Doc No: N3726
Date: 2013-09-02
Author: Pablo Halpern
phalpern@halpernwightsoftware.com

Polymorphic Memory Resources

(A Revision of [N3525](N3525) – Polymorphic Allocators)

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

1.1 Target

The original version of this proposal (N3525) was first discussed during the April 2013 meeting of WG21 in Bristol, UK. A straw poll of the Library Evolution Working Group indicated strong support for the concepts in this proposal and a decision was made to target these ideas for inclusion a forthcoming library Technical Specification (TS). **Nothing in this proposal should be construed as proposing a change to the Committee Draft currently in balloting.**

1.2 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

All section names and numbers are relative to the May 2013 Working Draft, N3691.

> 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 representating 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 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:

```
#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) { }

    // Copy construct with optional memory_resource.
    //  If alloc is not specified, uses pmr::get_default_resource().
    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).

```
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) { }

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

    virtual bool 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.

```
std::pmr::list<ShoppingList> folder;    // Default allocator resource
{
    char buffer[1024];
    FixedBufferResource buf_rsrc(&buffer, 1024);
    ShoppingList 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 pure virtual functions allocate(), deallocate(), and is_equal(). 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:

```
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 an 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.

The previous revision of this paper had a *type*, `new_delete_resource` and a function `new_delete_resource_`**`singleton`**`()`. The new proposal is a bit simpler and hides from the user the actual derived-class type of the returned object. Is that an improvement? There are two questions here:

- Should the derived-class type of the value returned by this function be made visible to the user?
- Should the `_singleton()` suffix be put back?

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 Class `unsynchronized_pool_resource`

A `unsynchronized_pool_resource` is a general-purpose resource that is intended for single-threaded access. It *owns* the allocated storage and frees it on destruction, even if `deallocate` is not called for some or all of the allocated blocks. Efficiency is obtained by avoiding the acquisition of locks 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 is 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):

```
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

```
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

8.1 Utility Classes

In section [utility] (20.2), **Header <utility> synopsis**, add a new type declaration:

```
// 20.2.x, erased-type placeholder
struct erased_type { };
```

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>.

Add a new subsection under 20.2:

20.2.x erased-type placeholder [utility.erased_type]

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

The `erased_type struct` is an empty `struct` used to as a placeholder for a type that is not known due to *type erasure*. Specifically, the nested type, `allocator_type`, is an alias for `erased_type` in classes that use *type-erased allocators* (see [type.erased.allocator]).

Modify section [allocator.uses] (2.8.7) as follows:

20.8.7 uses_allocator [allocator.uses]

20.8.7.1 uses_allocator trait [allocator.uses.trait]

```
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` is an alias for `erased_type` ([utility.erased_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`.

20.8.7.2 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 meets the requirements of an allocator ([allocator.requirements] or is a pointer to `pmr::memory_resource` or to a class derived from `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.

8.2 Polymorphic Memory Resources

Add a new subsection after section 20 [utilities] for the polymorphic memory resources.

20.x Polymorphic Memory Resources [memory.resource]

20.x.1 Header <memory_resources> synopsis [memory.resource.syn]

```
namespace std {
namespace pmr {

  class memory_resource;

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

```
    template <class Tp> class polymorphic_allocator;

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

    // 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>>;

    // Singleton 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
    class unsynchronized_pool_resource;
    class monotonic_buffer_resource;

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

8.2.1 Class `memory_resource`

20.x.2 Class `memory_resource` [memory.resource.class]

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

```
namespace std {
    namespace pmr {

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

        public:
          virtual ~memory_resource();
          virtual void* allocate(size_t bytes,
```

```
                                    size_t alignment = max_align) = 0;
        virtual void  deallocate(void *p, size_t bytes,
                                    size_t alignment = max_align) = 0;

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

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

At least one reviewer has expressed misgivings about having default arguments on the `allocate()` and `deallocate()` virtual functions, reasoning that a user could create an inconsistent derived class with a different default or with no default argument. The derived class would, therefore, give different results depending on whether `allocate()` and `deallocate()` were called through the base-class interface or through the derived-class interface. It is not clear to me that this is a real problem, as someone using the derived-class interface is not actually using the class hierarchy or the *is-a* relationship. Nevertheless, the alternative interface for `memory_resource` shown below eliminates the problem by using default arguments only on non-virtual functions, at the cost of some complication to the overall description:

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

  public:
    virtual ~memory_resource();
    virtual bool is_equal(const memory_resource& other) const
      noexcept = 0;

    void* allocate(size_t bytes, size_t alignment = max_align)
      { return do_allocate(bytes, alignment); }
    void  deallocate(void *p, size_t bytes, size_t alignment =
max_align)
      { do_deallocate(p, bytes, alignment); }

  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;
};
```

If the committee feels that this is a better interface, then I am willing to make this change.

20.x.2.1 `memory_resource` virtual member functions [memory.resource.mem]

```
~memory_resource();
```

Effects: Destroys the `memory_resource` base class.

