. A new member function, get_memory_resource() will return a pointer to the memory resource or, in the case where a C++11 allocator was provided at construction, a pointer to a resource_adaptor containing the original allocator. This pointer can be used to create other objects using the same allocator. If no allocator or resource was provided at construction, the value of get_default_resource() is used. To complete the idiom, classes that use type-erased allocators will declare

```cpp
typedef erased_type allocator_type;
```

indicating that the class uses allocators, but that the allocator is type-erased. (erased_type is an empty class that exists solely for this purpose.)

6 Impact on the standard

The facilities proposed here are mostly pure extensions to the library except for minor changes to the uses_allocator trait and to types that use type erasure for allocators: function, packaged_task, future, promise and the upcoming filepath type in the file-system TS [N3399]. No core language changes are proposed.

7 Implementation Experience

The implementation of the new memory_resource, resource_adaptor, and polymorphic_allocator features is very straightforward. A prototype implementation based on this paper is available at http://www.halpernwightsoftware.com/WG21/polymorphic_allocator.tgz. The prototype also includes a rework of the gnu function class template to add the functionality described in this proposal. Most of the work in adapting function was in adding allocator support without breaking binary (ABI) compatibility.

The memory_resource, polymorphic_allocator, monotonic_buffer_resource, and unsynchronized_pool_resource classes described in this proposal are minor variations of the facilities that have been in use at Bloomberg for over a decade (See the BSL open-source library). These facilities have dramatically improved testability of software (through the use of test resources) and provided performance benefits when using special-purpose allocators such as arena allocators and thread-specific allocators.

8 Formal Wording – new classes

Throughout this wording, Precondition clauses are distinguished from Requires clauses. The former clause is used for run-time conditions that, if violated, would produce undefined behavior. The latter clause is used for compile-time conditions.
which, if violated, would produce an ill-formed program. This pattern was used in anticipation of an often discussed initiative to distinguish between these clauses in the standard. However, if the committee decides not to pursue this distinction, or decides to defer it until a later time, all Precondition clauses can be replaced with Requires clauses by the editor.

8.1 Utility Class erased_type

Add a new section to the TS describing the following erased_type utility component:

u.1 Header <experimental/utility> synopsis [utility.syn]

```
#include <utility>
namespace std {
namespace experimental {

    // erased-type placeholder
    struct erased_type { };
}
}
```

Note to the editor: Other TS utility components from other papers would be merged into this synopsis here

```
}
}
```

u.2 Class erased_type [erased.type]

```
namespace std {
namespace experimental {
    struct erased_type { };
}
}
```

The erased_type struct is an empty struct that serves as a placeholder for a type $T$ in situations where the actual type $T$ is determined at runtime. For example, the nested type, allocator_type, is an alias for erased_type in classes that use type-erased allocators (see [type.erased.allocator]).

Although the first (and currently only) use of erased_type is in the context of memory allocation, the concept of type erasure is not allocator-specific. Since there may be new uses for this type in the future, I elected to put it in <utility> instead of in <memory>.

8.2 Polymorphic Memory Resources

Add a new subsection in the TS for the polymorphic memory resources.

w.x Polymorphic Memory Resources [memory.resource]

8.2.1 Header <experimental/memory_resource> synopsis

w.x.1 Header <experimental/memory_resource> synopsis [memory.resource.syn]

```
namespace std {
```
namespace experimental {
namespace pmr {

class memory_resource;

bool operator==(const memory_resource& a, const memory_resource& b) noexcept;
bool operator!=(const memory_resource& a, const memory_resource& b) noexcept;

template <class Tp> class polymorphic_allocator;

template <class T1, class T2>
bool operator==(const polymorphic_allocator<T1>& a, const polymorphic_allocator<T2>& b) noexcept;
template <class T1, class T2>
bool operator!=(const polymorphic_allocator<T1>& a, const polymorphic_allocator<T2>& b) noexcept;

// 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<
    allocator_traits<Allocator>::rebind_alloc<char>>;

// Global memory resources
memory_resource* new_delete_resource() noexcept;
memory_resource* null_memory_resource() noexcept;

// The default memory resource
memory_resource* set_default_resource(memory_resource* r) noexcept;
memory_resource* get_default_resource() noexcept;

// Standard memory resources
struct pool_options;
class synchronized_pool_resource;
class unsynchronized_pool_resource;
class monotonic_buffer_resource;

) // namespace pmr
) // namespace experimental
) // namespace std
8.2.2 Class memory_resource

w.x.2.1 Class memory_resource overview

The memory_resource class is an abstract interface to an unbounded set of classes encapsulating memory resources.

namespace std {
    namespace experimental {
        namespace pmr {

            class memory_resource {
                // For exposition only
                static constexpr size_t max_align = alignof(max_align_t);

                public:
                virtual ~memory_resource();
                void* allocate(size_t bytes, size_t alignment = max_align);
                void deallocate(void* p, size_t bytes, size_t alignment = max_align);

                bool is_equal(const memory_resource& other) const noexcept;

                protected:
                virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
                virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;

                virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
            }
        }
    }
}

The use of the pattern whereby a public non-virtual function calls a protected virtual function enables default arguments to be expressed only once, in the abstract base class. It has the additional benefit of allowing an implementation to instrument the function or for the meaning of the function to evolve in the standard without breaking existing derived classes. Finally, this pattern is convenient for specification because it separates the public interface from the derived-class requirements.

w.x.2.2 memory_resource public member functions

~memory_resource();

   Effects: Destroys this memory_resource.

   void* allocate(size_t bytes, size_t alignment = max_align);
**Effects:** equivalent to `return do_allocate(bytes, alignment);`

```c
void deallocate(void* p, size_t bytes, size_t alignment = max_align);
```

**Effects:** equivalent to `do_deallocate(p, bytes, alignment);

```c
bool is_equal(const memory_resource& other) const noexcept;
```

**Effects:** equivalent to `return do_is_equal(other);

```c
w.x.2.3 memory_resource protected virtual member functions [memory.resource.priv]
```

```c
virtual void* do_allocate(size_t bytes, size_t alignment) = 0;

**Preconditions:** alignment shall be a power of two.

**Returns:** A derived class shall implement this function to return a pointer to allocated storage (3.7.4.2) with a size of at least `bytes`. The returned storage is aligned to the specified `alignment`, if such alignment is supported; otherwise it is aligned to `max_align`.

[Note to editor: 3.7.4.2 [basic.stc.dynamic.deallocation] does not seem to actually define *allocated storage*, even though it is referenced in 3.8 [basic.life]. I could not find an actual definition of this term, but from the usage, it seems to mean storage that does not currently have an object constructed in it.]

**Throws:** a derived class implementation shall throw an appropriate exception if it is unable to allocate memory with the requested size and alignment.

