Doc No: N3525
Date: 2013-03-18
Author: Pablo Halpern
phalpern@halpernwightsoftware.com

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 allocators that interoperate with the existing compile-time polymorphic ones. In addition, this proposal improves the interface and allocation semantics of some of library classes, such as `std::function`, that use type erasure for allocators.

Contents

1 Document Conventions

All section names and numbers are relative to the November 2012 Working Draft**,** N3485**.**

> 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 LWG 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.

2 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 single allocator type, `polymorphic_allocator`, 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 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.

3 Summary of Proposal

3.1 Namespace `std::polyalloc`

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

The name, `polyalloc`, 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.

3.2 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.

3.3 Class Template `polymorphic_allocator<T>`

An instance of `polymorphic_allocator<T>` is a wrapper around an `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. Two objects `x` and y of type `list<int, polymorphic_allocator<int>>` are the same type, but may use different memory allocation mechanisms.

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

3.4 Aliases for container classes

There would be an alias in the `polyalloc` 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 polyalloc {

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

} // namespace polyalloc
} // namespace std
```

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

3.5 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 any `T` and `U`.

3.6 Typedef `new_delete_resource`

`new_delete_resource` is a typedef for `resource_adaptor<allocator<char>>`. In other words, it is a type derived from `memory_resource` that uses `operator new()` and `operator delete()` to manage memory.

3.7 Function `new_delete_resource_singleton()`

Since `std::allocator` is stateless, all instances of `new_delete_resource` are equivalent. The `new_delete_resource_singleton()` function simply returns a pointer to a singleton of that type.

3.8 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_singleton()`.

3.9 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 that 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.)

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 not be possible 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.  Because polymorphic_allocator is scoped,
    // every element of the vector will use the same allocator as the vector itself.
    typedef std::polyalloc::string  string_type; // string uses polymorphic allocator
    typedef std::polyalloc::vector<string_type> strvec_type;

    strvec_type m_strvec;

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

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

    // Copy construct with optional memory_resource.
    //  If alloc is not specified, uses polyalloc::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);
```

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::polyalloc::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))
            return m_next_alloc;
        else
            throw std::bad_alloc();
    }

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

    virtual bool is_equal(std::polyalloc::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::polyalloc::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::polyalloc::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::polyalloc::list`, `std::polyalloc::vector`, and `std::polyalloc::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 `ShoppingList`s that are inserted into the list (and to the strings within those `ShoppingList`s). 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 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.

6 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` and `polymorphic_allocator` classed described in this proposal are minor variations of the facilities that have been in use at Bloomberg for over a decade. These facilities have made dramatically improved testability of software (through the use of test allocators) and provided performance benefits when using special-purpose allocators such as arena allocators and thread-specific allocators.

7 Formal Wording

7.1 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 `<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] (2.6.7) as follows:

20.6.7 uses_allocator [allocator.uses]

20.6.7.1 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`.

20.6.7.2 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 `polyalloc::memory_resource` or to a class derived from `polyalloc::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.

7.2 Polymorphic Allocator

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

20.x Polymorphic Allocators [polymorphic.allocator]

20.x.1 Header <polymorphic_allocator> synopsis [polymorphic.allocator.syn]

```
namespace std {
namespace polyalloc {

  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>>;

  typedef resource_adaptor<allocator<char>> new_delete_resource;

  new_delete_resource* new_delete_resource_singleton() noexcept;
  memory_resource *set_default_resource(memory_resource *r)
    noexcept;
  memory_resource *get_default_resource() noexcept;

} // namespace polyalloc
} // namespace std
```

20.x.2 Polymorphic Memory Resource [polymorphic.resource]

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

```
namespace std {
namespace polyalloc {

  class memory_resource
  {
    public:
      virtual ~memory_resource();
      virtual void* allocate(size_t bytes, size_t alignment = 0) = 0;
      virtual void  deallocate(void *p, size_t bytes,
                               size_t alignment = 0) = 0;

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

  };

} // namespace polyalloc
```

```
} // namespace std
```

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

`~memory_resource();`

Effects: Destroys the `memory_resource` base class.