```
void* allocate(size_t bytes, size_t alignment = max_align) = 0;
```

Preconditions: `alignment` is 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` and an alignment not less than `alignment`. [Note to editor: 3.7.4.2 does not seem to actually define *allocated storage*, even though it is referenced in 3.8. 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.

```
void  deallocate(void *p, size_t bytes, size_t alignment = 0) = 0;
```

Preconditions: `p` was allocated from a prior call to `allocate(bytes, alignment)` and has not already 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.

```
bool 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*]

20.x.2.2 `memory_resource` equality [memory.resource.eq]

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

Returns: equivalent to `&a == &b || a.is_equal(b)`.

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

Returns: equivalent to `! (a == b)`.

8.2.2 Class template `polymorphic_allocator`

20.x.3 Class template `polymorphic_allocator` [polymorphic.allocator.class]

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 polymorphisms 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.

```
namespace std {
namespace pmr {

  template <class Tp>
  class polymorphic_allocator
  {
      memory_resource* m_resource;   // For exposition only

    public:
      typedef Tp value_type;

      polymorphic_allocator();
      polymorphic_allocator(memory_resource *r);

      polymorphic_allocator(const polymorphic_allocator& other)
        = default;

      template <class U>
        polymorphic_allocator(const polymorphic_allocator<U>& other);

      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..., Args2...>
        void construct(std::pair<T1,T2>* p, piecewise_construct_t,
                       tuple<Args1...> x, tuple<Args2...> y);
      template <class T1, class T2>
        void construct(std::pair<T1,T2>* p);
      template <class T1, class T2, class U, class V>
        void construct(std::pair<T1,T2>* p, U&& x, V&& y);
      template <class T1, class T2, class U, class V>
        void construct(std::pair<T1,T2>* p,
                       const std::pair<U, V>& pr);
      template <class T1, class T2, class U, class V>
        void construct(std::pair<T1,T2>* p, std::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 std
```

20.x.3.1 `polymorphic_allocator` constructors [polymorphic.allocator.ctor]

```
polymorphic_allocator();
```

Effects: set `m_resource` to `get_default_resource()`.

```
polymorphic_allocator(memory_resource *r);
```

Effects: If `r` is non-null, set `m_resource` to `r`; otherwise set `m_resource` to `get_default_resource()`.

Note: This constructor acts as an implicit conversion from `memory_resource*`.

```
template <class U>
  polymorphic_allocator(const polymorphic_allocator<U>& other);
```

Effects: sets `m_resource` to `other.resource()`.

Note: This constructor acts a conversion constructor from `polymorphic_allocator`s with different `value_type`s.

20.x.3.2 `polymorphic_allocator` member functions [polymorphic.allocator.mem]

```
Tp *allocate(size_t n);
```

Returns: Equivalent of `static_cast<Tp*>(m_resource->allocate(n * sizeof(Tp), alignof(Tp)))`.

```
void deallocate(Tp *p, size_t n);
```

Preconditions: `p` was allocated from an allocator, `x`, equal to `*this` using `x.allocate(n)`.

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

Throws: Nothing.

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

Effects: Construct a `T` object at `p` by *uses-allocator construction* with allocator `this->resource()` ([allocator.uses.construction] 20.6.7.2) and constructor arguments `std::forward<Args>(args)...`. If *uses-allocator construction* is ill-formed, then the call to `construct` is ill-formed. [*Note:* uses-allocator construction is always well formed for types that do not use allocators. – *end note*]

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

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

Effects: Constructs a `tuple`, `xprime`, from `x` by the following rules [*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()` ([allocator.uses.construction] 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(tuple<allocator_arg_t, memory_resource*>(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), tuple<memory_resource*>(this->resource())).
- Otherwise the program is ill formed.

and constructs a tuple, yprime, from y by the following rules:

- 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(tuple<allocator_arg_t, memory_resource*>(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), tuple<memory_resource*>(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().

```
template <class T1, class T2>
  void construct(std::pair<T1,T2>* p);
```

Effects: equivalent to this->construct(p, piecewise_construct, tuple<>(), 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(x.first), forward_as_tuple(x.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>(x.first)), forward_as_tuple(std::forward<V>(x.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`.