```c
virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;

**Preconditions:** `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:** A derived class shall implement this function to dispose of allocated storage.

**Throws:** Nothing.

Although this function throws nothing, it is not declared `noexcept` because it has a narrow interface. An implementation may choose to throw if a defensive test of the preconditions fails.

```c
virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;

**Returns:** A derived class shall implement this function to return `true` if memory allocated from `this` can be deallocated from `other` and vice-versa; otherwise it shall return `false`. [Note: The most-derived type of `other` might not match the type of `this`. For a derived class, `D`, a typical implementation of this function will compute `dynamic_cast<D*>(&other)` and go no further (i.e., return `false`) if it returns `nullptr.– end note]

For most classes derived from `memory_resource`, `do_is_equal` will return exactly this `== &other`. I.e., most memory resources are equal only if they are the same object. The `resource_adaptor` template (below) is a rare exception.

```c
w.x.2.4 memory_resource equality [memory.resource.eq]
```

```c
bool operator==(const memory_resource& a, const memory_resource& b) noexcept;

**Returns:** `&a == &b || a.is_equal(b).`
The explicit optimization of testing for \&a == \&b means that the implementation shall not invoke is_equal if the pointers compare equal. If this test were not explicit, then this important optimization would actually be illegal because the number of calls to is_equal is user-detectible.

\[
\text{bool operator!=(const memory_resource& a, const memory_resource& b) noexcept;}
\]

\textit{Returns:} !(a == b).

8.2.3 Class template polymorphic_allocator

w.x.3 Class template polymorphic_allocator [polymorphic_allocator.class]

w.x.3.1 Class template polymorphic_allocator overview [polymorphic_allocator.overview]

A specialization of class template pmr::polymorphic_allocator conforms to the Allocator requirements ([allocator.requirements] 17.6.3.5). Constructed with different memory resources, different instances of the same specialization of pmr::polymorphic_allocator can exhibit entirely different allocation behavior. This runtime polymorphism allows objects that use polymorphic_allocator to behave as if they used different allocator types at run time even though they use the same static allocator type.

```cpp
class polymorphic_allocator
{
    memory_resource* m_resource; // For exposition only

public:
    typedef Tp value_type;

    polymorphic_allocator() noexcept;
    polymorphic_allocator(memory_resource* r);
    polymorphic_allocator(const polymorphic_allocator& other) = default;
    template <class U>
        polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;

    polymorphic_allocator&
        operator=(const polymorphic_allocator& rhs) = default;

    Tp* allocate(size_t n);
    void deallocate(Tp* p, size_t n);
    template <typename T, typename... Args>
        void construct(T* p, Args&&... args);
```
// Specializations for pair using piecewise construction
template <class T1, class T2, class... Args1, class... Args2>
void construct(pair<T1,T2>* p, piecewise_construct_t,
tuple<Args1...> x, tuple<Args2...> y);
template <class T1, class T2>
void construct(pair<T1,T2>* p);
template <class T1, class T2, class U, class V>
void construct(pair<T1,T2>* p, U&& x, V&& y);
template <class T1, class T2, class U, class V>
void construct(pair<T1,T2>* p,
const std::pair<U, V>& pr);
template <class T1, class T2, class U, class V>
void construct(pair<T1,T2>* p, pair<U, V>&& pr);

template <typename T>
void destroy(T* p);

// Return a default-constructed allocator (no allocator propagation)
polymorphic_allocator select_on_container_copy_construction() const;

memory_resource* resource() const;
};
} // namespace pmr
} // namespace experimental
} // namespace std

w.x.3.2 polymorphic_allocator constructors [polymorphic.allocator.ctor]
polymorphic_allocator() noexcept;

Effects: Sets m_resource to get_default_resource().
polymorphic_allocator(memory_resource* r);

Precondition: r is non-null.
Effects: Sets m_resource to r.
Throws: nothing
Note: This constructor provides an implicit conversion from memory_resource*.
template <class U>
polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;

Effects: sets m_resource to other.resource().

w.x.3.3 polymorphic_allocator member functions [polymorphic.allocator.mem]
Tp* allocate(size_t n);

Returns: Equivalent to static_cast<Tp*>(m_resource->allocate(n * sizeof(Tp),
alignof(Tp))}.
void deallocate(Tp* p, size_t n);

**Preconditions:** p was allocated from a memory resource, x, equal to *m_resource, using
x.allocate(n * sizeof(Tp), alignof(Tp)).

**Effects:** Equivalent to m_resource->deallocate(p, n * sizeof(Tp), alignof(Tp)).

**Throws:** Nothing.

template <class T, class... Args>
void construct(T* p, Args&&... args);

**Requires:** uses-allocator construction of T with allocator this->resource() (see
[modsallocator.uses]) and constructor arguments std::forward<Args>(args)... is well-
formed. [Note: uses-allocator construction is always well formed for types that do not use allocators. –
end note]

**Effects:** Construct a T object at p by uses-allocator construction with allocator this->resource()
([allocatorusesconstruction] 20.6.7.2) and constructor arguments std::forward<Args>(args)....

**Throws:** Nothing unless the constructor for T throws.

template <class T1, class T2, class... Args1, class... Args2>
void construct(pair<T1,T2>* p, piecewise_construct_t,
tuple<Args1...> x, tuple<Args2...> y);

**Effects:** Let xprime be a tuple constructed from x according to the appropriate rule from the
following list. [Note: The following description can be summarized as constructing a
std::pair<T1,T2> object at p as if by separate uses-allocator construction with allocator this->resource() ([allocatorusesconstruction] 20.6.7.2) of p->first using the elements of x and
p->second using the elements of y. – end note]:

- If uses_allocator<T1,memory_resource*>::value is false and
  is_constructible<T,Args1...>::value is true, then xprime is x.

- Otherwise, if (uses_allocator<T1,memory_resource*>::value is true and
  is_constructible<T1,allocator_arg_t,memory_resource*,Args1...
  >::value is true, then xprime is tuple_cat(make_tuple(allocator_arg, this->resource())), move(x)).

- Otherwise, if (uses_allocator<T1,memory_resource*>::value is true and
  is_constructible<T1,Args1...,memory_resource*>::value) is true, then
  xprime is tuple_cat(move(x), make_tuple(this->resource()))).

