Making C++ Software *Allocator Aware*

Document Number:	P2127R0
Date:	2024-03-11
Project:	Programming Language C++
Audience:	LEWG, LWG

Reply to:	Pablo Halpern
	<phalpern@halpernwightsoftware.com>

Contents

A Note for the WG21 Audience 1
Abstract 2
Introduction 2
A Quick Reference for Allocator-Aware Interfaces 4
The Allocator-Aware Interface 5
Making a Simple `struct` AA 9
Making an Attribute Class AA 16
Implementing a Class That Allocates Memory 19
Implementing an AA Class Template 29
Implementing an AA Container 34
Pitfall: Inheriting from a bsl-AA Class 38
Testing AA Components 39
Conclusion 45
APPENDIX A. Converting from Legacy-AA to Bsl-AA 47
APPENDIX B. Mapping BDE AA Development to C++20 PMR AA Development 50
APPENDIX C. Allocator-Aware Move Operations in C++03 52
APPENDIX D. Alternatives to Storing the Allocator in the Object Footprint.. 55
Works Cited 57

A Note for the WG21 Audience

This paper is not a proposal; it is a view into the ways in which one organization, Bloomberg, writes allocator-aware software. Originally written as a how-to *document for Bloomberg engineers, we thought it would be instructive for a WG21 audience (especially LEWG).*

- *It is a case study detailing actual allocator use in a large organization.*
- *It contains historical context showing how Bloomberg's allocator-aware code base inspired the PMR allocator model in the C++ Standard Library and is, in turn, evolving toward adopting that model while still integrating with and supporting their existing codebase.*
- *It describes what is required of an allocator-aware type using a PMR-like allocator model.*

*This document is the third in a series of documents originally written for Bloomberg engineers and shared with the C++ Standards Committee. The previous two are **P2035**, **Value Proposition: Allocator-Aware (AA) Software** and **P2126**, **Unleashing the Power of Allocator-Aware Software Infrastructure**.*

Abstract

This paper teaches the reader how to write *Allocator-aware* (*AA*) software, a term for software that allows a client to supply an allocator at object construction. AA software provides the application developer with an effective, lower-cost alternative to writing bespoke types having individually customized memory management.[1] Creating AA software, however, can be considerably more complex than using existing AA software. After introducing the requirements for an AA type compatible with BDE[2] guidelines, this paper presents the steps of transforming a simple `struct` into an AA class and then explains how to accomplish this task for increasingly complex categories of types, culminating with container class templates.

BDE 4.0 introduced a number of facilities, described herein, that make it easier to author AA software. Though his paper is written primarily for Bloomberg engineers — both those who are new to AA concepts, and those who have experience using older patterns of writing AA software — developers outside of Bloomberg can also benefit from what we learned.

Introduction

Effective use of allocator-aware software infrastructure (AASI) is largely a matter of selecting the appropriate allocator when constructing allocator-aware

[1] Motivational background can be found in **halpern20a**. Information on using allocator-aware software infrastructure can be found in **halpern20b**.

[2] BDE is an initialism that began as Bloomberg Development Environment and is now understood to simply describe an engineering group within Bloomberg.

(AA) objects.[3] *Creating* AA classes, however, is another matter and a developer must learn specific techniques, described in this paper, to perform the task properly. Developers creating applications that necessitate writing custom AA classes (e.g., to be used within AASI containers) will also need to assimilate some subset of these techniques, which are covered, step-by-step, in this paper.

Making C++ software AA requires plumbing each class that might allocate memory to perform the following tasks.

- Accept an allocator on construction.
- Store the allocator internally, whether directly or within a subobject, and abstain from changing it throughout the lifetime of the object.
- Use the allocator to allocate and deallocate all owned memory.
- Supply the allocator as a constructor parameter to each AA subobject (i.e., member, base-class, contained-element, or any other logically owned object). Note that AA subobjects might also have AA subobjects and thus recursively require the same plumbing.
- Provide a member function that returns the allocator.

Depending on the nature of the class, the increase in source code needed to make reusable components AA is typically between 4% and 17%.[4] Despite this code-size increase, the task of transformation is — for the great majority of types written by application developers — straightforward and mechanical.[5,6] Unfortunately, undertaking this work at Bloomberg is complicated by a few factors.

- Continued use of pre-C++11 compilers
- Mismatches between the older *legacy-AA* interface used in the vast majority of pre-2023 AA code at Bloomberg, the newer *bsl-AA* interface recommended in this paper, and the C++17 Standard *pmr-AA* interface recommended for use outside of Bloomberg
- Inadequate infrastructure support in BDE 3.x,[7] especially for the bsl-AA and pmr-AA interfaces

[3] For a cost-benefit analysis of supporting AASI (including a description of various AA models), see **halpern20a**. For a tour of how to use AASI effectively from the application developer's perspective, see **halpern20b**.
[4] This data pertains to BDE (library) source code (c. May 2017); see **lakos19**, time 29:30.
[5] The `bde_verify` tool (**bloomberg**) already checks many of the AA requirements.
[6] Work is underway to integrate allocator awareness into the C++ language and compiler; see **meredith19**.
[7] Throughout this document, shaded text is used to describe issues in BDE 3.x that no longer apply to BDE 4.0 as well as workarounds for those issues when implementing an AA type in BDE 3.x.

This paper describes the preferred ways to write allocator-aware software using BDE 4.0, which is expected to be released in 2023.

Through a series of examples, this paper shows the reader how to transform a C++ class (or class template) into an AA class using the BDE model. The paper begins by introducing the interface and other requirements for a type to be AA and then moves on to five specific, highly structured categories of AA types.

1) **Simple structs with AA members**: demonstration of how to add the necessary member types, traits, and constructors so that an (optionally specified) allocator can be passed to all AA data members
2) **Attribute classes**: demonstration of how to identify missing constructors and add an optional allocator parameter to each existing constructor
3) **Classes that allocate memory**: demonstration of how to use the allocator directly in the constructors, destructor, assignment operators, and `swap` function
4) **Class templates**: demonstration of how to work with a type that is dependent on a template parameter that might or might not be AA
5) **Containers**: demonstration of how to extend allocator awareness beyond the constructors to include insertion and removal of (possibly AA) elements

The paper finishes by describing testing techniques specific to AA components.

This paper provides sufficient information to make most components consistent and interoperable with the BDE AASI. Certain wrapper classes, such as `std::optional` and `std::variant`, need to provide an allocator to an AA subobject at points other than wrapper construction, whereas smart pointers (e.g., `std::shared_ptr`) and some types with reference semantics allocate and construct the referenced object and control block only once and then share them among multiple pointer objects, thus tying the allocator to the *referenced* object rather than the *referencing* object. Such advanced classes use allocators in unique ways and require techniques that are beyond the scope of this paper and that do not generalize to most other classes. The author of such an advanced component is advised to look at the implementation of BDE equivalents, e.g., `bslstl_optional`, `bdlb_variant`, or `bslstl_sharedptr`.

A Quick Reference for Allocator-Aware Interfaces

For many years Bloomberg's allocator-aware types have been written following an interface style that we call the *legacy-AA model*. This model is still supported by the BDE infrastructure, but with the release of the allocator utilities in BDE 4.0, documented in this paper, we now recommend following the more modern *bsl-AA model* for writing allocator-aware types. The chief conceptual difference between the models is in the vocabulary type used to supply an allocator; the legacy-AA model directly uses a raw pointer to `bslma::Allocator`, whereas the bsl-AA model uses an instantiation of `bsl::allocator`, which is a Standard-compliant wrapper for `bslma::Allocator*`. Table 1 shows minimal examples of both interfaces, highlighting their differences. C++11 or later is assumed in these examples as well as most other examples in this paper; for a discussion of C++03 considerations, see Appendix C.

Table 1: Legacy-AA and bsl-AA interfaces

<table>
<tr><th>Legacy-AA interface</th><th>bsl-AA interface</th></tr>
<tr><td><pre>
class MyAAClass {
  public:
    // TRAITS
    BSLMF_NESTED_TRAIT_DECLARATION(
          MyAAClass,
          bslma::UsesBslmaAllocator);

    // CREATORS
    MyAAClass();
    explicit
      MyAAClass(bslma::Allocator *);
    MyAAClass(const MyAAClass&,
             bslma::Allocator * = 0);
    MyAAClass(MyAAClass&&);
    MyAAClass(MyAAClass&&,
              bslma::Allocator *);

    // ACCESSORS
    bslma::Allocator
        *allocator() const;
};
</pre></td><td><pre>
class MyAAClass {
  public:
    // TYPES
    typedef bsl::allocator<>
                        allocator_type;

    // CREATORS
    MyAAClass();
    explicit
      MyAAClass(const allocator_type&);
    MyAAClass(const MyAAClass&,
           const allocator_type& = {});
    MyAAClass(MyAAClass&&);
    MyAAClass(MyAAClass&&,
              const allocator_type&);

    // ACCESSORS
    allocator_type
        get_allocator() const;
};
</pre></td></tr>
</table>

Although they express it differently, each interface 1) indicates that the class is allocator-aware, 2) provides, for each constructor, a version having an allocator parameter, and 3) defines a function that returns the object's allocator. An additional C++17 *pmr-AA* model (not shown) has an interface identical to that of the bsl-AA model except that the allocator type is an instantiation of `std::pmr::polymorphic_allocator` instead of `bsl::allocator`. In all three models, the allocator is polymorphic at run time; thus AA objects having the same type can use different types of allocators.

The rest of this paper will describe the rationale behind the change of models presented above, how BDE 4.0 maximizes compatibility between the models, and how to apply this approach to *your* types.

The Allocator-Aware Interface

An allocator-aware class is supplied an allocator on construction, either as a constructor argument or using the current default allocator. This allocator is used to allocate all memory owned by the object, including memory owned by subobjects. Once constructed, an object's allocator does not change for the remainder of its lifetime.[8] This section describes the interface features common to all allocator-aware types consistent with the BDE infrastructure.[9] Subsequent sections describe how to transform a non-AA class into an AA class by adding and implementing these interface features.

The interface features described in this section comprise a *concept*, i.e., a set of supported operations on a type, including syntax and semantics, that can be used in a generic programming context. Even if a type uses an allocator, if it does not fully model the AA concept, then it will not be treated as an AA type by containers or AA utilities or, if it is recognized as an AA type, compilation will fail due to a missing interface component. For example, if a specific constructor does not have a variant that takes an allocator parameter, then that constructor cannot be used to `emplace` an object into a container because the container would not be able supply its allocator to the element. Similarly, if an object's allocator is allowed to change during the object's lifetime, it would violate the container's invariant that all its elements use the same allocator and could result in a mismatch between the container's lifetime and the lifetime of the allocators used by one or more of its elements.

This paper adheres to a new AA interface style based on the C++17 Standard.[10] To achieve compliance with this style, an AA class, `SomeClass`, must have the following six features.

1) The type, `SomeClass::allocator_type`, is a specialization of `bsl::allocator` (often `bsl::allocator<>`, which is an abbreviation for `bsl::allocator<std::byte>` in C++14 or `bsl::allocator<unsigned char>` prior to C++14[11]). The Standard Library class template `std::pmr::polymorphic_allocator` is modeled after `bsl::allocator`.

   In BDE 3.x, `bsl::allocator` does not have a default template argument; `bsl::allocator<unsigned char>` must be fully spelled out.

---

[8] Theoretically, this invariant would not hold if any of the C++ Standard Library traits `propagate_on_container_copy_assignment`, `propagate_on_container_move_assignment`, and `propagate_on_container_swap` evaluate to true for a given allocator. However, since these traits are all false for `bsl::allocator` and `bsl::polymorphic_allocator` (and `std::pmr::polymorphic_allocator` in C++17), we can ignore them when applying the polymorphic allocator style described in this paper.

[9] **bloomberga**

[10] The relationship between the C++ Standard and the style described in this paper is detailed in Appendix B.

[11] In most of this document, our use of the terms *C++14* and *C++17* refer to the *library standard* supplied by the platform rather than the *language standard* accepted by the compiler. Thus, the default argument to `bsl::allocator<>` will be `std::byte` if such a type exists, i.e., if the platform library conforms to the C++14 Standard Library specification.

2) Both of the following type traits are true:

```
bsl::uses_allocator<SomeClass, bsl::allocator<>>::value
bslma::UsesBslmaAllocator<SomeClass>::value
```

Both traits implicitly evaluate to true if the typedef `allocator_type` exists and is convertible to `bsl::allocator<>`. That is, by adhering to item (1), these traits are automatically correct.[12]

3) Every constructor has a variant that can be invoked with an allocator argument to be used by the constructed object; even the copy and move constructors must have variants (the *extended* copy and *extended* move constructors) where an allocator can be specified in addition to the object being copied or moved.[13] If an allocator is not specified, a default allocator is used except that the nonextended move constructor gets the allocator from the moved-from object.
4) All memory owned by the object or one of its logically owned subobjects is obtained from its allocator. A logically owned subobject is part of the object's state and is tied to the object's lifetime; such a subobject is not a temporary variable that exists only for the duration of a single member function invocation.[14] The well-known smart pointers `shared_ptr`, `unique_ptr`, and (at Bloomberg) `bslma::ManagedPtr` can use allocators but follow a different set of rules and do not conform to the AA interface. An object to which a smart pointer points is owned by the pointer but is not a *subobject* of the pointer.
5) An object's allocator does not change over the course of its lifetime.
6) The `get_allocator()` member function returns the object's allocator, i.e., the allocator used to construct the object.

These six features describe the allocator-specific requirements of a *bsl-AA* type. The requirements for a *pmr-AA* type are essentially the same except that `allocator_type` is a specialization of `std::pmr::polymorphic_allocator` instead of `bsl::allocator`. This document recommends idioms that are intended to work unchanged if third-party pmr-AA types are incorporated into the Bloomberg code base.

Existing Bloomberg software that predates these recommendations uses an older, *legacy-AA* model that is no longer recommended. Both AA models are

---

[12] Until spring 2023, having the appropriate `allocator_type` type did not automatically cause `UsesBslmaAllocator` to evaluate to true; thus, many classes have the trait defined explicitly even though doing so is now redundant.
[13] The reverse is not true: A constructor with an allocator parameter does *not* need to have a variant without an allocator parameter. Such a mandatory allocator might cause confusion, however, when using, for example, container `emplace` methods where the allocator is supplied implicitly by the called method and not by the caller.
[14] Local-scoped variables almost always use the default allocator. For more information about choosing the right allocator, see **halpern20b**, page 9.

compatible with the BDE infrastructure (e.g., container classes can work with AA elements using either style). The legacy-AA interface style has similar features to the bsl-AA style (described above) but renders them differently.