20.x.3.3 `polymorphic_allocator` equality [polymorphic.allocator.eq]

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

Returns: `a.resource() == b.resource() || *a.resource() == *b.resource()`.

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

Returns: `! (a == b)`

8.2.3 Class-alias template `resource_adaptor`

20.x.4 `resource_adaptor` [resource.adaptor]

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 of the class template, `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:

- `allocator_traits<Allocator>::pointer` shall be identical to `allocator_traits<Allocator>::value_type*`.
- `allocator_traits<Allocator>::const_pointer` shall be identical to `allocator_traits<Allocator>::value_type const*`.
- `allocator_traits<Allocator>::void_pointer` shall be identical to `void*`.

- `allocator_traits<Allocator>::const_void_pointer` shall be identical to `void const*`.

```
namespace std {
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_traits<Allocator>::rebind_alloc<char> m_alloc;

     public:
      typedef Allocator allocator_type;

      resource_adaptor_imp() = default;
      resource_adaptor_imp(const resource_adaptor_imp&) = default;

      // Does not participate in overload resolution unless
      // is_convertible<Allocator2, Allocator>::value != false
      template <class Allocator2> resource_adaptor_imp(Allocator2&& a2);

      virtual void *allocate(size_t bytes,
                             size_t alignment = max_align);
      virtual void deallocate(void *p, size_t bytes,
                              size_t alignment = max_align);

      virtual bool is_equal(const memory_resource& other) const;

      allocator_type get_allocator() const { return m_alloc; }
  };

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

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

20.x.4.1 `resource_adaptor_imp` constructor [resource.adaptor.ctor]

```
template <class Allocator2> resource_adaptor_imp(Allocator2&& a2);
```

Effects: Initializes `m_alloc` with `forward<Allocator2>(a2)`.

Remarks: Does not participate in overload resolution unless `is_convertible<Allocator2, Allocator>::value != false`.

20.x.4.2 `resource_adaptor_imp` member functions [resource.adaptor.mem]

```
virtual void *allocate(size_t bytes,
                       size_t alignment = max_align);
```

Returns: Allocated memory obtained by calling `m_alloc.allocate()`. The size and alignment of the allocated memory shall meet the requirements for a class derived from `memory_resource` ([memory.resource]).

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

Requires: `p` was previously allocated using `allocate()` and not deallocated.

Effects: Returns memory to the allocator using `m_alloc.deallocate()`.

```
virtual bool is_equal(const memory_resource& other) const;
```

Returns: `false` if `dynamic_cast<const resource_adaptor_imp*>(addressof(other))` is null, otherwise the value of `m_alloc == dynamic_cast<const resource_adaptor_imp&>(other).m_alloc`.

8.2.4 Program-wide `memory_resource` objects

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

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

Returns: A pointer to a static-duration object of type derived from `memory_resource` that can be used 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`.

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

Returns: A pointer to a static-duration object of 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.

```
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 set the default memory resource pointer to `new_delete_resource()`.

We have found it is convenient to use `nullptr` as a surrogate for the “default-default” handler in various interfaces. The use here simply provides consistency and makes it easy to reset the default resource to its initial state.

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

Remarks: The initial default memory resource pointer is `new_delete_resource()`. Calling the `set_default_resource` and `get_default_resource` functions shall not incur a data race. A call to `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`.

`memory_resource *get_default_resource() noexcept;`

Returns: The current default memory resource pointer.

8.3 Class `unsynchronized_pool_resource`

20.x.7 Class `unsynchronized_pool_resource` [unsynchronized.pool]

A `unsynchronized_pool_resource` is a general-purpose memory resource with the following qualities:

- It is designed 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` is not called for some or all of the allocated blocks.