- Otherwise the program is ill formed.

and let yprime be a tuple constructed from y according to the appropriate rule from the following list:

- If uses_allocator<T2,memory_resource*>::value is false and
  is_constructible<T,Args2...>::value is true, then yprime is y.

- Otherwise, if (uses_allocator<T2,memory_resource*>::value is true and
  is_constructible<T2,allocator_arg_t,memory_resource*,Args2...
  >::value) is true, then yprime is tuple_cat(make_tuple(allocator_arg, this->resource())), move(y)).
— Otherwise, if `uses_allocator<T2, memory_resource*>::value` is true and
  `is_constructible<T2, Args2..., memory_resource*>::value` is true, then
  `yprime` is `tuple_cat(move(y), make_tuple(this->resource()))`.
— Otherwise the program is ill formed.

then this function constructs a `std::pair<T1, T2>` object at `p` using constructor arguments
`piecewise_construct, xprime, yprime`.

The description above is almost identical to that in `scoped_allocator_adaptor` because a `polymorphic_allocator` is scoped. It differs in that, instead of passing
*this down to the constructed object, it passes `this->resource()`.

```cpp
template <class T1, class T2>
void construct(std::pair<T1, T2>* p);
  Effects: equivalent to `this->construct(p, piecewise_construct, tuple<>());`

template <class T1, class T2, class U, class V>
void construct(std::pair<T1, T2>* p, U&& x, V&& y);
  Effects: equivalent to `this->construct(p, piecewise_construct,
  forward_as_tuple(std::forward<U>(x)), forward_as_tuple(std::forward<V>(y)));`

template <class T1, class T2, class U, class V>
void construct(std::pair<T1, T2>* p, const std::pair<U, V>& pr);
  Effects: equivalent to `this->construct(p, piecewise_construct,
  forward_as_tuple(pr.first), forward_as_tuple(pr.second));`

template <class T1, class T2, class U, class V>
void construct(std::pair<T1, T2>* p, std::pair<U, V>&& pr);
  Effects: equivalent to `this->construct(p, piecewise_construct,
  forward_as_tuple(std::forward<U>(pr.first)),
  forward_as_tuple(std::forward<V>(pr.second)));`

template <typename T>
void destroy(T* p);
  Effects: `p->~T()`.

polymorphic_allocator select_on_container_copy_construction() const;
  Returns: `polymorphic_allocator()`.

memory_resource* resource() const;
  Returns: `m_resource`.

w.x.3.4 polymorphic_allocator equality [polymorphic_allocator.eq]

```cpp
template <class T1, class T2>
bool operator==(const polymorphic_allocator<T1>& a,
                const polymorphic_allocator<T2>& b) noexcept;
  Returns: `*a.resource() == *b.resource()`.
```
bool operator!=(const polymorphic_allocator<T1>& a,  
    const polymorphic_allocator<T2>& b) noexcept;

Returns: ! (a == b)

8.2.4 Class-alias template resource_adaptor

w.x.4 template alias resource_adaptor [resource.adaptor]

w.x.4.1 resource_adaptor [resource.adaptor.overview]

An instance of resource_adaptor<Allocator> is an adaptor that wraps a memory_resource  
interface around Allocator. In order 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 char 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 definition of the members of that class, whatever its name, are normative.

In addition to the Allocator requirements ([allocator.requirements] 17.6.3.4), the 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*.

namespace std {
namespace experimental {
namespace pmr {

// The name resource_adaptor_imp is for exposition only.

template <class Allocator>
  class resource_adaptor_imp : public memory_resource {

  // for exposition only
  Allocator m_alloc;

  public:
    typedef Allocator allocator_type;

    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);

```
\begin{verbatim}
resource_adaptor_imp& operator=(const resource_adaptor_imp&)
    = default;

allocator_type get_allocator() const { return m_alloc; }

protected:
    virtual void* do_allocate(size_t bytes, size_t alignment);
    virtual void do_deallocate(void* p, size_t bytes,
                                size_t alignment);

    virtual bool do_is_equal(const memory_resource& other) const
        noexcept;
};

template <class Allocator>
using resource_adaptor = typename
    resource_adaptor_imp<
        allocator_traits<Allocator>::template rebind_alloc<char>>;

} // namespace pmr
} // namespace experimental
} // namespace std
\end{verbatim}

\textbf{w.x.4.2 resource_adaptor_imp constructors [resource.adaptor ctor]}
\begin{verbatim}
explicit resource_adaptor_imp(const Allocator& a2);
    \textit{Effects:} Initializes \texttt{m\_alloc} with \texttt{a2}.

explicit resource_adaptor_imp(Allocator&& a2);
    \textit{Effects:} Initializes \texttt{m\_alloc} with \texttt{std::move(a2)}.
\end{verbatim}

\textbf{w.x.4.3 resource_adaptor_imp member functions [resource.adaptor mem]}
\begin{verbatim}
void* do_allocate(size_t bytes, size_t alignment);
    \textit{Returns:} Allocated memory obtained by calling \texttt{m\_alloc\_allocate}. The size and alignment of the allocated memory shall meet the requirements for a class derived from \texttt{memory_resource ([memory.resource])}.

void do_deallocate(void* p, size_t bytes, size_t alignment);
    \textit{Requires:} \texttt{p} was previously allocated using \texttt{A\_allocate}, where \texttt{A == m\_alloc}, and not subsequently deallocated.
    \textit{Effects:} Returns memory to the allocator using \texttt{m\_alloc\_deallocate()}.

bool do_is_equal(const memory_resource& other) const;
    \textit{Let} \texttt{p} be \texttt{dynamic\_cast<const resource\_adaptor\_imp*>(\&other)}.
    \textit{Returns:} false if \texttt{p} is null, otherwise the value of \texttt{m\_alloc \texttt{== p\_m\_alloc}}.
\end{verbatim}
### 8.2.5 Program-wide memory_resource objects

w.x.5 Access to program-wide memory_resource objects [memory.resource.global]

```cpp
memory_resource* new_delete_resource() noexcept;
```

*Returns:* A pointer to a static-duration object of a type derived from memory_resource that can serve as a resource for allocating memory using `::operator new` and `::operator delete`. The same value is returned every time this function is called. For return value `p` and memory resource `r`, `p->is_equal(r)` returns `&r == p`.

```cpp
memory_resource* null_memory_resource() noexcept;
```

*Returns:* A pointer to a static-duration object of a type derived from memory_resource for which `allocate()` always throws `bad_alloc` and for which `deallocate()` has no effect. The same value is returned every time this function is called. For return value `p` and memory resource `r`, `p->is_equal(r)` returns `&r == p`.