- The legacy-AA interface has no `allocator_type` member (or has one defined as `void`).
- Instead of a `bsl::allocator` object, an allocator in the legacy-AA model is represented by a raw pointer to `bslma::Allocator` which, at run time, points to a specific derived-class object.
- A class for which the `bslma::UsesBslmaAllocator` and `bsl::uses_allocator` traits both evaluate to true conforms to the bsl-AA interface, whereas a class for which only `bslma::UsesBslmaAllocator` is true conforms to the legacy-AA interface.
- Instead of a `get_allocator()` member function that returns `allocator_type`, a legacy-AA class has an `allocator()` member function that returns `bslma::Allocator*`.

A side-by-side comparison of these models is concisely illustrated in Table 1, earlier in this paper. Note that the `bslma::UsesBslmaAllocator` trait must be explicitly defined in the legacy-AA interface but is implicitly defined in the bsl-AA interface.

The bsl-AA model is simpler to use and prevents coding errors that are common when using raw pointers.[15] This newer model is also closer to the C++ Standard's PMR[16] model — `bsl::allocator` is nearly identical to `std::pmr::polymorphic_allocator` and `bslma::Allocator` is similar to `std::pmr::memory_resource`.[17] Moreover, when a C++17 or later library is available, `bsl::allocator` is derived from `bsl::polymorphic_allocator` and `bslma::Allocator` is derived from `bsl::memory_resource`, where `bsl::polymorphic_allocator` and `bsl::memory_resource` are aliases for the same-named types in the `std::pmr` namespace.

The bsl-AA model is also backward-compatible with the legacy-AA model because `bslma::Allocator*` is convertible to `bsl::allocator<>` (just as `bsl::memory_resource*` is convertible to `bsl::polymorphic_allocator<>`). The contained `bslma::Allocator*` can be retrieved using the `mechanism` method of `bsl::allocator`, should it be required for interoperability with legacy-AA components.

Types in the `bslstl` package that are adapted from the C++ Standard Library conform to the bsl-AA model when using the default allocator template parameter of `bsl::allocator`, but the legacy-AA model is still the norm in the rest of the Bloomberg codebase; to apply the recipes described in this paper, anyone developing AA software at Bloomberg should have at least a passing familiarity with the legacy-AA model. This paper is primarily concerned with developing new software, and thus it focuses on the bsl-AA model, touching on the legacy-AA model only in situations where the two come in contact.

Making a Simple `struct` AA

If a `struct` contains one or more AA data members, we take on the challenge of passing an allocator to those members when an instance of the `struct` is created. The currently supported way to add allocator awareness to a `struct` is to augment it with all the member types, traits, and constructors needed to give it a bsl-AA interface and bsl-AA semantics.

---

[15] The legacy-AA interface uses a null pointer as a default allocator argument. This pointer must be converted to a guaranteed-nonnull pointer by calling `bslma::Default::allocator(basicAllocator)`, which returns `basicAllocator` unchanged if it is non-null and otherwise the currently installed default allocator. Failing to perform this transformation can cause a null-pointer dereference, whereas doing it twice incurs the cost of unnecessary atomic reads. In contrast, `bsl::allocator` suffers none of these problems because it has a default constructor that always results in a valid (default) allocator.

[16] Polymorphic Memory Resource; see Appendix B.

[17] The standard types were modeled directly on the Bloomberg types but, with the benefit of hindsight, are a bit more evolved. For example, `std::pmr::memory_resource` lets the caller specify a required alignment when allocating memory, whereas `bslma::Allocator` did not until recent work made `bslma::Allocator` a derived class of `bsl::memory_resource`.

This section describes the steps needed to convert a simple `struct` having AA data members into a proper bsl-AA class. In these five steps, we must define

1. an `allocator_type` member type
2. regular and allocator-extended default constructors
3. regular and allocator-extended copy and move constructors
4. other regular and allocator-extended constructors (optional)
5. a `get_allocator` member function

Many of the steps needed to transform a simple `struct` into an AA class also apply (sometimes with small variations) to more complex categories of types.

In the C++ Standard, a simple `struct` without user-defined constructors belongs to the *aggregate* category of types and is compatible with member-by-member *aggregate initialization*.[18] An unfortunate consequence of making the `struct` AA is that it will no longer be an aggregate, so aggregate initialization will no longer be available.[19]

For the next few examples, assume the existence of a type, `DataManager`, that is AA using the legacy-AA interface. Using the following `struct Thing` as a starting point, we'll walk, one step at a time, through its transformation into an AA type:

```
namespace BloombergLP {
namespace xyzabc {
struct Thing {
    bsl::string d_name;  // bsl-AA member
    DataManager d_data;  // legacy-AA member
    int         d_score;
    int         d_rank;
};
} // close package namespace
} // close enterprise namespace
```

We begin by adding the `allocator_type` member type alias[20]:

```
struct Thing {
    // PUBLIC TYPES
    using allocator_type = bsl::allocator<>;
    //...
```

A pitfall of this `typedef` is that, unlike the `bslma::UsesBslmaAllocator` trait, `allocator_type` is inherited by derived classes, even though those derived

[18] **iso20**, section 9.4.1, "dcl.init.aggr," pp. 192-197

[19] Several methods, including using customization points and wrapper classes, have been proposed to pass an allocator to a `struct` without losing its aggregate classification. Direct language support for allocators is the least intrusive such proposal; see **meredith19**, time 49:32.

[20] In BDE 3.x, the type alias must be written

```
typedef bsl::allocator<unsigned char> allocator_type;
```

classes might not meet the other requirements of a bsl-AA type. See section "Pitfall: Inheriting from a bsl-AA Class."

To facilitate passing an allocator to the two AA data members (`d_name` and `d_data`), we will need to add constructors that have allocator parameters. An *allocator-extended* constructor is a constructor overload that takes, in addition to the usual parameters, an allocator parameter, either at the end of the parameter list or, less commonly, as the beginning of the parameter list preceded by a parameter of type `bsl::allocator_arg_t`. At a minimum, a default constructor, a copy constructor, an allocator-extended version of the default constructor, and an allocator-extended copy constructor are required. Adding a move constructor and an allocator-extended move constructor is also typically wise because classes that allocate memory (e.g., `bsl::string` and `DataManager`) usually have move constructors whose efficiency we would like to preserve.

The allocator-extended default constructor has a single parameter of type `const allocator_type&`. To prevent the one-parameter constructor from enabling implicit conversion from `allocator_type` to `Thing`, we must add the `explicit` keyword as well. The bold part of the comment in the example below models the typical allocator-related language for a constructor contract,[21] but for brevity, contract comments are omitted in subsequent examples:

```
// CREATORS
Thing();
explicit Thing(const allocator_type& allocator);
    // Create a 'Thing' object having a value-initialized (default)
    // value for each attribute.  Optionally specify an 'allocator'
    // (e.g., the address of a 'bslma::Allocator' object) to supply
    // memory; otherwise, the default allocator is used.
```

Combining these two constructors into one that has a default allocator argument would be tempting:

```
// BAD IDEA
explicit Thing(const allocator_type& allocator = {});
```

This approach, however, causes problems when using C++11 uniform initialization because the combined constructor, being explicit, cannot participate in several desirable patterns:

```
Thing t = {}; // won't compile
extern void f(Thing);
f({});        // won't compile
```

[21] If the two constructor overloads were documented separately, the comments, respectively, would be, "`Use the default allocator to supply memory,`" and "`Use the specified 'allocator' (e.g., the address of a 'bslma::Allocator' object) to supply memory.`"

The allocator-extended copy and move constructors have the same parameters as regular copy and move constructors but with an additional allocator parameter:

```
Thing(const Thing& original);
Thing(const Thing& original, const allocator_type& allocator);
Thing(Thing&& original) noexcept;
Thing(Thing&& original, const allocator_type& allocator);
```

Note that, unlike the move constructor, the *extended* move constructor cannot be `noexcept` because it will sometimes require memory allocation, as described later in section "Implementing a Class That Allocates Memory."

Because the original `struct` did not directly manage any resources, has no direct allocator member, and maintains no intra-member class invariants, the compiler can correctly automatically generate the remaining three *Rule of Five*[22] operations (destructor, copy-assignment operator, and move-assignment operator) — none of which have an allocator parameter — by defaulting them (preferred) or omitting them (for C++03 compatibility):

```
~Thing() = default;

Thing& operator=(const Thing&) = default;
Thing& operator=(Thing&&)      = default;
```

Within this paper, *modern* (i.e., C++11 and later) syntax is generally assumed, e.g., for declaring rvalue references. Code that must be compatible with C++03 can emulate rvalue references and move operations using the `bslmf::MovableRef` facility and must use `typedef` instead of `using` to declare type aliases. A detailed description of how to add move operations to an AA class such that they work correctly for both C++03 and C++11 can be found in Appendix C, Allocator-Aware Move Operations in C++03.

Although not required, we might want to add a constructor with one parameter for each data member, along with an optional allocator parameter:

```
Thing(const bsl::string_view& name,
      const DataManager&      data,
      int                     score,
      int                     rank,
      const allocator_type&   allocator = {});
```

The above constructor allows a `Thing` to be constructed using *list initialization*[23] with braces, which looks just like aggregate initialization and thus recovers an important feature that was lost when we changed `Thing` such that it is no longer an aggregate:

```
// Construct a 'Thing' using a default allocator (C++11 and later).
Thing theThing = { "hello", DataManager(), 2, 5 };
```

---

[22] **john18**

[23] **iso20**, section 9.4.4, "dcl.init.list," pp. 199-204

We complete the interface by adding an accessor named `get_allocator` to retrieve the allocator that was supplied at construction:

```
    // ACCESSORS
    allocator_type get_allocator() const;
```

The following is our new complete class interface:

```
namespace BloombergLP {
namespace xyzabc {
struct Thing {
    // PUBLIC TYPES
    using allocator_type = bsl::allocator<>;

    // PUBLIC DATA MEMBERS
    bsl::string d_name;
    DataManager d_data;
    int         d_score;
    int         d_rank;

    // CREATORS
    Thing();
    explicit Thing(const allocator_type& allocator);
    Thing(const Thing& original);
    Thing(const Thing& original, const allocator_type& allocator);
    Thing(Thing&&      original);
    Thing(Thing&&      original, const allocator_type& allocator);
    Thing(const bsl::string_view& name,
          const DataManager&      data,
          int                     score,
          int                     rank,
          const allocator_type&   allocator = {});  // optional

    // ACCESSORS
    allocator_type get_allocator() const;
};
}  // close package namespace
}  // close enterprise namespace
```

The implementation of the allocator-extended default constructor passes the allocator to the constructors for each of the AA members of the `struct` and value-initializes the non-AA members in the member initializer list:

```
// allocator-extended default ctor
Thing::Thing(const allocator_type& allocator)
  : d_name(allocator)
  , d_data(allocator.mechanism()) // BAD IDEA; see below
  , d_score()
  , d_rank()
{
}
```

Note that, for members having the legacy-AA interface (e.g., `d_data`), the `bslma::Allocator*` resource must be extracted from the `bsl::allocator` object by means of the `mechanism` method. Calling `mechanism` directly in this way will make the code brittle in the presence of pmr-AA types. A better approach to handling the mismatch between a bsl-AA class and a legacy-AA member is to transform allocator using `bslma::AllocatorUtil::adapt`, which returns an object convertible to `bsl::allocator<T>`, `bslma::Allocator*`, and `bsl::polymorphic_allocator<T>` and thus can be passed as the allocator type to a bsl-AA, legacy-AA, or pmr-AA constructor:

```
// allocator-extended default ctor
Thing::Thing(const allocator_type& allocator)
  : d_name(bslma::AllocatorUtil::adapt(allocator))
  , d_data(bslma::AllocatorUtil::adapt(allocator))
  , d_score()
  , d_rank()
{
}
```

If `DataManager` is eventually converted to the bsl-AA model, as we recommend, the code in the constructor implementation above will compile and work correctly without change. Similarly, the implementation is robust if `bsl::string` is replaced by a different string-like type, such as `std::pmr::string`. The same pattern applies to the other allocator-aware constructors[24]:

```
// allocator-extended copy ctor
Thing::Thing(const Thing& original, const allocator_type& allocator)
  : d_name(original.d_name, bslma::AllocatorUtil::adapt(allocator))
  , d_data(original.d_data, bslma::AllocatorUtil::adapt(allocator))
  , d_score(original.d_score)
  , d_rank(original.d_rank)
{
}

// allocator-extended move ctor; may throw
Thing::Thing(Thing&& original, const allocator_type& allocator)
  : d_name(bsl::move(original.d_name),
           bslma::AllocatorUtil::adapt(allocator))
  , d_data(bsl::move(original.d_data),
           bslma::AllocatorUtil::adapt(allocator))
  , d_score(original.d_score)
  , d_rank(original.d_rank)
{
}

Thing::Thing(const bsl::string&    name,
             const DataManager&    data,
             int                   score,
             int                   rank,
             const allocator_type& allocator)
  : d_name(name, bslma::AllocatorUtil::adapt(allocator))
  , d_data(data, bslma::AllocatorUtil::adapt(allocator))
```

[24]The `bsl::move` function used in this and subsequent examples is equivalent to `std::move` and is available via `#include <bsl_utility.h>`.

```
  , d_score(score)
  , d_rank(rank)
{
}
```

The *non*extended default and copy constructors *could be* compiler generated (explicitly defaulted), resulting in member-wise construction. However, as each member is initialized, the default memory resource must be looked up. Theoretically, the nonextended move constructor can *also* be defaulted, but only if all AA members are known to have well-behaved move constructors that copy the allocator from the moved-from object. Such an approach to the move constructor is brittle, and writing the move constructor explicitly is safer to ensure that each AA member is constructed using the allocator of the moved-from object.

