D2127R0  
2020-07-16

Pablo Halpern (on behalf of Bloomberg):   
[phalpern@halpernwightsoftware.com](mailto:phalpern@halpernwightsoftware.com)  
John Lakos: [jlakos@bloomberg.net](mailto:jlakos@bloomberg.net)

Making C++ Software *Allocator Aware*

*NOTE: This white paper is intended to motivate continued investment in developing and maturing better memory allocators in the C++ Standard. Although its instructional content is targeted at Bloomberg engineers, the wider C++ community can benefit from this illustration of the use of memory allocators in a large industrial codebase, including the related challenges that the authors are working to alleviate both at Bloomberg and in the C++ Standard.*

# Abstract

*Allocator-aware* (*AA*) software — software that allows a client to supply an allocator at object construction — provides the application developer with an effective, lower-cost alternative to writing bespoke types having individually customized memory management.[[1]](#footnote-2) Creating AA software, however, can be considerably more complex than using existing AA software. After introducing the requirements for an AA type compatible with the BDE[[2]](#footnote-3) Development Environment, this paper walks the reader through 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.

# Introduction

Effective use of an allocator-aware software infrastructure (AASI) is largely a matter of selecting the appropriate allocator when constructing allocator-aware (AA) objects.[[3]](#footnote-4) *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.

Making C++ software AA requires “plumbing” each class that might allocate memory to

1. accept an allocator on construction,
2. store the allocator internally and refrain from changing it throughout the lifetime of the object,
3. use the allocator to allocate and deallocate all owned memory, and
4. make the allocator available to AA subobjects (i.e., member, base-class, contained-element, or any other logically owned objects).

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]](#footnote-5) Despite this code-size increase, the task of transformation is — for the great majority of types written by application developers — straightforward and mechanical.[[5]](#footnote-6),[[6]](#footnote-7) Unfortunately, undertaking this work at Bloomberg is complicated by three factors:

1. Continued use of pre-C++11 compilers.
2. A mismatch between the new-style AA interface recommended in this paper and the current-style interface used in the vast majority of AA code at Bloomberg.
3. Inadequate infrastructure support, especially for the new-style interface.

The process described in this paper is, therefore, even more difficult than it otherwise would be; the paper presents the current state with the expectation that future revisions thereof will be simpler and more concise as these factors are resolved.

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 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, where that type 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.

Next, the paper addresses special considerations involving move and swap operations. It then finishes by describing testing techniques specific to AA components.

Adding allocator support to special classes such as std::optional, std::variant, or smart pointers (e.g., std::shared\_ptr) is beyond the scope of this paper. Such classes use allocators in unique ways and require techniques 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,[[7]](#footnote-8) bdlb\_variant, or bslstl\_sharedptr. This paper, however, provides sufficient information to render most common components consistently and interoperably with the BDE AASI.

# 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]](#footnote-9) This section describes the interface features common to all allocator-aware types consistent with the BDE infrastructure.[[9]](#footnote-10) Subsequent sections describe how to transform a class that is not AA 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, it cannot be used in AA containers or processed by AA utilities. 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 of its elements use the same allocator and could result in a mismatch between the container’s lifetime and the lifetime of one or more of its elements.

This paper adheres to a new AA interface style based on the C++17 standard.[[10]](#footnote-11) To achieve compliance with this style, an AA class, SomeClass, must have the following features:

* The type, SomeClass::allocator\_type, is a specialization of bsl::allocator (often bsl::allocator<char>).
* Both of the following type traits evaluate true:
  + bsl::uses\_allocator<SomeClass, bsl::allocator<char>>::value
  + bslma::UsesBslmaAllocator<SomeClass>::value

The former trait evaluates to true if allocator\_type exists and is convertible to bsl::allocator<char>; the latter trait must be defined explicitly.[[11]](#footnote-12)

* Every constructor has a variant that can be invoked with an allocator to be used by the constructed object.[[12]](#footnote-13) If an allocator is not specified, a default-constructed allocator is used.
* All memory belonging to 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; it is not a temporary variable that exists only for the duration of a single member function invocation.[[13]](#footnote-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 that a smart pointer points to is owned by the pointer but is not a *subobject* of the pointer.
* An object’s allocator does not change over the course of its lifetime.
* The get\_allocator() member function returns the allocator used to construct the object.

Most of the components in the BDE library use an older interface style that is no longer recommended. Both styles are compatible with the BDE infrastructure (e.g., container classes can work with AA elements using either style). The two styles render the features listed above differently:

* The old-style interface has no allocator\_type member.
* Instead of a bsl::allocator object, an allocator in the old style is represented by a raw pointer to bslma::Allocator.
* A class for which the bslma::UsesBslmaAllocator and bsl::uses\_allocator traits both evaluate to true conforms to the new interface whereas a class for which only bslma::UsesBslmaAllocator is true conforms to the old.
* Instead of a get\_allocator() member function that returns allocator\_type, a class using the older interface has an allocator() member function that returns bslma::Allocator\*.

The newer style is simpler to use and prevents coding errors that are common when using raw pointers.[[14]](#footnote-15) The newer interface is also closer to the C++ Standard’s pmr[[15]](#footnote-16) interface — bsl::allocator is nearly identical to std::pmr::polymorphic\_allocator and bslma::Allocator is nearly identical to std::pmr::memory\_resource.[[16]](#footnote-17) The new style is backward-compatible with the old one because bslma::Allocator\* is convertible to bsl::allocator<char> (just as std::pmr::memory\_resource\* is convertible to std::pmr::polymorphic\_allocator<>) and the contained bslma::Allocator\* can be retrieved using the mechanism method of bsl::allocator should it be required for interoperability with old-style components.

Types that are adapted from the C++ Standard library conform to the new style, but the old style 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 old style. Because this paper is primarily concerned with developing new software, it focuses on the new style interface, touching on the old style only in situations where the two come in contact.