A memory resource may obtain memory using another resource for replenishing its pool. The null memory resource is useful for situations where the original pool is not expected to become exhausted.

The *default memory resource pointer* is a pointer to a memory resource that is used by certain facilities when an explicit memory resource is not supplied through the interface. Its initial value is the return value of `new_delete_resource()`.

```cpp
memory_resource* set_default_resource(memory_resource* r) noexcept;
```

*Effects:* If `r` is non-null, sets the value of the default memory resource pointer to `r`, otherwise sets the default memory resource pointer to `new_delete_resource()`.

*Post-condition:* `get_default_resource() == r`.

*Returns:* The previous value of the default memory resource pointer.

*Remarks:* Calling the `set_default_resource` and `get_default_resource` functions shall not incur a data race. A call to the `set_default_resource` function shall synchronize with subsequent calls to the `set_default_resource` and `get_default_resource` functions.

These synchronization requirements are the same as for `set/get_new_handler` and `set/get_terminate`.

```cpp
memory_resource* get_default_resource() noexcept;
```

*Returns:* The current value of the default memory resource pointer.

### 8.3 Classes synchronized_pool_resource and unsynchronized_pool_resource

w.x.6 Pool resource classes [pool.resource]

w.x.6.1 Classes synchronized_pool_resource and unsynchronized_pool_resource [pool.resource.overview]

The `synchronized_pool_resource` and `unsynchronized_pool_resource` classes (collectively, *pool resource classes*) are general-purpose memory resources having the following qualities:

— Each resource *owns* the allocated memory, and frees it on destruction – even if `deallocate` has not been called for some of the allocated blocks.
A pool resource (see Figure 1) consists of a collection of pools, serving requests for different block sizes. Each individual pool manages a collection of chunks that are in turn divided into blocks of uniform size, returned via calls to do_allocate. Each call to do_allocate(size, alignment) is dispatched to the pool serving the smallest blocks accommodating at least size bytes.

When a particular pool is exhausted, allocating a block from that pool results in the allocation of an additional chunk of memory from the upstream allocator (supplied at construction), thus replenishing the pool. With each successive replenishment, the chunk size obtained increases geometrically. [Note: By allocating memory in chunks, the pooling strategy increases the chance that consecutive allocations will be close together in memory. – end note]

Allocation requests that exceed the largest block size of any pool are fulfilled directly from the upstream allocator.

A pool_options struct may be passed to the pool resource constructors to tune the largest block size and the maximum chunk size.

[Example: Figure 1 shows a possible data structure that implements a pool resource.

![Figure 1: pool resource](image)

A synchronized_pool_resource may be accessed from multiple threads without external synchronization and may have thread-specific pools to reduce synchronization costs. An unsynchronized_pool_resource class may not be accessed from multiple threads simultaneously and thus avoids the cost of synchronization entirely in single-threaded applications.

```cpp
namespace std {
namespace experimental {
namespace pmr {
```
struct pool_options
{
    size_t max_blocks_per_chunk = 0;
    size_t largest_required_pool_block = 0;
};

By bundling the options together into a struct, the interface is substantially more future-proof than if the options were specified individually in the constructors. It is much easier to add (named) fields to a struct than to overload on a list of (unnamed) arguments to a constructor.

class synchronized_pool_resource : public memory_resource
{
    public:
        synchronized_pool_resource(const pool_options& opts,
                                    memory_resource* upstream);

        synchronized_pool_resource()
        : synchronized_pool_resource(pool_options(),
                                    get_default_resource()) { }

        explicit synchronized_pool_resource(memory_resource* upstream)
        : synchronized_pool_resource(pool_options(), upstream) { }

        explicit synchronized_pool_resource(pool_options& opts)
        : synchronized_pool_resource(opts,
                                    get_default_resource()) { }

        synchronized_pool_resource(
            const synchronized_pool_resource&) = delete;
    virtual ~synchronized_pool_resource();

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

        void release();
        memory_resource* upstream_resource() const;
        pool_options options() const;

    protected:
        virtual void* do_allocate(size_t bytes, size_t alignment);
        virtual void  do_deallocate(void* p, size_t bytes,
                                    size_t alignment);

        virtual bool do_is_equal(const memory_resource& other) const
            noexcept;
};

class unsynchronized_pool_resource : public memory_resource
{
    public:
        unsynchronized_pool_resource(const pool_options& opts,
memory_resource* upstream);

unsynchronized_pool_resource()
    : unsynchronized_pool_resource(pool_options(),
        get_default_resource()) { }
explicit unsynchronized_pool_resource(memory_resource* upstream)
    : unsynchronized_pool_resource(pool_options(), upstream) { }
explicit unsynchronized_pool_resource(const pool_options& opts)
    : unsynchronized_pool_resource(opts,
        get_default_resource()) { }

unsynchronized_pool_resource(
    const unsynchronized_pool_resource&) = delete;
virtual ~unsynchronized_pool_resource();
unsynchronized_pool_resource& operator=(
    const unsynchronized_pool_resource&) = delete;
void release();
memory_resource* upstream_resource() const;
pool_options     options() const;

protected:
    virtual void*
        do_allocate(size_t bytes, size_t alignment);
    virtual void
        do_deallocate(void* p, size_t bytes,
                       size_t alignment);

    virtual bool do_is_equal(const memory_resource& other) const
        noexcept;
};

} // namespace pmr
} // namespace experimental
} // namespace std

w.x.6.2 pool_options data members

The members of pool_options comprise a set of constructor options for pool resources. The effect of each option on the pool resource behavior is described below:

size_t max_blocks_per_chunk;
The maximum number of blocks that will be allocated at once from the upstream memory resource to replenish a pool.

If the value of max_blocks_per_chunk is zero or is greater than an implementation-defined limit, that limit is used instead. The implementation may choose to use a smaller value than is specified in this field and may use different values for different pools.

size_t largest_required_pool_block;
The largest allocation size that is required to be fulfilled using the pooling mechanism. Attempts to allocate a single block larger than this threshold will be allocated directly from the upstream memory
If \texttt{largest\_required\_pool\_block} is zero or is greater than an implementation-defined limit, that limit is used instead. The implementation may choose a pass-through threshold larger than specified in this field.