An easy-to-follow rule of thumb is to write each nonextended default, copy, and move constructor to simply delegate to its extended counterpart, passing the appropriate allocator to the extended constructor:

```
// regular default ctor
Thing::Thing()
  : Thing(allocator_type())
{
}

// regular copy constructor
Thing(const Thing& other)
  : Thing(other, allocator_type())
{
}

// regular move ctor.
Thing::Thing(Thing&& original) noexcept
  : Thing(bsl::move(original), original.get_allocator())
{
}
```

Alternatively, the copy constructor and extended copy constructor can be merged into a single function having a default constructor argument:

```
// regular and copy constructor
Thing(const Thing& other, const allocator_type& allocator = {});
```

The implementation of this combined copy constructor is identical to the implementation of the extended copy constructor shown previously. Though compact and easy to understand, this version of the copy constructor breaks the regular pattern of the other two special constructors. Whether to combine the copy constructors or have the regular copy constructor delegate to the extended one is a matter of taste.

Next, we declare and implement the `get_allocator()` method:

```
    allocator_type get_allocator() const;
```

The allocator can be retrieved from any of the AA data members. Thus,

```
Thing::allocator_type Thing::get_allocator() const
    { return d_name.get_allocator(); }
```

is equivalent to

```
Thing::allocator_type Thing::get_allocator() const
    { return d_data.allocator(); }
```

Both implementations rely on the specific AA model used by the data member: `d_name` is bsl-AA, so `d_name.get_allocator()` returns a `bsl::allocator<>`; `d_data` is legacy-AA, so `d_data.allocator()` returns a `bslma::Allocator*` that is then implicitly converted to the `bsl::allocator<>` return value. Again, the code will break if the data member's AA model changes. To make the implementation robust if `d_name` becomes pmr-AA or `d_data` becomes bsl-AA, use the `bslma::AATypeUtil::getAllocatorFromSubobject` function,[25] which retrieves the allocator from an AA subobject regardless of its AA model and converts it to `Thing`'s `allocator_type`:

```
Thing::allocator_type Thing::get_allocator() const {
    return bslma::AATypeUtil::getAllocatorFromSubobject<allocator_type>(d_name);
}
```

or

```
Thing::allocator_type Thing::get_allocator() const {
    return bslma::AATypeUtil::getAllocatorFromSubobject<allocator_type>(d_data);
}
```

Because a pure `struct` like `Thing` belongs to the simplest category of would-be AA types, it suffers the most relative increase in size and complexity when transforming from non-AA to AA.[26] The size and complexity of this interface comes from the addition of a type alias, four constructors, and an accessor (getter).

Providing maximal C++03 compatibility (see Appendix C) would require manually implementing both assignment operators. C++03 compatibility also substantially complicates the implementation of the move operations in ways unrelated to allocators.

Subsequent sections will illustrate how the percentage increase in code size becomes progressively smaller as the complexity of the starting point increases.

Making an Attribute Class AA

An attribute class is similar to a simple `struct` except its data members are private and managed by invariant-preserving public manipulators and

[25] `getAllocatorFromSubobject` provides lossless recovery of an allocator value when a bsl-AA class contains a pmr-AA subobject.

[26] **meredith19** describes an approach being considered for integrating allocators into the C++ language. If adopted, this method would completely eliminate both the interface complexity and the effort needed to make most allocating types AA.

accessors. As before, if any of the data members are AA, the attribute class itself should also be AA. Relative to the simple `struct`, a new challenge is that existing constructors may need to be adapted.

Let's start with a non-AA attribute-class version of our `Thing` example:

```
namespace BloombergLP {
namespace xyzabc {

class Thing {
    // DATA
    bsl::string d_name;
    DataManager d_data;
    int         d_score;
    int         d_rank;
  public:
    // CREATORS
    Thing();
    explicit Thing(bool dataMode);
    Thing(bsl::string_view   name,
          const DataManager& data,
          int                score = 0,
          int                rank  = 5);

    // MANIPULATORS
    void setName(bsl::string_view name);
    void setScore(int score);
    ...
};

}  // close package namespace
}  // close enterprise namespace
```

Defining the `allocator_type` and `get_allocator()` members is achieved exactly as it was for the simple `struct`. We must declare extended copy and move constructors just as we did for the simple `struct` example, using identical implementations.[27] The Rule of Five operations are compiler-generated in this example, but we should implement the nonextended copy and move constructors for the same reason as we did for the plain `struct`.

Unlike a simple `struct`, our `Thing` attribute class has existing constructors that we must extend with an allocator parameter. We add a default allocator to the constructor taking a `bool` parameter like this:

```
    explicit Thing(bool dataMode, const allocator_type& allocator = {});
```

The four-parameter constructor in our `Thing` class is a bit trickier than the preceding one. Just adding an allocator to the end of the parameter list does not suffice:

---

[27] Similarly, if we choose to provide C++03 move operations, we declare and implement those operations identically to the simple `struct` example. See Appendix C for detailed instructions and caveats for implementing move operations in C++03.

```
// BAD IDEA: partial solution at best
Thing(bsl::string_view      name,
      const DataManager&    data,
      int                   score = 0,
      int                   rank  = 5,
      const allocator_type& allocator = {});
```

The problem is that, although we can specify a `score`, `rank`, and `allocator`, we cannot specify just a `score` and an `allocator`. As was stated in section "The Allocator-Aware Interface," *every* constructor argument list must be usable with an allocator argument.[28] Typically, the way we fix this problem is to create an allocator-parameter overload for the case of no optional parameters, one optional parameter, and two optional parameters:

```
Thing(bsl::string_view      name,
      const DataManager&    data,
      const allocator_type& allocator = {});
Thing(bsl::string_view      name,
      const DataManager&    data,
      int                   score,
      const allocator_type& allocator = {});
Thing(bsl::string_view      name,
      const DataManager&    data,
      int                   score,
      int                   rank,
      const allocator_type& allocator = {});
```

The solution above uses the *trailing-allocator convention* specifying the allocator parameter — one of two conventions recognized by containers and AA utilities. The other option is to use the *leading-allocator convention*, a C++11 Standard alternative for specifying the allocator whereby the allocator parameter appears at the *start* of a parameter list, preceded by the special marker type `bsl::allocator_arg_t`. Using this convention, we need only two overloads for the preceding constructor:

```
Thing(bsl::string_view      name,
      const DataManager&    data,
      int                   score = 0,
      int                   rank  = 5);
Thing(bsl::allocator_arg_t  ,
      const allocator_type& allocator,
      bsl::string_view      name,
      const DataManager&    data,
      int                   score = 0,
      int                   rank  = 5);
```

The second constructor is invoked by specifying `bsl::allocator_arg` as the first argument and an allocator as the second argument:

```
Thing myThing(bsl::allocator_arg, myAlloc, "Fred", myData, myScore);
```

28 A complaint when developing AA software is that convenient language features such as default arguments and aggregate initialization become so much more difficult. This complaint provides significant motivation for language support for allocators described in **meredith19**.

If the constructor is a template with a C++11 variadic parameter list (i.e., a parameter list where the last deduced parameter contains an ellipsis), the leading-allocator convention is the *only* way to add an allocator parameter.[29] If there are no variadic constructors, then whether to provide multiple overloads or to use the leading-allocator convention is a matter of practicality; if the issue arises for only one or two constructors, having only one or two default parameters each, most programmers prefer to keep the trailing-allocator convention with a trailing defaulted allocator parameter. Because the (C++03-compatible) BDE infrastructure has no way to detect which of the two allocator-passing conventions is used for a specific constructor, a class using the leading-allocator convention must define the trait `bslmf::UsesAllocatorArgT` to be true, even if used in C++11. If we choose this convention, the preceding declaration for the constructor taking only a bool would also need to change; the `bslmf::UsesAllocatorArgT` trait is defined on a *per-class* basis, not a *per-constructor* basis; all constructors (including the extended copy and move constructors) *must use the same allocator-passing convention.*

Implementing a Class That Allocates Memory

Up until now, all memory allocation and deallocation has been managed by the member variables of our class. If a class needs to allocate memory directly, we need to understand additional, not necessarily intuitive, rules and apply them in the destructor, assignment operators, and `swap`.

Let's assume that `DataManager` is a large type and is unused much of the time in our `Thing` objects. Allocating space (from the allocator) to hold the `DataManager` on an as-needed basis is more sensible than having the `DataManager` object be an always-present data member. For illustrative purposes, the `Thing` class below uses a raw pointer to hold the address of allocated memory, although, in practice, a smart pointer (`bslma::ManagedPtr` or `std::unique_ptr`) might be a better choice. For classes that directly manage memory, whether or not they use allocators or smart pointers, the Rule of Five members must be defined by the user and *must not* be compiler generated:

[29] Technically, extracting the last argument from a variadic argument list and determining whether it is an allocator at compile time should be possible. Getting this right (including avoiding bad or ambiguous overload resolution) requires a lot of template metaprogramming for which neither BDE nor the Standard Library currently have support.

```
class Thing {
    // DATA
    bsl::string  d_name;
    DataManager *d_data_p;
    int          d_score;
    int          d_rank;
    // ...
  public:
    // TYPES
    using allocator_type = bsl::allocator<>;

    // CREATORS
    Thing() noexcept;
    explicit Thing(const allocator_type& allocator) noexcept;
    explicit Thing(bool                  dataMode,
                   const allocator_type& allocator = {});
    Thing(const Thing& original);
    Thing(const Thing& original, const allocator_type& allocator);
    Thing(Thing&& original) noexcept;
    Thing(Thing&& original, const allocator_type& allocator);
    ~Thing();

    // MANIPULATORS
    Thing& operator=(const Thing& rhs);
    Thing& operator=(Thing&& rhs);

    void swap(Thing& other);

    // ...
    // ACCESSORS
    // ...
    allocator_type get_allocator() const;
};
```

The `get_allocator` method returns the allocator held by the string member. If we did not already have an AA member, we would need to store the allocator separately in a new member of type `allocator_type`.

All memory allocation and deallocation of owned subobjects should go through the allocator. The `bslma::AllocatorUtil::newObject` method is the most effective way to allocate and initialize a single object. If the object being created is AA, `newObject` automatically passes the allocator to the object's constructor, as shown in the `dataMode` and extended copy constructors:

```
Thing::Thing(bool dataMode, const allocator_type& allocator)
  : d_name(allocator), d_data_p(nullptr), d_score(0), d_rank(5)
{
    if (dataMode) {
        d_data_p =
            bslma::AllocatorUtil::newObject<DataManager>(allocator);
    }
}
```

```
Thing::Thing(const Thing& original, const allocator_type& allocator)
  : d_name(original.name(), allocator)
  , d_data_p(nullptr)
  , d_score(original.d_score)
  , d_rank(original.d_rank)
{
    if (original.d_data_p) {
        d_data_p = bslma::AllocatorUtil::newObject<DataManager>(allocator
                                                    *original.d_data_p);
    }
}
```

We destroy an object and release its footprint memory back to the allocator by calling `bslma::AllocatorUtil::deleteObject`:

```
Thing::~Thing()
{
    Bslma::AllocatorUtil::deleteObject(get_allocator(), d_data_p);
}
```

The constructors above are exception safe because only one object is being allocated and constructed. Any memory allocated by `d_name` is freed by the `bsl::string` destructor if any subsequent operation in the `Thing` constructor throws an exception. `AllocatorUtil::newObject` is atomic with respect to exceptions in that it either succeeds entirely or cleans up after itself if the memory allocation or `DataManager` constructor throw an exception.

The BDE 3.x library does not have the `AllocatorUtil` package. The most straightforward way to allocate and construct an object is to use the `bslma::Allocator` overload of `operator new`. This version of `operator new` takes a reference — not a pointer[30] — to a `bslma::Allocator` object and allocates memory from that allocator. To destroy and deallocate an object created in this way, we use the `deleteObject` method of `bslma::Allocator` (or, when the most-derived type of the object is known statically, `deleteObjectRaw`). BDE 3.x versions of the `dataMode` and extended copy constructors and destructor would look different:

```
Thing::Thing(bool dataMode, const allocator_type& allocator)
  : d_name(allocator), d_data_p(nullptr), d_score(0), d_rank(5)
{
    if (dataMode) {
        d_data_p = new(*allocator.mechanism())
                       DataManager(bslma::AllocatorUtil::adapt(allocator));
    }
}
```

[30] Warning: Passing a pointer to an allocator would still compile but with the undesirable runtime semantics of corrupting the allocator.

```
Thing::Thing(const Thing& original, const allocator_type& allocator)
  : d_name(original.name(), allocator)
  , d_data_p(nullptr)
  , d_score(original.d_score)
  , d_rank(original.d_rank)
{
    if (original.d_data_p) {
        d_data_p = new(*allocator.mechanism())
                       DataManager(*original.d_data_p,
                             bslma::AllocatorUtil::adapt(allocator));
    }
}

Thing::~Thing()
{
    get_allocator().mechanism()->deleteObject(d_data_p);
}
```

Note that the constructors pass the allocator not only to `operator new` for allocation, but also to the `DataManager` constructor for use within the allocated object. The constructors are exception safe because `operator new` is atomic with respect to exceptions, just as `newObject` is. Unlike `newObject`, however, `operator new` does not automatically determine whether `DataObject` is AA nor which AA model or allocator-passing convention it uses. In generic BDE 3.x code, therefore, the construction of a member like `d_data_p` would need to be separate from its allocation, with a proctor definition in between for exception safety (see proctor discussion, below):

```
        d_data_p = allocator.mechanism()->allocate(sizeof(DataManager));
        bslma::DeallocatorProctor<bslma::Allocator>
                               proctor(d_data_p, allocator.mechanism());
        bslma::ConstructionUtil::construct(d_data_p,
                                           allocator.mechanism());
```

Let's say, however, that `DataManager` has a method, `idStr`, that returns a unique string for each `DataManager` object, and let's say that we want to append that string to the `d_name` field. The straightforward modification to the constructor is not exception safe:

```
Thing::Thing(bool dataMode, const allocator_type& allocator)
  : d_name(allocator), d_data_p(nullptr), d_score(0), d_rank(5)
{
    // NOT YET SAFE IN THE PRESENCE OF EXCEPTIONS (FIX TO FOLLOW)!
    if (dataMode) {
        d_data_p =
            bslma::AllocatorUtil::newObject<DataManager>(allocator);
        d_name += ':';                // might throw
        d_name += d_data_p->idStr();  // might throw
    }
}
```

If appending the data ID to the name throws an exception, the object created by `newObject` will become orphaned, resulting in a memory leak. To be exception safe, we need a way to reverse the effect of a successful call to newObject if a function subsequently fails (e.g., because an exception was thrown). The best way to achieve this exception safety is to use an RAII[31] object whose destructor will automatically rewind a specified step if the current function returns prematurely. The BDE library refers to such an object by the unconventional term *proctor*. Every proctor type has a constructor that gives it control over some resource, a release method that releases control over the resource without destroying it, and a destructor that destroys any resource(s) still under the proctor's control. In the Thing constructor example, we can use a bslma::DeleteObjectProctor to destroy and deallocate the DataManager object if an exception occurs during the string append operations[32]:

```
Thing::Thing(bool dataMode, const allocator_type& allocator)
  : d_name(allocator), d_data_p(nullptr), d_score(0), d_rank(5)
{
    if (dataMode) {
        d_data_p =
            bslma::AllocatorUtil::newObject<DataManager>(allocator);
        bslma::DeleteObjectProctor<allocator_type, DataManager>
                                   delProct(get_allocator(), d_data_p);
        d_name += ':';                 // might throw
        d_name += d_data_p->idStr();   // might throw
        delProct.release();            // no more ops that might throw
    }
}
```

In the revised constructor above, a DeleteObjectProctor object is created immediately after the DataManager object is returned from AllocatorUtil::newObject and takes responsibility for destroying and deallocating it in the event of an exception. Once all potentially throwing operations have completed successfully, the release method is called to deactivate the proctor.