The example below shows a minimal class interface using both styles, with the differences highlighted in bold typeface. Move operations were omitted for brevity. Note that the bsl::uses\_allocator trait automatically evaluates to the correct value for both interfaces (false for the old style and true for the new style).

|  |  |
| --- | --- |
| Old-style AA interface | New-style AA interface |
| class MyAAClass {  public:  BSLMF\_NESTED\_TRAIT\_DECLARATION(  MyAAClass,  bslma::UsesBslmaAllocator);  MyAAClass();  explicit  MyAAClass(**bslma::Allocator \***);  MyAAClass(const MyAAClass&,  **bslma::Allocator \* = 0);**  **bslma::Allocator**  **\*allocator**() const;  }; | class MyAAClass {  public:  **using allocator\_type =**  **bsl::allocator<char>;**  BSLMF\_NESTED\_TRAIT\_DECLARATION(  MyAAClass,  bslma::UsesBslmaAllocator);  MyAAClass();  explicit  MyAAClass(**const allocator\_type&**);  MyAAClass(const MyAAClass&,  **const allocator\_type& = {});**  **allocator\_type**  **get\_allocator()** const;  }; |

# Making a Simple struct AA

If a struct contains one or more AA data members, the challenge we take on is to pass 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 of the member types, traits, and constructors needed to give it an AA interface and AA semantics. In the C++ Standard, a simple struct without user-defined constructors belongs to a category of types known as *aggregates* and is compatible with member-by-member *aggregate initialization*.[[17]](#footnote-18) After the transformation described here, the result will, unfortunately, no longer be an aggregate.[[18]](#footnote-19) Many of the steps needed to transform a simple struct into an AA class apply (sometimes with small variations) to more complex categories of types.

For the next few examples, assume the existence of a type, DataManager, that is AA using the old-style 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;

DataManager d\_data;

int d\_score;

int d\_rank;

};

} // close package namespace

} // close enterprise namespace

Our first step is to add the allocator\_type member type alias:[[19]](#footnote-20)

struct Thing {

// PUBLIC TYPES

using allocator\_type = bsl::allocator<char>;  
 //...

To facilitate passing an allocator to the two AA data members (d\_name and d\_data), we will need to add constructors that take allocator arguments. An *allocator-extended* constructor is a constructor overload that takes an allocator argument in addition to the usual arguments, typically as an extra argument at the end of the argument list. At a minimum, a default constructor, an allocator-extended version of the default constructor, and an allocator-extended copy constructor are required. It is also typically wise to add an allocator-extended move constructor because classes that allocate memory (e.g., bsl::string and DataManager) usually have efficient move semantics.

The allocator-extended default constructor takes a single argument of type const allocator\_type&. To prevent the one-argument constructor from enabling implicit conversion from allocator\_type to Thing, we must add the explicit keyword, as well. This example includes a comment that models (in bold) the typical allocator-related language for a constructor contract.[[20]](#footnote-21) For brevity, contract comments are omitted in subsequent examples.

// CREATORS

Thing();

**explicit Thing(const allocator\_type& allocator);**

// Create a 'Thing' object having value-initialized (default)

// values for all attributes. **Optionally specify an 'allocator'**

**// (e.g., the address of a 'bslma::Allocator' object) to supply**

**// memory; otherwise the default allocator is used.**

It would be tempting to combine these two into a single constructor that takes a default allocator argument:

**// 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

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**, const allocator\_type& allocator**);  
 Thing(Thing&& original**, const allocator\_type& allocator**);

The “Rule of Five”[[21]](#footnote-22) operations (destructor, regular copy constructor, regular move constructor, copy-assignment operator, and move-assignment operator) — none of which take an allocator argument — are automatically generated correctly by the compiler and don’t need to be declared. Note that, unlike the (compiler-generated) move constructor, the extended move constructor cannot be noexcept because it will sometimes require memory allocation, as described later in the section, Implementing a Class That Allocates Memory.

Within this paper, C++11 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. Though desirable for most for AA types, adding a regular move constructor and move-assignment operator in C++03 complicates the scenario for C++11 because automatic generation of the copy operations is suppressed. Appendix A details how to add move operations to an AA class such that they work correctly for both C++03 and C++11.

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

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 (brace-style) list initialization,[[22]](#footnote-23) which looks just like aggregate initialization and thus regains an important feature lost when we changed Thing to no longer be an aggregate:

// Construct a 'Thing' using a default allocator (C++11 and later).

Thing theThing = { "hello", DataManager(), 2, 5 };

The implementations of the allocator-extended constructors simply pass the allocator to the constructors for each of the AA members of the struct in the member initializer list. For members having the old-style AA interface, the bslma::Allocator\* resource must be extracted from the bsl::allocator object by means of the mechanism method:

// Regular default ctor

Thing::Thing()

: d\_name(), d\_data(), d\_score(), d\_rank()

{

}

// Allocator-extended default ctor

Thing::Thing(const allocator\_type& allocator)

: d\_name(**allocator**)

, d\_data(**allocator.mechanism()**) // Old AA interface style

, d\_score()

, d\_rank()

{

}

// Allocator-extended copy ctor

Thing::Thing(const Thing& original, **const allocator\_type& allocator**)

: d\_name(original.d\_name, **allocator**)

, d\_data(original.d\_data, **allocator.mechanism()**)

, 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(std::move(original.d\_name), **allocator**)

, d\_data(std::move(original.d\_data), **allocator.mechanism()**)

, 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, **allocator**)

, d\_data(data, **allocator.mechanism()**)

, d\_score(score)

, d\_rank(rank)

{

}

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()**; }

Note that d\_data.allocator() returns a bslma::Allocator\* that is then implicitly converted to bsl::allocator<char>.

Finally, we use the BSLMF\_NESTED\_TRAIT\_DECLARATION macro to declare that the bslma::UsesBslmaAllocator trait is true for our AA type.

struct Thing {

...

**// TRAITS**

**BSLMF\_NESTED\_TRAIT\_DECLARATION(Thing, bslma::UsesBslmaAllocator);**

An alternative way to define the bslma::UsesBslmaAllocator trait is to specialize it outside of the class, in the BloombergLP::bslma namespace (not in Thing’s namespace):

namespace BloombergLP {

namespace bslma {

**template<> struct UsesBslmaAllocator<xyzabc::Thing> : bsl::true\_type {};**

} // close bslma namespace

} // close enterprise namespace

Both styles are acceptable; some developers prefer the external trait style because it removes implementation details, clutter, and mysterious macro invocations from the public class interface, while others find the nested macro to be less cumbersome, especially for templates. The bsl::uses\_allocator trait does not need to be specialized because bsl::uses\_allocator<Thing, bsl::allocator<char>> will automatically evaluate to true due to the presence of allocator\_type within the class.