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

Preconditions: `alignment` is either zero or 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 implementation-defined alignment of *Q*. [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.] If `0 < alignment && alignment <= sizeof(max_align_t)`, then *Q* shall be not less than `alignment`. If `alignment > sizeof(max_align_t)`, the *Q* shall be not less than `sizeof(max_align_t)`. [*Note:* ideally, a user-defined derived class will always choose *Q* `== alignment`. – *end note*] If `alignment == 0`, then *Q* shall be no less than the maximum possible alignment required for an object of size `bytes` assuming no extended alignment (3.11 [basic.align]). [*Note:* The maximum alignment possible for an object of size `bytes` is the largest power of two ≤ `sizeof(max_align_t)` that evenly divides bytes. Thus, it is possible for *Q* to be less than both `bytes` and `sizeof(max_align_t)`. – *end note*]

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`. Tor 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 [polymorphic.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)`.

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

A specialization of class template `polyalloc::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 `polyalloc::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 polyalloc {

  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);

      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 polyalloc
  } // 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 as both a copy constructor and 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.

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: [*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*]

Constructs a `tuple`, `xprime`, from `x` by the following rules:

— 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()`.

The non-normative comment at the beginning is new. Does it help?

```
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()`.

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

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

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

An instance of `resource_adaptor<Allocator>` is an adaptor that wraps an `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 polyalloc {

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

      Allocator m_alloc;    // for exposition only

    public:
      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 = 0);
      virtual void deallocate(void *p, size_t bytes, size_t alignment =0);

      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 polyalloc
} // namespace std
```

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

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

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

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

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

> *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` ([polymorphic.resource]).

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

> *Requires:* `p` was previously allocated using `allocate()` and not deallocated.
>
> *Effects:* Returns memory 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`.

20.x.5 Access to program-wide `memory_resource` objects [polymorphic.globals]

`new_delete_resource* new_delete_resource_singleton() noexcept;`

> *Returns:* A pointer to a static-duration object of `new_delete_resource` type 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.

`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_singleton()`.

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_singleton()`. 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.

7.3 Allocator Type Erasure

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 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 Allocator type erasure [allocator.type.erasure]

Allocator type erasure is the process of obtaining a pointer `r` to a `polyalloc::memory_resource` ([polymorphic.resource]) from an argument `alloc` to a constructor template for some object `X` of class `T`. Throughout its lifetime, `X` uses `r` to allocate any needed memory (i.e., for internal data structures). The process by which this `r` is computed from `alloc` depends on the type of `alloc` and is described in Table Q:

Table Q – memory_resource for Allocator type erasure

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

Additionally, a class `C` that uses allocator type erasure shall meet the following requirements:

- `C::allocator_type` shall be identical to `erased_type`.
- `X.get_memory_resource()` returns `r`.

In 20.8.11.2 [func.wrap.func], add the following declarations to class template `function`:

```
    typedef erase_type allocator_type;

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

Change the first paragraph of section 20.8.11.2.1 [func.wrap.func.con] as follows:

When any `function` constructor that takes a first argument of type `allocator_arg_t` is invoked, the second argument shall be handled in accordance with *allocator type erasure* ([allocator.type.erasure]). ~~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`

7.4 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 polyalloc {

// basic_string using polymorphic_allocator in namespace polyalloc
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 polyalloc
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;

} // namespace polyalloc
```

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

In section 23.3.3.1 [deque.overview], add polymorphic allocator aliases as follows:

```
// specialized algorithms:
template <class T, class Allocator>
  void swap(deque<T,Allocator>& x, deque<T,Allocator>& y);
// deque with polymorphic allocator
namespace polyalloc {

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

}
```

Continue in this way for remaining containers.

8 Appendix: Section 4.3 from N1850

8.1 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.

9 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.

10 References

N1850 *Towards a Better Allocator Model*, Pablo Halpern, 2005

jsmith *C++ Type Erasure*, JSmith, Published on www.cplusplus.org, 2010-01-27

N3399 *Filesystem Library Proposal (Revision 3),* Beman Dawes, 2012-09-21