\textbf{w.x.6.3 pool resource constructors and destructors [pool.ctor]}

\begin{verbatim}
synchronized_pool_resource(const pool_options& opts,
   memory_resource* upstream);
unsynchronized_pool_resource(const pool_options& opts,
   memory_resource* upstream);
\end{verbatim}

\textit{Precondition:} upstream is the address of a valid memory resource.

\textit{Effects:} Constructs a pool resource object that will obtain memory from \texttt{upstream} whenever the pool resource is unable to satisfy a memory request from its own internal data structures. The resulting object will hold a copy of \texttt{upstream}, but will not own the resource to which \texttt{upstream} points. [\textit{Note:} The intention is that calls to \texttt{upstream->allocate()} will be substantially fewer than calls to \texttt{this->allocate()} in most cases. – end note] The behavior of the pooling mechanism is tuned according to the value of the \texttt{opts} argument.

\textit{Throws:} Nothing unless \texttt{upstream->allocate()} throws. It is unspecified if or under what conditions this constructor calls \texttt{upstream->allocate()}.

\begin{verbatim}
virtual ~synchronized_pool_resource();
virtual ~unsynchronized_pool_resource();
\end{verbatim}

\textit{Effects:} calls \texttt{this->release()}.

\textbf{w.x.6.4 pool resource members [pool.mem]}

\begin{verbatim}
void release();
\end{verbatim}

\textit{Effects:} Calls \texttt{upstream\_resource()->deallocate()} as necessary to release all allocated memory. [\textit{Note:} memory is released back to \texttt{upstream\_resource()} even if \texttt{deallocate} has not been called for some of the allocated blocks. – end note]

\begin{verbatim}
memory_resource* upstream\_resource() const;
\end{verbatim}

\textit{Returns:} The value of the \texttt{upstream} argument provided to the constructor of this object.

\begin{verbatim}
pool_options options() const;
\end{verbatim}

\textit{Returns:} The options that control the pooling behavior of this resource. The values in the returned \texttt{struct} may differ from those supplied to the pool resource constructor in that values of zero will be replaced with implementation-defined defaults and sizes may be rounded to unspecified granularity.

\begin{verbatim}
virtual void* do_allocate(size_t bytes, size_t alignment);
\end{verbatim}

\textit{Returns:} A pointer to allocated storage (3.7.4.2) with a size of at least \texttt{bytes}. The size and alignment of the allocated memory shall meet the requirements for a class derived from \texttt{memory\_resource} ([memory.resource]).

\textit{Effects:} If the pool selected for a block of size \texttt{bytes} is unable to satisfy the memory request from its own internal data structures, it will call \texttt{upstream\_resource()-\texttt{allocate}} to obtain more memory. If \texttt{bytes} is larger than that which the largest pool can handle, then memory will be allocated using \texttt{upstream\_resource()-\texttt{allocate}}.

\textit{Throws:} Nothing unless \texttt{upstream\_resource()-\texttt{allocate}} throws.
virtual void do_deallocate(void* p, size_t bytes, size_t alignment);

Effects: Return the memory at p to the pool. It is unspecified if or under what circumstances this operation will result in a call to upstream_resource() -> deallocate().

Throws: Nothing

ev不出 bool

unsynchronized_pool_resource::do_is_equal(const memory_resource& other)
    const noexcept;

Returns: this == dynamic_cast<unsynchronized_pool_resource*>(&other).

virtual bool

synchronized_pool_resource::do_is_equal(const memory_resource& other)
    const noexcept;

Returns: this == dynamic_cast<synchronized_pool_resource*>(&other).

8.4 Class monotonic_buffer_resource

w.x.7 Class monotonic_buffer_resource [monotonic.buffer]

w.x.7.1 Class monotonic_buffer_resource overview [monotonic.buffer]

A monotonic_buffer_resource is a special-purpose memory resource intended for very fast memory allocations in situations where memory is used to build up a few objects and then is released all at once when the memory resource object is destroyed. It has the following qualities:

— A call to deallocate has no effect, thus the amount of memory consumed increases monotonically until the resource is destroyed.

— The program can supply an initial buffer, which the allocator uses to satisfy memory requests.

— When the initial buffer (if any) is exhausted, it obtains additional buffers from an upstream memory resource supplied at construction. Each additional buffer is larger than the previous one, following a geometric progression.

— It is intended for access from one thread of control at a time. Specifically, calls to allocate and deallocate do not synchronize with one another.

— It owns the allocated memory and frees it on destruction, even if deallocate has not been called for some of the allocated blocks.

namespace std {
namespace experimental {
namespace pmr {

class monotonic_buffer_resource : public memory_resource {
{
    memory_resource* upstream_rsrc;  // exposition only
    void* current_buffer;  // exposition only
    size_t next_buffer_size;  // exposition only

public:
    explicit monotonic_buffer_resource(memory_resource* upstream);
    monotonic_buffer_resource(size_t initial_size,
                               memory_resource* upstream);
monotonic_buffer_resource(void* buffer, size_t buffer_size,
        memory_resource* upstream);

monotonic_buffer_resource()
        : monotonic_buffer_resource(get_default_resource()) { }
explicit monotonic_buffer_resource(size_t initial_size)
        : monotonic_buffer_resource(initial_size,
                        get_default_resource()) { }
monotonic_buffer_resource(void* buffer, size_t buffer_size)
        : monotonic_buffer_resource(buffer, buffer_size,
                        get_default_resource()) { }

monotonic_buffer_resource(const monotonic_buffer_resource&) = delete;
virtual ~monotonic_buffer_resource();

monotonic_buffer_resource operator=(
        const monotonic_buffer_resource&) = delete;

void release();
memory_resource* upstream_resource() const;

protected:
virtual void* do_allocate(size_t bytes, size_t alignment);
virtual void do_deallocate(void* p, size_t bytes,
        size_t alignment);

virtual bool do_is_equal(const memory_resource& other) const
        noexcept;
};

} // namespace pmr
} // namespace experimental
} // namespace std

w.x.7.2 monotonic_buffer_resource constructor and destructor [monotonic.buffer ctor]
explicit monotonic_buffer_resource(memory_resource* upstream);
monotonic_buffer_resource(size_t initial_size, memory_resource* upstream);

Preconditions: upstream shall be the address of a valid memory resource; initial_size, if specified, shall be greater than zero.