Our complete class interface now looks like this:

namespace BloombergLP {

namespace xyzabc {

struct Thing {

// PUBLIC TYPES

using allocator\_type = bsl::allocator<char>;

// TRAITS

BSLMF\_NESTED\_TRAIT\_DECLARATION(Thing, bslma::UsesBslmaAllocator);

// PUBLIC DATA MEMBERS

bsl::string d\_name;

DataManager d\_data;

int d\_score;

int d\_rank;

// CREATORS

Thing();

explicit Thing(const allocator\_type& allocator);

// Compiler generates regular (nonextended) copy and move constructors

Thing(const Thing& original, const allocator\_type& allocator);

Thing(Thing&& original, const allocator\_type& allocator);

// ACCESSORS

allocator\_type get\_allocator() const;

};

} // close package namespace

} // close enterprise namespace

This simplest category of would-be AA types suffers the most relative increase in size and complexity when transforming to AA.[[23]](#footnote-24) The size and complexity of this interface comes from two sources:

1. Making the class AA added a type alias, four constructors, an accessor (getter), and a trait definition.
2. Providing maximal C++03 compatibility (see Appendix A) would add an additional constructor and 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, but its data members are private and protected by invariant-preserving public manipulators and accessors. As before, if any of the data members are AA, the attribute class 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 variant of our Thing example:

namespace BloombergLP {

namespace xyzabc {

class Thing {

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);

~Thing();

// MANIPULATORS

setName(bsl::string\_view name);

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, as is adding the UsesBslmaAllocator trait. The Rule of Five operations are compiler-generated, but we must declare extended copy and move constructors just as we did for the simple struct example, with identical implementations.[[24]](#footnote-25)

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

explicit Thing(bool dataMode, **const allocator\_type& allocator = {}**);

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

**// 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 the section titled, “The Allocator-Aware Interface,” *every* constructor argument list be usable with an allocator argument.[[25]](#footnote-26) Typically, the way we fix this problem is to create an allocator-parameter overload for the case of no optional arguments, one optional argument, and two optional arguments, as follows:

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 = {}**);

A second option is to use the *leading-allocator convention*, a standard (since C++11) 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);

If the constructor is a template with a C++11 variadic parameter list (i.e., a parameter list where the last deduced argument contains an ellipsis), the leading-allocator convention is the only way to add an allocator argument.[[26]](#footnote-27) If there are no variadic constructors, then it is a matter of practicality whether to have multiple overloads or to switch to the leading-allocator convention; if the issue arises for only one or two constructors, with only one or two default arguments each, most people prefer to keep the trailing-allocator convention with a trailing defaulted allocator argument. Because the (C++03 compatible) BDE infrastructure does not have a 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. Note that this trait is defined on a per-class 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 for the DataManager (from the allocator) on an as-needed basis is more sensible than having the DataManager object as an always-present data member. For illustrative purposes, the Thing class below uses a raw pointer to hold the address of allocated memory, as though Thing were a low-level data-management component. For classes that directly manage memory, whether they use allocators or not, the Rule of Five members must be defined by the user and *must not* be compiler–generated as they were in the prior examples. We save ourselves a little work by using a default allocator argument to combine the regular copy constructor and the extended copy constructor into one:

class Thing {

// DATA

bsl::string d\_name;

DataManager **\*d\_data\_p**;

int d\_score;

int d\_rank;

// ...

public:

// TYPES

using allocator\_type = bsl::allocator<char>;

// TRAITS

BSLMF\_NESTED\_TRAIT\_DECLARATION(Thing, bslma::UsesBslmaAllocator);

// CREATORS

Thing() noexcept;

explicit Thing(const allocator\_type& allocator) noexcept;

explicit Thing(bool dataMode,

const allocator\_type& allocator = {});

**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{ **return d\_name.get\_allocator();** }

};

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. Using the bslma::Allocator overload of operator new is currently the simplest way to allocate and initialize a single object. This version of operator new takes a reference (not a pointer[[27]](#footnote-28)) to a bslma::Allocator object and allocates memory from that allocator, as shown in the dataMode and extended copy constructors below:

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(allocator.mechanism());**

}

}

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, allocator.mechanism());**

}

}

Note that these constructors pass the allocator not only to operator new for allocation, but also to the DataManager constructor for use within the allocated object; since DataManager is an old-style AA type, we must pass the allocator as a pointer to bslma::Allocator (hence the call to allocator.mechanism()).

We destroy an object and release its footprint memory back to the allocator by calling bslma::Allocator::deleteObject:[[28]](#footnote-29)

Thing::~Thing()

{

**get\_allocator().mechanism()->deleteObject(d\_data\_p);**

}

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 it points to and also not to leave the assigned-to object in an invalid state if an exception is thrown:

Thing& operator=(const Thing& rhs)

{

if (this == &rhs) {

return \*this;

}

// Copy name from rhs using correct allocator

bsl::string rhsName(rhs.d\_name, get\_allocator());

if (!rhs.d\_data\_p) {

**get\_allocator().mechanism()->deleteObject(d\_data\_p);**

d\_data\_p = nullptr;

}

else if (d\_data\_p) {

\*d\_data\_p = \*rhs.d\_data\_p;

}

else {

**d\_data\_p = new(\*get\_allocator().mechanism())**

**DataManager(\*rhs.d\_data\_p, get\_allocator().mechanism());**

return \*this;

}

// The following operations are guaranteed not to throw

d\_name.swap(rhsName);

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 fastMove member function that will be used to implement the public move operations:

class Thing {  
 // ...

// PRIVATE MANIPULATORS

void fastMove(Thing& other) noexcept;

// Move the specified 'other' Thing into '\*this'.

// The behavior is undefined unless 'other' and '\*this'

// have the same allocator.

//...

};

void

Thing::fastMove(Thing& other) noexcept

{

BSLS\_ASSERT\_SAFE(get\_allocator() == other.get\_allocator());

get\_allocator().mechanism()->deleteObject(d\_data\_p);

d\_name = std::move(other.d\_name);

**d\_data\_p = other.d\_data\_p; // Pointer move**

d\_score = other.d\_score;

d\_rank = other.d\_rank;

other.d\_data\_p = nullptr; // Clear duplicate pointer to data

}

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.[[29]](#footnote-30) Thus, while the allocator-extended version of most constructors can be expressed using an optional allocator argument, as we did for the copy constructor, a defaulted argument would produce the 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, then delegate to fastMove:

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

{

**fastMove(original);**

}

The extended move constructor can be invoked with an allocator different than 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 move only when the allocators are the same and falling back to a copy if they are not[[30]](#footnote-31):

Thing::Thing(Thing&& original, const allocator\_type& allocator)

: Thing(**original.get\_allocator()**)

{

// '\*this' is in a valid (empty) state

**if (allocator == original.get\_allocator()) {**

**fastMove(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()) {**

if (this != &rhs) {

**fastMove(rhs);**

}

}

else {

**operator=(rhs); // Copy assignment**

}

return \*this;

}

The third move operation is swap. Following a BDE rule, 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) noexcept;

// ...