[*Note:* Implementations are encouraged to choose allocation algorithms and data structures such that objects allocated from a single `unsynchronized_pool_resource` are likely to be close together in memory. – *end note*]

```
namespace std {
namespace pmr {

  class unsynchronized_pool_resource : public memory_resource
  {
    public:
      unsynchronized_pool_resource(
          memory_resource* upstream = get_default_resource());
      virtual ~unsynchronized_pool_resource();

      virtual void* allocate(size_t bytes,
                             size_t alignment = max_align);
      virtual void  deallocate(void *p, size_t bytes,
                               size_t alignment = max_align);

      virtual bool is_equal(const memory_resource& other) const
        noexcept;

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

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

20.x.7.1 `unsynchronized_pool_resource` constructor and destructor [unsynchronized.pool.ctor]

```
unsynchronized_pool_resource(
        memory_resource* upstream = get_default_resource());
```

Precondition: `upstream` is the address of a valid memory resource.

Effects: Constructs a memory pool which will obtain memory from `upstream` whenever it is unable to satisfy a memory request from its own internal data structures. The resulting `unsynchronized_pool_resource` will hold a copy of `upstream`, but will not own the resource to which it points. [*Note:* The intension is that calls to `upstream->allocate()` will be substantially fewer than calls to `this->allocate()` in most cases. – *end note*]

Throws: Nothing unless `upstream->allocate()` throws. It is not specified whether this constructor calls `upstream->allocate()`.

The constructor described does not give the user an opportunity to specify tuning parameters. There are two reasons for this choice: 1) It is not clear that the user can actually benefit much from tuning, e.g., the growth factor of the memory pool. It seems likely that the defaults chosen by the implementation (e.g., doubling the pool each time up to a limit) will suffice for most use cases and it is therefore not worth the complexity of adding the tuning parameters. 2) Almost any tuning parameters would imply a specific implementation strategy which, up until now, has been left unspecified.

I would consider reasoned arguments for a limited set of tuning parameters, despite the rationale described in the previous paragraph. If a general set of parameters could be specified, there is also the question as to whether they should be specified directly in the constructor or as separate modifiers after construction. The latter is more future-proof, as it allows any number of parameters to be added without a combinatorial explosion in the set of constructor arguments.

```
virtual ~unsynchronized_pool_resource();
```

Effects: calls `this->release()`.

20.x.7.2 `unsynchronized_pool_resource` members [unsynchronized.pool.mem]

```
virtual void* allocate(size_t bytes, size_t alignment = max_align);
```

Returns: A pointer to allocated storage (3.7.4.2) with a size of at least `bytes` and an alignment not less than `alignment`.

Effects: If this pool is unable to satisfy the memory request from its own internal data structures, it will call `upstream_resource()->allocate()` to obtain more memory.

Throws: Nothing unless `upstream_resource()->allocate()` throws.

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

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

Throws: Nothing

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

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

`void release();`

Effects: Calls `upstream_resource()->deallocate()` as necessary to release all allocated memory. [*Note:* memory is released back to `upstream_resource()` even if some blocks that were allocated from `this` were never deallocated from `this`. – *end note*]

`memory_resource* upstream_resource() const;`

Returns: the value of the `upstream` argument provided to the constructor of this object.

8.4 Class `monotonic_buffer_resource`

20.x.8 Class `monotonic_buffer_resource` [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 objects are 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 is exhausted, it obtains additional buffers from an "upstream" memory resource supplied at construction.

— If a memory request size exceeds a specified threshold, the internal buffer is bypassed and the request is passed directly to the constructor-supplied upstream resource.

— It is designed 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` is not called for some or all of the allocated blocks.

```
namespace std {
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:
      monotonic_buffer_resource(
          memory_resource* upstream = get_default_resource());
      monotonic_buffer_resource(size_t initial_size,
          memory_resource* upstream = get_default_resource());
      monotonic_buffer_resource(void* buffer, size_t buffer_size,
          memory_resource* upstream = get_default_resource());

      virtual ~monotonic_buffer_resource();

      virtual void* allocate(size_t bytes,
```

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

        virtual bool is_equal(const memory_resource& other) const
          noexcept;

        void max_buffer_size(size_t v);
        size_t max_buffer_size() const;

        void passthrough_threshold(size_t v);
        size_t passthrough_threshold() const;

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

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

20.x.8.1 `monotonic_buffer_resource` constructor and destructor [monotonic.buffer.ctor]

```
monotonic_buffer_resource(
          memory_resource* upstream = get_default_resource());
monotonic_buffer_resource(size_t initial_size,
          memory_resource* upstream = get_default_resource());
```

Preconditions: `upstream` is the address of a valid memory resource; `initial_size`, if specified, is positive.

Effects: Sets `upstream_rsrc` to `upstream and` `current_buffer` to `nullptr`. If `initial_size` is specified, sets `next_size` to at least `initial_size`; otherwise sets `next_size to` an implementation-defined size.