Effects: Sets upstream_rsrc to upstream and current_buffer to nullptr. If initial_size is specified, sets next_buffer_size to at least initial_size; otherwise sets next_buffer_size to an implementation-defined size.
monotonic_buffer_resource(void* buffer, size_t buffer_size,
        memory_resource* upstream);

Preconditions: upstream shall be the address of a valid memory resource. buffer_size shall be no larger than the number of bytes in buffer.
Effects: Sets upstream_rs to upstream, current_buffer to buffer, and next_buffer_size to initial_size (but not less than 1), then increases next_buffer_size by an implementation-defined growth factor (which need not be integral).

~monotonic_buffer_resource();

Effects: Calls this->release().

w.x.7.3 monotonic_buffer_resource members [monotonic_buffer_mem]

void release();

Effects: Calls upstream_rs->deallocate() as necessary to release all allocated memory.

[Note: memory is released back to upstream_rs even if some blocks that were allocated from this have not been deallocated from this.—end note]

memory_resource* upstream_resource() const;

Returns: the value of upstream_rs.

void* do_allocate(size_t bytes, size_t alignment);

Returns: A pointer to allocated storage (3.7.4.2) with a size of at least bytes. The size and alignment of the allocated memory shall meet the requirements for a class derived from memory_resource ([memory.resource]).

Effects: If the unused space in current_buffer can fit a block with the specified bytes and alignment, then allocate the return block from current_buffer; otherwise set current_buffer to upstream_rs->allocate(n, m), where n is not less than max(bytes, next_buffer_size) and m is not less than alignment, and increase next_buffer_size by an implementation-defined growth factor (which need not be integral), then allocate the return block from the newly-allocated current_buffer.

Throws: Nothing unless upstream_rs->allocate() throws.

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

Effects: None

Throws: Nothing

Remarks: Memory used by this resource increases monotonically until its destruction.

bool do_is_equal(const memory_resource& other) const noexcept;

Returns: this == dynamic_cast<monotonic_buffer_resource*>(&other).

8.5 String Aliases Using Polymorphic Allocators

Create an experimental extension to <string> to add variations of the standard string types that allocate memory using pmr::polymorphic_allocator:

w.x.8 Header <experimental/string> synopsis:

#include <string>

namespace std {
namespace experimental {
namespace pmr {

N3916: Polymorphic Memory Resources - r2
// basic_string using polymorphic allocator in namespace pmr
template <class charT, class traits = char_traits<charT>>
using basic_string =
    std::basic_string<charT, traits, polymorphic_allocator<charT>>;

// basic_string typedef names using polymorphic allocator in namespace
// std::experimental::pmr
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;

} // namespace std
} // namespace pmr
} // namespace experimental

With this change pmr::wstring is a wstring that uses a polymorphic allocator.

8.6 Containers Aliases Using Polymorphic Allocators

Create experimental extensions to most of the container headers to add variations of
the standard containers that allocate memory using pmr::polymorphic_allocator:

w.x.9 Header <experimental/deque> synopsis [deque.syn]
#include <deque>

namespace std {
namespace experimental {
namespace pmr {

    template <class T>
    using deque = std::deque<T,polymorphic_allocator<T>>;

};
} // namespace pmr
} // namespace experimental

w.x.10 Header <experimental/forward_list> synopsis [forward_list.syn]
#include <forward_list>

namespace std {
namespace experimental {
namespace pmr {

    template <class T>
    using forward_list =
        std::forward_list<T,polymorphic_allocator<T>>;

};
} // namespace pmr
} // namespace experimental


w.x.11 Header <experimental/list> synopsis [list.syn]
#include <list>

namespace std {
namespace experimental {
namespace pmr {

    template <class T>
    using list = std::list<T,polymorphic_allocator<T>>;

}
}
}

w.x.12 Header <experimental/vector> synopsis [vector.syn]
#include <vector>

namespace std {
namespace experimental {
namespace pmr {

    template <class T>
    using vector = std::vector<T,polymorphic_allocator<T>>;

}
}
}

w.x.13 Header <experimental/map> synopsis [map.syn]
#include <map>

namespace std {
namespace experimental {
namespace pmr {

    template <class Key, class T, class Compare = less<Key>>
    using map = std::map<Key, T, Compare,
                         polymorphic_allocator<pair<const Key,T>>>;

    template <class Key, class T, class Compare = less<Key>>
    using multimap = std::multimap<Key, T, Compare,
                                    polymorphic_allocator<pair<const Key,T>>>;

}
}
}

w.x.14 Header <experimental/set> synopsis [set.syn]
#include <set>

namespace std {


namespace experimental {
namespace pmr {

    template <class Key, class Compare = less<Key>>
    using set = std::set<Key, Compare,
        polymorphic_allocator<Key>>;

    template <class Key, class Compare = less<Key>>
    using multiset = std::multiset<Key, Compare,
        polymorphic_allocator<Key>>;

}
}

w.x.15 Header <experimental/unordered_map> synopsis [unordered_map.syn]
#include <unordered_map>
namespace std {
namespace experimental {
namespace pmr {

    template <class Key, class T,
        class Hash = hash<Key>,
        class Pred = std::equal_to<Key>>
    using unordered_map =
        std::unordered_map<Key, T, Hash, Pred,
            polymorphic_allocator<pair<const Key,T>>>;

    template <class Key, class T,
        class Hash = hash<Key>,
        class Pred = std::equal_to<Key>>
    using unordered_multimap =
        std::unordered_multimap<Key, T, Hash, Pred,
            polymorphic_allocator<pair<const Key,T>>>;

}
}
}

w.x.16 Header <experimental/unordered_set> synopsis [unordered_set.syn]
#include <unordered_set>
namespace std {
namespace experimental {
namespace pmr {

    template <class Key,
        class Hash = hash<Key>,
        class Pred = equal_to<Key>>
    using unordered_set =
std::unordered_set<Key, Hash, Pred,
polymorphic_allocator<Key>>;

template <class Key,
class Hash = hash<Key>,
class Pred = equal_to<Key>>
using unordered_multiset =
std::unordered_multiset<Key, Hash, Pred,
polymorphic_allocator<Key>>;

w.x.16 Header <experimental/regex> synopsis [regex.syn]
#include <regex>
#include <experimental/string>

namespace std {
namespace experimental {
namespace pmr {

template <class BidirectionalIterator>
using match_results =
std::match_results<BidirectionalIterator,
polymorphic_allocator<sub_match<BidirectionalIterator>>>;

typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t*> wcmatch;
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;
}
}
}

8.7 Type-erased allocators

Insert a new section into the TS as follows:

x.y.z Type-erased allocator [type.erased.allocation]