};

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

If a and b 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.[[31]](#footnote-32) 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 be declared noexcept, and can execute in constant time.[[32]](#footnote-33)
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 the BDE rule that the member swap must not throw limits its implementation to option 1. By using the regular move constructor and fastMove, 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(this->get\_allocator() == other.get\_allocator());

// All three of the following calls are 'noexcept'.

Thing temp1(std::move(\*this));

this->fastMove(other);

other.fastMove(temp1);

}

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

void swap(Thing& a, Thing& b)

{

// Creation of temporaries might allocate and copy, which might throw.

Thing temp1(std::move(a), b.get\_allocator());

Thing temp2(std::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 swap. We can use this fact to simplify the move assignment operator and retain the strong guarantee:

Thing& Thing::operator=(Thing&& rhs)

{

Thing(std::move(rhs), get\_allocator())**.swap(\*this)**;

return \*this;

}

In this rewritten move-assignment operator, the swap method is invoked only if the extended 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 move constructor fails, then \*this is not modified.

# Implementing an AA Class Template

Before a class template is instantiated, whether a type that relies on a template parameter (known in the C++ Standard as a *dependent type*) is or is not AA is unknown. One challenge resulting from this uncertainty is that objects of 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<char>;

// TRAITS

BSLMF\_NESTED\_TRAIT\_DECLARATION(Thing, bslma::UsesBslmaAllocator);

// 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 constructor prototypes need not change. The problem comes in the implementation of the constructors. Let’s look at one example:

template <class TYPE>

Thing::Thing(bsl::string\_view name,

const TYPE& data,

const allocator\_type& allocator)

: d\_name(name, 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, allocator). No method exists for writing an initializer that is correct for both cases.

We solve the problem by using the wrapper class bslalg::ConstructorProxy<TYPE>, which offers a unified interface over whatever constructor convention TYPE uses (no allocator, trailing allocator, or leading allocator):

**template <class TYPE>**

class Thing {

bsl::string d\_name;

**bslalg::ConstructorProxy<TYPE>** d\_data**Proxy**;

int d\_score;

int d\_rank;

In all cases ConstructorProxy is passed a bslma::Allocator\* using the trailing-allocator convention.[[33]](#footnote-34) The allocator is either ignored (for non-AA types) or is correctly passed to the object’s constructor (for AA types). Thus, the initializer for d\_dataProxy unconditionally becomes d\_dataProxy(data, allocator.mechanism()).

Throughout Thing’s implementation, we replace all uses of d\_data with d\_dataProxy.object():

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 very different:

template <class TYPE>

class Thing {

**~~bsl::string d\_name;~~**

bslalg::ConstructorProxy<TYPE> d\_dataProxy;

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<char> d\_allocator;**

bslalg::ConstructorProxy<TYPE> d\_dataProxy;

int d\_score;

int d\_rank;

// ...

The allocator, though unused when TYPE is not AA, takes up space in the object footprint, but that is often an acceptable cost to pay for the simplicity of this approach.

If the extra pointer-sized space consumption is an issue or if a pure AA or non-AA interface is important, then some refactoring and metaprogramming will be required. We begin by declaring our Thing template with an extra Boolean parameter that defaults to true if TYPE is AA and false otherwise:[[34]](#footnote-35)

template <class TYPE**, bool USES\_ALLOC =**

**BloombergLP::bslma::UsesBslmaAllocator<TYPE>::value**>

class Thing;

The partial specialization where USES\_ALLOC is true supplies the entire AA interface:

template <class TYPE>

class Thing**<TYPE, true>** {

**bslalg::ConstructorProxy<TYPE>** d\_data**Proxy**;

int d\_score;

int d\_rank;

public:

// PUBLIC TYPES

**using allocator\_type = bsl::allocator<char>;**

**// TRAITS**

**BSLMF\_NESTED\_TRAIT\_DECLARATION(Thing, bslma::UsesBslmaAllocator);**

// 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;**

};

Although the TYPE is known to be AA, we still use bslalg::ConstructorProxy because it automatically handles the leading-allocator vs. trailing-allocator convention for us. 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 Thing constructors pass an allocator to d\_dataProxy:

template <class TYPE>

Thing<TYPE, true>::Thing(const TYPE& data,

const allocator\_type& allocator)

: **d\_dataProxy(data, allocator.mechanism())**

, d\_score(0)

, d\_rank(5)

{

}

The Thing::get\_allocator method retrieves the allocator from the d\_dataProxy member[[35]](#footnote-36):

template <class TYPE>

allocator\_type Thing<TYPE, true>::get\_allocator() const  
{

**return d\_dataProxy.object().get\_allocator();**

}

We create the partial specialization where USES\_ALLOC is false to contain a member variable of the other (AA) specialization, forwarding most types, constructors, and member function implementations, but leaving out allocator\_type, get\_allocator, the bslma::UsesBslmaAllocator trait, and allocator function parameters[[36]](#footnote-37):

template <class TYPE>

class Thing<TYPE, **false**> {

// DATA

**Thing<TYPE, true> d\_imp;**

public:

// CREATORS

Thing() **: d\_imp() { }**

explicit Thing(const TYPE& data) **: d\_imp(data) { }**

...

const TYPE& data() const **{ return d\_imp.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 ignored by bslalg::ConstructorProxy. The get\_allocator() method is never instantiated for non-AA TYPEs, so no compilation errors result from its use of TYPE::get\_allocator.

Unfortunately, duplicate declarations of each member function are present in both specialized versions 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.

# 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 implement MyList, a simple doubly-linked list AA container template. Our simplified template declaration looks as follows:

template <class TYPE>

struct MyList\_Node;

template <class TYPE>

class MyList {

bsl::allocator<char> d\_allocator;

MyList\_Node \*d\_head\_p, \*d\_tail\_p;

public:

using allocator\_type = bsl::allocator<char>;

// TRAITS

BSLMF\_NESTED\_TRAIT\_DECLARATION(MyList, bslma::UsesBslmaAllocator);

// 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 on top of 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 in the node.
3. Link the new node onto the list.

Separating step 1 from step 2 is simpler than combining them (explanation follows shortly). To achieve this separation, we need to be able to construct a node without constructing the object that will be stored within that node. The bsls::ObjectBuffer template is designed for this purpose, reserving properly aligned space for an object without actually constructing or destroying the object:

template <class TYPE>

struct MyList\_Node {

**bsls::ObjectBuffer<TYPE> d\_value**;

MyList\_Node \*d\_prev\_p;

MyList\_Node \*d\_next\_p;

};

This simple node type is trivially constructible and destructible, so default construction and destruction can be considered free. The node template is also agnostic to whether or not TYPE is AA. The alternative to using bsls::ObjectBuffer would be to use bslalg::ConstructorProxy, but that would require MyList\_Node to itself be conditionally AA, with all of the complications and extra work that entails.