```
monotonic_buffer_resource(void* buffer, size_t buffer_size,
          memory_resource* upstream = get_default_resource());
```

Preconditions: `upstream` is the address of a valid memory resource. `buffer_size` is no larger than the number of bytes in `buffer`.

Effects: Sets `upstream_rsrc` to `upstream`, `current_buffer` to `buffer`, and `next_size` to at least 2*`initial_size` (but not less than 1).

```
virtual ~monotonic_buffer_resource();
```

Effects: Calls `this->release()`.

20.x.8.2 `monotonic_buffer_resource` members [monotonic.buffer.mem]

```
virtual void* allocate(size_t bytes, size_t alignment = max_align);
```

Returns: A pointer to allocated storage (3.7.4.2) with a size of at least `bytes` and an alignment not less than `alignment`.

Effects: If `bytes` >= `passthrough_threshold()`, return `upstream_rsrc->allocate(bytes, alignment)`; otherwise, if the unused space in current_buffer can fit a buffer of the specified bytes and alignment, then allocate the return block from current_buffer; otherwise, set current_buffer to upstream_rsrc->allocate(n, m), where n is not less than max(n, next_size) and m is not less than alignment, and set next_size to at least min(2*n, max_buffer_size()), then allocate the return block from the newly-allocated current_buffer.

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

The description above is very algorithmic and specific. It is intended to give the user a firm grasp of the expected behavior and use of memory so that they may choose their tuning parameters accordingly to avoid large amounts of wasted memory. Is such detail necessary or desirable?

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

Effects: None

Throws: Nothing

Remarks: Memory use by this resource increases monotonically until destruction.

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

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

```
void max_buffer_size(size_t v);
```

Precondition: v > 0.

Effects: Sets the size of the largest buffer that this resource will request from the upstream resource. The implementation may choose to use a larger value than v (e.g., rounding up to the next cache line or page). If passthrough_threshold() > v, then calls passthrough_threshold(v).

```
size_t max_buffer_size() const;
```

Returns: The size of the largest buffer that this resource will request from the upstream resource. Unless changed by the user, this function will return an implementation-defined value.

```
void passthrough_threshold(size_t v);
```

Precondition: v >= 0.

Effects: The size of the largest object that would be allocated from a buffer. Objects larger than v will be allocated by directly passing the allocation request through to upstream_rsrc. If max_buffer_size() < v, then calls max_buffer_size(v).

```
size_t passthrough_threshold() const;
```

Retruns: The size of the largest object that would be allocated from a buffer. Objects larger than passthrough_threshold() will be allocated by directly passing the allocation request through to upstream_rsrc. Unless changed by the user, this function will return an implementation-defined value.

```
void release();
```

Effects: Calls upstream_rsrc->deallocate() as necessary to release all allocated memory. [*Note:* memory is released back to upstream_rsrc even if some blocks that were allocated from this were never deallocated from this. – *end note*]

```
memory_resource* upstream_resource() const;
```

Returns: the value of the `upstream` argument provided to the constructor of this object.

8.5 Type-erased allocator

8.5.1 In General

Insert a new section into the standard as follows:

The following describes an idiom that is followed by several types in the standard. It is unclear where in the standard this description belongs. Should it arranged as a set of requirement and added to section 17.6.3? If so, what is it? Is it a requirement of the allocator parameter? Should it be a definition, like *INVOKE* and *COPY_DECAY* or *uses-allocator construction*. Please advise. Once it is correctly categorized, I can complete tweaking the wording and format.

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

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 of 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 `rptr` of static type `pmr::memory_resource*` ([memory.resource.class]). The process by which `rptr` is computed from `alloc` depends on the type of `alloc` as described in Table Q:

Table Q – Computed `memory_resource` for type-erased allocator

If the type of `alloc` is	then the value of `rptr` is
non-existent – no `alloc` specified	The value of `pmr::get_default_resource()` at the time of construction.
`nullptr_t`	The value of `pmr::get_default_resource()` at the time of construction.
pointer convertible to `pmr::memory_resource*`	`static_cast< pmr::memory_resource*>(rptr)`
`pmr::polymorphic_allocator<U>`	`alloc.resource()`
a type meeting the Allocator requirements ([allocator.requirements])	a pointer to a value of type `pmr::resource_adaptor<A>` where A is the type of alloc. `rptr` remains valid only for the lifetime of X
None of the above	The program is ill-formed

Additionally, class `C` shall meet the following requirements:

- `C::allocator_type` is identical to `erased_type`.
- `X.get_memory_resource()` returns `rptr`.

8.5.2 Type-erased allocator for `function`

In 20.10.11.2 [func.wrap.func], add the following declarations to class template `function`:

```
    typedef erased_type allocator_type;
```

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

Change the first paragraph of section 20.10.11.2.1 [func.wrap.func.con] as follows:

When a `function` constructor that takes a first argument of type `allocator_arg_t` is invoked, the second argument shall is treated as a *type-erased allocator* ([type.erased.allocator]). ~~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 shall be performed by *using-allocator construction* with allocator `get_memory_resource()`.

And correct the definitions of `operator=` as follows:

```
function& operator=(const function& f);
```

Effects: `function(allocator_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(allocator_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(allocator_arg, get_memory_resource(), std::forward<F>(f)).swap(*this);`

Returns: `*this`

```
template<class F> function& operator=(reference_wrapper<F> f) noexcept;
```

Effects: `function(allocator_arg, get_memory_resource(), f).swap(*this);`

Returns: `*this`

8.5.3 Type-erased allocator for `promise`

In section 30.6.5 [futures.promise], add the following declarations to class template `promise`:

```
typedef erased_type allocator_type;

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

Add the following paragraph before 30.6.5 [futures.promise] paragraph 1:

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

8.5.4 Type-erased allocator for `packaged_task`

In section 30.6.9 [futures.task], add the following declarations to class template `packaged_task`:

```
    typedef erased_type allocator_type;

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

Add the following paragraph before 30.6.9 [futures.task] after paragraph 2:

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.erased.allocator]).

8.6 Containers Aliases Using Polymorphic Allocators

In section 21.3 [string.classes], Header `<string>` synopsis, add aliases as follows:

```
// basic_string typedef names
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;

namespace pmr {

// 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 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 pmr
```

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

In sections 23.3.1 [sequences.general], 23.4.2 [associative.map.syn], 23.4.3 [associative.set.syn], 23.5.2 [unord.map.syn], and 23.5.3 [unord.set.syn], add polymorphic allocator aliases to the end of specified headers as follows (these insertions should not appear in the WP text as a table):

Header	polymorphic allocator aliases

`<deque>`	`namespace pmr {`
	`  template <class T>`
	`    using deque = std::deque<T,polymorphic_allocator<T>>;`
	`}`

Header	polymorphic allocator aliases

<forward_list>	namespace pmr {
	  template <class T>
	    using forward_list =
	      std::forward_list<T,polymorphic_allocator<T>>;
	}

<list>	namespace pmr {
	  template <class T>
	    using list = list<T,polymorphic_allocator<T>>;
	}

<vector>	namespace pmr {
	  template <class T>
	    using vector = vector<T,polymorphic_allocator<T>>;
	}

<map>	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>>>;
	}

<set>	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>>;
	}

<unordered_map>	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>>>;
	}

Header	polymorphic allocator aliases

<unordered_set>	`namespace pmr {`
	`  template <class Key,`
	`            class Hash = hash<Key>,`
	`            class Pred = std::equal_to<Key>>`
	`    using unordered_set =`
	`      std::unordered_set<Key, Hash, Pred,`
	`           polymorphic_allocator<pair<const Key,T>>>;`
	`  template <class Key,`
	`            class Hash = hash<Key>,`
	`            class Pred = std::equal_to<Key>>`
	`    using unordered_multiset =`
	`      std::unordered_multiset<Key, Hash, Pred,`
	`           polymorphic_allocator<pair<const Key,T>>>;`
	`}`

9 Appendix: Section 4.3 from N1850

9.1 Template Implementation Policy

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:

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

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:

```
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:

```
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`:

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

or, better yet:

```
template <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-interoperablity gap. Consider trying to compare a shared list of shared strings with a regular list of regular strings:

```
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:

```
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

```
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.

10 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.

11 References

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