BDE 3.x does not have DeleteObjectProctor but does have a similar RawDeleterProctor that assumes the legacy-AA model. Compared to DeleteObjectProctor, the ALLOCATOR and TYPE parameters are reversed, as are the constructor arguments. The ALLOCATOR parameter is a *reference* (not a *pointer*) type, typically bslma::Allocator or a pool type, whereas the allocator constructor argument *is* a pointer. When using bsl::allocator, we must retrieve that pointer by means of the mechanism accessor:

```
    bslma::RawDeleterProctor<DataManager, bslma::Allocator>
                             delProct(d_data_p, get_allocator().mechanism());
```

---

[31] RAII is an initialism for *Resource Acquisition Is Initialization*, a common C++ idiom whereby an object acquires a resource (in this case, memory) upon construction and automatically relinquishes it upon destruction.

[32] The C++ Standard's unique_ptr (**iso20**, section 20.11.1, "unique.ptr," pp. 630–639) can be used as a proctor but requires a special deleter that is not yet standard (see **köppe20**). Additionally, bdlb::ScopeExit in BDE and the scope_exit and scope_fail templates described in **sommerlad19** can take the place of proctors but are not yet standardized.

The BDE 4.0 library provides three new proctor class templates.

Proctor template	**Reverses this operation**
`bslma::DeleteObjectProctor`	`bslma::AllocatorUtil::newObject`
`bslma::DeallocateObjectProctor`	`bslma::AllocatorUtil::allocateObject`
`bslma::DeallocateBytesProctor`	`bslma::AllocatorUtil::allocateBytes`

Note that each new proctor's name and constructor parameter list corresponds to the operation it performs on premature destruction; for example, `bslma::DeleteObjectProctor`'s constructor takes the same arguments as `bslma::AllocatorUtil::deleteObject`. At the end of this section, however, we'll see a way to use the `Thing` class as its *own* proctor.

As with any class that allocates memory, the copy-assignment operator must take care not to overwrite the `d_data_p` pointer before deallocating the memory to which it points nor to leave the assigned-to object in an invalid state if an exception is thrown:

```
Thing& operator=(const Thing& rhs)
{
    d_name  = rhs.d_name;
    if (d_data_p && rhs.d_data_p) { // Assign data object.
        *d_data_p = *rhs.d_data_p;
    }
    else if (d_data_p) {            // Delete data object.
        bslma::AllocatorUtil::deleteObject(get_allocator(), d_data_p);
        d_data_p = nullptr;
    }
    else if (rhs.d_data_p) {        // Allocate and copy data object.
        d_data_p = bslma::AllocatorUtil::newObject<DataManager>(
                                  get_allocator(), *rhs.d_data_p);
    }
    d_score = rhs.d_score;
    d_rank  = rhs.d_rank;
    return *this;
}
```

Move operations (move construction, move assignment, and swap) require special consideration. Ideally, a move operation on an allocating type requires moving only the pointers to allocated memory, without copying the contents of allocated memory. Let's encapsulate this ideal in a private `fastMoveFrom` member function that will be used to implement the public move operations:

```
class Thing {
    // ...
    // PRIVATE MANIPULATORS
    void fastMoveFrom(Thing& other) noexcept;
        // Move the specified 'other' Thing into '*this'.
```

```
        // The behavior is undefined unless 'other' and '*this'
        // are distinct objects that have the same allocator.
    //...
};

void
Thing::fastMoveFrom(Thing& other) noexcept
{
    BSLS_ASSERT_SAFE(get_allocator() == other.get_allocator());

    bslma::AllocatorUtil::deleteObject(get_allocator(), d_data_p);

    d_name   = bsl::move(other.d_name);
    d_data_p = other.d_data_p; other.d_data_p = nullptr;  // pointer move[33]
    d_score  = other.d_score;
    d_rank   = other.d_rank;
}
```

When the caller does not explicitly provide an allocator, the (nonextended) move constructor should use the *moved-from* object's allocator for the newly constructed object. This behavior is unlike that of all other constructors, which use the *default* allocator when the caller does not provide one.[34] Thus, while we can express the allocator-extended version of most constructors using an optional allocator parameter, as we did for the copy constructor, a defaulted parameter would produce an incorrect result for the move constructor:

```
// BAD IDEA
Thing(Thing&& original, const allocator_type& allocator = {});
```

Another important difference between the regular (nonextended) move constructor and the extended move constructor is that the former is usually declared `noexcept`, whereas the latter might allocate memory so it's not necessarily `noexcept`.

The nonextended move constructor can simply invoke the extended default-constructor and delegate to `fastMoveFrom`:

```
Thing::Thing(Thing&& original) noexcept
  : Thing(original.get_allocator())   // C++11 delegating constructor
//: d_name(original.get_allocator()), d_data_p(nullptr) // C++03 version
{
    fastMoveFrom(original);
}
```

The extended move constructor can be invoked with an allocator different from that of the moved-from object. In that case, moving the pointer is problematic because the moved-to object's destructor will attempt to deallocate the memory from the wrong allocator. The correct behavior, therefore, is performing a fast

---

[33] Using the C++14 and later Standard Library `exchange` template, the pointer move can be expressed more simply as `d_data_p = std::exchange(other.d_data_p, nullptr);`.
[34] In theory, the result of a nonextended move constructor should be the same as that of constructing the moved-from object directly at the moved-to location.

move only when the allocators are the same and otherwise falling back to a copy[35]:

```
Thing::Thing(Thing&& original, const allocator_type& allocator)
  : Thing(allocator)                     // C++11 delegating constructor
//: d_name(allocator), d_data_p(nullptr) // C++03 version
{
    // '*this' is in a valid (empty) state.
    if (allocator == original.get_allocator()) {
        fastMoveFrom(original);
    }
    else {
        operator=(original);  // copy assignment
    }
}
```

The same issue affects the move-assignment operator. The allocator used by the moved-to object does not change during move assignment and may differ from the allocator used by the moved-from object. As in the case of the extended move constructor, we must test for the same allocator and move or copy as appropriate:

```
Thing& Thing::operator=(Thing&& rhs)
{
    if (get_allocator() != rhs.get_allocator()) {
        operator=(rhs);  // copy assignment
    else if (this != &rhs) {
        fastMoveFrom(rhs);
    }

    return *this;
}
```

The third move operation is `swap`. Following a recent BDE convention, an AA class *should* provide a public member function `swap` that never throws an exception and *may* provide an ADL-discoverable free function `swap` that might throw:

```
class Thing {
    // ...
  public:
    // ...
    void swap(Thing& other);
    // ...
};

void swap(Thing& a, Thing& b);
```

---

[35] The BDE rule is that if the allocators match, the extended move constructor must behave like the regular move constructor, and if the allocators do not match, the extended move constructor must behave like the extended *copy* constructor. See “Expected Properties of Types Declaring the `bslma::UsesBslmaAllocator` Trait” in **bloomberg**.

If the arguments use the same allocator, then either swap operation is performed in constant time (no allocations or deallocations) and never throws an exception. Swapping the allocators for the purpose of guaranteeing the O(1), nonthrowing behavior may be tempting, but allowing the allocator to change during an object's lifetime violates important invariants, especially within containers.[36] For implementing swap when the objects being swapped have different allocators, we have three options, each of which is slower and uses more temporary memory than the one before.

1) Require allocator equality as a precondition (typically verified with an assertion). With this implementation, swap can execute in constant time.[37]
2) Perform the traditional three-move swap, where two of the moves will degenerate to copies that allocate memory and might throw. This option is the equivalent of a fully qualified call to std::swap.
3) Provide the strong exception guarantee, whereby each object is carefully copied using the other object's allocator before attempting any moves.

All three options are reasonable choices, but our member swap implementation is limited to option 1 due to the BDE rule that the member swap must not throw. By using the regular move constructor and fastMoveFrom, we need not concern ourselves with the allocator at all, except (optionally) to assert that they are equal:

```
void Thing::swap(Thing& other) noexcept {
    BSLS_ASSERT_SAFE(get_allocator() == other.get_allocator());
    // All three of the following calls are 'noexcept'.
    Thing temp1(bsl::move(*this));
    this->fastMoveFrom(other);
    other.fastMoveFrom(temp1);
}
```

For the swap free function, all three implementation options are possible, but we'll implement the third one for illustrative purposes. The strong exception guarantee means that the original objects are left unchanged if the swap exits with an exception:

```
void swap(Thing& a, Thing& b)
{
    if (a.get_allocator() == b.get_allocator()) {
        a.swap(b);
    }
```

[36] The bsl-AA interface is designed so that all the elements in a container of Thing will use the same allocator, thus ensuring locality of reference and consistent allocator lifetime throughout the container. It is critical, therefore, that nothing outside the container can change the allocator of a container element; e.g., swap(v[0],x) must not swap the allocators of v[0] and x.

[37] Some algorithms cannot meet their complexity guarantees unless elements can be swapped in constant time. The PMR containers in the Standard Library have this precondition on swap.

```
    else {
        // Creating temporaries might allocate and copy, which might throw
        Thing temp1(bsl::move(a), b.get_allocator());
        Thing temp2(bsl::move(b), a.get_allocator());

        a.swap(temp2); // no allocation, copy, or throw
        b.swap(temp1); // no allocation, copy, or throw
    }
}
```

Note that if the objects being swapped have the same allocator, which is always the case when the objects are elements of the same container, then the swap itself requires no allocations and will not throw an exception, regardless of whether we call member `swap` or free-function `swap`. We can use this fact to simplify the implementations of both the copy and move assignment operators, in the process bestowing on them the strong exception guarantee:

```
Thing& Thing::operator=(const Thing& rhs)  // simplified implementation
{
    Thing(rhs, get_allocator()).swap(*this);
    return *this;
}

Thing& Thing::operator=(Thing&& rhs)
{
    Thing(bsl::move(rhs), get_allocator()).swap(*this);
    return *this;
}
```

In these rewritten assignment operators, the `swap` method is invoked only if the extended copy or move constructor succeeds without throwing an exception. The constructed temporary object has the same allocator as `*this`, so the subsequent `swap` is guaranteed to succeed in constant time. If the extended copy or move constructor fails, then `*this` is not modified. Note that these simplified assignment operators will sometimes result in an additional allocation and deallocation relative to the previous implementations.

Taking advantage of a swap operation that is guaranteed not to throw under certain conditions, we can eliminate the proctor in some or all of the `Thing` constructors:

```
Thing::Thing(bool dataMode, const allocator_type& allocator)
  : d_name(allocator), d_data_p(nullptr), d_score(0), d_rank(5)
{
    if (dataMode) {
        Thing temp(allocator);  // will clean up on exception
        temp.d_data_p =
            bslma::AllocatorUtil::newObject<DataManager>(allocator);
        temp.d_name += ':';                      // might throw
        temp.d_name += temp.d_data_p->idStr();  // might throw
        this->swap(temp)            // no more ops that might throw
    }
}
```

This version of the constructor depends on `temp`'s destructor to clean up the data manager if an exception is thrown subsequent to its creation. Effectively, `Thing` is using itself as a proctor with `swap` being used instead of `release`. An important caveat with this technique is that we often assert that class invariants are preserved on entry to a destructor. When using this class-as-its-own-proctor technique, it is critical that an exception cannot be thrown while the `temp` object is in an invalid state.

Implementing an AA Class Template

Until a class template is instantiated, we cannot know whether a type related to a template parameter (known in the C++ Standard as a *dependent type*) is AA. Subobjects of a dependent type must be constructed in such a way that an allocator is passed to the constructor if and only if the dependent type is AA. An instantiation of a class template might not be AA unless at least one dependent type is AA, posing the additional challenge of writing an adaptable interface.

For this example, we'll begin with our `Thing` attribute class, modified such that, instead of a `DataManager` member, it holds an object of a type specified by the user as a template parameter:

```
namespace BloombergLP {
namespace xyzabc {

template <class TYPE>
class Thing {
    // DATA
    bsl::string d_name;
    TYPE        d_data;
    int         d_score;
    int         d_rank;

  public:
    // TYPES
    using allocator_type = bsl::allocator<>;

    // CREATORS
    Thing();
    explicit Thing(const allocator_type& allocator);
    Thing(bsl::string_view      name,
          const TYPE&           data,
          const allocator_type& allocator = {});
    ...
```

Our `Thing` class template is already AA, so the type aliases, traits, and default constructor prototype need not change. The problem comes in the implementation of the allocator-extended constructors:

```
template <class TYPE>
Thing::Thing(bsl::string_view      name,
             const TYPE&           data,
             const allocator_type& allocator)
  : d_name(name, bslma::AllocatorUtil::adapt(allocator))
  , d_data(data, ?allocator?)
  , d_score(0)
  , d_rank(5)
{
}
```

If `TYPE` is *not* AA, then the initializer for `d_data` should be simply `d_data(data)`, but if `TYPE` *is* AA, then the initializer should be `d_data(data, bslma::AllocatorUtil::adapt(allocator))`. We can achieve this conditionally AA initialization by calling `bslma::ConstructionUtil::make<TYPE>(allocator, data)` and initializing `d_data` with the return value of this call:

```
template <class TYPE>
Thing::Thing(bsl::string_view      name,
             const TYPE&           data,
             const allocator_type& allocator)
  : d_name(name, bslma::AllocatorUtil::adapt(allocator))
  , d_data(bslma::ConstructionUtil:make<TYPE>(allocator, data))
  , d_score(0)
  , d_rank(5)
{
}
```

All the C++11 compilers in use at Bloomberg will perform the above initialization without making extra copies. Any number of constructor arguments can be supplied after the allocator. If `TYPE` is not AA, then the allocator is ignored; if it is AA using the trailing allocator convention, then `ConstructionUtil::make` will *append* `allocator` to the argument list; and if it is AA using the leading allocator convention, then it will *prepend* `bsl::allocator_arg` and `allocator` to the argument list. Furthermore, if `TYPE` is legacy-AA, `ConstructionUtil::make` will call `allocator.mechanism()` to retrieve a `bslma::Allocator` pointer to pass to the `TYPE` constructor, obviating a call to `AllocatorUtil::adapt`.