We create a new node (step 1 in our list above) by using placement operator new, as we did to initialize d\_data\_p earlier:

template <class TYPE>

template <class... ARGS>

void MyList<TYPE>::emplace\_back(ARGS&&... args)

{

MyList\_Node<TYPE> \*node\_p =

**new(\*get\_allocator().mechanism()) MyList\_Node<TYPE>;**

Next, we construct a TYPE object in the node (step 2). If TYPE is AA, we want to construct it using the list’s allocator. We’ll use the bslma::ConstructionUtil::construct function template which, like bslalg::ConstructorProxy, normalizes the constructor argument list. The second argument to construct is an allocator that is either ignored or passed to the object’s constructor using the leading-allocator or trailing-allocator convention, based on TYPE’s traits.

// NOT YET SAFE IN THE PRESENCE OF EXCEPTIONS (FIX TO FOLLOW)!

bslma::ConstructionUtil::construct(node\_p->d\_value.address(),

**get\_allocator()->mechanism(),**

std::forward<ARGS>(args)...);

This call to construct might throw an exception from TYPE’s constructor, which would cause emplace\_back to exit without returning the node to the allocator. We’ll address this memory leak in a moment, but let’s first finish our exposition of the nonexceptional path.

The final step is to link the new node into the existing list:

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;

}

}

Returning to the potential memory leak we just identified, we need a way to reliably deallocate the node (i.e., return the memory to the allocator) if the emplace\_back operation fails to complete after already having allocated memory (e.g., as a result of an exception). The best way to achieve this exception safety is to use an RAII[[37]](#footnote-38) object whose destructor will automatically rewind a 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 still under the proctor’s control. In the emplace\_back example, we will use a bslma::RawDeleterProctor to unwind the allocation of the node and a bslma::DestructorProctor to unwind the construction of the element if an exception occurs during the emplace\_back operation.[[38]](#footnote-39) The complete emplace\_back method looks as follows:

template <class TYPE>

template <class... ARGS>

void MyList<TYPE>::emplace\_back(ARGS&&... args)

{

MyList\_Node<TYPE> \*node\_p =

new(\*get\_allocator().mechanism()) MyList\_Node;

**bslma::RawDeleterProctor<bslma::Allocator> delProct(node\_p,**

**get\_allocator().mechanism());**

bslma::ConstructionUtil::construct(node\_p->d\_value.address(),

get\_allocator()->mechanism(),

std::forward<ARGS>(args)...);

**// Proctor to unwind the 'construct' is not needed (see below).**

**//bslma::DestructorProctor<TYPE> dtorProct(node\_p->d\_value.address());**

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;

}

// Success! Release all proctors.

**// dtorProct.release();**

**delProct.release();**

}

The bslma::DestructorProctor is shown here for illustrative purposes but is commented out because, in this example, no potentially throwing operations occur after construction of the element that would necessitate unwinding the construction.

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, using bslma::DestructionUtil to destroy the element and deleteObject to release its node back to the allocator.[[39]](#footnote-40) Using a proctor is unnecessary because neither destruction nor deallocation should ever throw an exception and, even if they did, no method exists to unwind these operations:

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::DestructionUtil::destroy(node\_p->d\_value.address());**

**get\_allocator().mechanism()->deleteObject(node\_p);**

}

}

This separation between allocating memory and constructing an element in the allocated memory applies as well to array-based containers such as vector. Instead of allocating one node at a time containing one element each, array-based containers allocate a block of memory suitable for holding a contiguous array of elements using the allocator’s allocate method:

// Following is unnecessary if allocator\_type is bsl::allocator<TYPE>

bsl::allocator<TYPE> typedAlloc = get\_allocator();

d\_data = typedAlloc.allocate(*numElements*);

Operations that construct multiple elements, such as a resize operation, would use a bslma::AutoDestructor to protect all currently constructed elements that need to be unwound instead of creating a separate DestructorProctor for each element constructed.[[40]](#footnote-41)

# 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]](#footnote-42) When testing an AA class, we should test that

1. all memory belonging to the object is allocated from its allocator and not from global operator new or the default allocator;
2. the object gives all owned memory back 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 during copy or move assignment;
5. AA elements in containers use the same allocator as the container;
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; and
7. the object allocates exactly as much memory (blocks and/or bytes) as expected (i.e., a white-box test). (Note that this test is optional.)

The bslma\_testallocator and bslma\_testallocatormonitor components facilitate achieving these test goals.[[42]](#footnote-43) 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 on a specific allocation attempt 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 gone up, down, or stayed the same since it was created or last reset, and by how much.

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 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

theList.pop\_back();

**ASSERT(tam.isInUseSame());** // Back to original memory use

// ... Other code that uses 'ta' could go here 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 with one that directs all allocation requests to a specific test allocator (which gets its memory from malloc, not operator new, thus avoiding recursion)[[43]](#footnote-44):

namespace { **bslma::TestAllocator opNewAllocator**; }

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 in 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:

**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 unused in block**

**ASSERT(onm.isTotalSame()); // 'operator new' 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 propagation of the allocator itself. For example, once we have tested that calling get\_allocator returns the expected allocator (defaulted or not), we can choose any allocator we want for performing the allocation and deallocation tests; testing allocation and deallocation with different allocator arguments is unnecessary:

// Test of default and extended-default constructor

MyList<int> list1;  
ASSERT(&da == list1.get\_allocator());

MyList<int> list2(&ta);

ASSERT(&ta == list2.get\_allocator());

**ASSERT(0 == ta.numBlocksInUse()); // Needs to be tested only once per ctor**

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 requirements:

1. The nonextended copy constructor must create an object with the default allocator and allocate from only that allocator, regardless of the allocator held by the copied-from object.
2. The nonextended move constructor typically should not allocate from any allocator. If allocation is required, then such allocation must draw from the same allocator as the moved-from object.
3. The extended copy constructor must allocate only from the specified allocator.
4. The extended move constructor should behave like the regular move constructor if the allocator designated for the moved-to object is the same as the allocator of the moved-from object. Otherwise the extended move constructor should behave like the extended copy constructor.
5. The copy-assignment operator must allocate only from the allocator of the left-hand operand.
6. The move-assignment operator should not allocate if the moved-to object uses the same allocator as the moved-from object; otherwise, the move-assignment operator should behave like the copy-assignment operator.