A type-erased allocator is an allocator or memory resource, alloc, used to allocate internal data structures for an object X of type C, but where C is not dependent on the type of alloc. Once alloc has been supplied to X (typically as a constructor argument), alloc can be retrieved from X only as a pointer rp.ptr of static type std::experimental::pmr::memory_resource* ([memory.resource.class]). The process by which rp.ptr is computed from alloc depends on the type of alloc as described in Table Q:

Table Q – Computed memory_resource for type-erased allocator

<table>
<thead>
<tr>
<th>If the type of alloc is</th>
<th>then the value of rp.ptr is</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
If the type of alloc is
then the value of rptr is

<table>
<thead>
<tr>
<th>Type of alloc</th>
<th>Value of rptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-existent – no alloc specified</td>
<td>The value of experimental::pmr::get_default_resource() at the time of construction.</td>
</tr>
<tr>
<td>nullptr_t</td>
<td>The value of experimental::pmr::get_default_resource() at the time of construction.</td>
</tr>
<tr>
<td>a pointer type convertible to pmr::memory_resource*</td>
<td>static_cast<a href="">experimental::pmr::memory_resource*</a>(alloc)</td>
</tr>
<tr>
<td>pmr::polymorphic_allocator&lt;U&gt;</td>
<td>alloc.resource()</td>
</tr>
<tr>
<td>any other type meeting the Allocator requirements ([allocator.requirements])</td>
<td>a pointer to a value of type experimental::pmr::resource_adaptor&lt;A&gt; where A is the type of alloc. rptr remains valid only for the lifetime of X.</td>
</tr>
<tr>
<td>None of the above</td>
<td>The program is ill-formed.</td>
</tr>
</tbody>
</table>

Additionally, class C shall meet the following requirements:
- C::allocator_type shall be identical to std::experimental::erased_type.
- X.get_memory_resource() returns rptr.

9 Formal wording – Changes to classes in the standard

Although this proposal is targeted towards a TS, there are a small number of C++14 standard library classes that would need to be adjusted in order for users of the TS to get maximum value from the features proposed here. These changes are expressed as deltas from the standard, but new classes are still within the experimental namespace. Implementers might wish have a macro to turn these new features on or off, depending on whether TS functionality is desired. (Such a macro could be added to the feature-test recommendations listed in N3694.)

Note: the section numbers below are relative to the October 2013 Committee Draft, N3797 and will need to be updated when the FDIS and eventually the IS is issued.

Add the following section to the TS:

**t.u Departure from the ISO standard [standard.departure]**

**t.u.1 In general**

Although most of the facilities described in this technical specification are strictly supplements to the ISO C++ Standard, a few facilities depend on extensions to entities within the standard itself. The following sections describe these extensions by quoting the affected parts of the standard and using **underlining** to represent added text and **strike-through** to represent deleted text.

The description above is a placeholder for text that might appear only once within the TS. I expect the project editors to edit or replace it as appropriate.

**t.u.2 Uses-allocator construction [mods.allocate.uses]**

The following changes to the uses_allocator trait and to the description of uses-allocator construction allow a memory_resource pointer act as an allocator in many circumstances. **[Note: Existing programs that uses standard allocators would be unaffected by this change. – end note]**
20.7.7 uses_allocator [allocator.uses]

20.7.7.1 uses_allocator trait [allocator.uses.trait]

```cpp
template <class T, class Alloc> struct uses_allocator;
```

**Remark:** automatically detects whether \( T \) has a nested `allocator_type` that is convertible from `Alloc`. Meets the `BinaryTypeTrait` requirements (20.9.1). The implementation shall provide a definition that is derived from `true_type` if a type `T::allocator_type` exists and `either` is `convertible<Alloc, T::allocator_type>::value` `false` or `T::allocator_type` `alias` `std::experimental::erased_type` ([utility.erasd_type]), otherwise it shall be derived from `false_type`. A program may specialize this template to derive from `true_type` for a user-defined type `T` that does not have a nested `allocator_type` but nonetheless can be constructed with an allocator where either:

- the first argument of a constructor has type `allocator_arg_t` and the second argument has type `Alloc` or
- the last argument of a constructor has type `Alloc`.

20.7.7.2 uses_allocator construction [allocator.uses.construction]

*Uses-allocator construction with allocator* `Alloc` refers to the construction of an object `obj` of type `T`, using constructor arguments `v1`, `v2`, ..., `vN` of types `V1`, `V2`, ..., `VN`, respectively, and an allocator `alloc` of type `Alloc`, where `Alloc` either (1) meets the requirements of an allocator ([allocator.requirements]), or (2) is a pointer type convertible to `std::experimental::pmr::memory_resource*` ([polymorphic.allocator]), according to the following rules:

The new text for *Uses-allocator construction* is not strictly necessary, but it is intended to clarify that two different kinds of thing can be passed as `alloc` in `uses-allocator construction`.

9.1 Type-erased allocator for function

t.u.3 Additions to `std::function` [mods.func.wrap]

In section 20.9.11.2 [func.wrap.func], the following declarations are added as public members of class template `function`:

```cpp
    typedef experimental::erased_type_allocator_type;

    experimental::pmr::memory_resource* get_memory_resource();
```

In section 20.9.11.2.1 [func.wrap.func.con], the introductory paragraph is changed as follows, giving the constructors of the `function` class template support for a type-erased allocator:

When a `function` constructor that takes a first argument of type `allocator_arg_t` is invoked, the second argument is treated as a type-erased allocator ([type.erasd_allocator]), shall have a type that conforms to the requirements for Allocator (Table 17.6.3.5). A copy of the allocator argument is used to allocate memory, if necessary, for the internal data structures of the constructed function object. If the constructor moves or makes a copy of a `function` object (including an instance of the `function` class template), then that move or copy is performed by using-allocator construction with allocator `get_memory_resource()`.
In section 20.9.11.2.1 [func.wrap.func.con], the assignment operators are enhanced to take the type-erased allocator into account:

```cpp
function& operator=(const function& f);
    Effects: function(allocation_arg, get_memory_resource(),
    f).swap(*this);
    Returns: *this

function& operator=(function&& f);
    Effects: Replaces the target of *this with the target of f. function(allocation_arg,
    get_memory_resource(), std::move(f)).swap(*this);
    Returns: *this

function& operator=(nullptr_t);
    Effects: If *this != NULL, destroys the target of this.
    Postconditions: !(*this).
    Returns: *this

template<class F> function& operator=(F&& f);
    Effects: function(allocation_arg, get_memory_resource(),
    std::forward<F>(f)).swap(*this);
    Returns: *this

template<class F> function& operator=(reference_wrapper<F> f) noexcept;
    Effects: function(allocation_arg, get_memory_resource(),
    f).swap(*this);
    Returns: *this
```

In section 20.9.11.2.2 [func.wrap.func.mod] a precondition is added to the definition of swap:

```cpp
void swap(function& other) noexcept;
    Precondition: this->get_memory_resource() == other->get_memory_resource().
    Effects: Interchanges the targets of *this and other.
```

### 9.2 Type-erased allocator for promise
t.u.4 Additions to std::promise [mods.futures.promise]

In section 30.6.5 [futures.promise], the following declarations are added as public members of class template promise:

```cpp
typedef experimental::erased_type_allocator_type;

experimental::pmr::memory_resource* get_memory_resource();
```

and the following paragraph is inserted before the first (introductory) paragraph of the section.