Unfortunately, using `ConstructionUtil::make` does have some limitations.

- `ConstructionUtil::make` is unavailable for the Sun and IBM compilers because they do not reliably prevent extra copy operations, so it is probably inappropriate for any code that might be compiled on those platforms, including most code supporting C++03.
- In language versions prior to C++17, `TYPE` is required to be move-constructible; otherwise, the instantiation of `ConstructionUtil::make` will yield a compilation error.

Most types are move-constructible, including virtually all types that are copy-constructible (because a copy constructor is a perfectly valid move constructor). Nonmovable types are typically mechanisms such as mutexes, for which the object's location in memory is as important as its state. If you are content limiting `TYPE` to movable types, `ConstructionUtil::make` is the cleanest way to initialize a member of template-parameter type.

When `ConstructionUtil::make` does not work, we can achieve conditionally AA initialization by using the `bslalg::ConstructorProxy<TYPE>`, which wraps an object of `TYPE` and presents a consistent AA constructor interface regardless of whether `TYPE` is AA and regardless of whether it uses the leading or trailing allocator convention:

```
template <class TYPE>
class Thing {
    bsl::string                      d_name;
    bslalg::ConstructorProxy<TYPE>   d_dataProxy;
    int                              d_score;
    int                              d_rank;
```

The `ConstructorProxy` constructor takes 0 to 14 arguments of arbitrary type *always* followed by an allocator, i.e., it follows the trailing-allocator convention, but note that the allocator is *not* optional. As for `ConstructionUtil::make`, nonallocator arguments are passed to the proxied object's constructor and the allocator is either ignored (for non-AA types) or passed to the object's constructor (for AA types). Thus, the initializer for `Thing`'s constructor now looks somewhat different:

```
template <class TYPE>
Thing::Thing(bsl::string_view      name,
             const TYPE&           data,
             const allocator_type& allocator)
  : d_name(name, bslma::AllocatorUtil::adapt(allocator))
  , d_dataProxy(data, allocator)
  , d_score(0)
  , d_rank(5)
{
}
```

Note that we don't need to use `bslma::AllocatorUtil::adapt` for the `d_dataProxy` initializer because the allocator is guaranteed to be `bsl::allocator` or `bsl::polymorphic_allocator`, either of which can be initialized from the `allocator` argument.

In BDE 3.x, the allocator parameter to `ConstructorProxy` has type `bslma::Allocator*`, so using `bslma::AllocatorUtil::adapt` is required:

```
  , d_dataProxy(data, bslma::AllocatorUtil::adapt(allocator))
```

In addition to the constructor changes, we must replace all uses of `d_data` with `d_dataProxy.object()` throughout `Thing`'s implementation:

```
template <class TYPE>
TYPE& Thing<TYPE>::data() { return d_dataProxy.object(); }
```

The Thing template described so far is always AA because it contains a bsl::string, which is known to be AA. If we were to remove the string, then the situation would be different:

```
template <class TYPE>
class Thing {
    bsl::string d_name;
    TYPE        d_data;
    int         d_score;
    int         d_rank;
    // …
```

If TYPE is AA, then Thing<TYPE> should be AA; otherwise, Thing<TYPE> need not be AA. The easiest way to handle this situation is to *artificially* make Thing AA in all circumstances by adding an allocator member:

```
template <class TYPE>
class Thing {
    bsl::allocator<> d_allocator;
    TYPE             d_data;
    int              d_score;
    int              d_rank;
    // ...
```

The allocator, though unused when TYPE is not AA, always takes up space in the object footprint; this wasted space is often an acceptable cost for the simplicity of this approach.

If the extra pointer-sized space consumption is an issue or if having the interface be pure AA or pure non-AA is important, then some refactoring and metaprogramming will be required; let's consider one such approach. We begin by declaring our Thing template with an extra Boolean parameter that defaults to true if TYPE is AA and false otherwise[38]:

```
template <class TYPE, bool USES_ALLOC =
                      BloombergLP::bslma::UsesBslmaAllocator<TYPE>::value>
class Thing;
```

The partial specialization for which USES_ALLOC is true supplies the entire AA interface:

```
template <class TYPE>
class Thing<TYPE, true> {
    TYPE        d_data;
    int         d_score;
    int         d_rank;

  public:
    // PUBLIC TYPES
    using allocator_type = bsl::allocator<>;
```

[38] This metaprogramming technique is difficult to use for variadic class templates; indirection through inheritance or alias templates is required in this case.

```
    // CREATORS
    Thing();
    explicit Thing(const allocator_type& allocator);
    explicit Thing(const TYPE& data,
                   const allocator_type& allocator = {});
    // ..
    const TYPE& data() const;
    allocator_type get_allocator() const;
};
```

By supplying an explicit `true` value for `USES_ALLOCATOR`, this partial specialization can also be instantiated for *non-AA* types, and we will shortly exploit this feature.

The allocator-extended constructors in this specialization initialize `d_dataProxy` using the specified allocator:

```
template <class TYPE>
Thing<TYPE, true>::Thing(const TYPE&           data,
                         const allocator_type& allocator)
  , d_data(bslma::ConstructionUtil:make<TYPE>(allocator, data))
  , d_score(0)
  , d_rank(5)
{
}
```

Although the `TYPE` is known to be AA, we still use `ConstructionUtil:make` because it automatically handles potential AA model and constructor-convention mismatches for us. We could similarly use `ConstructorProxy` to achieve the same effect.

The `Thing::get_allocator` method retrieves the allocator from the object stored within the `d_data` member:

```
template <class TYPE>
allocator_type Thing<TYPE, true>::get_allocator() const
{
    using bslma::AATypeUtil;
    return AATypeUtil::getAllocatorFromSubobject<allocator_type>(d_data);
}
```

We create the partial specialization where `USES_ALLOC` is false to inherit from the other (AA) specialization, hard-coding `USES_ALLOC` to true in the base class. The new specialization defines only the non-AA constructors and disables `allocator_type` (by redefining it to `void`) and `get_allocator` (by redefining it as private):

```
template <class TYPE>
class Thing<TYPE, false> : Thing<TYPE, true> {

    // PRIVATE TYPES
    using Base = Thing<TYPE, true>;

    // NOT IMPLEMENTED
    void get_allocator() const;
```

```
  public:
    // TYPES
    using allocator_type = void;

    // CREATORS
    Thing() : Base() { }
    explicit Thing(const TYPE& data) : Base(data) { }
    ...
};
```

This layering of the non-AA specialization on top of the AA specialization works because the (default-constructed) allocator in the AA implementation is discarded by `bslma::ConstructionUtil::make`. The base-class `get_allocator()` method is never instantiated for non-AA `TYPE`s, so no compilation errors result from its otherwise-invalid use of `getAllocatorFromSubobject`.

Unfortunately, duplicate declarations of nonextended constructors are present in the two partial specializations of our class template, so any interface maintenance must be done in both places. Implementation changes, however, affect only the AA specialization, mitigating the maintenance issue caused by this duplication.

Several other metaprogramming approaches exist for implementing a conditionally AA template. Work is in progress on a set of tools to make the task simpler, especially when more than one dependent type is involved.

Implementing an AA Container

The archetypal AA type is a container class (or container class template). The new challenge when implementing a container is insertion and removal of elements (each of which might be AA) outside of the constructors and destructor, especially in the presence of exceptions.

Let’s look at a simplified implementation of `MyList`, an AA doubly linked list container template:

```
template <class TYPE>
struct MyList_Node;

template <class TYPE>
class MyList {

    // PRIVATE TYPES
    using Node = MyList_Node<TYPE>;

    bsl::allocator<>  d_allocator;
    Node             *d_head_p, *d_tail_p;

  public:
    // TYPES
    using allocator_type = bsl::allocator<>;
```

```
    // CREATORS
    MyList();
    explicit MyList(const allocator_type& allocator);
    MyList(const MyList& original, const allocator_type& allocator = {});
    MyList(MyList&& original);
    MyList(MyList&& original, const allocator_type& allocator);
    ~MyList();

    // MANIPULATORS
    MyList& operator=(const MyList& rhs);
    MyList& operator=(MyList&& rhs);
    template <class... ARGS>
      void emplace_back(ARGS&&... args);
    void pop_back();
    TYPE& front();
    TYPE& back();

    // ACCESSORS
    const TYPE& front() const;
    const TYPE& back() const;
    allocator_type get_allocator() const { return d_allocator; }
};

// FREE FUNCTIONS
bool operator==(const MyList& a, const MyList& b);
bool operator!=(const MyList& a, const MyList& b);
```

For brevity, the example omits iterators and other operations that a reusable list class would normally supply. We'll focus on the `emplace_back` and `pop_back` member functions, which respectively insert and remove elements at the end of the list. The implementation of the constructors, destructor, assignment operators, accessors, and equality comparison operators present no allocator-related challenges beyond those discussed in previous sections. For example, the destructor can be implemented using `pop_back`:

```
template <class TYPE>
MyList<TYPE>::~MyList()
{
    while (d_head_p) {
        pop_back();
    }
}
```

The implementation of `emplace_back` involves three main steps.

1) Allocate a new `MyList_Node` object.
2) Construct the new element within the node.
3) Link the new node onto the list.

The `MyList_Node` class template holds an object that might or might not be AA. Because this class template is private to the component, however, we can take some shortcuts on the interface. Specifically, a node never needs to hold its own allocator, so we omit the `get_allocator` member as well as the copy and move constructors and assignment operators. The constructor for

`MyList_Node` must conditionally pass an allocator to `TYPE`'s constructor, which we accomplish using `ConstructionUtil::make`:

```
template <class TYPE>
struct MyList_Node {
    using allocator_type = allocator_type<>;

    TYPE  d_value;
    Node *d_prev_p;
    Node *d_next_p;

    template <class... ARGS>
    MyList_Node(const allocator_type& allocator, ARGS&&... ctorArgs)
      : d_value(bslma::ConstructionUtil::make<TYPE>(allocator,
                                                     std::forward<ARGS>(args)...))
    { }
};
```

We create a new node (steps 1 and 2 in our list above) using `bslma::AllocatorUtil::newObject`:

```
template <class TYPE>
template <class... ARGS>
void MyList<TYPE>::emplace_back(ARGS&&... args)
{
    Node *node_p = bslma::AllocatorUtil::newObject<Node>(get_allocator(),
                                                     std::forward<ARGS>(args)...);
    // No potentially throwing operations after this point.
    node_p->d_prev_p = d_tail_p;
    node_p->d_next_p = nullptr;
    d_tail_p = node_p;
    if (!d_head_p) {
        d_head_p = node_p;
    }
}
```

Note that all potentially throwing operations are bundled into the single call to `newObject`; as tempting as managing `node_p` with a proctor seems, doing so is unnecessary in this case.

Finally, let's look at the `pop_back` member function, which removes the last element from our list. The steps in the implementation are basically the reverse of `emplace_back`. First, we unlink the last node from the list, and then we call `bslma::AllocatorUtil::deleteObject` to destroy the node (including its `TYPE` member) and release its storage back to the allocator:

```
template <class TYPE>
void MyList<TYPE>::pop_back()
{
    MyList_Node<TYPE> *node_p = d_tail_p;
    if (node_p) {
        d_tail_p = node_p.d_prev_p;
        if (d_tail_p) {
            d_tail_p->d_next_p = nullptr;
        }
```

```
        else {
            d_head_p = nullptr;
        }

        bslma::AllocatorUtil::deleteObject(get_allocator(), node_p);
    }
}
```

Using a proctor is unnecessary for this implementation because neither destruction nor deallocation should ever throw an exception, and even if they did, no method exists to unwind these operations.

Although combining memory allocation and element construction into one step is practical for node-based containers like `MyList`, separating allocation from construction is required for most array-based containers such as `vector`. Instead of creating nodes containing one element each, array-based containers allocate a block of memory suitable for holding a variable-length, contiguous array of elements and then construct a subset of those elements as a separate step during insertion. A partial implementation of a vector-like class template, `MyVector`, might hold an allocator, pointers to the start and end of the data elements, and a capacity indicating how many data elements the current block can hold:

```
template <class TYPE>
class MyVector {
    // DATA
    bsl::allocator<>  d_allocator;
    TYPE             *d_data;
    TYPE             *d_dataEnd;
    std::size_t       d_capacity;

    // PRIVATE MANIPULATORS
    void reallocate(std::size_t newCapacity);
        // Change the capacity of the vector to the specified
        // 'newCapacity' by reallocating the data array and moving all
        // existing elements to the new array.

    // ...
};
```

Operations that construct multiple elements, such as the `reallocation` operation, could use `AllocatorUtil::allocateObject` to allocate the raw memory and `ConstructionUtil::construct` to construct each element within the allocated storage. If an exception is thrown after some elements have been constructed, those elements must be destroyed. The `bslma::AutoDestructor` proctor is used to manage all currently constructed elements and automatically destroy them in the case of an exceptional exit. Note that this example uses `AutoDestructor` to manage elements in forward order (incrementing the proctor as new elements are constructed at the end of the array), but it can also be used in reverse order (decrementing the proctor as new elements are constructed at the front)[39]:

```
template <class TYPE>
void MyVector::reallocate(std::size_t newCapacity)
{
    TYPE *p = bslma::AllocatorUtil::allocateObject<TYPE>(d_allocator,
                                                          newCapacity);

    bslma::DeallocateObjectProctor<allocator_type, TYPE>
                            dataProctor(d_allocator, p, newCapacity);
    bslma::AutoDestructor dtorProctor(p);
    for (TYPE *e = d_data; e != d_dataEnd; ++e, ++p) {
        bslma::ConstructionUtil::construct(p, d_allocator, bsl::move(*e));
        ++dtorProctor;  // Manage the newly constructed element.
    }
    for (TYPE *e = d_data; e != d_dataEnd; ++e) {
        bslma::DestructionUtil::destroy(e);
    }
    bslma::AllocatorUtil::deallocateObject(d_allocator, d_data);
    dtorProctor.release();              // Commit constructions.
    d_data     = dataProctor.release();  // Commit allocation.
    d_dataEnd  = p;
    d_capacity = newCapacity;
}
```

Pitfall: Inheriting from a bsl-AA Class

Bloomberg code has a number of classes that inherit from `bsl::string`, `bsl::vector`, and other bsl-AA classes.[40] The derived class is often not AA:

```
class StringLike : public bsl::string {
    // String-like class that inherits from 'string' but is distinct for
    // overload-resolution purposes.
  public:
    StringLike() {}
    StringLike(const char *s) : bsl::string(s) {}    // IMPLICIT
    // ...
};
```

Unfortunately, `StringLike` inherits the `allocator_type` member from `bsl::string` causing `bslma::UsesBslmaAllocator<StringLike>` to evaluate true, even though `StringLike` provides none of the allocator-extended constructors needed for it to be treated as an AA type. Thus, inserting into an AA container such as `bsl::vector<StringLike>` will result in compilation errors as the vector code tries to pass an allocator to each `StringLike` element.