When testing a class template, the facilities in the bsltf package[[44]](#footnote-45) can be of help. The bsltf::AllocTestType, for example, is a very simple 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.allocator()); // Uses specified allocator

theList.emplace\_back(val5);

**ASSERT(&ta1 == theList.back().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 to run an exception-safety test on a block of code having a deterministic allocation pattern. The macros execute the code within a try block, catching the exceptions thrown by the test allocator. Starting 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. Thus, to test the exception safety of emplace\_back, we write:

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

**BSLMA\_TESTALLOCATOR\_EXCEPTION\_TEST\_BEGIN(ta) {**

MyList<bsltf::AllocTestType> theList(&ta);

for (int i = 0; i <= 5; ++i) {

bsltf::AllocTestType v(i); // Uses default allocator

theList.emplace\_back(v);

}

ASSERT(5 == theList.back().data()); // Minimal sanity 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 forgotten to add the proctor in our emplace\_back implementation, 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 a class with a destructor that encapsulates those postcondition checks. The class can be either general or tailored for a very 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 of 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 as follows:

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.

# Conclusion

Ensuring that reusable components are allocator aware 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.

For simple types such as structs and attribute types, the basic guidelines shown in this paper allow readers to swiftly and correctly plumb their own types to be AA. For more sophisticated components, the (open-source) BDE[[45]](#footnote-46) infrastructure provides low-level components (e.g., bslma\_constructionutil and bslma\_testallocator) and tools[[46]](#footnote-47) 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: 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 in the bslmf\_movableref component. Using bslmf::MovableRef instead of rvalue references, we are able to 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.

A quick summary of bslmf\_movableref[[47]](#footnote-48):

* An instantiation of bslmf::MovableRef<T> emulates T&& (i.e., an rvalue reference) when compiled with a pre-C++11 compiler and is identical to T&& when compiled with a C++11 or later compiler.
* An expression 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(*movableref*) (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 (lvalue or rvalue) ref.

To express move operations in C++03, we will need to declare not only the extended move constructor, but also the normal move constructor and move- assignment operator using bslmf::MovableRef. We will also have to define the normal copy constructor and copy-assignment operator because compiler generation of these operations are suppressed in C++11 by the presence of the user-defined move constructor and move-assignment operator, 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 Rule of Five-plus-two set for a Thing class is shown below. Note that the set includes only six separate members because the copy constructor and extended copy constructor are combined into one:

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();

Thing& operator=(const Thing& rhs);

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

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 optional in both C++03 and C++11 for a nonallocating type because the compiler can generate a correct destructor automatically. Two additional adaptations for C++03 are 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 the section, “Making a Simple struct AA”:

// Combined regular and allocator-extended copy ctor

Thing::Thing(const Thing& original, const allocator\_type& allocator)

: d\_name(original.d\_name, allocator)

, d\_data(original.d\_data, allocator.mechanism())

, 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)),

allocator)

, d\_data(**bslmf::MovableRefUtil::move(**

**bslmf::MovableRefUtil::access(original)**.d\_data)),

allocator.mechanism())

, d\_score(original.d\_score)

, d\_rank(original.d\_rank)

{

}

// Destructor

Thing::~Thing()

{

}

// 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 B: Alternatives to Storing the Allocator Pointer 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 methods 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 of 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 instructions executed.

The most practical approach is to store the allocator on the heap as part of the object’s allocated memory and in the object footprint when the object is empty. In our vector example, we could create a custom vector, UsuallyEmptyVector,[[48]](#footnote-49) where the allocator and the first data pointer share space in a union. The allocator is stored in the union when the vector’s capacity is 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 pmr::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; a one would indicate an empty vector, where the rest of the pointer points to the memory resource, and a zero 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 technically falls into the realm of undefined behavior 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[[49]](#footnote-50) doesn’t actually reduce the object footprint but uses the footprint more efficiently 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 other than 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 particular 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 allocators*.[[50]](#footnote-51) 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.

# Appendix C: 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 standard std::pmr::memory\_resource abstract base class is nearly identical to Bloomberg’s bslma::Allocator class, the major differences being 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 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.[[51]](#footnote-52) Note that BDE’s old-style 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 nearly identically 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 C++20 Standard Library contains no proctors, but std::unique\_ptr can be even more effective in this role with the appropriate use of deleters, easy-to-author types that operate on a specified address when a unique\_ptr goes out of scope. Creating deleters that mimic the behavior of the destructors for bslma::DeallocatorProctor and bslma::DestructorProctor would be a trivial task. A proposal in the C++ Standards Committee awaits and 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.[[52]](#footnote-53) Additionally, bslma::TestAllocator has no equivalent in the C++ Standard Library, but one has been proposed and source code is available.[[53]](#footnote-54)

# Appendix D: Converting from Old-style AA to New-style AA

As of June 2020, most AA classes at Bloomberg use 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 new style provides the following benefits.

* The new style is more similar to the C++ Standard’s pmr style and will eventually be 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 new style is compatible with C++11 AA constructs. For example, a new-style AA type X can be stored in a vector<X, std::scoped\_allocator\_adaptor<bsl::allocator<X>> and will correctly inherit its allocator instance from the vector.

To convert a class from the old-style AA interface to the new-style interface, follow the listed steps.

1. Make bsl::allocator available for use:

|  |  |
| --- | --- |
| Add | #include <bslma\_stdallocator.h> |

1. Add an allocator\_type alias (or typedef in C++03) into the public part of the class:

|  |  |
| --- | --- |
| Add | public:  // TYPES  **using allocator\_type = bsl::allocator<char>;** |

1. 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; |

1. Change any constructors that take a bslma::Allocator\* parameter to take a const allocator\_type& parameter instead. Since it is no longer a base-class type, change the name basicAllocator to just allocator. If the allocator parameter has a 0 default value, change it 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.