When a promise constructor that takes a first argument of type allocation_arg_t is invoked, the second argument is treated as a type-erased allocator ([type.eraser.allocator]).
9.3 Type-erased allocator for packaged_task

In section 30.6.9 [futures.task], the following declarations are added as public members of class template packaged_task:

```cpp
typedef experimental::erased_type allocator_type;
experimental::pmr::memory_resource* get_memory_resource();
```

and the following paragraph is inserted before the first (introductory) paragraph of the section.

When a packaged_task constructor that takes a first argument of type allocator_arg_t is invoked, the second argument is treated as a type-erased allocator ([type.eraser.allocator]).

10 Appendix: Template Implementation Policy (Section 4.3 from N1850)

The first problem most people see with the allocator mechanism as specified in the Standard is that the choice of allocator affects the type of a container. Consider, for example, the following type and object definitions:

```cpp
typedef std::list<int, std::allocator<int> > NormIntList;
typedef std::list<int, MyAllocator<int> > MyIntList;
```

```cpp
NormIntList list1(5, 3);
MyIntList list2(5, 3);
```

list1 and list2 are both lists of integers, and both contain five copies of the number 3. Most people would say that they have the same value. Yet they belong to different types and you cannot substitute one for the other. For example, assume we have a function that builds up a list:

```cpp
int build(std::list<int>& theList);
```

Because we did not specify an allocator parameter for the argument type, the default, std::allocator<int> is used. Thus, theList is a reference to the same type as list1. We can use build to put values into list1, but we cannot use it to put values into list2 because MyIntList is not compatible with std::list<int>. The following operations are also not supported:

```cpp
list1 == list2
list1 = list2
MyIntList list3(list1);
NormIntList* p = &list2;
// etc.
```

Now, some would argue that the solution to the build function problem is to templatize build:

```cpp
template <typename Alloc>
typename Alloc>
int build(std::list<int, Alloc>& theList);
```

or, better yet:

```cpp
template <typename OutputIterator>
typename OutputIterator>
int build(OutputIterator theIter);
```
Both of these templatized solutions have their place, but both add substantial complexity to the development process. Templates, if overused, lead to long compile times and, sometimes, bloated code. If build were a template and passed its arguments on to other functions, those functions would also need to be templates. This chained instantiation of templates produces a deep compile-time dependency such that a change to any of those modules would result in a recompilation of a significant part of the system. For thorough coverage of the benefits of reducing physical dependencies, see [Lakos96].

Even if the templatization solution were acceptable, once a nested container (e.g. a list of strings) is involved, even the simplest operations require many layers of code to bridge the type-interoperability gap. Consider trying to compare a shared list of shared strings with a regular list of regular strings:

```cpp
typedef std::basic_string<
    char,
    std::char_traits<char>,
    shared_alloc<char>
> shared_string;

std::list<shared_string, shared_alloc<shared_string> > SharedList;
std::list<std::string> TestList;

Not only will SharedList == TestList fail to compile, but employing iterators and standard algorithms will not work either:

```cpp
bool same = std::range_equal(SharedList.begin(), SharedList.end(),
    TestList.begin(), TestList.end());
```

The types to which the iterators refer are not equality-compatible (std::string vs. shared_string). The interoperability barrier caused by the use of template implementation policies impedes the straightforward use of vocabulary types – ubiquitous types used throughout the internal interfaces of a program. For example, to declare a string, s using MyAllocator we would need to write

```cpp
std::basic_string<char, std::char_traits<char>, MyAllocator<char> > s;
```

Many people find this hard to read, but the more important fact is that s is not an std::string object and cannot be used wherever std::string is expected. Similar problems exist for other common types like std::vector<int>. The use of a well-defined set of vocabulary types like string and vector lends simplicity and clarity to a piece of code. Unfortunately, their use hinders the effective use of STL-style allocators and vice-versa.

Finally, template code is much harder to test than non-template code. Templates do not produce executable machine code until instantiated. Since there are an unbounded number of possible instantiations for any given template, the number of test cases needed to ensure that every path is covered can grow by an order of magnitude for each template parameter. Subtle assumptions that the template writer makes about the template’s parameters may not become apparent until someone instantiates the template with an innocent-looking, but not-quite-
compatible parameter, long after the engineer who created the template has left the project.

Template implementation policies can be very useful when constructing mechanisms, as in the case of a function object (functor) type being used to specify an implementation policy for a standard algorithm template. Alexandrescu makes a compelling case for the use of template class policies in situations where instantiations are not expected to interoperate. However, template implementation policies are detrimental when used to control the memory allocation mechanisms of basic types that could otherwise interoperate.

11 Acknowledgements

I’d like to thank John Lakos for his careful review of my introductory text and for showing me what allocators can really do, if correctly conceived. Also, a big thank you to the members of my former team at Bloomberg for your help in defining the concepts in this paper and reviewing the result, especially Alexander Beels, Henry Mike Verschell, and Alisdair Merideth, who reworked the usage example for me. Thanks to Mark Boyall for promoting the addition of new allocators to the standard and for reviewing an early draft of this paper.

12 References

N3726 Polymorphic Memory Resources, Pablo Halpern, 2013
N3525 Polymorphic Allocators, Pablo Halpern, 2013
N3575 Additional Standard allocation schemes, Mark Boyall, 2013
N1850 Towards a Better Allocator Model, Pablo Halpern, 2005
BSL The BSL open-source library, BLoomberg
jsmith C++ Type Erasure, JSmith, Published on www.cplusplus.org, 2010-01-27
N3399 Filesystem Library Proposal (Revision 3), Beman Dawes, 2012-09-21