---

[39] See **bloombergc**.

[40] Sometimes inheriting from a class that was not intended for inheritance has legitimate engineering reasons, but doing so is generally a dubious practice.

A similar problem occurs when a legacy-AA class inherits from a bsl-AA class:

```
class IntCollection : public std::vector<int> {
    // Vector-like class that inherits from 'vector' and caries some
    // extra data.  This is a legacy-AA class.

    std::string d_name;

  public:
    BSLMF_NESTED_TRAIT_DECLARATION(IntCollection, UsesBslmaAllocator);

    explicit IntCollection(const std::string&  name,
                           bslma::Allocator   *basicAllocator = 0);
    IntCollection(const IntCollection&  original,
                  bslma::Allocator     *basicAllocator = 0);

    // ...
    bslma::Allocator *allocator() const;
};
```

Again, `allocator_type` is inherited, so while `IntCollection` is AA, a container like `bsl::vector<IntCollection>` will treat it as *bsl-AA* and produce a compilation error when it attempts to pass a `bsl::allocator` to the `IntCollection` constructor.

Inheriting from a library class is always a dubious proposition with potential unintended consequences, so the first and best approach is often to change the class to use composition instead of inheritance. A second and complementary approach is to make the derived class fully bsl-AA (often not difficult if its constructors simply forward to the AA member or base class); see Appendix A for guidance on converting a legacy-AA class to bsl-AA.

The simplest fix that applies to both problematic uses of inheritance is to neutralize the inherited `allocator_type` by redefining it to `void`:

```
    using allocator_type = void;  // Neutralize inherited type
```

When fixing a class that suffers from one of these inheritance-related problems, applying the `void` allocator type fix in the short term and one or both of the other fixes later on is safe.

Testing AA Components

If we hope to produce quality software, then instrumenting a class to be AA must be paired with testing the AA aspects of that class[41] and verifying seven qualities.

1) All memory belonging to the object is allocated from its allocator and not from global `operator new` or the default allocator.

---

[41] Some of the techniques described in this section are demonstrated in **fehér19a**, starting at time 18:08.

2) The object returns all owned memory to the allocator on destruction.
3) The class object retains a copy of the specified allocator on construction (or the default allocator if one isn't specified).
4) The allocator doesn't change, for either operand, during copy or move assignment.
5) AA elements in a container use the same allocator as the container throughout the container's lifetime.
6) Memory is not leaked and objects are not corrupted if an exception occurs while allocating memory or while constructing or modifying an AA subobject.
7) (Optional white-box test) The object allocates memory only when expected and in the expected quantities (number of blocks, bytes, and/or allocator calls).

The `bslma_testallocator` and `bslma_testallocatormonitor` components facilitate achieving these test goals.[42] The `bslma::TestAllocator` class is an allocator that tracks blocks and bytes allocated, deallocated, and currently in use. It detects attempts to deallocate the same block twice, to deallocate a block from a different allocator than was used to allocate it, and to destroy the allocator while blocks are still outstanding (i.e., leaked). A `bslma::TestAllocator` can also be set to throw an exception after a specific number of allocation attempts for testing exception safety in the AA type. The `bslma::TestAllocatorMonitor` class captures the state of a specific `bslma::TestAllocator` and provides concise Boolean queries for whether allocation has increased or decreased (and by how much) or stayed the same since it was created or last reset.

Let's look at a simple example using `bslma::TestAllocator` and `bslma::TestAllocatorMonitor` to verify correct allocator-related behavior in our primitive linked-list container. Note that we use the common idiom of passing the address of a `bslma::Allocator`-derived class (`bslma::TestAllocator`) to the `MyList` constructor, taking advantage of the implicit conversion from `bslma::Allocator*` to `bsl::allocator`:

```
{
    // If veryVeryVeryVerbose is true, 'ta' prints data on every operation
    bslma::TestAllocator ta("list alloc", veryVeryVeryVerbose);
    // ... Other code that uses 'ta' could go here. ...
    bslma::TestAllocatorMonitor tam(&ta);

    MyList<int> theList(&ta);
    ASSERT(&ta == theList.get_allocator());
    ASSERT(tam.isTotalSame());    // no 'ta' allocations in the constructor

    theList.emplace_back(3);
    ASSERT(1 == tam.numBlocksInUseChange());  // exactly one block used
```

[42] Documentation is available in **bloombergd** and **bloomberge**.

```
        theList.pop_back();
        ASSERT(tam.isInUseSame());    // back to original memory use
        // ... Other code that uses 'ta' could go here. ...
    } // 'ta' destructor checks for memory leaks.
```

A class that has been converted from non-AA to AA might erroneously have residual calls to `operator new` that bypass the allocator. Even more common is forgetting to pass the allocator to a subobject's constructor, resulting in the subobject erroneously using the default allocator. We can detect improper use of `operator new` by replacing global `operator new` and `operator new[]` with ones that direct all allocation requests to a specific test allocator (which gets its memory from `malloc`, not `operator new`, thus avoiding recursion)[43]:

```
namespace {
bslma::TestAllocator& opNewAllocator() {
    static bslma::TestAllocator ret;
    return ret;
}
}
void *operator new(std::size_t size)
    { return opNewAllocator().allocate(size); }
void operator delete(void *block_p) noexcept
    { return opNewAllocator().deallocate(block_p); }
```

We can detect improper use of the default allocator by setting it, either at the top of `main` or in a specific test case, to a designated test allocator using `bslma::DefaultAllocatorGuard`. When the guard goes out of scope, the default allocator is automatically restored to its previous value:

```
bslma::TestAllocator defaultTestAllocator;
bslma::DefaultAllocatorGuard daGuard(&defaultTestAllocator);
```

Using a `bslma::TestAllocatorMonitor`, we can verify that operations on the type being tested result in no *net* allocations from either `opNewAllocator` or `defaultTestAllocator`, though *transient* allocations for local variables, if any, are expected. We'll improve our previous test by adding these additional checks for incorrect use of `operator new` or the default allocator. We expect no transient allocations from the default allocator or `operator new`, so we use the `isTotalSame` method instead of `isInUseSame` to verify that no allocations at all were done from those sources:

[43] In modern C++, `operator new` has no exception specification and `operator delete` is marked `noexcept`; in C++03, they are marked `throw(std::bad_alloc)` and `throw()`, respectively. Thus, conditional compilation on `BSLS_COMPILERFEATURES_SUPPORT_NOEXCEPT` is needed to support both. Later C++ versions also have aligned overloads of `operator new`, requiring even more overloads in the most thorough tests.

```
bslma::TestAllocator da("default alloc", veryVeryVeryVerbose);
{
    bslma::DefaultAllocatorGuard daGuard(&da);
    bslma::TestAllocatorMonitor onm(&opNewAllocator());
    bslma::TestAllocatorMonitor dam(&da);

    bslma::TestAllocator ta("list alloc", veryVeryVeryVerbose);
    bslma::TestAllocatorMonitor tam(&ta);

    MyList<int> theList(&ta);
    ASSERT(&ta == theList.get_allocator());
    ASSERT(tam.isInUseSame());   // No memory was consumed by constructor.

    theList.emplace_back(3);
    ASSERT(1 == tam.numBlocksInUseChange());  // exactly one block used

    theList.pop_back();
    ASSERT(tam.isInUseSame());   // back to original memory use

    ASSERT(dam.isTotalSame());   // Default allocator is unused in block.
    ASSERT(onm.isTotalSame());   // 'operator new' is unused in block.

    // 'ta' destructor checks for memory leaks.
}
```

Allocating memory correctly is a distinct concern that is conditioned, but not ensured, by the correct setting of the allocator itself. After each constructor call, we verify that `get_allocator` returns the expected result and that the expected number of bytes or blocks was allocated from the allocator.

An object should allocate and deallocate only from the allocator acquired on construction, except that transient allocations and their corresponding deallocations should use the global allocator or a locally defined allocator. Special care must be taken to ensure that copy and move constructors and assignment operators allocate the correct amount of memory from the correct allocator. The following list summarizes the six requirements.

1) The nonextended copy constructor creates an object having the default allocator and allocates from only that allocator, regardless of the allocator held by the copied-from object.
2) The extended copy constructor creates an object having the specified allocator and allocates from only that allocator, regardless of the allocator held by the copied-from object.
3) The nonextended move constructor creates an object having a copy of the moved-from object's allocator. Typically, the move constructor would not allocate any memory.
4) The extended move constructor behaves like the regular move constructor if the allocator specified in the constructor arguments compares equal to the allocator of the moved-from object. Otherwise, the extended move constructor behaves like the extended copy constructor. Alternatively, if the element type is not AA and has a nonthrowing move operation, a container can perform element-by-element move, rather

than copy. (For example, `bsl::shared_ptr` has optimized move construction but is not AA.)

5) The copy-assignment operator allocates only from the allocator of the left-side operand, which does not change as a result of the assignment.
6) The move-assignment operator should not allocate if the moved-to object has an allocator equal to that of the moved-from object; otherwise, the move-assignment operator should behave like the copy-assignment operator or, as in the case of the extended move constructor, perform element-by-element move construction or assignment.

When testing a class template, the facilities in the `bsltf` package[44] can help. The `bsltf::AllocTestType`, for example, is a simple bsl-AA type that we use to verify that our `MyList` container propagates its allocator to its contained elements:

```
bslma::TestAllocator ta1("test alloc 1", veryVeryVeryVerbose);
bslma::TestAllocator ta2("test alloc 2", veryVeryVeryVerbose);
MyList<bsltf::AllocTestType> theList(&ta1);
...
bsltf::AllocTestType val5(5, &ta2);
ASSERT(&ta2 == val5.get_allocator());  // uses specified allocator
theList.emplace_back(val5);
ASSERT(&ta1 == theList.back().get_allocator());  // uses list's allocator
```

Our last testing goal is informally referred to as *allocation-caused exception safety*. Any class that allocates memory might encounter an out-of-memory exception, especially when using an allocator that obtains memory from an intentionally limited pool. The test allocator has a `setAllocationLimit(n)` manipulator that will cause the allocator to throw an exception at the *n*th allocation (counting from 0). This manipulator is typically used idiomatically by the `BSLMA_TESTALLOCATOR_EXCEPTION_TEST_BEGIN/_END` macros (defined in `bslma_testallocator.h`) to run an exception-safety test on a block of code having a deterministic allocation pattern. The macros execute the code in a `try` block within a loop, catching the exceptions thrown by the test allocator. Beginning with an allocation limit of 0, the limit is incremented each time an exception is caught. The process is repeated until the code being tested completes without throwing. This idiom tests that the block of code handles failure cleanly at each possible allocation point. After the `_END` macro, we should verify that no memory was leaked from the allocator. We use this idiom to test `emplace_back`'s exception safety:

```
bslma::TestAllocator ta("list alloc", veryVeryVeryVerbose);
BSLMA_TESTALLOCATOR_EXCEPTION_TEST_BEGIN(ta) {
    MyList<bsltf::AllocTestType> theList(&ta);
```

[44] **bloombergf**

```
        for (int i = 0; i <= 5; ++i) {
            bsltf::AllocTestType v(i);  // uses default allocator
            theList.emplace_back(v);
        }
        ASSERT(5 == theList.back().data());  // minimal correctness check
    } BSLMA_TESTALLOCATOR_EXCEPTION_TEST_END
    ASSERT(0 == ta.numBlocksInUse());
```

Each iteration of the `for` loop requires two allocations: one for the new list node and one for the value of the test object stored in the node. If we had needed a proctor in our `emplace_back` implementation and forgotten to add one, this test would have detected a leak in the final `ASSERT`.

In rare cases, we want to test specific postconditions for an exception beyond the absence of leaks and corruption. Our `emplace_back` operation, for example, has the strong guarantee; the list should be unmodified if the operation does not succeed. One way to test these postconditions is to create an RAII class with a destructor that encapsulates those postcondition checks. The class can be either general or tailored for a specific test. We create such a tailored class for our `emplace_back` test as follows:

```
class PushBackExcCheck {
    MyList<bsltf::AllocTestType> const *d_list_p;
    MyList<bsltf::AllocTestType>        d_snapshot; // uses default alloc
  public:
    explicit PushBackExcCheck(const MyList<bsltf::AllocTestType> *list_p)
      : d_list_p(list_p), d_snapshot() { }

    ~PushBackExcCheck() {
        if (d_list_p) ASSERT(*d_list_p == d_snapshot);
    }

    void checkpoint() { d_snapshot = *d_list_p; }
    void release() { d_list_p = nullptr; }
};
```

The `checkpoint` method takes a snapshot of the list and should be called before any potentially throwing operation having the strong guarantee. The `release` method should be called when all the operations have completed successfully, so that the destructor does not test the exceptional conditions when an exception was not thrown. Employing this checker class, we can enhance the previous exception test:

```
bslma::TestAllocator ta("list alloc", veryVeryVeryVerbose);
BSLMA_TESTALLOCATOR_EXCEPTION_TEST_BEGIN(ta) {
    MyList<bsltf::AllocTestType> theList(&ta);
    PushBackExcCheck excChecker(&theList);
    for (int i = 0; i <= 5; ++i) {
        bsltf::AllocTestType v(i);  // uses default allocator
        excChecker.checkpoint();
        theList.emplace_back(v);
    }
```

```
        excChecker.release();
        ASSERT(5 == theList.back().data()); // minimal sanity check
    } BSLMA_TESTALLOCATOR_EXCEPTION_TEST_END
    ASSERT(0 == ta.numBlocksInUse());
```

Although these allocator-specific tests add bulk to the test driver, they actually highlight one of the strengths of making a type AA: We can instrument every aspect of memory allocation easily, without modifying the allocating class and we can use forced allocation failures as a technique to validate exception safety.