1. Remove the use of bslma::Default::allocator for unnullifying 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) |

1. If the class code initializes any old-style AA class members, then call the mechanism() method on the allocator to get the bslma::Allocator pointer:

|  |  |
| --- | --- |
| Before | Thing::Thing(int, bslma::Allocator \*basicAllocator)  : d\_name(basicAllocator) // new-style AA member  , d\_data(basicAllocator) // old-style AA member |
| After | Thing::Thing(int, const allocator\_type& allocator)  : d\_name(allocator) // new-style AA member  , d\_data(allocator**.mechanism()**) // old-style AA member |

1. Add a new get\_allocator method but keep a modified version of the old-style allocator method for backward compatibility.

|  |  |
| --- | --- |
| Before | inline bslma::Allocator \*Thing::allocator() const  { return d\_alloc; } |
| After | inline Thing::allocator\_type Thing::get\_allocator() const  { return d\_alloc; }  inline bslma::Allocator \*Thing::allocator() const  { return **get\_allocator().mechanism();** } |

1. Until the BDE infrastructure catches up to the new style (expected in 2020) raw allocation, deallocation, and construction of dynamically-allocated objects will rely on the bslma::Allocator\*-based tools in the bslma package, again through the use of get\_allocator().mechanism(). Because the allocator() method remains for backward compatibility, the “Before” code below will continue to work properly but doesn’t reveal that that a compatibility feature is being used.

|  |  |
| --- | --- |
| Before | MyList\_Node<TYPE> \*node\_p = new(\***allocator()**) MyList\_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 | MyList\_Node<TYPE> \*node\_p =  new(\***get\_allocator().mechanism()**) MyList\_Node;  bslma::DeallocatorProctor<bslma::Allocator>  nodeProct(node\_p, **get\_allocator().mechanism()**);  bslma::ConstructionUtil::construct(node\_p->d\_value.address(),  **get\_allocator().mechanism()**,  value);  ...  bslma::DestructionUtil::destroy(node\_p->d\_value.address());  **get\_allocator().mechanism()**->deleteObject(node\_p); |

Alternatively, the allocator member can be stored as an appropriate instantiation of bsl::allocator and used directly to allocate and deallocate, but that does not make the code any more compact in most cases.

|  |  |
| --- | --- |
| After (Alter­native) | **bsl::allocator<MyList\_Node<TYPE>> nodeAlloc(get\_allocator());**  **MyList\_Node<TYPE> \*node\_p = nodeAlloc.allocate(1);**  **nodeAlloc.construct(node\_p);** // Begin lifetime  bslma::DeallocatorProctor<bslma::Allocator>  nodeProct(node\_p, **nodeAlloc**.mechanism());  bslma::ConstructionUtil::construct(node\_p->d\_value.address(),  **nodeAlloc**.mechanism(),  value);  ...  bslma::DestructionUtil::destroy(node\_p->d\_value.address());  **nodeAlloc.destroy(node\_p);**  **nodeAlloc.dallocate(node\_p);** |

1. The existing test driver should compile and run with no changes. However, you should at least add tests for the bsl::uses\_allocator trait and the get\_allocator method:

|  |  |
| --- | --- |
| 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<char>>::value));**  ...  Obj mX(&theTestAlloc); const Obj& X = mX;  ASSERT(&theTestAlloc == X.allocator());  **ASSERT(&theTestAlloc == X.get\_allocator());** |

# Works Cited

**alexandrescu04.** A. Alexandrescu, “Write Less Code and More Software,” Amazon, June 4, 2004.  
<https://youtu.be/Lv5vQXraGJM?t=2773>

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

**bloombergb.** “Component bslmf\_movableref,” *BDE API Documentation,* Bloomberg, accessed May 6, 2020.  
[https://bloomberg.github.io/bde‑resources/doxygen/bde\_api\_prod/  
group\_\_bslmf\_\_movableref.html](https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bslmf__movableref.html)

**bloombergc.** “Component bslma\_autodestructor,” *BDE API Documentation,* Bloomberg, accessed May 6, 2020.  
[https://bloomberg.github.io/bde‑resources/doxygen/bde\_api\_prod/  
group\_\_bslma\_\_autodestructor.html](https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bslma__autodestructor.html)

**bloombergd.** “Component bslma\_testallocator,” *BDE API Documentation,* Bloomberg, accessed May 6, 2020.  
[https://bloomberg.github.io/bde‑resources/doxygen/bde\_api\_prod/  
group\_\_bslma\_\_testallocator.html](https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bslma__testallocator.html)

**bloomberge.** “Component bslma\_testallocatormonitor,” *BDE API Documentation,* Bloomberg, accessed May 6, 2020.  
[https://bloomberg.github.io/bde‑resources/doxygen/bde\_api\_prod/  
group\_\_bslma\_\_testallocatormonitor.html](https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bslma__testallocatormonitor.html)

**bloombergf.** “Package bsltf,” *BDE API Documentation,* Bloomberg, accessed May 6, 2020.  
[https://bloomberg.github.io/bde‑resources/doxygen/bde\_api\_prod/  
group\_\_bsltf.html](https://bloomberg.github.io/bde-resources/doxygen/bde_api_prod/group__bsltf.html)

**bloombergg.** “Component bslma\_usesbslmaallocator,” *BDE API Documentation,* Bloomberg, accessed May 6, 2020.  
[https://bloomberg.github.io/bde‑resources/doxygen/bde\_api\_prod/  
group\_\_bslma\_\_usesbslmaallocator.html](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>

**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://www.youtube.com/watch?v=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, 2019.  
[http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/  
p1160r1.pdf](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, 2020.  
[http://www.open‑std.org/jtc1/sc22/wg21/docs/papers/2020/  
p2035r0.pdf](http://www.openstd.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, 2020.  
[http://www.open‑std.org/jtc1/sc22/wg21/docs/papers/2020/  
p2126r0.pdf](http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2020/p2126r0.pdf).

**halpern20c**. P. Halpern. “A Language Facility for Making Types Allocator Aware (AA),” forthcoming.

**iso17.** “ISO/IEC 14882:2017 Programming Language C++,” *International Standards Organization (ISO) Draft International Standard N4660,* March 2017. (Public version available as N4659.)  
[http://www.open‑std.org/JTC1/SC22/WG21/docs/papers/2017/  
n4659.pdf](http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2017/n4659.pdf)

**john18.** G. John, “Rule of Three vs Rule of Five in C++?” *Tutorialspoint*, 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 2020.  
[http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/  
p0211r3.html](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 2019.   
<https://youtu.be/ebn1C-mTFVk?t=1770>

**llvm19.** “llvm::SmallVector< T, N > Class Template Reference,” *LLVM 10.0.0 Documentation*, November 2019.

**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 2018.  
[http://www.open‑std.org/jtc1/sc22/wg21/docs/papers/2019/  
p0052r10.pdf](http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2019/p0052r10.pdf)

1. Motivational background can be found in **halpern20a**. Information on using AASI can be found in **halpern20b**. [↑](#footnote-ref-2)
2. BDE is an initialism that began as Bloomberg Development Environment and is now understood to simply describe a group within Bloomberg. [↑](#footnote-ref-3)
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**. [↑](#footnote-ref-4)
4. This data pertains to BDE (library) source code (c. May 2017); see **lakos19**, time index 29:30. [↑](#footnote-ref-5)
5. The bde\_verify tool [**bloombergh**] already checks and even automates many of the mundane tasks. [↑](#footnote-ref-6)
6. Work is underway to integrate allocator awareness into the C++ language and compiler; see **meredith19** and **halpern20c**. [↑](#footnote-ref-7)
7. As of April 2020, the bslstl\_optional component is still under development. [↑](#footnote-ref-8)
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 std::pmr::polymorphic\_allocator in C++17) we can ignore them when applying the polymorphic allocator style described in this paper. [↑](#footnote-ref-9)
9. [**bloomberga**] [↑](#footnote-ref-10)
10. The relationship between the C++ Standard and the style described in this paper is detailed in Appendix C. [↑](#footnote-ref-11)
11. Logically, uses\_allocator evaluating to true would automatically cause UsesBslmaAllocator to evaluate to true, but, as of June 2020, that is not the case. [↑](#footnote-ref-12)
12. 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, e.g., container emplace methods where the allocator is supplied implicitly by the called method and not by the caller. [↑](#footnote-ref-13)
13. Local-scoped variables almost always use the default allocator. For more information about choosing the right allocator, see **halpern20b**, page 9. [↑](#footnote-ref-14)
14. The older 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 nonnull; otherwise the currently installed default allocator. Failing to do 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. [↑](#footnote-ref-15)
15. Polymorphic Memory Resource; see Appendix C [↑](#footnote-ref-16)
16. The standard types were modeled directly on the Bloomberg types but, with the benefit of hindsight, are a bit more evolved. For example, pmr::memory\_resource lets the caller specify a required alignment when allocating memory whereas bslma::Allocator does not. Work is ongoing to improve bslma::Allocator and bsl::allocator to be fully compatible with the standard types. [↑](#footnote-ref-17)
17. **iso17**, section 11.6.1, “dcl.init.aggr,” pp. 224–228 [↑](#footnote-ref-18)
18. Several methods, including 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 index 49:32. [↑](#footnote-ref-19)
19. In C++03, the type alias must be written “typedef bsl::allocator<char> allocator\_type;”. [↑](#footnote-ref-20)
20. If the two constructor overloads are documented separately, the comments would say, “Use the default allocator to supply memory,” and “Use the specified 'allocator' (e.g., the address of a 'bslma::Allocator' object) to supply memory,” respectively. [↑](#footnote-ref-21)
21. **john18** [↑](#footnote-ref-22)
22. **iso17**, section 11.6.4, “dcl.init.list,” pp. 231–236 [↑](#footnote-ref-23)
23. **halpern20c** 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. [↑](#footnote-ref-24)
24. Similarly, if we choose to provide C++03 move operations, we declare and implement those operations identically to the simple struct example. See Appendix A for detailed instructions and caveats for implementing move operations in C++03. [↑](#footnote-ref-25)
25. 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 **merideth19** and **halpern20c**. [↑](#footnote-ref-26)
26. 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. [↑](#footnote-ref-27)
27. Warning: passing a pointer to an allocator would still compile but with the undesirable runtime semantics of corrupting the allocator. [↑](#footnote-ref-28)
28. For this particular case, where the most-derived class for \*d\_data\_p is known statically, deleteObjectRaw might be slightly faster than deleteObject. [↑](#footnote-ref-29)
29. 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. [↑](#footnote-ref-30)
30. The BDE rule is that the extended move constructor must behave like the regular move constructor if the allocators match and must behave like the extended *copy* constructor if the allocators do not match. See “Expected Properties of Types Declaring the bslma::UsesBslmaAllocator Trait” in **bloombergg**. [↑](#footnote-ref-31)
31. The AA interface is designed so that all of 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 of 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). [↑](#footnote-ref-32)
32. Some algorithms cannot meet their complexity guarantees unless elements can be swapped in constant time. [↑](#footnote-ref-33)
33. As Bloomberg migrates more of the code base to the new AA interface style, utilities such as ConstructorProxy will be enhanced to work with bsl::allocator as well. [↑](#footnote-ref-34)
34. This metaprogramming technique is difficult to use for variadic class templates; indirection through inheritance or alias templates is required in this case. [↑](#footnote-ref-35)
35. This implementation of get\_allocator depends on TYPE using the new-style AA interface. Work is underway on a utility that will provide a unified interface for both AA styles. [↑](#footnote-ref-36)
36. Forwarding functions in the class definition are for simplicity of exposition. BDE style dictates that the function bodies should appear outside of the class, to avoid clutter. [↑](#footnote-ref-37)
37. RAII is an acronym 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. [↑](#footnote-ref-38)
38. The C++ standard unique\_ptr [**iso17**, section 23.11.1, “unique.ptr,” pp. 609-620]can be used as a proctor but requires a special deleter that is not yet standard (see **köppe20**). Additionally, the scope\_exit and scope\_fail templates described in **sommerlad19** can take the place of proctors, but are not yet standardized. [↑](#footnote-ref-39)
39. Because the node destructor is trivial, we are allowed to skip it by calling deallocate instead of deleteObject, but that is unlikely to yield any speed or code-size improvement in this case. [↑](#footnote-ref-40)
40. See **bloombergc**. [↑](#footnote-ref-41)
41. Some of the techniques described in this section are demonstrated in **fehér19a**, starting at time index 18:08. [↑](#footnote-ref-42)
42. Documentation is available in **bloombergd** and **bloomberge**. [↑](#footnote-ref-43)
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. [↑](#footnote-ref-44)
44. **bloombergf** [↑](#footnote-ref-45)
45. **bloombergi** [↑](#footnote-ref-46)
46. See also **bloombergh** for a description of the bde\_verify tool, which detects a number of allocator-related errors and offers other guidance. [↑](#footnote-ref-47)
47. See complete documentation in **bloombergb**. [↑](#footnote-ref-48)
48. 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. [↑](#footnote-ref-49)
49. **alexandrescu04**, time index 46:13 [↑](#footnote-ref-50)
50. *Inverse Lookup Allocator Registry*, invented by Hyman Rosen of Bloomberg [↑](#footnote-ref-51)
51. This functionality is being considered for bsl::allocator in a future release of the BDE library. [↑](#footnote-ref-52)
52. **köppe20** [↑](#footnote-ref-53)
53. **fehér19b** provides the proposal, and **fehér19c** offers the source for a reference implementation. [↑](#footnote-ref-54)