Conclusion

Ensuring that reusable components are AA is a critical part of Bloomberg's success in achieving performance, object placement, metrics gathering, thorough testing, and effective debugging. The effort of converting a non-AA component to an AA one, though not negligible, need not be excessive and is in most cases straightforward in nature.

The steps defined in this paper can be condensed into the following **quick reference** for defining a bsl-AA type.

1. Define an `allocator_type` member type that is an alias for `bsl::allocator<>`.

   ```
   class Thing {
       // ...
     public:
       using allocator_type = bsl::allocator<>
   ```

2. If you need a default constructor, also define an *extended* default constructor (using either the trailing- or leading-allocator convention).

   ```
   Thing();
   explicit Thing(const allocator_type& allocator);
   ```

3. Define all seven of the *Rule of Five Plus Two* members: copy constructor, extended copy constructor, move constructor, extended move constructor, copy-assignment operator, move-assignment operator, and destructor. Typically, only the move constructor (and, implicitly, the destructor) can be `noexcept`.

   ```
   Thing(const Thing&);
   Thing(const Thing&, const allocator_type&);
   Thing(Thing&&) noexcept;
   Thing(Thing&&, const allocator_type&);
   ~Thing();

   Thing& operator=(const Thing&);
   Thing& operator=(Thing&&);
   ```

4. For all other constructors, ensure that there also exists an extended version that takes an allocator, either by adding a defaulted allocator parameter to the end of the parameter list or by defining an overload with

an allocator parameter (using either the trailing- or leading-allocator convention).

```
          explicit Thing(int);
          Thing(int, const allocator_type&);
          // ...
      };
```

5. The implementation of each regular (nonextended) constructor should delegate to its corresponding extended constructor. The move constructor should get its allocator from its argument; all other nonextended constructors should use a default-constructed allocator.

```
Thing::Thing() : Thing(allocator_type()) { }
Thing::Thing(const Thing& original)
   : Thing(original, allocator_type()) { }
Thing::Thing(Thing&& original)
   : Thing(original, original.get_allocator()) { }
Thing(int i) : Thing(i, allocator_type()) { }
```

6. If the type does not manage its own memory resources and has no intra-member invariants, then the move and copy assignment operators and the destructor can be defaulted; otherwise, they all need to be user provided.

```
Thing(const Thing& original, const allocator_type& allocator)
    : d_allocator(allocator)
    , d_data_p(bslma::AllocatorUtil::newObject<DATA>(
                                           *original.d_data_p))
    // ...
{
    // ...
}
```

7. If you must define a member variable of `allocator_type` (as opposed to retrieving it from a subobject), **never assign to that allocator member**. Similarly, never swap allocators between objects.
8. Whenever possible, provide a `get_allocator` accessor that either returns a copy of the allocator member (if any) or retrieves the allocator from a subobject.

```
Thing::allocator_type Thing::get_allocator() const
{
    return d_allocator;
}
```

For simple types, such as `struct`s and attribute classes, the basic guidelines shown in this paper allow readers to plumb — swiftly and correctly — their own types to be AA. For more sophisticated components, the (open-source) BDE[45] infrastructure provides low-level components (e.g., `bslma_aatypeutil`,

---

[45] **bloombergi**

bslma_constructionutil, and bslma_testallocator) and tools[46] to facilitate the creation, testing, and validation of AA components. Along with these assets, the recipes delineated in this paper should yield robust and reusable AA software.

APPENDIX A. Converting from Legacy-AA to Bsl-AA

As of June 2020, most AA classes at Bloomberg are legacy-AA, using an old interface style in which allocators are conveyed as raw pointers of type bslma::Allocator* instead of as objects of type bsl::allocator<T>. Transitioning to the bsl-AA model provides the following benefits.

- The bsl-AA model is more like the C++ Standard's pmr-AA model and, for C++17 and later platform libraries, is built on and compatible with pmr.
- The allocator can never accidentally be null because bsl::allocator has a default constructor that selects the currently installed default allocator.
- The bsl-AA model is compatible with C++11 AA constructs. For example, a bsl-AA type X can be stored in a std::vector<X, bsl::allocator<X>>, and will correctly inherit its allocator from the vector.

To convert a class from the legacy-AA interface to the bsl-AA interface, follow the nine steps described here.

1) Make bsl::allocator available for use:

Add
```
#include <bslma_bslallocator.h>
```

In BDE 3.x, the header was named <bslma_stdallocator.h>. The old name continues to work.

2) Add an allocator_type alias (or typedef in C++03) in the public part of the class. An existing bslma::UsesBslmaAllocator trait declaration is harmless but no longer necessary; the same applies to #include <bslma_usesbslmaallocator.h>:

Before
```
public:
  // TRAITS
  BSLMF_NESTED_TRAIT_DECLARATION(Thing,
                                 bslma::UsesBslmaAllocator);
```

[46] Refer to **bloombergh** for a description of the bde_verify tool, which detects a number of allocator-related errors and offers other guidance.

After	`public:`
	`  // TYPES`
	`  **using allocator_type = bsl::allocator<>;**`

3) If the class has a member variable of type `bslma::Allocator*`, change it to `allocator_type` (and remove the `_p` suffix in the name, if any):

Before	`bslma::Allocator *d_alloc_p;`
After	`allocator_type d_alloc;`

4) Change any constructors that take a `bslma::Allocator*` parameter to instead take a `const allocator_type&` parameter. Since the allocator parameter can no longer be confused with the `bslma::Allocator` type, change the name `basicAllocator` to just `allocator`. If the allocator parameter has a `0` default value, change the default expression to an empty initializer list:

Before	`explicit Thing(int, bslma::Allocator *basicAllocator = 0);`
After	`explicit Thing(int, const allocator_type& allocator = {});`

Note that if C++03 compatibility is needed, the empty initializer list (`{}`) must be replaced with an explicit constructor call (`allocator_type()`). Don't forget to update the constructor documentation to reflect the new parameter names and defaults.

5) Remove the use of `bslma::Default::allocator` for denullifying an allocator argument:

Before	`Thing::Thing(int, bslma::Allocator *basicAllocator)` `  : d_alloc_p(bslma::Default::allocator(basicAllocator))`
After	`Thing::Thing(int, const allocator_type& allocator)` `  : d_alloc(allocator)`

6) If the class code initializes any legacy-AA class members, then pass the allocator to `bslma::AllocatorUtil::adapt` when initializing those members:

Before	`Thing::Thing(int, bslma::Allocator *basicAllocator)`
	`  : d_name(basicAllocator)          // bsl-AA member`
	`  , d_data(basicAllocator)          // legacy-AA member`

After	`Thing::Thing(int, const allocator_type& allocator)`
	`  : d_name(allocator)               // bsl-AA member`
	`  , d_data(bslma::AllocatorUtil::adapt(allocator))`
	`        // legacy-AA member`

Note that this conversion can also be applied when initializing `d_name` but is unnecessary in that case because we know that `d_name` is bsl-AA.

7) Add a new `get_allocator` method. For now, keeping a modified version of the `allocator` method is recommended for compatibility with existing legacy-AA client code:

Before	`inline bslma::Allocator *Thing::allocator() const { return d_alloc_p; }`

After	`inline Thing::allocator_type Thing::get_allocator() const { return d_alloc; }`
	
	`inline bslma::Allocator *Thing::allocator() const { return get_allocator().mechanism(); }`

8) Replace uses of `operator new(bslma::Allocator&)` and `bslma::Allocator::deleteObject` with `bslma::AllocatorUtil::newObject` and `bslma::AllocatorUtil::deleteObject`, respectively, and replace old proctors with their newer versions:

Before:

```
Node *node_p = new(*allocator()) Node;
bslma::DeallocatorProctor<bslma::Allocator>
     nodeProct(node_p, allocator());
bslma::ConstructionUtil::construct(node_p->d_value.address(),
                                   allocator(), value);
...
bslma::DestructionUtil::destroy(node_p->d_value.address());
allocator()->deleteObject(node_p);
```

After:

```
Node *node_p =
    bslma::AllocatorUtil::newObject<Node>(get_allocator());
bslma::DeleteObjectProctor<allocator_type, Node>
    nodeProct(get_allocator(), node_p);
bslma::ConstructionUtil::construct(node_p->d_value.address(),
                                   get_allocator(), value);
...
bslma::DestructionUtil::destroy(node_p->d_value.address());
bslma::AllocatorUtil::deleteObject(get_allocator(), node_p);
```

Alternatively, though the allocator member can be stored as an appropriate instantiation of `bsl::allocator` and used directly to allocate, deallocate, construct, and destroy an object, doing so does not make the code any more compact in most cases:

After (Alter-native)	`bsl::allocator<Node> nodeAlloc(get_allocator());`
	`Node *node_p = nodeAlloc.allocate(1);`
	`bslma::DeallocateObjectProctor<bsl::allocator<Node>>`
	`    nodeProct(nodeAlloc, node_p);`
	`nodeAlloc.construct(node_p->d_value.address(), value);`
	`...`
	`nodeAlloc.destroy(node_p->d_value.address());`
	`nodeAlloc.dallocate(node_p);`

9) The existing test driver should compile and run with no changes; we should, however, at least add tests for the `bsl::uses_allocator` trait and the `get_allocator` method and test each constructor with a `bsl::allocator<>` argument:

Before	`typedef MyList<int> Obj;`
	`ASSERT(bslma::UsesBslmaAllocator<Obj>::value);`
	`...`
	`Obj mX(&theTestAlloc); const Obj& X = mX;`
	`ASSERT(&theTestAlloc == X.allocator());`

After	`typedef MyList<int> Obj;`
	`ASSERT(bslma::UsesBslmaAllocator<Obj>::value);`
	`ASSERT((bsl::uses_allocator<Obj, bsl::allocator<>>::value));`
	`...`
	`bsl::allocator<> bslTestAlloc(&theTestAlloc);`
	`Obj mX(bslTestAlloc); const Obj& X = mX;`
	`ASSERT(&theTestAlloc == X.allocator());`
	`ASSERT(bslTestAlloc == X.get_allocator());`

APPENDIX B. Mapping BDE AA Development to C++20 PMR AA Development

The C++17/C++20 *Polymorphic Memory Resource* (PMR) library is an offshoot of the Bloomberg allocator library. The C++17 Standard `std::pmr::memory_resource` abstract base class is nearly identical to Bloomberg's `bslma::Allocator` class; the major differences are that the PMR `allocate` method takes an alignment argument in addition to the number of bytes and that the PMR public member functions are nonvirtual functions that call private virtual functions (e.g., `allocate` is a nonvirtual function that calls the virtual function `do_allocate`), following the pattern for other abstract base classes in the Standard. Similarly, the C++17 Standard `std::pmr::polymorphic_allocator` class template is identical to Bloomberg's `bsl::allocator` template except that the former stores a pointer to a `pmr::memory_resource` (returned by the `resource` method) instead of a pointer to a `bslma::Allocator` (returned by the `mechanism` method). In

C++20, polymorphic_allocator has additional methods, new_object and delete_object, that simplify allocating and deallocating as well as constructing and destroying an object from an allocator and virtually eliminate the need to call the memory_resource accessor.[47] Note that BDE's legacy-AA interface has no equivalent in the C++ Standard Library; i.e., none of the AA types use pmr::memory_resource* in their interfaces except indirectly through pmr::polymorphic_allocator.

To construct an object in uninitialized memory, the C++20 library has std::uninitialized_construct_using_allocator, which works similarly to bslma::ConstructionUtil::construct in the BDE library, ignoring the allocator for non-AA types and passing it to the constructor for AA types. The easiest way to initialize a member variable or local variable in C++20 is with std::make_using_allocator, which constructs and returns a value of specified type, again handling or ignoring the allocator as appropriate:

```
template <class TYPE>
class Thing {
    std::pmr::polymorphic_allocator<> d_allocator;
    TYPE                              d_data;
    // ...
  public:
    using allocator_type = std::pmr::polymorphic_allocator<>;

    Thing();
    explicit Thing(const allocator_type& alloc);
    // ...
};

template <class TYPE>
Thing<T>::Thing(const allocator_type& alloc)
  : d_allocator(alloc), d_data(make_using_allocator<TYPE>(alloc)) { }
```

This idiom relies on C++20's rules for materialization of temporary variables (sometimes referred to as guaranteed copy elision); the return value of make_using_allocator is constructed directly into the d_data member without invoking a copy or move constructor. These construction rules make it unnecessary to use a wrapper class like bslalg::ConstructorProxy. The BDE equivalent is bslma::ConstructionUtil::make, which takes advantage of copy elision in all the C++11 compilers in use at Bloomberg, even though the new materialization rules were not standardized until C++17.

The C++20 Standard Library contains no proctors, but std::unique_ptr can be even more effective in this role with the appropriate use of custom deleters, easy-to-author types that operate on a specified address when a unique_ptr goes out of scope. A unique_ptr with a deleter that invokes

[47] This functionality is being considered for bsl::allocator in a future release of the BDE library.

bslma::AllocatorUtil::deleteObject is straightforward to define and use as a proctor:

```
class pmr_deleter
{
    bsl::polymorphic_allocator<> d_alloc;

  public:
    template <class TYPE>
    pmr_deleter(const bsl::polymorphic_allocator<TYPE>& alloc)
        : d_alloc(alloc) {}

    template <class TYPE>
    void operator()(TYPE *p)
        { bslma::AllocatorUtil::deleteObject(d_alloc, p); }
};

...
    my_Class *p = AllocUtil::newObject<my_Class>(alloc, &counter);
    std::unique_ptr<my_Class, pmr_deleter> proctor(p, alloc);
    // ...
    proctor.release();
```

A proposal currently in review in the C++ Standards Committee calls for (among other things) an allocate_unique function that allocates memory, constructs an object in the memory, and returns a unique_ptr, thus combining object allocation, construction, and exception-protection into one step.[48] Additionally, bslma::TestAllocator has no equivalent in the C++ Standard Library, but one has been proposed and source code is available.[49]

APPENDIX C. Allocator-Aware Move Operations in C++03

Types that allocate memory often benefit from efficient move constructors and move-assignment operators that transfer pointers rather than copying objects. Move operations depend on rvalue references, which were introduced in C++11 but are partially emulated for C++03 in the bslmf_movableref component. Using bslmf::MovableRef instead of rvalue references, we can write move constructors and move-assignment operators, as well as their allocator-extended variants, that are portable between C++03 and C++11 and later.

Let's summarize the bslmf_movableref component.[50]

- An instantiation of bslmf::MovableRef<T> emulates T&& (i.e., an rvalue reference) when compiled with a pre-C++11 compiler and is a

---

48 **köppe20**
49 **fehér19b** provides the proposal, and **fehér19c** offers the source for a reference implementation.
50 See complete documentation in **bloombergb**.

nondeducible alias for `T&&` when compiled with a C++11 or later compiler.

- An expression, *`expr`*, of type `bslmf::MovableRef<T>` is implicitly convertible to `T&` (i.e., an lvalue reference). The conversion can be made explicit by calling `bslmf::MovableRefUtil::access(`*`expr`*`)` (e.g., to access a member of `T` through a `MovableRef<T>`).
- A call to `bslmf::MovableRefUtil::move(ref)` emulates `std::move(ref)`, returning a `bslmf::MovableRef` to the object referenced by (an lvalue or rvalue) `ref`.

To express move operations in C++03, we will need to use `bslmf::MovableRef` to declare not only the regular and extended move constructors, but also the move-assignment operator which, in C++11, often could have been implicitly declared. We also cannot implicitly declare the copy constructor and copy-assignment operator because the presence of the user-defined move constructor and move-assignment operator would cause a C++11 compiler to suppress automatic generation of these operations for reasons unrelated to allocators. Adding the allocator-extended copy and move constructors to the Rule of Five gives us the *Rule of Five Plus Two*, the complete set of which are now required for our C++03-compatible `Thing` class, as shown in the example below. Note that the set includes only six separate members because the copy constructor and extended copy constructor are combined into one:

```
typedef bsl::allocator<> allocator_type;

Thing(const Thing&          original,
      const allocator_type& allocator = allocator_type());
Thing(bslmf::MovableRef<Thing> original) BSLS_KEYWORD_NOEXCEPT;
Thing(bslmf::MovableRef<Thing> original,
      const allocator_type&    allocator);
~Thing();  // OPTIONAL

Thing& operator=(const Thing& rhs);
Thing& operator=(bslmf::MovableRef<Thing> rhs);

allocator_type get_allocator() const;
```

The above C++03 declarations require no modifications to work in C++11, though the presence of the user-defined copy-constructor, move-constructor, copy-assignment, and move-assignment declarations add significant clutter in cases in which the C++ compiler could have generated them automatically. The destructor declaration is typically required in both C++03 and C++11 because the compiler can seldom generate a correct destructor automatically for an allocating type but can be omitted for a type, like this one, that delegates all allocations and deallocations to its member variables. Three additional adaptations for C++03 are the use of `typedef` instead of `using` to declare `allocator_type`, the use of `allocator_type()` instead of `{}` to initialize the default allocator parameters, and the use of `BSLS_KEYWORD_NOEXCEPT` instead of the `noexcept` keyword to indicate that an operation cannot throw an exception.

To conclude our exposition of C++03 compatibility, let's look at the complete implementations of the Rule of Five Plus Two operations for the `Thing` type described in section "Making a Simple `struct` AA." Note that the absence of delegating constructors in C++03 requires a good deal of code duplication between the regular and extended move constructors:

```
// combined regular and allocator-extended copy ctor
Thing::Thing(const Thing& original, const allocator_type& allocator)
  : d_name(original.d_name, bslma::AllocatorUtil::adapt(allocator))
  , d_data(original.d_data, bslma::AllocatorUtil::adapt(allocator))
  , d_score(original.d_score)
  , d_rank(original.d_rank)
{
}

// regular move ctor
Thing::Thing(bslmf::MovableRef<Thing> original) BSLS_KEYWORD_NOEXCEPT
  : d_name(bslmf::MovableRefUtil::move(
              bslmf::MovableRefUtil::access(original).d_name))
  , d_data(bslmf::MovableRefUtil::move(
              bslmf::MovableRefUtil::access(original).d_data))
  , d_score(original.d_score)
  , d_rank(original.d_rank)
{
}

// allocator-extended move ctor; may throw
Thing::Thing(bslmf::MovableRef<Thing> original,
             const allocator_type&    allocator)
  : d_name(bslmf::MovableRefUtil::move(
              bslmf::MovableRefUtil::access(original).d_name)),
           bslma::AllocatorUtil::adapt(allocator))
  , d_data(bslmf::MovableRefUtil::move(
              bslmf::MovableRefUtil::access(original).d_data)),
           bslma::AllocatorUtil::adapt(allocator))
  , d_score(original.d_score)
  , d_rank(original.d_rank)
{
}

// destructor
Thing::~Thing()  // OPTIONAL
{
}

// copy-assignment operator
Thing& Thing::operator=(const Thing& rhs) {
    d_name  = rhs.d_name;
    d_data  = rhs.d_data;
    d_score = rhs.d_score;
    d_rank  = rhs.d_rank;
    return *this;
}
```

```
// move-assignment operator
Thing& Thing::operator=(bslmf::MovableRef<Thing> rhs) {
    Thing& rhsRef = rhs;
    d_name  = bslmf::MovableRefUtil::move(rhsRef.d_name);
    d_data  = bslmf::MovableRefUtil::move(rhsRef.d_data);

    // The use of 'bslmf::MovableRefUtil::move' is optional for the
    // two integer members.
    d_score = bslmf::MovableRefUtil::move(rhsRef.d_score);
    d_rank  = bslmf::MovableRefUtil::move(rhsRef.d_rank);
    return *this;
}
```

APPENDIX D. Alternatives to Storing the Allocator in the Object Footprint

The allocator for an AA class object is typically stored as a data member, contributing the size of a `bsl::allocator` (one pointer size) to the footprint of the object. This overhead is unacceptable in some applications. For example, given a vector of ten million vectors, where 90% of the inner vectors are empty, the wasted space due to the allocator members is about 69MB (assuming 64-bit pointers), which might be significant in memory-constrained environments. The allocator member might cause a class that would fit into a single cache line without allocators to instead straddle two cache lines.

If measurement and calculation show that the allocator within the memory footprint of a class is a problem for a specific application or library, numerous techniques are available to preserve allocator awareness while eliminating the allocator from the object's footprint; what follows is just a small sampling. Note that all these examples involve creating classes not usually found in existing AASI libraries, but these new classes are themselves reusable for other situations where a small footprint is critical. Also note that these designs tend to favor reducing memory consumption at a (sometimes significant) cost in the number of (potentially branching) instructions executed.

The most practical approach is to store the allocator within the object footprint only when the object is holding no other data (e.g., an empty container), and otherwise store the allocator on the heap as part of the object's allocated memory. In our vector example, we could create a custom vector, `UsuallyEmptyVector`,[51] where the allocator and the data pointer share space in a `union`. The allocator is stored in the `union` when the vector's capacity is

---

[51] Calling our customized vector `SmallVector` or `TinyVector` might be tempting, but the vector is actually the reverse of what most people mean by a "small vector." Existing `SmallVector` classes (e.g., see **llvm19**) store a small number of elements directly in the object footprint, making the footprint *bigger* rather than *smaller* and using much more space for empty vectors than our `UsuallyEmptyVector` would.

zero (i.e., no memory has been allocated) and in a prefix to the allocated chunk otherwise. We can do better though. If footprint size truly is critical, we can make our `UsuallyEmptyVector` footprint just one pointer. Assuming that `bslma::Allocator` (or `bsl::memory_resource`) has an alignment requirement of at least 2 bytes, we can steal the low-order bit of a multipurpose pointer to indicate whether the vector is empty; 1 would indicate an empty vector, where the rest of the pointer points to the memory resource, and 0 would indicate a nonempty vector, where the rest of the pointer points to the start of the allocated data area. The data area would contain the length, capacity, and allocator, followed by the actual vector elements. Stealing a bit from a pointer can be challenging to do correctly and portably but is exactly the kind of practical engineering that is worth doing (taking advantage of known platform behavior) when constraints are especially tight.

Another approach[52] that doesn't actually reduce the object footprint but uses the footprint more efficiently is to implement the small-string optimization. Storing a data pointer, size, capacity, and allocator are all unnecessary when the bytes that make up the string value fit within the string footprint, and this approach capitalizes on that knowledge. Two bits of the last byte of the string footprint are used to hold bookkeeping information indicating a) whether the string exceeds the small-string capacity and b) whether the memory resource is not the same as the one returned by `pmr::new_delete_resource`. If both are zero, then the last byte becomes the null terminator for the string and the entire footprint can be used for the small string optimization. Otherwise, the string representation trades off small-string space for storing the additional data necessary for the allocator and capacity. This approach must be applied carefully, with attention to the pointer layout on the target hardware. It can be combined with the previous approach to produce a one-pointer string that can still use the small-object optimization for up to 7 bytes if the `new`/`delete` resource is used.

Finally, let's consider *ILAR*[53] *allocators*. This framework involves an external lookup table that maps address ranges to allocators. Instead of storing the allocator in the object footprint, an object finds its allocator by looking up its own address in the external table. In the case of our vector of usually empty vectors, the outer vector's allocator would register the blocks it allocates in the lookup table so that the inner (usually empty) vectors could find themselves there. ILAR allocators require significant collaboration between allocators and clients, and the lookup table must be carefully managed, especially in a multithreaded environment, but for a memory-constrained application, an ILAR allocator might be a reasonable engineering choice.

---

[52] **alexandrescu04**, time 46:13

[53] *Inverse Lookup Allocator Registry*, invented by Hyman Rosen of Bloomberg

Works Cited

alexandrescu04. A. Alexandrescu, "Write Less Code and More Software," talk given at Amazon, June 4, 2004.
https://youtu.be/Lv5vQXraGJM?t=2773

bloomberga. *BDE API Documentation,* v. 3.93.1.0, Bloomberg, accessed May 6, 2020.
https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/

bloombergb. "Component `bslmf_movableref`," *BDE API Documentation,* v. 3.93.1.0, Bloomberg, accessed May 6, 2020.
https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bslmf__movableref.html

bloombergc. "Component `bslma_autodestructor`," *BDE API Documentation,* v. 3.93.1.0, Bloomberg, accessed May 6, 2020.
https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bslma__autodestructor.html

bloombergd. "Component `bslma_testallocator`," *BDE API Documentation,* v. 3.93.1.0, Bloomberg, accessed May 6, 2020.
https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bslma__testallocator.html

bloomberge. "Component `bslma_testallocatormonitor`," *BDE API* v. 3.93.1.0, *Documentation,* Bloomberg, accessed May 6, 2020.
https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bslma__testallocatormonitor.html

bloombergf. "Package `bsltf`," *BDE API Documentation,* v. 3.93.1.0, Bloomberg, accessed May 6, 2020.
https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bsltf.html

bloomberggg. "Component `bslma_usesbslmaallocator`," *BDE API Documentation,* v. 3.93.1.0, Bloomberg, accessed May 6, 2020.
https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bslma__usesbslmaallocator.html

bloombergh. "`bde_verify`," *BDE Tools Documentation*, Bloomberg, accessed May 6, 2020.
https://bde.bloomberg.com/bde-verify/index.html (Bloomberg internal access required)

bloombergi. "BDE Libraries," GitHub source repository, accessed June 30, 2020.
https://github.com/bloomberg/bde

fehér19a. A. Fehér. "`test_resource`: The pmr Detective," *C++ Conference* (CppCon), Aurora, CO, September, 2019. https://youtu.be/vijveMT2OCY

fehér19b. A. Fehér and A. Meredith. "Add Test Polymorphic Memory Resource to the Standard Library," *C++ Standards Committee Working Group ISOCPP*, Technical Report P1160R1, October 7, 2019. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1160r1.pdf

fehér19c. A. Fehér. "P1160 Add Test Polymorphic Memory Resource to Standard Library," GitHub source repository, 2019. https://github.com/bloomberg/p1160

halpern20a. P. Halpern and J. Lakos. "Value Proposition: Allocator-Aware (AA) Software," *C++ Standards Committee Working Group ISOCPP*, Technical Report P2035R0, January 12, 2020. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2035r0.pdf

halpern20b. P. Halpern. "Unleashing the Power of Allocator-Aware Software Infrastructure," *C++ Standards Committee Working Group ISOCPP*, Technical Report P2126R0, March 2, 2020. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2126r0.pdf

iso20. "ISO/IEC 14882:2020 Programming Language C++," *International Standards Organization (ISO) Draft International Standard N4860,* March 31, 2020. (Free version available as N4861.) http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2020/n4860.pdf

john18. G. John, "Rule of Three vs Rule of Five in C++?" *Tutorials Point*, February, 28, 2018. https://www.tutorialspoint.com/Rule-of-Three-vs-Rule-of-Five-in-Cplusplus

köppe20. T. Köppe. "Allocator-Aware Library Wrappers for Dynamic Allocation," *C++ Standards Committee Working Group ISOCPP*, Technical Report P0211R3, January 14, 2020. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0211r3.html

lakos19. J. Lakos. "Value Proposition: Allocator-Aware (AA) Software," *C++ Conference* (CppCon), Aurora, CO, September 16, 2019. https://youtu.be/ebn1C-mTFVk?t=1770

llvm19. "`llvm::SmallVector< T, N >` Class Template Reference," *LLVM 15.0.0 Documentation*, accessed March 24, 2022.

https://llvm.org/doxygen/classllvm_1_1SmallVector.html?msclkid=bbbdecd1abb411ecab8bd2a9bb9805d0

meredith19. A. Meredith and P. Halpern, "Getting Allocators Out of Our Way," *C++ Conference* (CppCon), Aurora, CO, September 18, 2019. https://www.youtube.com/watch?v=RLezJuqNcEQ

sommerlad19. P. Sommerlad and A. L. Sandoval, "Generic Scope Guard and RAII Wrapper for the Standard Library," *C++ Standards Committee Working Group ISOCPP*, Technical Report P0052R10, February 19, 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0052r10.pdf